

A Partial Evaluator for the Untyped Lambda Calculus ¹

Carsten K. Gomard, Neil D. Jones

gomard@diku.dk, neil@diku.dk
DIKU, Department of Computer Science
University of Copenhagen

Summary

This article describes theoretical and practical aspects of an implemented self-applicable partial evaluator for the untyped lambda calculus with constants and a fixed point operator. To the best of our knowledge, it is the first partial evaluator that is simultaneously higher-order, nontrivial, and self-applicable.

Partial evaluation produces a *residual program* from a source program and some of its input data. When given the remaining input data the residual program yields the same result that the source program would when given all its input data. Our partial evaluator produces a residual λ -expression given a source λ -expression and the values of some of its free variables. By self-application, the partial evaluator can be used to compile and to generate stand-alone compilers from a denotational or interpretive specification of a programming language.

An essential component in our self-applicable partial evaluator is the use of explicit *binding time information*. We use this to annotate the source program, marking as *residual* the parts for which residual code is to be generated and marking as *eliminable* the parts that can be evaluated using only the data that is known during partial evaluation. We give a simple criterion, *well-annotatedness*, that can be used to check that the partial evaluator can

¹An extended abstract of this paper has appeared in the proceedings of the IEEE Conference on Programming Languages [Jones *et al.* 1990].

handle the annotated higher order programs without committing errors.

Our partial evaluator is surprisingly simple, is implemented in a side-effect free subset of Scheme, and has been used to compile, to generate compilers, and to generate a compiler generator. In this article we examine two machine generated compilers and find that their structures are surprisingly natural.

Contents

Preface	3
Acknowledgements	4
1 Introduction	5
1.1 Background	5
1.2 Prerequisites, Overview and Outline	7
1.3 Partial Evaluation and the Futamura Projections	8
1.3.1 Partial Evaluation	9
1.3.2 The Futamura Projections	10
2 Partial Evaluation Using a Two-Level Lambda Calculus	11
2.1 An Untyped Lambda Calculus	12
2.1.1 Syntax	13
2.1.2 Semantics	14
2.2 Two-level Syntax	14
2.3 Two-level Semantics	16
2.4 A Bird's Eye View of How Mix Works	19
2.5 "Well-Annotated Programs Do Not Go Wrong"	19
2.5.1 Well-Annotated Expressions	20
2.5.2 The Existence of Best Completions	23
2.6 The Mix Equation Revisited	24
3 Aspects of Binding Time Analysis by Type Inference	25
3.1 BTA: Type Analysis of Untyped Programs?!	25
3.2 Viewing Binding Time Analysis as Type Inference	27
3.3 Finiteness of Partial Evaluation	29
3.4 Code Duplication	30
4 Experiments with Mix	31
4.1 A Self-Interpreter	32
4.1.1 "L mix [sint, p] = p" is important!	33
4.1.2 A Self-Compiler	34

4.1.3	A Compiler Generator	34
4.1.4	Performance	35
4.2	An Interpreter for a Tiny Imperative Language	36
4.3	An Example of Compiler Generation	38
4.3.1	Performance	39
5	Perspectives and Conclusions	40
5.1	Related Work	40
5.1.1	Partial Evaluation	40
5.1.2	Binding Time Analysis	42
5.1.3	Semantics-directed compiler generation	42
5.2	Future Work	44
5.2.1	Target Code Quality	44
5.2.2	Specialization of Named Combinators	44
5.2.3	Applications of BTA	46
5.3	Conclusion	47
	Bibliography	48
	A A Mix Session	54
	B T Annotated	59
	C The Generated Self-Compiler	60
	D The Generated Compiler Generator	62
	E Tiny Interpreter	64
	F The Generated Tiny Compiler	66
	Last Page	68

Preface

This article develops a simple self-applicable partial evaluator for a higher order language, the lambda calculus. Examples demonstrate that the partial evaluator can be used automatically to generate a language implementation in the form of a compiler, given as input a from language definition in the form of a denotational semantics.

Self-applicable partial evaluators for first order languages had been around for four years before this project was begun. The goal was to generalize the techniques that worked well for first order languages to include the higher order languages with their higher expressive power. We have succeeded in doing higher order partial evaluation, but the techniques used are somewhat different—and simpler!—than those used in earlier partial evaluation projects.

To read this article, a superficial knowledge of denotational semantics, typed programming languages and type inference systems will be beneficial.

Acknowledgements

The goal of partially evaluating lambda expressions has been in the minds of the programming language theory group at DIKU since the first successes with first order languages in 1984. Recent insights grew from numerous discussions in this group, including in particular Anders Bondorf, Olivier Danvy, and Torben Mogensen, as well as Carsten Kehler Holst, Thomas Jensen, and Peter Sestoft. We also thank the guests and other acquaintances of the group that have stimulated the work by showing their interest (John Hughes, John Launchbury, Carolyn Talcott, Phil Wadler, and many others).

1 Introduction

1.1 Background

A language's implementation should be guided by its precise semantic definition. It seems an impossible task to implement a realistic language correctly on the basis of a loose idea of how it should work. Even with a precise—formal or informal—semantic description it is neither a small nor an easy task to implement a compiler correctly.

Therefore the mechanical derivation of a language implementation from a semantic definition has received great attention as an area of research during the last ten years. The goal is clear: From a language definition expressed in a not too cumbersome formalism automatically to derive an efficient implementation that is faithful to the semantic definition.

An important formalism for assigning meanings to programs is *denotational semantics* [Stoy 1977, Schmidt 1986] founded by Scott and Strachey. A denotational definition assigns to each program a lambda expression denoting the input–output function computed by the program. Denotational semantics was intended to be the mathematical theory of programming languages, saying *what* the meaning of a program is, but nothing about *how* to compute it. How to compute it was considered irrelevant to understanding the meaning of a program.

Since the lambda calculus can be implemented², a denotational definition of a programming language can actually run directly on a machine. We may thus view a denotational definition as an *interpreter* for the defined language, meaning that we have for free a language implementation derived (without doing anything) from a formal language definition. Could we ask for more? Yes, we ask for *efficiency*.

A denotational semantics defines the meanings of programs in a programming language by translating them into the lambda calculus. The lambda expression resulting from the translation has very large computational overhead; for efficient implementation, it is necessary to simplify the expression before it is applied to the program input and executed. This was the strat-

²Lisp, Scheme, ML, Miranda *etc.* are all based on the lambda calculus

egy used in SIS: the Semantics Implementation System of [Mosses 1979], the first in a long series of systems to derive implementations from denotational definitions. To our knowledge none of these systems is so powerful (or so simple) that one could consider using the system to construct its own components. All are quite complex, with many stages of processing and intermediate languages. In contrast, the partial evaluator presented here involves only one language (with annotations), and all components are derived from a single program, “mix”.

Partial evaluation is another approach to the generation of language implementations. Partial evaluation is a program transformation technique specializes programs with respect to given, incomplete input data. During the seventies it was realized independently in Japan and the Soviet Union [Futamura 1971, Turchin 1980, Ershov 1978] that given a self-applicable partial evaluator it was possible to generate a compiler for a language, given as input an operational definition in the form of an interpreter. It was not, however, until 1984 that the first non-trivial self-application was realized on the computer [Jones *et al.* 1985, Jones *et al.* 1989].

The language used in that project was first order recursion equations. Since then partial evaluators have been developed for various other first order languages, but until now no higher order solution has been developed. Partial evaluation of higher order languages is clearly desirable because of their expressive power and elegance.

When a compiler is generated by self-application of a partial evaluator, the partial evaluator is the only program involved apart from the language definition. Furthermore a partial evaluator is usually a relatively small program. It is thus much easier to prove the generated compiler correct (meaning: faithful to the input language definition) because “all” it takes is to prove the partial evaluator correct. Once this is done (and it has been done: see [Gomard 1989] and part II of this paper), every generated compiler will be faithful to the language definitions from which they were derived. This completely obviates the need for the difficult intellectual work involved in proving individual compilers correct [Goguen *et al.* 1977, Morris 1973].

In this article we merge the two threads of development sketched above. We construct a self-applicable partial evaluator for the lambda calculus with unrestricted use of higher order functions and apply it to some small example denotational definitions of programming languages. The emphasis will be on *what* the partial evaluator does, including self-application, and on *how* it is done, via the two-level lambda calculus and binding time analysis. Part II of this article is more mathematical, showing *why* mix functions correctly.

1.2 Prerequisites, Overview and Outline

The reader should be familiar with denotational semantics (e.g. [Schmidt 1986]) and should have some knowledge of partial evaluation.

After a summary of this article's contents, the rest of section 1 is devoted to a review and overview of the basic theory of partial evaluation and compiler generation (more details may be found in [Jones *et al.* 1985, Jones *et al.* 1989]).

A central part of the article is section 2 where the syntax and semantics of our one- and two-level lambda calculi are presented. The task of partial evaluation is split into two: First we add annotations to the one-level (normal) lambda expression to obtain a two-level expression, and then evaluate the annotated expression. A type system that assures the partial evaluation against type errors is derived from the semantics of the two-level lambda calculus.

Section 3 demonstrates how the notion of “type analysis of untyped programs” provides a convenient framework for doing binding time analysis.

Section 4 reports the results of experiments with our partial evaluator. We use an interpreter for a small imperative language to demonstrate compiling and compiler generation, and describe experiments with a metacircular interpreter for the lambda calculus itself. We investigate the structure of the target programs, the generated compiler, and the generated compiler generator, all of which turn out to be very natural. Finally the run times of our experimental program executions are given.

Section 5 discusses related work and natural work to follow after this. Finally we summarize the achievements of this article.

In the appendices a Scheme session shows how the implemented partial evaluator works. Various program texts and generated residual programs may also be found there.

1.3 Partial Evaluation and the Futamura Projections

The concept of partial evaluation and the possibility of compiler generation by self-application of a partial evaluator have been exploited in a number of papers, e.g., [Futamura 1971, Jones *et al.* 1985, Jones *et al.* 1989]. In the following we review the basic definitions.

In partial evaluation we treat programs as data objects and it is therefore natural to use a universal domain D from which we draw both programs and their data. We identify a programming language \mathbf{L} with its semantic (partial) function $D \xrightarrow{p} D \xrightarrow{p} D$ which maps each valid \mathbf{L} -program into its input-output function. From now on \mathbf{L} is the lambda calculus and D contains the set of all Lisp S-expressions. An expression is identified with its representation as a list, e.g., `(lam x (@ x x))` for $\lambda x. x @ x$ (a notational convention known at MIT as “Cambridge Polish”).

We supply the input to a lambda calculus program p through its free variables $\{x_1, \dots, x_m\}$ so the input to the program is thus m values, v_1, \dots, v_m . We define

$$\mathbf{L} p [v_1, \dots, v_m]$$

to be equal to $\mathcal{E}[[p]]\rho$ (the semantic evaluation function \mathcal{E} will be formally defined in section 2.1), where $\rho = [x_1 \mapsto v_1 \dots, x_m \mapsto v_m]$. Consider, for example, the following exponentiation program with free variables n and x . (This program, called *power*, is written in an informal notation; properly it should be in Cambridge Polish.)

```

letrec p n' x' =
  if n' = 0
  then 1
  else x' * p (n'-1) x

```

in $p \ n \ x$

Calling the program *power* and letting $\rho = [n \mapsto 2, x \mapsto 3]$, we have

$$\mathbf{L} \text{ power } [2, 3] = \mathcal{E} \llbracket \text{power} \rrbracket \rho = 9.$$

1.3.1 Partial Evaluation

Suppose now that p is a lambda calculus program with two free variables. Then $\mathbf{L} \ p \ [d1, d2]$ denotes the value of p with input $d1$ and $d2$ substituted for the free variables. If only $d1$ is available, application of the semantic function $\mathbf{L} \ p \ d1$ does not make sense, as the result is likely to depend on $d2$. However, $d1$ might be used to perform *some* of the computations in p , yielding as result a transformed, optimized version of p . We use the term *partial evaluation* for the process of doing such computations on basis of incomplete input data. Its outcome is a program, so "partial evaluation" is really a form of *program specialization* (or transformation).

A *residual* program of an \mathbf{L} -program p with respect to partial data $d1$ is a program p_{d1} such that for all $d2$ the following holds (where $=$ denotes equality of partial values):

$$\mathbf{L} \ p \ [d1, d2] = \mathbf{L} \ p_{d1} \ d2$$

A *partial evaluator* is a program *mix* that given p and the partial data $d1$ produces the residual program p_{d1} . This is captured by the *mix equation* for all p and $d1$:

$$\mathbf{L} \ p \ [d1, d2] = \mathbf{L} \ (\mathbf{L} \ \text{mix} \ [p, d1]) \ d2$$

This is just a restatement of Kleene's S_n^m theorem from recursive function theory, with $m = n = 1$. If p is a lambda expression, the free variable in it bound to $d1$ is called *static*, while that bound to $d2$ is called *dynamic*. Generalization to other values of m and n is straightforward.

Existence of *mix* is classically proved in a rather trivial way. For example, if p is the *power* program and $d1 = 2$ is the value of free variable n , the classical residual program $\mathbf{L} \ \text{mix} \ [p, 2]$ would be:

```

letrec p n' x' =   if n'= 0
                   then 1
                   else x' * p (n'-1) x
in p 2 x

```

For our purposes we want more efficient residual programs in which all computations depending on $d1$ have been done by *mix*. A better residual power program for $d1 = 2$ can be got by unfolding applications of the function *p* and doing the computations involving n , yielding the residual program:

```

x * (x * 1)

```

1.3.2 The Futamura Projections

Let **S** and **T** be programming languages, perhaps (but not necessarily) different from **L**. An **S**-interpreter *int* written in **L** is a program that fulfills

$$\mathbf{S} \text{ } pgm \text{ } data = \mathbf{L} \text{ } int \text{ } [pgm, data]$$

for all *data*. An **S**-to-**T**-compiler *comp* written in **L** is a program that fulfills

$$\mathbf{S} \text{ } pgm \text{ } data = \mathbf{T} \text{ } (\mathbf{L} \text{ } comp \text{ } pgm) \text{ } data$$

for all *pgm* and *data*.

The *Futamura projections* [Futamura 1971, Ershov 1978] state that given a partial evaluator *mix* and an interpreter *int* it is possible to compile programs, and even to generate stand-alone compilers and compiler generators by self-applying *mix*. The three Futamura projections are:

$$\begin{aligned}
\mathbf{L} \text{ } mix \text{ } [int, source] &= target \\
\mathbf{L} \text{ } mix \text{ } [mix, int] &= compiler \\
\mathbf{L} \text{ } mix \text{ } [mix, mix] &= compiler \text{ } generator
\end{aligned}$$

Program *target* is a specialized version of **L**-program *int* and so itself an **L**-program, so translation has occurred from the *interpreted* language to the language in which the interpreter itself is written. That the target program is faithful to its source is easily verified using the definitions of interpreters, compilers and the *mix* equation:

$$\begin{aligned}
output &= \mathbf{S} \text{ source input} \\
&= \mathbf{L} \text{ int [source, input]} \\
&= \mathbf{L} (\mathbf{L} \text{ mix [int, source]}) \text{ input} \\
&= \mathbf{L} \text{ target input}
\end{aligned}$$

Verification that program *compiler* correctly translates source programs into equivalent target programs is also straightforward:

$$\begin{aligned}
target &= \mathbf{L} \text{ mix [int, source]} \\
&= \mathbf{L} (\mathbf{L} \text{ mix [mix, int]}) \text{ source} \\
&= \mathbf{L} \text{ compiler source}
\end{aligned}$$

Finally, we can see that *cogen* transforms interpreters into compilers by the following:

$$\begin{aligned}
compiler &= \mathbf{L} \text{ mix [mix, int]} \\
&= \mathbf{L} (\mathbf{L} \text{ mix [mix, mix]}) \text{ int} \\
&= \mathbf{L} \text{ cogen int}
\end{aligned}$$

These proofs and a more detailed discussion can be found in [Jones *et al.* 1989].

2 Partial Evaluation Using a Two-Level Lambda Calculus

To do partial evaluation of a lambda expression p it is tempting to insert the partial input data in p , and apply one of the usual reduction strategies, modified not to reduce when insufficient information is available. But the standard call-by-name and call-by-value reduction strategies (for example as defined in [Plotkin 1975]) do not reduce inside the bodies of abstractions. This approach yields trivial results in practice.

It is no solution either to reduce indiscriminately inside abstractions since this can lead to infinite reduction if the expression contains a fixed point operator, or the Y-combinator written as a lambda expression. The point is that for partial evaluation to succeed, *some, but not all* of the redexes in the expression should be reduced. Many, so as little work as

possible will be left to be done by the residual program; but not so many as to risk nontermination. Thus our main task is to determine which ones to reduce to yield efficient residual programs without risking nontermination.

The classical deterministic reduction orders are *uniform*, meaning: independent of the program being evaluated. This is insufficient for our purposes since the best reduction order may be program-dependent. A solution is to use a *non-uniform reduction strategy*, one which selects redexes in a way depending on the particular program being partially evaluated. A simple technique is to mark parts of the program as “residual”, so these will not be reduced but the remaining ones will.

We therefore do partial evaluation of lambda expressions in two phases. In the first phase, we determine *which* redexes should be reduced at compile-time and which ones should be *residual*, meaning that they should be suspended until run-time. This determination, which is done before knowing the static data, is called *binding time analysis*, henceforth called *BTA* for short. Its importance for efficient self-application is discussed in [Bondorf *et al.* 1988]. The result of applying the BTA to an expression exp is an *annotated* expression exp^{ann} in a two-level lambda calculus. In exp^{ann} the BTA has marked the parts of exp that should *not* be reduced at partial evaluation time.

In the second phase we blindly obey the annotations, reducing redexes not marked as residual, and generating residual target code (also a lambda expression) for the operations marked residual. The reduction phase is performed by applying a semantic function \mathcal{T} —which is an extension of the usual evaluation function \mathcal{E} —to the annotated expression. \mathcal{T} maps residual operators to code pieces for execution at run-time and non-residual operators to their “usual” meanings.

2.1 An Untyped Lambda Calculus

We develop a partial evaluator for a very simple language, the classical lambda calculus. Such a simple language allows a more complete description than would be possible for a larger and more practical language, and makes it possible to carry out proofs of correctness and optimality (see part II).

The lambda calculus forms the basis of modern functional programming languages (e.g., Scheme, ML, Miranda, Haskell), so the results obtained here should not be too hard to adapt to more practical frameworks. The fact that we use an untyped language makes it easier to write interpreters (and thus partial evaluators) and it also allows complete removal of a level of interpretation overhead. We will return to this in section 4.1.1.

2.1.1 Syntax

A lambda calculus program is an *expression* with free variables. The program takes its input through its free variables whose values are supplied by an *initial environment*. This environment is also expected to map base function names, such as `cons`, to the corresponding functions (syntactically these are predefined variables). First order base values include natural numbers and S-expressions and are written `const base-value`.

The expression abstract syntax is as follows, where `@` denotes application and `fix` the least fixed point operator.

$$\text{exp} ::= \text{var} \mid \text{exp} @ \text{exp} \mid \lambda \text{var} . \text{exp} \mid \text{if exp exp exp} \mid \text{fix exp} \mid \text{const base-value}$$

Since we use lambda calculus both as a programming language and as a meta-language, we need to distinguish notationally lambdas that appear in source programs from lambdas that denote functions. Syntactic (source program) lambda expressions are written in sans serif style: `exp @ exp`, `λvar . exp`, `fix exp ...`, and the meta-language is in *slanted* style: *exp @ exp*, *λvar . exp*, *fix exp ...*. When a lambda expression is presented as generated by machine, it is written in **type-writer** style using “Cambridge polish” notation: **(exp exp)**, **(lam var exp)** *etc.*.

For an example, consider a program to compute the function `x` to the `nth` where `x` and `n` are free (input) variables. (For readability we omit some of the explicit `consts` and application nodes in the concrete syntax. We thus write `(= n' 0)` instead of `(= @ n' @ const 0)`.)

$$(\text{fix } \lambda p . \lambda n' . \lambda x' . \text{if } (= n' 0) \quad 1 \quad (* x' (p @ (- n' 1) @ x'))) @ n @ x$$

2.1.2 Semantics

We use denotational semantics [Stoy 1977, Schmidt 1986] to assign meanings to programs, rather than the more traditional $\alpha\beta\eta$ -reduction approach. There are at least two reasons for this:

1. It allows a cleaner analysis of possible errors at partial evaluation time.
2. It is more natural (and yields much better results) for the self-application used in compiler generation.

The Scott domain of expression values Val is the separated sum of the flat domain of base values and the domain of function values: $Val = Base + Funval$.

If f is a value from $Funval$ we take $f \uparrow Funval$ to be the value f injected into summand $Funval$ of Val . Informally: $f \uparrow Funval$ puts a type tag on f . If val is a value from Val , $val \downarrow Funval$ removes the type tag, provided val is from the $Funval$ summand. If not, $val \downarrow Funval$ produces an error. (It is assumed that Val has an error element and that applications *etc.*, are strict in errors; details are omitted for notational simplicity.)

The valuation functions for lambda calculus programs are the usual ones, given in figure 1 with the notational conventions usual in denotational semantics (\uparrow and \downarrow bind less strongly than application). The denotational definition of figure 1 may be regarded as a self-interpreter for the lambda calculus since it is easily transformed into a lambda calculus program (of form $\text{fix } \lambda \mathcal{E} . \dots$).

The untyped lambda calculus program to be interpreted might contain type errors and hence the type checks such as $\mathcal{E} \llbracket \text{exp}_1 \rrbracket \rho \downarrow Funval$ are necessary so the self-interpreter can report "error" when this happens. If the subject program is known to be *well-typed* it is safe to omit the type tags [Milner 1978]. This is in general not the case for an untyped language.

2.2 Two-level Syntax

The two-level lambda calculus contains two versions of each operator in the ordinary lambda calculus: for each of the "normal" operators and base

Semantic Domains	
Val	$= Base + Funval$
$Funval$	$= Val \rightarrow Val$
Env	$= Var \rightarrow Val$
$\mathcal{E}: Expression \rightarrow Env \rightarrow Val$	
$\mathcal{E}[\text{var}]\rho$	$= \rho(\text{var})$
$\mathcal{E}[\lambda \text{var}. \text{exp}]\rho$	$= \lambda \text{value}. (\mathcal{E}[\text{exp}]\rho[\text{var} \mapsto \text{value}])\uparrow Funval$
$\mathcal{E}[\text{exp}_1 @ \text{exp}_2]\rho$	$= (\mathcal{E}[\text{exp}_1]\rho \downarrow Funval) (\mathcal{E}[\text{exp}_2]\rho)$
$\mathcal{E}[\text{fix exp}]\rho$	$= \text{fix } (\mathcal{E}[\text{exp}]\rho \downarrow Funval)$
$\mathcal{E}[\text{if exp}_1 \text{ exp}_2 \text{ exp}_3]\rho$	$= (\mathcal{E}[\text{exp}_1]\rho \downarrow Base) \rightarrow \mathcal{E}[\text{exp}_2]\rho, \mathcal{E}[\text{exp}_3]\rho$
$\mathcal{E}[\text{const } c]\rho$	$= c\uparrow Base$

Figure 1: Lambda Calculus Semantics

functions: $\lambda, @, \dots$, cons, \dots there is also a residual version: $\underline{\lambda}, \underline{@}, \dots$, cons, \dots in the two-level calculus. The abstract syntax of two-level expressions is

$$\begin{array}{l} \text{texp} ::= \text{texp } @ \text{texp} \mid \lambda \text{var}. \text{texp} \mid \text{if } \text{texp } \text{texp } \text{texp} \mid \text{fix } \text{texp} \mid \text{const base-value} \mid \\ \text{texp } \underline{@} \text{texp} \mid \underline{\lambda} \text{var}. \text{texp} \mid \underline{\text{if}} \text{texp } \text{texp } \text{texp} \mid \underline{\text{fix}} \text{texp} \mid \underline{\text{const}} \text{base-value} \mid \\ \text{var} \mid \text{lift texp} \end{array}$$

Intuitively, in the two-level semantics all operators $\lambda, @, \dots$ have the same denotations by the semantic function \mathcal{T} in figure 2 as in the one-level call-by-value semantic function of figure 1, while the residual operators: $\underline{\lambda}, \underline{@}, \dots$ are suspended yielding as result a piece of *code* (a one-level expression) for execution at run-time. The lift operator builds a residual constant expression with the same value as lift's argument. The lift operator is used when a residual expression has a constant value that must be computed at partial evaluation time. We will give an example of this in section 2.5.1.

Note that we do not distinguish syntactically between "normal" and residual variables (elsewhere called dynamic and static). The reason for this is that the universality of the value domain, which can hold both "normal" values and residual code.

2.3 Two-level Semantics

The value of a two-level expression ranges over a domain $2Val$:

$$\begin{aligned} 2Val &= Base + 2Funval + Code \\ 2Funval &= 2Val \rightarrow 2Val \\ Code &= Expression \end{aligned}$$

which is the normal expression value domain extended by an extra summand, the flat domain of one-level expressions, $Code$. The value of a two-level expression might thus be an (ordinary) expression. The valuation functions for two-level lambda calculus programs are given in figure 2. The rules contain explicit type checks; section 2.5 will discuss sufficient criteria for omitting these (and so the error element as well). The \mathcal{T} -rules for the non-residual syntactic constructs look the same as the corresponding \mathcal{E} -rules but the values range over the larger domain of two-level values $2Val$.

To obtain self-applicability the rules should be rewritten as a single expression in our lambda calculus language. This is straightforward, the result being of the form $\text{fix } \lambda \mathcal{T}. \lambda \text{exp}. \lambda \rho. \dots$.

The \mathcal{T} -rule for a residual application is

$$\begin{aligned} \mathcal{T}[\text{texp}_1 @ \text{texp}_2]\rho &= build-@(\mathcal{T}[\text{texp}_1]\rho \downarrow Code, \\ \mathcal{T}[\text{texp}_2]\rho \downarrow Code) \uparrow Code \end{aligned}$$

The recursive calls $\mathcal{T}[\text{texp}_1]\rho$ and $\mathcal{T}[\text{texp}_2]\rho$ produce reduced operator and operand and the function $build-@$ “glues” them together with an application operator $@$ to appear in the residual program (concretely, an expression of the form $\text{texp}_1\text{-code} @ \text{texp}_2\text{-code}$). All the $build$ -functions are strict. (For a concrete example of machine generated code, the reader can consult the session in appendix A.)

The projections check that both operator and operand reduce to code pieces since it does not make sense to glue, e.g., functions together to appear in the residual program. Finally the newly composed expression is tagged as being code.

The \mathcal{T} -rule for variables is

$$\mathcal{T}[\text{var}]\rho = \rho(\text{var})$$

The environment ρ is expected to hold the values of all variables regardless of whether they are predefined constants, functions, or code pieces. The environment is updated in the usual way in the rule for non-residual λ , and in the rule for $\underline{\lambda}$, the formal parameter is bound to a fresh variable name (which we assume available whenever needed):

$$\begin{aligned} \mathcal{T}[\underline{\lambda}\text{var}.\text{texp}]\rho = & \text{let } nvar = \text{newname} \\ & \text{in } \text{build-}\lambda(nvar, \mathcal{T}[\text{texp}]\rho[\text{var} \mapsto nvar] \downarrow \text{Code}) \uparrow \text{Code} \end{aligned}$$

Each occurrence of var in texp will then be looked up in ρ , causing var to be replaced by some var_{new} . Since $\underline{\lambda}\text{var}.\text{texp}$ might be duplicated, and thus become the “father” of many λ -abstractions in the residual program, this renaming is necessary to avoid name confusion in residual programs. The free dynamic variables must be bound to their new names in the initial static environment ρ_s . The generation of new variable names relies on a side effect on a global state (a name counter). In principle this could have been avoided by adding an extra parameter to the semantic function, but for the sake of notational simplicity we have used a less formal solution.

EXAMPLE 1 Suppose we are given the power program power with free variables n and x :

$$\begin{aligned} & (\text{fix } \lambda p. \lambda n'. \lambda x'. \text{if } (= n' 0) \quad \text{const } 1 \quad (* @ x') @ (p @ (- n' 1) \\ & @ x')) @ n @ x \end{aligned}$$

with $n = 2$ as static variable. We annotate the irreducible parts to yield the program power^{ann} :

$$\begin{aligned} & (\text{fix } \lambda p. \lambda n'. \lambda x'. \text{if } (= n' 0) \quad \underline{\text{const}} 1 \quad (\underline{*} @ x') @ (p @ (- n' 1) \\ & @ x')) @ n @ x \end{aligned}$$

With $\rho_s = [n \mapsto 2 \uparrow \text{Base}, x \mapsto x_{new} \uparrow \text{Code}]$ we can evaluate power^{ann} (i.e. partially evaluate power), yielding:

Semantic Domains

$$\begin{aligned}
2Val &= Base + 2Funval + Code \\
2Funval &= 2Val \rightarrow 2Val \\
Code &= Expression \text{ (= the set of one-level expressions)} \\
2Env &= Var \rightarrow 2Val
\end{aligned}$$

$$\mathcal{T}: 2Expression \rightarrow 2Env \rightarrow 2Val$$

$$\mathcal{T}[\text{var}]\rho = \rho(\text{var})$$

$$\mathcal{T}[\lambda \text{var} . \text{texp}]\rho = (\lambda \text{value} . (\mathcal{T}[\text{texp}]\rho[\text{var} \mapsto \text{value}])) \uparrow 2Funval$$

$$\mathcal{T}[\text{texp}_1 @ \text{texp}_2]\rho = (\mathcal{T}[\text{texp}_1]\rho \downarrow 2Funval) (\mathcal{T}[\text{texp}_2]\rho)$$

$$\mathcal{T}[\text{fix } \text{texp}]\rho = \text{fix } (\mathcal{T}[\text{texp}]\rho \downarrow 2Funval)$$

$$\mathcal{T}[\text{if } \text{texp}_1 \text{ texp}_2 \text{ texp}_3]\rho = (\mathcal{T}[\text{texp}_1]\rho \downarrow Base) \rightarrow \mathcal{T}[\text{texp}_2]\rho, \mathcal{T}[\text{texp}_3]\rho$$

$$\mathcal{T}[\text{const } c]\rho = c \uparrow Base$$

$$\mathcal{T}[\text{lift } \text{texp}]\rho = \text{build-const}(\mathcal{T}[\text{texp}]\rho \downarrow Base) \uparrow Code$$

$$\mathcal{T}[\lambda \text{var} . \text{texp}]\rho = \text{let } nvar = \text{newname} \text{ in } \text{build-}\lambda(nvar, \mathcal{T}[\text{texp}]\rho[\text{var} \mapsto nvar] \downarrow Code) \uparrow Code$$

$$\mathcal{T}[\text{texp}_1 @ \text{texp}_2]\rho = \text{build-@}(\mathcal{T}[\text{texp}_1]\rho \downarrow Code, \mathcal{T}[\text{texp}_2]\rho \downarrow Code) \uparrow Code$$

$$\mathcal{T}[\text{fix } \text{texp}]\rho = \text{build-fix}(\mathcal{T}[\text{texp}]\rho \downarrow Code) \uparrow Code$$

$$\mathcal{T}[\text{if } \text{texp}_1 \text{ texp}_2 \text{ texp}_3]\rho = \text{build-if}(\mathcal{T}[\text{texp}_1]\rho \downarrow Code, \mathcal{T}[\text{texp}_2]\rho \downarrow Code, \mathcal{T}[\text{texp}_3]\rho \downarrow Code) \uparrow Code$$

$$\mathcal{T}[\text{const } c]\rho = \text{build-const}(c) \uparrow Code$$

Figure 2: Two-level Lambda Calculus Semantics

$$\begin{aligned}
&\mathcal{T}[\text{power}^{ann}]\rho_s \\
&= \mathcal{T}[(\text{fix } \lambda p . \lambda n' . \lambda x' . \text{if } (= n' 0) \quad \text{const } 1 \quad (* @ x') @ (p @ (- n' 1) @ x')) @ n @ x]\rho_s \\
&= (* x_{new} (* x_{new} 1))
\end{aligned}$$

In appendix A a Scheme session shows how this residual program is generated by our partial evaluator.

In the power example it is quite clear that with $\rho = [n \mapsto 2, x \mapsto d2]$, $\rho_s = [n \mapsto 2, x \mapsto x_{new}]$, and $\rho_d = [x_{new} \mapsto d2]$ (omitting injections for brevity) it holds for all $d2$

$$\mathcal{E}[\llbracket power \rrbracket \rho] = \mathcal{E}[\llbracket \mathcal{T}[\llbracket power^{ann} \rrbracket \rho_s] \rrbracket \rho_d]$$

This is the kind of correctness property we want to hold in general. In section 2.6 we state a general correctness theorem concerning two-level evaluation.

□ Ex1

2.4 A Bird’s Eye View of How Mix Works

Mix specializes programs in two steps. If given program p and static data $d1$, its actions are:

1. Binding time analysis. This annotates p , giving p^{ann} .
2. Evaluate $\mathcal{T}[\llbracket p^{ann} \rrbracket \rho_s]$, where ρ_s binds p ’s free static variable to $d1$.

Free variables in p will only be bound to first order values, *i.e.*, values in the *Base* or *Code* summands of $2Val$.

2.5 “Well-Annotated Programs Do Not Go Wrong”

The semantic rules of figure 2 check explicitly that the values of subexpressions are in the appropriate summands of the value domain, in the same way that a type-checking interpreter for an untyped language would check types on the fly. Type-checking on the fly is clearly necessary to prevent mix from committing type errors itself on a poorly annotated program.

Doing type checks on the fly in mix is not very satisfactory for practical reasons. Mix is supposed to be a general and automatic program generation tool, and it should for obvious reasons be impossible for a mix generated compiler to go down with an error message.

Note that it is in principle possible—but unacceptably inefficient in practice—to avoid mix-time errors by annotating as residual all operators in the input program to mix. This would place all values in the code summand so all type checks would succeed; but the residual program would always be isomorphic to the source program, so it would not be optimized at all.

The aim of this section is to develop a more efficient strategy, ensuring two things prior to partial evaluation: that the partial evaluator will not

fail a type check, thus rendering the type checks superfluous; and ensuring that as many operations as possible are performed at mix time. This section title “Well-Annotated Programs Do Not Go Wrong” is thus a paraphrase of Milner’s slogan [Milner 1978].

2.5.1 Well-Annotated Expressions

Given a one-level expression exp , an *annotated* lambda expression exp^{ann} is a two-level expression obtained by replacing some occurrences of \textcircled{e} , λ , \dots in exp by the corresponding marked operator: \textcircled{e} , $\textcircled{\lambda}$, \dots and inserting some lift-operators. Clearly the annotations have to be placed consistently not to produce a projection error according to the rules in figure 2.

A simple and traditional solution to our problem is to devise a type system. In typed functional languages, a type inference algorithm such as algorithm W of [Milner 1978, Damas and Milner 1982], checks that a program is well-typed prior to program execution. If it is, no run-time summand tags or checks are needed. Type correctness is quite well understood and can be used to get a nice formulation of the problem to be solved by binding time analysis. It also turns out that we can adapt some of the type inference ideas of [Damas and Milner 1982] (and many other papers) to get a nice algorithm for doing binding time analysis. The type system used here is very simple, but it should not be too difficult to adapt the ideas to a more powerful system including more base types, constructors, and polymorphism.

DEFINITION 2 The abstract syntax of a two-level type t is given by

$$\text{type} ::= \text{base} \mid \text{type} \rightarrow \text{type} \mid \text{code}$$

A *type environment* is a mapping from variables to types. \square DEF2

DEFINITION 3 Let τ be a type environment mapping the free variables of a two-level expression texp to their types. Then texp is *well-annotated* if $\tau \vdash \text{texp} : t$ can be deduced from the inference rules in figure 3 for some type

t . \square DEF3

Note that type *unicity* does not hold: t is not uniquely determined by τ and texp . Given any type environment τ and the expression $\lambda x. x$ it holds, e.g., that $\tau \vdash \lambda x. x: \text{base} \rightarrow \text{base}$ and $\tau \vdash \lambda x. x: (\text{base} \rightarrow \text{base}) \rightarrow (\text{base} \rightarrow \text{base})$.

Our lambda calculus is basically untyped, but the well-annotatedness ensures that the program parts evaluated at partial evaluation time are well-typed, thus ensuring mix against type errors. The well-annotatedness criterion is completely permissive concerning the run-time part of a two-level expression. An extreme case: every lambda expression with only residual operators is well-typed—at *partial evaluation time*.

Two-level expressions of type *base* evaluate (completely) to constants, and expressions of type $t_1 \rightarrow t_2$ evaluate to some function “living” only at partial evaluation time. The mix-value of a two-level expression texp of type *code* is a one-level expression exp . For partial evaluation we are only interested in fully annotated programs p^{ann} that have type *code*. If so, $\mathcal{T}[p^{ann}]_{\rho_s}$ (if defined) will be the residual program.

Suppose an expression of type *base* is evaluated at mix-time yielding *value* as result, and suppose *value* is needed at runtime. The lift annotation is then used to indicate that the computed *value* must be turned into a constant *expression* to appear in the residual program. The type inference rules accordingly state that if texp has type *base* then $\text{lift } \text{texp}$ has type *code*.

EXAMPLE 4 Consider the following program that computes n times x to the n th where x and n are free (input) variables.

$$\begin{aligned} & (\text{fix } \lambda p. \lambda n'. \lambda x'. \text{if } (= n' 0) \quad n \quad (* x' (p @ (- n' 1) @ x')) @ \\ & n @ x \end{aligned}$$

If we take x to be of type *code* (dynamic) and n to be of type *base* (static), we observe that this program cannot be well-annotated without using lift.

The reason is that the multiplication branch must have type *code* since the multiplication cannot be performed at partial evaluation time, but the *n* branch of the conditional has type *base*. This incompatibility cannot be resolved using the type deduction rules without the rule for lift.

If, however, we “lift” the *n* that provides the base case value, we obtain a well-annotated program:

$$(\text{fix } \lambda p. \lambda n'. \lambda x'. \text{if } (= n' 0) \quad \text{lift } n \quad \underline{*} \underline{@} x' \underline{@} (p \underline{@} (- n' 1) \underline{@} x')) \underline{@} n \underline{@} x$$

□ Ex4

The result on error freedom of well-typed programs can be formulated as follows. Proof is omitted since the result is well-known.

DEFINITION 5 Let *t* be a two-level type and *v* be a two-level value. We say that *t* *suits* *v* iff one of the following holds

1. *t* = *base* and *v* = *ct*↑*Base* for some *ct*.
2. *t* = *code* and *v* = *cd*↑*Code* for some *cd*.
3. (a) *t* = *t*₁ → *t*₂, *v* = *f*↑*2Funval* for some *f*, and
(b) $\forall v \in 2Val: t_1 \text{ suits } v \text{ implies } t_2 \text{ suits } f(v).$

A type environment τ suits an environment ρ if for all variables *x* bound by ρ , $\tau(x)$ suits $\rho(x)$. □ DEF5

Recall that the initial static environment ρ_s maps static variables to their (first order) values and dynamic variables to their new name. A type environment τ that suits ρ_s thus maps static variables to *base*, and dynamic variables to *code*. The following is a standard result [Milner 1978].

PROPOSITION 6 If $\tau \vdash \text{texpr}: t$ and τ suits ρ_s then $\mathcal{T}[\![\text{texpr}]\!]\rho_s$ does not yield a projection error. □ PROP6

2.5.2 The Existence of Best Completions

In general we want to perform as much computation as possible at partial evaluation time. This means that as few annotations as possible should be added to make the expression well-annotated.

DEFINITION 7 The *annotation forgetting* function $\phi: 2Exp \rightarrow Exp$, when applied to a two level expression texp returns an expression exp which differs from texp only in that all annotations and lift operators are removed.

□ DEF7

DEFINITION 8 Given two-level expressions, texp and texp' , define $\text{texp} \sqsubseteq \text{texp}'$ by

1. $\phi(\text{texp}) = \phi(\text{texp}')$
2. All operators marked as residual in texp are also marked as residual in texp'

□ DEF8

\sqsubseteq defines a preorder on the set of two-level expressions. If we restrict \sqsubseteq to the set of two-level expressions without lift-operators, \sqsubseteq is a partial order possessing greatest lower bounds (written \sqcap).

DEFINITION 9 Given a two-level expression texp and a type environment τ , a *completion* of texp for τ is a two-level expression texp' with $\text{texp} \sqsubseteq \text{texp}'$ and $\tau \vdash \text{texp}': t$ for some type t . A *best completion* (if it exists) is an expression texp'' which is a completion of texp fulfilling $\text{texp}'' \sqsubseteq \text{texp}'$ for all completions texp' of texp . A *liftless completion* texp' of texp is a completion of texp in which lift does not occur. □ DEF9

As we saw in example 4 not all two-level expressions have liftless completions. If a two-level expression has liftless completions, it has a unique best liftless completion.

THEOREM 10 Given a two-level expression texp and a type environment τ mapping the free variables of texp to either *base* or *code*. Assume texp has at least one liftless completion. Then it also has a best liftless completion.

PROOF Found in [Gomard 1989] and in part II of this article.

□ **THEOREM10**

There does not in general exist a unique best completion of two-level expressions. Suppose n has type *base* and that we want a completion of $\lambda x. x \text{ @ } n$ of type *code*. There are two candidates: $\text{lift } ((\lambda x. x) \text{ @ } n)$ and $(\lambda x. x) \text{ @ } \text{lift } n$ which, though different, yield equal amounts of static computation. In [Gomard 1989] and in part II there is a discussion of the general existence of best completions.

2.6 The Mix Equation Revisited

With $m = n = 1$ the mix equation is

$$\mathbf{L} \ p \ [d1, d2] = \mathbf{L} \ (\mathbf{L} \ \text{mix} \ [p, d1]) \ d2$$

In terms of \mathcal{E} and \mathcal{T} the equation is

$$\mathcal{E}[[p]]\rho = \mathcal{E}[[\mathcal{T}[[p^{ann}]]\rho_s]]\rho_d$$

where p^{ann} is a completion of p , $\rho = [x_1 \mapsto d1, x_2 \mapsto d2]$, $\rho_s = [x_1 \mapsto d1, x_2 \mapsto x_{new}]$, and $\rho_d = [x_{new} \mapsto d2]$.

We now state the general correctness theorem for two-level evaluation of untyped lambda expressions.

THEOREM 11 (Main Correctness Theorem.) Suppose we are given

1. A two-level expression texp with free static variables $x_1 \dots x_m$ and free dynamic variables $x_{m+1} \dots x_{m+n}$.
2. Environments $\rho, \rho_d \in Env$ and $\rho_s \in 2Env$ such that for $i \in 0..m$: $\rho(x_i) = \rho_s(x_i)$ and for $i \in m+1..m+n$: $\rho(x_i) = \rho_d(\rho_s(x_i) \downarrow Code)$.

3. $\tau \vdash \text{texp}$: *code* for some τ that suits ρ_s .

If both $\mathcal{E}[\mathcal{T}[\text{texp}]\rho_s]\rho_d$ and $\mathcal{E}[\phi(\text{texp})]\rho$ are not \perp then

$$\mathcal{E}[\mathcal{T}[\text{texp}]\rho_s]\rho_d = \mathcal{E}[\phi(\text{texp})]\rho$$

PROOF Found in [Gomard 1989] and in part II of this article.

□ THEOREM11

3 Aspects of Binding Time Analysis by Type Inference

3.1 BTA: Type Analysis of Untyped Programs?!

Given an expression exp in the untyped, one-level lambda calculus and some assumptions on its free variables, it is always possible to add enough annotations to obtain a well-annotated two-level expression texp , in the worst case by making all operators residual and adding lift operators where needed. Operators in texp can be forced to be residual for two somewhat different reasons:

1. The computation cannot be performed when the input data is incomplete, or
2. The non-residual part of the type system could not assign a proper type to the subexpression.

The first reason forces one, for example, to make the sum of a static and a dynamic variable residual. The second reason would force $\lambda x. x @ x$ to be annotated $\underline{\lambda}x. x @ x$ no matter what the context, since $\lambda x. x @ x$ cannot be assigned a type in our system. Similarly any occurrence of the lambda notation equivalent of the Y-combinator in a subject program will be made residual, so to define mix-time recursive functions we have to use the explicit fixed point operator.

Our favorite subject programs are denotational language definitions. In these the “syntactic dispatch” and the environment lookup operations are

usually “well-behaved” and do not need to be made residual. Consider for example the following (in which we have again omitted some application operators). It is a syntactically sugared fragment of a lambda calculus self-interpreter (and thus also a fragment of mix):

$$\begin{aligned}\mathcal{E} = \lambda \text{exp}.\lambda \rho. \text{ case exp of} \\ \text{var(id)} & : \rho(\text{id}) \\ \text{app}(\text{exp}_1, \text{exp}_2) & : (\mathcal{E} \text{ exp}_1 \rho) @ (\mathcal{E} \text{ exp}_2 \rho) \\ \text{abs}(x, \text{exp}_1) & : \lambda \text{val}.(\mathcal{E} \text{ exp}_1 (\lambda \text{id}. \text{ if id}=x \text{ then val else } \rho(\text{id})))\end{aligned}$$

We assume that the expression exp fed to \mathcal{E} is static. Clearly $(\mathcal{E} \text{ exp } \rho)$ must have the same type as $(\mathcal{E} \text{ exp}_1 \rho)$ (call it t). On the other hand, the application branch of the case expression demands $(\mathcal{E} \text{ exp } \rho)$ to have the same type as $(\mathcal{E} \text{ exp}_1 \rho) @ (\mathcal{E} \text{ exp}_2 \rho)$, so the same expression must have type $t \rightarrow t$. These demands are incompatible; consequently $(\mathcal{E} \text{ exp } \rho)$, the results of the case branches, and the application operator must be retyped: annotated as residual and so of type *code*.

The environment ρ is well-behaved, since the application of ρ and its updating (in the abstraction branch) are type consistent. The minimally annotated self-interpreter thus becomes:

$$\begin{aligned}\mathcal{E} = \lambda \text{exp}.\lambda \rho. \text{ case exp of} \\ \text{var(id)} & : \rho(\text{id}) \\ \text{app}(\text{exp}_1, \text{exp}_2) & : (\mathcal{E} \text{ exp}_1 \rho) @ (\mathcal{E} \text{ exp}_2 \rho) \\ \text{abs}(x, \text{exp}_1) & : \underline{\lambda} \text{val}.(\mathcal{E} \text{ exp}_1 (\lambda \text{id}. \text{ if id}=x \text{ then val else } \rho(\text{id})))\end{aligned}$$

where exp has type *base*, ρ has type *base* \rightarrow *code*, and \mathcal{E} has type *base* \rightarrow (*base* \rightarrow *code*) \rightarrow *code*.

By now the reader may be confused: this type seems quite different from the type $\mathcal{E}: \text{Expression} \rightarrow \text{Env} \rightarrow \text{Val}$ with $\text{Env} = \text{Var} \rightarrow \text{Val}$ seen in the denotational semantics of the lambda calculus of figure 1. But there is no conflict, just a different viewpoint: the types used here are for a different purpose than those of the semantics, namely to identify what can be performed at partial evaluation time.

To explain the type of \mathcal{E} in the well-annotated two-level self-interpreter, first recall that expressions are base values. Second, *Val* is a sum domain,

and the result of applying \mathcal{E} to an expression can be in any summand (or even be the error element), depending on dynamic program input, which is unavailable at partial evaluation time. The result of evaluation must thus have type *code*.

3.2 Viewing Binding Time Analysis as Type Inference

Given an expression exp , a naive exponential time algorithm could generate all well-annotated completions and pick the best. We would like to do better than that, and this section sketches ideas for a more efficient BTA. Detailed algorithms will appear in part II.

Given a well-annotated expression texp , algorithm W of [Damas and Milner 1982] is able to assign types to texp and its subexpressions. Given an expression that is not well-annotated, the algorithm would at some stage fail to unify two type terms and report an error. Since all such errors can in principle be fixed by annotating some operators as residual, a good question is *which* operators it should change.

Algorithm W manipulates type terms that denote types of the subexpressions of texp . It appears possible to associate information with each type term about which of the program's subexpressions *must* have that type.³ When a unification fails the relevant list of subexpressions is returned. This points out which parts of texp should be given type *code* for texp to have a chance of being well-annotated.

In our framework we have three type constructors: \rightarrow , *base*, and *code*. During type inference we will mark each such constructor with a subscript: a list of occurrences of subexpressions of the expression whose type is being inferred. This means that if the constructor causes a unification to fail, then the subexpression(s) pointed out by the occurrence(s) should be made residual.

EXAMPLE 12 Suppose $\text{texp} =$

$$\frac{\text{if } x \quad x \quad \lambda y. y}{\quad}$$

³In [Wand 1986] a similar idea is used to make type inference systems give better error messages.

with initial type assumption $\tau = [x \mapsto \text{code}]$. The condition of any non-residual if-expression must have type *base* (i.e. boolean), so x must also be a base value. The unification of *code* with *base* fails, and accordingly the conditional must be residual, so texp is transformed into

$$\underline{\text{if}} \ x \ \times \ \lambda y. y$$

and W tries to check the types of this new candidate. This time the unification of the types of the two branches again fails, since $\lambda y. y$ has type $t \rightarrow t$, so accordingly λy is made residual. Now $\text{texp} =$

$$\underline{\text{if}} \ x \ \times \ \underline{\lambda} y. y$$

which is well-annotated. \square Ex12

EXAMPLE 13 In the power example the unannotated program is

$$(\text{fix } \lambda p. \lambda n'. \lambda x'. \text{if } (= \ n' \ 0) \ \text{const } 1 \ \ (* \ @ \ x') \ @ \ (p \ @ \ (- \ n' \ 1) \ @ \ x')) \ @ \ n \ @ \ x$$

with initial type assumption $\tau = [n \mapsto \text{base}, x \mapsto \text{code}]$. This implies expression $(\text{fix } \dots)$ has type $\text{base} \rightarrow \text{code} \rightarrow \alpha_2$ where α_2 is a type variable. In turn it is found that p must have type $\text{base} \rightarrow \text{code} \rightarrow \alpha_2$, n' has type *base*, x' has type *code*, and that both branches in the conditional have type α_2 .

The type of the non-residual (binary) multiplication operator $*$ is $\text{base} \rightarrow \text{base} \rightarrow \text{base}$. Since x has type *code* the type assignment algorithm will fail. This forces $*$ and the corresponding applications to be residual. In turn $\text{const } 1$ must also be made residual yielding the well-annotated (and best) completion:

$$(\text{fix } \lambda p. \lambda n'. \lambda x'. \text{if } (= \ n' \ 0) \ \underline{\text{const}} \ 1 \ \ (\underline{*} \ @ \ x') \ @ \ (p \ @ \ (- \ n' \ 1) \ @ \ x')) \ @ \ n \ @ \ x$$

\square Ex13

It is possible to augment the type representations in the almost linear unification algorithm of [Huet 1976] with type origin information without significant increase in its running time. Algorithm W scans the program once and makes one unification for each application and each conditional. In our algorithm, each time algorithm W fails, at least one operator will be made residual, and algorithm W will be re-applied to the result. This does not seem prohibitively expensive, especially in light of the fact that binding time analysis is only performed once: before the static data are available for specialization. Part II will contain the details of the algorithm.

What this binding time analysis does is different from the lambda calculus binding time analyses described in [Nielson and Nielson 1988b, Schmidt 1988] in that the base language is not required to be well-typed.

Earlier work in binding time analysis (e.g. [Jones *et al.* 1989]) has been based on abstract interpretation, but it seems to us that the framework of type inference provides a simple and efficient alternative. This idea of using type inference on “typical” abstract interpretation tasks has also been used in very recent work on strictness analysis [Kuo and Mishra 1989] where the result was a much more efficient but sometimes less precise strictness analysis than that done by evaluation over a higher order abstract strictness domain. In [Wadler 1989] a type system is also used to determine which variables are referenced exactly once; again a typical application area where abstract interpretation has been used.

3.3 Finiteness of Partial Evaluation

Mix can “go wrong” in other ways than by committing type errors. Reduction might proceed infinitely if \mathcal{T} reduces too often. To avoid this some redexes should be left in the residual program, and since mix obeys the annotations blindly it is the responsibility of the BTA to decide which. Some attention has been paid to this problem in the literature and it is generally recognized as being difficult to ensure termination and yet to do nontrivial computation at partial evaluation time.

A variety of BTA algorithms for first order functional languages have been published, e.g., [Jones *et al.* 1989, Mogensen 1988], but they do not

ensure termination. [Jones 1988] outlines a BTA algorithm with strong termination properties for a flow chart language, but even though the language is simple the algorithm is not. A further problem with the first order languages is that it is a too conservative restriction to demand *compositionality*: static parameters that become strictly smaller in every recursive function call. To ensure safe BTA without this restriction some abstract interpretations are needed.

The higher order lambda calculus on the other hand still has significant expressive power when compositionality is imposed. Let us first note that partial evaluation of a well-annotated expression *without* non-residual fix-operators is guaranteed to terminate. Second, a non-residual fix-expression

$$\text{fix } \lambda f. \lambda x_1. \dots \lambda x_n. \text{body}$$

defining a function f is safe if for some i , the i 'th argument has type *base* and in all recursive calls to f , that argument is always strictly smaller than x_i (according to some well-founded ordering on the domain of possible argument values). Compositionality of recursive function definitions is easy to check, and it is always possible to transform a well-annotated expression not satisfying the criterion into one that does, by simply making any offending fixed point operators residual. As long as compositional definitions are used the criterion is strong enough to ensure termination of partial evaluation without forcing to be residual any fix-operators that safely could have been annotated as non-residual.

3.4 Code Duplication

The following two-level expression `texp` is well-annotated with $\tau = [y \mapsto \text{code}]$

$$(\lambda x. x \underline{+} x) @ (y * y)$$

In the proof of $\tau \vdash \text{texp} : \text{code}$, subexpression $\lambda x. x \underline{+} x$ has type $\text{code} \rightarrow \text{code}$ and $(y * y)$ has type code . Partial evaluation yields (we do not rename y) the residual program

$$(y * y) + (y * y)$$

which has the unfortunate feature that the computation $(y*y)$ has been duplicated. If $(y*y)$ had been computationally heavier or had been contained in a recursive call, this could have serious impacts on the efficiency of the residual program. To avoid the code duplication a more conservative annotation of the subject program would suffice:

$$(\underline{\lambda}x. x \underline{+} x) @ (y*y)$$

This program is also well-annotated and partial evaluation yields

$$(\lambda x. x + x) @ (y*y)$$

It is always possible to solve the problem by making enough operators residual but it is clearly desirable not to do this when not strictly necessary.

In the first annotated program $\lambda x. x \underline{+} x$ had type $code \rightarrow code$, and in the second $\underline{\lambda}x. x \underline{+} x$ had type $code$. To avoid code duplication we would have to restrict the number of references to the formal parameter in functions of type $code \rightarrow t$ to at most one. To be sure to preserve termination properties under call-by-value we would have to insist on exactly one reference.

Quite similar problems are well known from other partial evaluators. In [Sestoft 1988] a method to detect which parts of a program should be made residual to avoid duplication of function calls is presented. In the Similix project [Bondorf and Danvy 1989, Bondorf 1990] a let-expression is inserted whenever there is a risk of duplication.

4 Experiments with Mix

The partial evaluator mix that we have implemented in and for the lambda calculus realizes in practice the three Futamura projections mentioned in section 1.3.2. In this chapter we demonstrate this by doing compilation and compiler generation from mix and from two different language definitions written in the lambda calculus. The first example, a self-interpreter for the lambda calculus, may seem to be only of academic interest. It serves, however, to demonstrate rigorously that mix can reduce away *all* of the computational overhead traditionally associated with interpretation. Further, it

shows that *mix* can generate a “self-compiler” and a compiler generator with natural and understandable structure. For a second and less introspective example, we present a denotational semantics for a small imperative language, *Tiny*, and study the structure and performance of the *Tiny*-to-lambda calculus compiler generated from it by *mix*.

4.1 A Self-Interpreter

From the Futamura projections it is not at all clear how good residual programs we can expect *mix* to produce, and it not even clear how to measure the quality of residual programs. We now examine the structure of some *mix* generated programs, and measure the actual run time on the computer of some standard examples.

To get an idea of how much speed-up we can expect partial evaluation to yield, consider the partial evaluation of a self-interpreter with respect to a known program p .

$$\mathbf{L} \text{ mix } [sint, p] = r$$

By the *mix* equation $\mathbf{L} p d = \mathbf{L} sint [p, d] = \mathbf{L} (\mathbf{L} \text{ mix } [sint, p]) d = \mathbf{L} r d$, so the programs p and r should have the same meaning. But what about efficiency?

If we required r to be more efficient than p this would mean that *mix* was able to optimize *any* program written in the lambda calculus, needing only a self-interpreter to help. Considering the simplicity of *mix*, this is asking too much. Introducing a self-interpreter gives a roughly linear slow-down of the execution speed of p due to the interpretation overhead. It thus seems unreasonable to expect *mix* in general to give more than a linear speed-up, *i.e.* to remove the interpretation factor. If we can achieve $\mathbf{L} \text{ mix } [sint, p] = p$, then *all* interpretation overhead has been removed.

The self-interpreter we used for the one-level lambda calculus was a direct implementation of the semantic rules in figure 1 without the injections and projections. The main part of the annotated self-interpreter is in figure 4; note that we use the annotation $\text{-}\mathbf{r}$ instead of underlining. We have

not included the lengthy definition of the initial environment holding all predefined functions.

4.1.1 “ $\mathbf{L\ mix\ [sint, p] = p}$ ” is important!

It turns out that our partial evaluator yields a residual program r structurally equal to p , the only difference being the variable names. This structural equality happens only because the self-interpreter does not use indirect representation of values. If the interpreter had, for instance, used closures to represent functional values, then the residual program r would have done so too, since r is derived from the interpreter in a very straightforward way.

For a concrete example, figure 5 contains two programs, one a hand-written fibonacci program, the other result of running `mix` to specialize the self-interpreter with respect to the first program.

The absence of the type checks is also necessary to obtain $\mathbf{L\ mix\ [sint, p] = p}$ since hardly any of the type checks would be performable at partial evaluation time. Since `mix` only operates on well-annotated programs we have complete assurance against type errors at mix time, but a type check of the residual program would be needed to ensure absence of run-time type errors. Further, if the interpreter were required to be strongly typed, then some indirect value representation would be needed, meaning that `mix` would not be able to remove a complete layer of interpretation.

In the development phase of a self-applicable partial evaluator, the cited condition on the result of specializing a self-interpreter often provides a natural and useful stepping stone in the process of getting self-application to work. To see why, compare the equations

$$\begin{aligned}\mathbf{L\ mix\ [sint, p]} &= p \\ \mathbf{L\ mix\ [mix, int]} &= \textit{compiler}\end{aligned}$$

and recall that `mix` roughly consists of a self-interpreter augmented with semantic rules for the residual operators. It has been the experience of several partial evaluation projects, e.g., [Jones *et al.* 1989] and now this project too, that once a self-interpreter was properly handled by `mix`, the first successful compiler generation was not too far away! (We assume tacitly

that binding time analysis is performed. Without it, there is a long way from the self-interpreter to mix.)

4.1.2 A Self-Compiler

The result of running $\mathbf{L} \text{ mix } [mix, sint]$ is a “self-compiler”, a program that transforms one lambda expression into another semantically equivalent expression. The generated self-compiler is structure preserving as well: the target expressions generated by the compiler have machine generated variable names but are otherwise identical to the source expressions.

Figure 6 contains the part of the generated compiler that compiles abstractions and applications. The program is presented as generated by machine, except that the variable names been changed into some more natural ones. The compiler has a “100% natural” recursive descent structure which is a little surprising considering that it is generated by self-application of a partial evaluator. The compilers generated in [Jones *et al.* 1989, Gomard and Jones 1989] can also be read by humans but their structures are not more than “75% natural”.

It is interesting to compare the compiler’s clause for application with the clause in the annotated self-interpreter. Where the interpreter has a non-residual operation so has the compiler, and where the interpreter has a residual operator, $\underline{\textcircled{Q}}$, the compiler has a call to the corresponding expression building function, *build-@*. We now show the compiler fragment from figure 6 in a syntactically sugared notation, where we use the “syntactic” operators “@”, “ λ ”, ... in the meta language to denote construction of those operators. Thus *build-@*($\mathcal{C}[\text{exp}_1]\rho, \mathcal{C}[\text{exp}_2]\rho$) may be written $\mathcal{C}[\text{exp}_1]\rho \text{ “@” } \mathcal{C}[\text{exp}_2]\rho$.

$$\begin{aligned} \mathcal{C}[\lambda \text{var. exp}]\rho &= \text{let } nvar = \text{newname in} \\ &\quad \text{“}\lambda\text{” } nvar. \mathcal{C}[\text{exp}]\rho[\text{var} \mapsto nvar] \\ \mathcal{C}[\text{exp}_1 \text{ @ exp}_2]\rho &= \mathcal{C}[\text{exp}_1]\rho \text{ “@” } \mathcal{C}[\text{exp}_2]\rho \end{aligned}$$

4.1.3 A Compiler Generator

The third Futamura projection states that $\mathbf{L} \text{ mix } [mix, mix]$ yields a compiler generator *cogen*. When the compiler generator *cogen* is applied to an

interpreter a compiler is generated. It holds that

$$\mathbf{L} \text{ cogen } int = \mathbf{L} \text{ mix } [mix, int]$$

In section 4.1.2 we looked at the structure of a little piece of the self-compiler, so we have an idea of what *cogen* does. The question is now whether it does it in a natural way, and the answer turns out to be yes.

Figure 7 shows the part of the machine generated *cogen* that treats application nodes in the subject program which is typically an annotated interpreter. A non-residual application node is treated in a very straight forward manner: an application node to appear in the compiler is built exactly as in the self-compiler above. In the residual case the processed branches are not glued together by an application node but by a call to the function *build-@* to appear in the compiler. With the same notational conventions as in section 4.1.2:

$$\begin{aligned} C[\exp_1 @ \exp_2]\rho &= C[\exp_1]\rho \text{ “@” } C[\exp_2]\rho \\ C[\exp_1 @ \exp_2]\rho &= \text{“build-@”}(C[\exp_1]\rho, C[\exp_2]\rho) \end{aligned}$$

4.1.4 Performance

The table below shows the run-times of our example programs. All timings are measured in Sun 3/50 cpu milliseconds using Chez Scheme. The run-times for *mix* are just for two-level evaluation; they do not include binding time analysis.

run		run-time	ratio
$\mathbf{L} \text{ sint } [fib, 15]$	=	30400	
$\mathbf{L} \text{ target } 15$	= 987	830	36.6
$\mathbf{L} \text{ mix } [sint, fib]$	=	1980	
$\mathbf{L} \text{ scomp } fib$	= <i>target</i>	60	33.0
$\mathbf{L} \text{ mix } [mix, sint]$	=	55400	
$\mathbf{L} \text{ cogen } sint$	= <i>scomp</i>	1280	43.0
$\mathbf{L} \text{ mix } [mix, mix]$	=	64600	
$\mathbf{L} \text{ cogen } mix$	= <i>cogen</i>	1330	48.6

The interpretational overhead in the self-interpreter and the mix program is rather large since all free variables (input variables and predefined function names) are looked up in the initial environment, which is a quite large case expression. Mix is able to remove this interpretational overhead, so the speed-ups gained when mix and the self-interpreter are partially evaluated are accordingly large (thus perhaps artificially so).

The table below shows the sizes of our example programs. The sizes are measured as the number of cons cells + the number of atoms in the S-expressions representing the programs in Chez Scheme.

program	size	ratio
<i>fib</i>	101	1.0
<i>target</i>	101	
<i>sint</i>	2826	1.2
<i>scomp</i>	3375	
<i>mix</i>	3206	1.2
<i>cogen</i>	3811	

The programs *fib* and *target* are virtually identical, and the size ratio between *scomp* and *sint* is quite small since the two programs are very similar in structure. Some operations in *sint* have been replaced by the corresponding code generating operations. *Cogen* is similarly a transformed version of *mix*.

4.2 An Interpreter for a Tiny Imperative Language

In this section we give an interpreter for an imperative language, *Tiny*, with while-loops and assignments. We give a denotational semantics in lambda calculus form and discuss how annotation should be done. The syntax of *Tiny*-programs is

```

program      ::=  var-declaration command
var-declaration ::=  variables variable*
command      ::=  while expression do command |
                  command; command |
                  variable := expression

```

The semantic functions are given in figure 8. The semantic functions may easily be written in lambda calculus form to be partially evaluated (seen in an appendix).

The resulting lambda calculus program, a *Tiny*-interpreter, has two free variables: the initial store `istore` and the `program` to be interpreted. Suppose that a *Tiny*-program is given as static data, but `istore` is unknown. In other words, suppose that we have the type assumptions

$$\begin{aligned}\tau \vdash \text{istore} &: \text{code} \\ \tau \vdash \text{program} &: \text{base}\end{aligned}$$

We will now informally discuss how to add annotations such that the interpreter becomes well-annotated according to the rules of figure 3. The commands and expressions are all subparts of the static program, and so have type *base*. The environment, ρ , can be computed completely since it does not depend on the store. It is applied to subparts of the program (variable names) and it returns static locations. Hence $\tau \vdash \rho: \text{base} \rightarrow \text{base}$. Since the initial store is of type *code* all subsequent stores also have type *code*. When the semantic function \mathcal{C} is applied to a command of type *base*, an environment of type $\text{base} \rightarrow \text{base}$ and a store of type *code* the result is an updated store, also of type *code*.

The mix-time application of $\lambda \text{store} . \dots$ can result in duplication of store updating code. To avoid this the $\lambda \text{store} . \dots$ has been annotated $\underline{\lambda} \text{store} . \dots$, and the corresponding applications have accordingly been annotated as residual. BTA could have classified $\lambda \text{store} . \dots$ as non-residual (thus getting type $\text{code} \rightarrow \text{code}$) and still meet the type checking rules, but to avoid code duplication the classification $\underline{\lambda} \text{store} . \dots$ (of type *code*) is a better choice. The type of \mathcal{C} (as annotated for partial evaluation) is thus

$$\tau \vdash \mathcal{C}: \text{base} \rightarrow (\text{base} \rightarrow \text{base}) \rightarrow \text{code}$$

Figure 9 shows the function \mathcal{C} in its annotated lambda calculus form. When we apply the annotated *Tiny*-interpreter to the following program

variables result x;

```

result := 1;
x      := 6;
while x do
  result := result * x;
  x      := x - 1

```

which computes the factorial of 6, the residual program in figure 10 is produced. We have changed some names and omitted (const ..) around the integers 0 and 1 for readability. Furthermore we have postreduced a few trivial redexes. A redex $(\lambda x. \text{body}) @ \text{arg}$ is trivial if x appears at most once in body .

The store is explicitly passed around as a parameter, but since the program is *single-threaded* in the store variables [Schmidt 1985] these could all be replaced by one global variable. Such a transformation is called *globalization* [Sestoft 1989]. With the store arguments removed the residual program would almost look like (nested) assembly code.

4.3 An Example of Compiler Generation

When *mix* is applied to itself and the *Tiny*-interpreter, a compiler from *Tiny* to lambda calculus is generated. When we examine the structure of the generated compiling function \mathcal{C}_c we notice a strong resemblance with that of the semantic function \mathcal{C} . Those operators annotated as residual in figure 9 have been replaced by the corresponding code generating actions.

To emphasize the structural similarities we have changed the machine generated names into names close to those of \mathcal{C} . Figure 12 contains the part that compiles commands as generated by machine. Figure 11 contains a part of the generated compiling function \mathcal{C}_c syntactically sugared. As in section 4.1.2, we use for brevity the syntax-font in citation marks: “@”, “ λ ”, ... instead of writing *build-@*, *build- λ* To reduce the number of quotes we write “if = @ 0” instead of quoting all constructors in the term. A comparison of figures 8 and 11 shows that in the generated compiler the run-time (residual) actions of the interpreter have been replaced by code building operations.

4.3.1 Performance

The table below shows the run-times of our example programs. In the following, *fac* denotes the factorial program written in *Tiny*, *target* denotes the factorial residual program (figure 10), *tiny* denotes the *Tiny*-interpreter, *comp* denotes the generated *Tiny*-compiler. All the timings are measured in Sun 3/50 cpu milliseconds using Chez Scheme.

run		run-time	ratio
L <i>tiny</i> [<i>fac</i> ,6]	=	70	
L <i>target</i> 6	= 720	10	7.0
L <i>mix</i> [<i>tiny</i> , <i>fac</i>]	=	700	
L <i>comp</i> <i>fac</i>	= target	20	35.0
L <i>mix</i> [<i>mix</i> , <i>tiny</i>]	=	17600	
L <i>cogen</i> <i>tiny</i>	= comp	380	46.3
L <i>mix</i> [<i>mix</i> , <i>mix</i>]	=	64600	
L <i>cogen</i> <i>mix</i>	= cogen	1330	48.6

The table below shows the sizes of our example programs. The sizes are measured as the number of cons cells + the number of atoms in the S-expressions representing the programs in Chez Scheme.

program	size	ratio
fac	71	
target	221	3.1
tiny	743	
comp	927	1.3
mix	3206	
cogen	3811	1.2

From the two tables above (and the tables in section 4.1.4) we see that in the case where *mix* performs a compilation from *Tiny* to lambda calculus the size ratio is significantly larger and the speed-up smaller than in all other cases. In the case of self-compilation there is a canonical target program—which our methods produce—but this is generally not the case for other compilation tasks. When programs written in arbitrary programming languages are translated into lambda calculus it is not clear (nor always true)

that the target program derived from the source and an interpreter in the straightforward way is the best. Therefore to get really good results some post processing, like globalization of parameters, is likely to be needed. Alternatively, the interpreters can be written with explicit attention to the fact that they will be partially evaluated. This, however, requires good insight into the partial evaluation algorithm.

5 Perspectives and Conclusions

5.1 Related Work

The present work overlaps with two areas: partial evaluation (and its recent offspring: BTA) that has emphasized automatic program optimization and transformation; and semantics-directed compiler generation, whose main goal has been to take as input a denotational semantics definition of a programming language, and to obtain automatically a compiler that efficiently implements the defined language.

5.1.1 Partial Evaluation

Early work in partial evaluation viewed partial evaluation as an optimizing phase in a compiler (constant folding), as a device for incremental computations [Lombardi 1967], or as a method to transform imperative Lisp programs [Beckman *et al.* 1976]. The latter system was able to handle FUNARGs, but it was not self-applicable (although the Redcompile program amounts to a hand-written version of *cogen*). Later work aimed to partially evaluate higher-order and imperative Scheme programs [Schooler 1984, Guzowski 1988], but did not achieve self-application. Ershov and Itkin have proved correctness of a partial evaluation schemes for a flow chart language [Ershov and Itkin 1977], but the schemes did not allow transfer of static information across dynamic conditionals. In [Hansen and Träff 1989] an evaluation strategy that involves specialization of first order functions is proved correct.

The potential of self-application was realized independently in Japan and the Soviet Union [Futamura 1971, Turchin 1980, Ershov 1978] in the

early 1970's and experiments were made without conclusive results. The first non-trivial self-application was realized in 1984 [Jones *et al.* 1985, Jones *et al.* 1989] for first order recursive equations. Since then several other self-applicable systems have been developed ([Bondorf 1989] for programs in the form of term rewriting systems, [Gomard and Jones 1989] for a simple imperative language, [Fuller and Abramsky 1988] for Prolog, [Romanenko 1988] for a subset of Turchin's Refal language, [Consel 1988] and [Bondorf and Danvy 1989] for stronger systems handling first order Scheme programs).

These systems are reasonably efficient for first order languages, the generated compilers were typically between 3 and 10 times faster than compiling by partial evaluation of an interpreter.

Recent work by Bondorf extends that of [Bondorf and Danvy 1989]. The result is a self-applicable partial evaluator for higher order subset of Scheme where user-named functions are specialized with respect to higher order values [Bondorf 1990]. This ability makes Bondorf and Danvy's system more powerful than ours, the price being that it is far more complicated. The system begins with a BTA that uses a *closure analysis* along the lines of [Sestoft 1989], prior to a traditional abstract interpretation based binding time analysis. The closure analysis yields control flow information that is used to determine which program parts must be dynamic as a consequence of something else being dynamic. Several other static program analyses are performed, for example to detect and avoid code duplication. This abstract interpretation based approach provides an alternative to the one proposed by us in section 3.2: to annotate type terms with some program point information.

The resulting system is surely of more practical utility than ours, especially in a Scheme context. In contrast our system uses the classical lambda calculus, and in our opinion illustrates a fundamental principle: that correctness of binding time annotations is well viewed as type correctness, and that BTA can be done by type inference. Further, we have constructed a complete correctness proof.

5.1.2 Binding Time Analysis

Some basic ideas saw the light of day in [Jones and Muchnick 1978] but binding time analysis was not recognized as a central concept before the explicit use of binding time information in [Jones *et al.* 1985] gave a breakthrough in practice. Since then binding time analysis has become a main ingredient in all self-applicable partial evaluation systems, and it has become a research area on its own. The reasons why binding time analysis is essential for efficient self-application are detailed in [Bondorf *et al.* 1988].

The Nielsons have written several interesting papers on binding time analysis of a typed lambda calculus, e.g., [Nielson and Nielson 1988b]. They introduce a *well-formedness* criterion for typed two-level lambda expression. In their approach it is necessary for x to be of run-time *kind* for $\underline{\lambda}x. \text{body}$ to be well-formed. This is quite parallel to our demand that x must have type *code*. Similarly, it is necessary for x to be of compile-time kind for $\lambda x. \text{body}$ to be well-formed. Their run-time type system is not “flat” as is our *code*, but has function types, products, *etc.*. In their framework there is no construct like our *lift*, which to us seems necessary in practice, for example to write an interpreter suited for partial evaluation. As we saw in section 2.5.1, the presence of *lift* complicates the notion of best completion and calls for more work.

In [Mogensen 1989] a binding time analysis for polymorphically typed higher order languages is devised. In contrast with [Nielson and Nielson 1988b] (and us), Mogensen uses an abstract closure consisting of the function name and the binding times for the free variables to describe the binding time of a function. Neither [Mogensen 1989] nor [Nielson and Nielson 1988b] develop a partial evaluator.

5.1.3 Semantics-directed compiler generation

The pathbreaking work in this field was SIS: the Semantics Implementation System of [Mosses 1979]. SIS implements a pure version of the untyped lambda calculus using the call by need reduction strategy. Compiling from

a denotational semantics is done by translating the definition into a lambda expression, applying the result to the source program, and simplifying the result by reducing wherever possible. This is clearly a form of partial evaluation. SIS has a powerful notation for writing definitions, but it is unfortunately extremely slow, and is prone to infinite loops when using, for example, recursively defined environments. In our opinion this is because the reduction strategy is “on-line”, and the problem could be eliminated by annotations such as we have used. (Choosing annotations to avoid nontermination and code duplication is admittedly a challenging problem, but we feel it is one that should be solved *before* doing partial evaluation rather than during it.)

Systems based on the pure (typed) lambda calculus include [Paulson 1982, Weis 1987, Nielson and Nielson 1988a]. The first uses partial evaluation at compile time. It is considerably faster at compile time than SIS, but still very slow at run time. Weis’s system [Weis 1987] is probably the fastest in this category that has been used on large language definitions. In the Nielsons’s work so far, the greatest emphasis has been on correctness rather than efficient running systems.

Systems by Pleban and Appel [Pleban 1984, Appel 1985] achieve greater run-time efficiency at the expense of less pure semantic languages – one for each language definition in the former case, and a lambda calculus variant with special treatment of environments and stores in the latter. Finally Wand’s methodology is very powerful [Wand 1984, Wand 1982], but it seems to require so much cleverness from the user that it is not clear how it may be automated.

The main strength of our system is that is simple enough to be understood and proven correct and yet able to perform non-trivial compilation and compiler generation. The main weakness of our system seems to be that there is still a long way to the generation of “real” compilers, e.g. generating target code nearer to machine level, and applying the many forms of analysis and optimization seen in handwritten compilers. We think that we have a clear understanding of basic principles for compiler generation by partial evaluation, but to get past the toy level some hard work remains.

5.2 Future Work

5.2.1 Target Code Quality

All target programs generated by our system are written in the lambda calculus and though lambda calculus can be implemented quite efficiently, this is a source of inefficiency since it is far from traditional machine architectures. One possibility is to apply relevant optimizing transformations to the lambda calculus target programs, such as globalization of [Schmidt 1985, Sestoft 1989], detection of tail recursion *etc.*, and to compile the target program into machine code.

To avoid this two pass style the machine code can be generated during partial evaluation as in [Holst 1988]. In our framework the idea is to redefine the functions *build- λ* , *build-@* ... such that they construct a piece of machine code instead of a lambda expression. One might define, for example, that *build-if*(c_1 , c_2 , c_3) =

```

       $c_1$ 
      jump-null?  label1;
       $c_2$ 
      jump      label2;
label1:   $c_3$ 
label2:
```

A problem with this approach is that it is not (immediately) possible to apply transformations that require global analysis of the target program. An interesting idea [Schmidt 1988] is to examine the subject program for useful properties before partial evaluation, and let the partial evaluation phase use these properties to generate better code. A natural candidate for such an analysis is the detection of single-threaded variables in a language definition. In [Bondorf and Danvy 1989] a partial evaluator that can handle the presence of global variables is described.

5.2.2 Specialization of Named Combinators

The fundamental concept in most partial evaluators, and one which we do not use in any explicit way, is that of *program point specialization*. The idea

is that if, say, a binary function $f \times y = \text{body}$ at partial evaluation time is seen to be called with a static first argument (say 2) and a dynamic second argument, then a function $f_2 \ y = \text{optimized-body}$ is added to the residual program. We call f a program point (in an imperative language a program point is a label), and we call the pair $(f, 2)$ a specialized program point. A partial evaluator often uses two sets *pending* and *out* to keep track of the specialized program points for which code must be or has been generated.

Program point specialization has some advantages over our approach: Code need only be generated once for each specialized program point and it is thus easier to control code duplication. Some applications of partial evaluation rely heavily on sharing specialized functions in the residual program [Consel and Danvy 1989], but we have seen that compilation and compiler generation can be done nontrivially without this sharing.

A larger class of programs can be partially evaluated nontrivially (see example 14). There are also some disadvantages: The partial evaluator becomes more complicated, termination problems get harder, and since the two sets *pending* and *out* inevitably are dynamic the results of self-application are not as nice as the ones we get.

EXAMPLE 14 Consider Ackermann's function

$$\begin{aligned}\text{ack } 0 \ n &= n+1 \\ \text{ack } m \ 0 &= \text{ack } (m-1) \ 1 \\ \text{ack } m \ n &= \text{ack } (m-1) (\text{ack } m \ (n-1))\end{aligned}$$

and suppose that m is known to have the value 2. With specialization of named functions we can get the following residual program (about twice as fast as the original):

$$\begin{aligned}\text{ack}_1 \ 0 &= 2 \\ \text{ack}_1 \ n &= (\text{ack}_1 \ (n-1)) + 1 \\ \text{ack}_2 \ 0 &= 3 \\ \text{ack}_2 \ n &= \text{ack}_1 (\text{ack}_2 \ (n-1))\end{aligned}$$

The introduction of specialized named functions, ack_1 and ack_2 is crucial to the quality of the residual program. Without program point specialization no optimization would have occurred, since a residual fix can only define one function at a time, and three were used above. \square Ex14

5.2.3 Applications of BTA

The job of the BTA is to replace just enough of the program's operators with their residual counterparts so the non-residual program parts are well-typed. We now discuss some possible applications of such a BTA algorithm that have nothing to do with partial evaluation.

If a BTA algorithm was applied to a well-typed program with no free variables of type *code*, the result would be that of ordinary monomorphic type checking, namely a message of acceptance together with the program's type. If the BTA accepted a program to be (normally) executed without adding any annotations, it would mean that all type checks during evaluation could safely be omitted. This is the motivation for having strongly typed languages.

Untyped languages like Scheme are compiled to machine code that has type checks at run-time, though many of the type checks could easily be seen in advance always to succeed. For example, when an interpreter is run, it is clear that the syntactic dispatch will never make a type error as long as the input is atomic, and the environment lookup can also be seen to succeed. If BTA was applied to such an interpreter, some operators would be made residual (see section 3.1). For some of these operators type checking code should be generated, but no such code is needed for the rest. We suspect this could yield some speed-up since Lisp and Scheme programs often make limited use of untyped features, which are of course enough to make a traditional type checker fail.

A BTA algorithm could also be used instead of a traditional type checker to allow better type error messages to be given. Consider the interpreter from section 3.1, which would not pass a traditional type checker without annotations. We believe that seeing the well-annotated interpreter text:

$$\begin{aligned}
\mathcal{E} &= \lambda \text{exp} . \lambda \rho . \text{ case exp of} \\
&\quad \text{var(id)} && : \rho(\text{id}) \\
&\quad \text{app}(\text{exp}_1, \text{exp}_2) && : (\mathcal{E} \text{ exp}_1 \rho) \ @ \ (\mathcal{E} \text{ exp}_2 \rho) \\
&\quad \text{abs}(x, \text{exp}_1) && : \underline{\lambda} \text{val} . (\mathcal{E} \text{ exp}_1 (\lambda \text{id} . \text{ if id=x then val else } \rho(\text{id})))
\end{aligned}$$

together with the information that `exp` has type *base*, ρ has type *base* \rightarrow *code*, and \mathcal{E} has type *base* \rightarrow (*base* \rightarrow *code*) \rightarrow *code*, is much more useful to the user than the traditional error message: “failed to unify” followed by two type terms.

5.3 Conclusion

We have solved the open problem of developing and implementing a self-applicable partial evaluator for a higher order language (and so has [Bondorf 1990]). As programming language, we used an untyped lambda calculus with a fixed point operator and an explicit conditional. The solution turned out to be surprisingly simple and one reason for this is that we found it unnecessary to specialize program points to obtain nontrivial results. The main area of application of our methods is the automatic transformation of interpretive language specifications into compilers. From denotational definitions of small languages our partial evaluator can generate compilers with a very natural structure—they look almost “handwritten”.

As in other successful partial evaluation projects we use annotations added in a prephase to guide partial evaluation. We introduced the concept of well-annotatedness by means of a type system to ensure that the partial evaluator does not commit errors during partial evaluation. This approach also gives a new approach to the problem of ensuring finiteness of partial evaluation. Ideas for a binding time analysis have been discussed, and an implementation will be described in part II of this paper.

We tried the partial evaluator on two small language definitions and investigated the structure of the generated compilers. The compilers, as well as the generated compiler generator, were found to be readable—if the machine generated names are manually changed!

[Gomard 1989] and part II of this article contain a detailed proof of a

central theorem: the correctness of partial evaluation. This theorem guarantees that compilers generated from a language definition will always be faithful to the programming language semantics. It also contains a proof that there is a unique “best” way to annotate a given program (without lifts).

References

- [Appel 1985] A. Appel, Semantics-Directed Code Generation, in *12th ACM Symposium on Principles of Programming Languages*, pp. 315–324, 1985.
- [Beckman *et al.* 1976] L. Beckman *et al.*, A Partial Evaluator, and Its Use as a Programming Tool, *Artificial Intelligence* 7,4 (1976) 319–357.
- [Bondorf 1989] A. Bondorf, A Self-Applicable Partial Evaluator for Term Rewriting Systems, in *TAPSOFT '89. Proc. Int. Conf. Theory and Practice of Software Development, Barcelona, Spain, March 1989. (Lecture Notes in Computer Science, Vol. 352)*, edited by J. Diaz and F. Orejas, pp. 81–95, Springer-Verlag, 1989.
- [Bondorf 1990] A. Bondorf, Automatic Autoprojection of Higher Order Recursive Equations, 1990. (to appear in the proceedings of ESOP 90).
- [Bondorf and Danvy 1989] A. Bondorf and O. Danvy, Automatic Autoprojection for Recursive Equations with Global Variables and Abstract Data Types, 1989. (submitted for publication).
- [Bondorf *et al.* 1988] A. Bondorf, N.D. Jones, T. Mogensen, and P. Sestoft, *Binding Time Analysis and the Taming of Self-Application*, Draft, 18 pages, DIKU, University of Copenhagen, Denmark, August 1988.
- [Consel 1988] C. Consel, New Insights into Partial Evaluation: The Schism Experiment, in *ESOP'88, 2nd European Symposium on Programming, Nancy, France, March 1988. (Lecture Notes in Computer Science, Vol. 300)*, edited by H. Ganzinger, pp. 236–246, Springer-Verlag, 1988.
- [Consel and Danvy 1989] C. Consel and O. Danvy, Partial Evaluation of Pattern Matching in Strings, *Information Processing Letters* 30,2 (1989) 79–86.

- [Damas and Milner 1982] L. Damas and R. Milner, Principal Type-Schemes for Functional Programs, in *9th ACM Symposium on Principles of Programming Languages*, pp. 207–212, 1982.
- [Ershov 1978] A.P. Ershov, On the Essence of Compilation, in *Formal Description of Programming Concepts*, edited by E.J. Neuhold, pp. 391–420, North-Holland, 1978.
- [Ershov and Itkin 1977] A.P. Ershov and V.E. Itkin, Correctness of Mixed Computation in Algol-like Programs, in *Mathematical Foundations of Computer Science, Tatranská Lomnica, Czechoslovakia. (Lecture Notes in Computer Science, Vol. 53)*, edited by J. Gruska, pp. 59–77, Springer-Verlag, 1977.
- [Fuller and Abramsky 1988] D.A. Fuller and S. Abramsky, Mixed Computation of Prolog Programs, *New Generation Computing* 6,2,3 (1988) 119–141.
- [Futamura 1971] Y. Futamura, Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler, *Systems, Computers, Controls* 2,5 (1971) 45–50.
- [Goguen *et al.* 1977] G. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, Initial Algebra Semantics and Continuous Algebras, *Journal of the ACM* 24 (1977) 68–95.
- [Gomard 1989] C. K. Gomard, *Higher Order Partial Evaluation – HOPE for the Lambda Calculus*, Master’s thesis, DIKU, University of Copenhagen, Denmark, September 1989.
- [Gomard and Jones 1989] C. K. Gomard and N. D. Jones, Compiler Generation by Partial Evaluation: a Case Study, in *Information Processing 89*, edited by H. Gallaire, IFIP, 1989.
- [Guzowski 1988] M.A. Guzowski, *Towards Developing a Reflexive Partial Evaluator for an Interesting Subset of LISP*, Master’s thesis, Dept. of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio, January 1988.
- [Hansen and Träff 1989] T. A. Hansen and J. L. Träff, *Memoization and its Use in Lazy and Incremental Program Generation*, Master’s thesis, DIKU, University of Copenhagen, 1989.

- [Holst 1988] N.C.K. Holst, Language Triplets: The AMIX Approach, in *Partial Evaluation and Mixed Computation*, edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pp. 167–185, North-Holland, 1988.
- [Huet 1976] G. Huet, *Resolution d'équations dans les langages d'ordre 1, 2, ..., ω* , PhD thesis, Univ. de Paris VII, 1976.
- [Jones 1988] N.D. Jones, Automatic Program Specialization: A Re-Examination from Basic Principles, in *Partial Evaluation and Mixed Computation*, edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pp. 225–282, North-Holland, 1988.
- [Jones and Muchnick 1978] Neil D. Jones and Steven S. Muchnick, *TEMPO: A Unified Treatment of Binding Time and Parameter Passing Concepts in Programming Languages. (Lecture Notes in Computer Science, vol. 66)*, Springer-Verlag, 1978.
- [Jones *et al.* 1985] N.D. Jones, P. Sestoft, and H. Søndergaard, An Experiment in Partial Evaluation: The Generation of a Compiler Generator, in *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, Vol. 202)*, edited by J.-P. Jouannaud, pp. 124–140, Springer-Verlag, 1985.
- [Jones *et al.* 1989] N.D. Jones, P. Sestoft, and H. Søndergaard, Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation, *Lisp and Symbolic Computation* 2,1 (1989) 9–50.
- [Jones *et al.* 1990] N. D. Jones, C. K. Gomard, A. Bondorf, O. Danvy, and T. Mogensen, A Self-applicable Partial Evaluator for the Lambda Calculus, in *IEEE Computer Society 1990 International Conference on Computer Languages*, March 1990.
- [Kuo and Mishra 1989] T-M. Kuo and P. Mishra, Strictness Analysis: A New Perspective Based on Type Inference, in *Functional Programming Languages and Computer Architecture, London, September 89*, ACM Press and Addison-Wesley, 1989.
- [Lombardi 1967] L.A. Lombardi, Incremental Computation, in *Advances in Computers, Vol. 8*, edited by F.L. Alt and M. Rubinoff, pp. 247–333, Academic Press, 1967.
- [Milner 1978] R. Milner, A Theory of Type Polymorphism in Programming, *Journal of Computer and System Sciences* 17,3 (1978) 348–375.

- [Mogensen 1988] T. Mogensen, Partially Static Structures in a Self-Applicable Partial Evaluator, in *Partial Evaluation and Mixed Computation*, edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pp. 325–347, North-Holland, 1988.
- [Mogensen 1989] T. Mogensen, Binding Time Analysis for Polymorphically Typed Higher Order Languages, in *TAPSOFT '89. Proc. Int. Conf. Theory and Practice of Software Development, Barcelona, Spain, March 1989. Lecture Notes in Computer Science 352*, edited by J. Diaz and F. Orejas, pp. 298–312, Springer-Verlag, 1989.
- [Morris 1973] F. L. Morris, Advice on Structuring Compilers and proving them correct, in *1st ACM Symposium on Principles of Programming Languages*, pp. 144–152, 1973.
- [Mosses 1979] P. Mosses, *SIS — Semantics Implementation System, Reference Manual and User Guide*, DAIMI Report MD-30, DAIMI, University of Århus, Denmark, 1979.
- [Nielson and Nielson 1988a] F. Nielson and H. R. Nielson, *The TML-Approach to Compiler-Compilers*, Technical Report 1988-47, Department of Computer Science, Technical University of Denmark, 1988.
- [Nielson and Nielson 1988b] H. R. Nielson and F. Nielson, Automatic Binding Time Analysis for a Typed λ -calculus, in *15th ACM Symposium on Principles of Programming Languages*, pp. 98–106, 1988.
- [Paulson 1982] L. Paulson, A Semantics-Directed Compiler Generator, in *9th ACM Symposium on Principles of Programming Languages*, pp. 224–233, 1982.
- [Pleban 1984] U. Pleban, Compiler Prototyping Using Formal Semantics, *SIGPLAN Notices* 19,6 (1984) 94–105.
- [Plotkin 1975] G. Plotkin, Call-by-name, Call-by-value and the lambda calculus, *Theoretical Computer Science* 1 (1975) 125–159.
- [Romanenko 1988] S.A. Romanenko, A Compiler Generator Produced by a Self-Applicable Specializer Can Have a Surprisingly Natural and Understandable Structure, in *Partial Evaluation and Mixed Computation*, edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pp. 445–463, North-Holland, 1988.

- [Schmidt 1985] D. A. Schmidt, Detecting Global Variables in Denotational Specifications, *ACM Transactions on Programming Languages and Systems* 7,2 (1985) 299–310.
- [Schmidt 1986] D. A. Schmidt, *Denotational Semantics*, Allyn and Bacon, 1986.
- [Schmidt 1988] D.A. Schmidt, Static Properties of Partial Evaluation, in *Partial Evaluation and Mixed Computation*, edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pp. 465–483, North-Holland, 1988.
- [Schooler 1984] R. Schooler, *Partial Evaluation as a Means of Language Extensibility*, Master’s thesis, 84 pages, MIT/LCS/TR-324, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, August 1984.
- [Sestoft 1988] P. Sestoft, Automatic Call Unfolding in a Partial Evaluator, in *Partial Evaluation and Mixed Computation*, edited by D. Bjørner, A.P. Ershov, and N.D. Jones, pp. 485–506, North-Holland, 1988.
- [Sestoft 1989] P. Sestoft, Replacing Function Parameters by Global Variables, in *Functional Programming Languages and Computer Architecture, London, September 89*, ACM Press and Addison-Wesley, 1989.
- [Stoy 1977] J. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.
- [Turchin 1980] V.F. Turchin, The Use of Metasystem Transition in Theorem Proving and Program Optimization, in *Automata, Languages and Programming. Seventh ICALP, Noordwijkerhout, The Netherlands. (Lecture Notes in Computer Science, Vol. 85)*, edited by J. De Bakker and J. van Leeuwen, pp. 645–657, Springer-Verlag, 1980.
- [Wadler 1989] P. Wadler, Linear Types Can Change the World!, 1989. (draft).
- [Wand 1982] M. Wand, Semantics-Directed Machine Architecture, in *9th ACM Symposium on Principles of Programming Languages*, pp. 234–241, 1982.
- [Wand 1984] M. Wand, A Semantic Prototyping System, in *SIGPLAN ’84 Symposion on Compiler Construction*, pp. 213–221, 1984.

- [Wand 1986] M. Wand, Finding the Source of Type Errors, in *13th ACM Symposium on Principles of Programming Languages*, pp. 38–43, 1986.
- [Weis 1987] P. Weis, *Le Systeme SAM: Metacompilation tres efficace a l'aide d'operateur semantiques*, PhD thesis, l'Universite Paris VII, 1987. (in french).

A A Mix Session

```
> (define (fix f) (lambda (a) ((f (fix f)) a)))
fix
```

```
> (define L-interpreter-in-Scheme      ; Scheme code implementing L
    (fix (lambda (e)
            (lambda (exp)
              (lambda (env)
                (if (atom? exp)
                    (env exp)
                    ((lambda (head-exp)
                      (lambda (tail-exp)
                        (if ((eq? head-exp) 'const)
                            (car tail-exp)
                            (if ((eq? head-exp) 'lam)
                                (lambda (value)
                                  ((e (take-body exp))
                                     (lambda (y)
                                       (if ((eq? y) (take-var exp))
                                           value
                                           (env y))))))
                        (if ...
```

L-interpreter-in-Scheme

```
> (define L                                ; Scheme code implementing L.
    (lambda (pgm d1 d2)                   ; pgm is expected to have two free
      ((L-interpreter-in-Scheme pgm)     ; variables, x1 and x2.
       (lambda (id)                       ;
         (if (eq? id 'x1)                 ; the standard initial environment
             d1                           ; is extended with the bindings
             (if (eq? id 'x2)             ; [x1 -> d2, x2 -> d2]
                 d2
                 (initial-env id))))))
L
```

```
> (define power                            ; power computes x2 to the x1'st (L code)
    '(@ (@ (fix (lam p (lam n (lam x
      (if (@ (@ eq? n) (const 0))
          (const 1)
          (@ (@ * x)
              (@ (@ p (@ (@ - n) (const 1))) x))))))
      x1) x2))
power
```

```

> (L power (const 2) (const 3))
9

> (define E '(fix (lam E (lam exp (lam env          ; core of the self-interpreter for L
  (if (@ atom? exp)
      (@ env exp)
      (@ (@ (lam head-exp (lam tail-exp
        (if (@ (@ eq? head-exp) (const const))
            (@ car tail-exp)
            (if (@ (@ eq? head-exp) (const lam))
                (lam value (@ (@ E (@ take-body exp))
                    (lam y (if (@ (@ eq? y)
                        (@ take-var exp))
                        value
                        (@ env y))))))
        (if (@ (@ eq? head-exp) (const @))
            (@ (@ (@ E (@ car tail-exp)) env)
                (@ (@ E (@ cadr tail-exp)) env))
            (if (@ (@ eq? head-exp) (const if))
                (if (@ (@ E (@ car tail-exp)) env)
                    (@ (@ E (@ cadr tail-exp)) env)
                    (@ (@ E (@ caddr tail-exp)) env))
            (if (@ (@ eq? head-exp) (const fix))
                (fix (@ (@ E (@ car tail-exp)) env))
                (@ (@ error exp) (const "Wrong syntax"))
                ))))))
      (@ car exp))
      (@ cdr exp))))))
E

; x1 must be bound to a binary L-pro-
> (define self-int '(@ (@ ,E x1)          ; gram, x2 to a pair (d1, d2) of the
  (lam id          ; input to program x1
    (if (@ (@ eq? id) (const x1))
        (@ car x2)
        (if (@ (@ eq? id) (const x2))
            (@ cdr x2)
            (@ ,i-env-txt id))))))
self-int

> (L self-int power (cons (const 2) (const 3)))
9

> (define T '(fix (lam T (lam exp (lam env          ; evaluator for 2-level L

```



```

(if (@ atom? exp)
  (@ env exp)
  (@ (@ (lam head-exp (lam tail-exp
    (if (@ (@ eq? head-exp) (const const))
      (@ car tail-exp)
      (if (@ (@ eq? head-exp) (const const-r))
        (@ build-const (@ car tail-exp))
        (if (@ (@ eq? head-exp) (const lam))
          (lam value (@ (@ T (@ take-body exp))
            (lam y (if (@ (@ eq? y)
              (@ take-var exp))
              value
              (@ env y))))))
          (if (@ (@ eq? head-exp) (const lam-r))
            (@ (lam par
              (@ (@ build-lam par)
                (@ (@ T (@ cadr tail-exp))
                  (lam x ...

```

T

```

; mix takes two arguments x1 & x2. x1 is a
; binary L-program and x2 is the first input
; to x1. Thus x1 is T-evaluated with its first
; free variable (also named x1!) bound to input
> (define mix '(@ (@ ,T x1) ; x2, and its second variable (named x2) bound
  (lam id ; to its own name.
    (if (@ (@ eq? id) (const x1))
      x2
      (if (@ (@ eq? id) (const x2))
        (const x2) ; x2 is bound to the variable
        (@ ,i-env-txt id)))))) ; name x2. The residual program
mix ; thus has free variable, x2.

> (define power-ann
  '(@ (@ (fix (lam p (lam n (lam x
    (if (@ (@ eq? n) (const 0))
      (const 1)
      (@-r (@-r *-r x)
        (@ (@ p (@ (@ - n) (const 1))) x))))))
    x1) x2))
power-ann

> (L mix power-ann (const 2))
(@ (@ * x2) (@ (@ * x2) 1))

```

```

; the definition of E-ann is
> (define self-int-ann '(@ (@ ,E-ann x1) ; not in this session. (Essen-
    (lam id ; tially a subset of T-ann.)
      (if (@ (@ eq? id) (const x1))
          (@-r car-r x2)
          (if (@ (@ eq? id) (const x2))
              (@-r cdr-r x2)
              (@ ,i-env-ann id))))))

self-int-ann

> (define power-target (L mix sint-ann power))
power-target

> power-target
(@ (@ (fix (lam value-001
    (lam value-002
    (lam value-003
    (if (@ (@ eq? value-002) (const 0))
        (const 1)
        (@ (@ * value-003)
            (@ (@ value-001
                (@ (@ - value-002) (const 1)))
                value-003)))))))
    (@ car x2))
    (@ cdr x2))

; The residual program of the self-interpreter, power-target, has one free
; variable x2. This variable should be bound to the pair of input values
; (d1, d2) to the program power.

> (L power-target 'dummy (cons (const 2) (const 3)))
9

> (define T-ann '(fix (lam mix (lam exp (lam env ; Full listing in appendix B
    (if (@ atom? exp)
        (@ env exp)
        (@ (@ (lam head-exp (lam tail-exp
            (if (@ (@ eq? head-exp) (const const))
                (@ lift (@ car tail-exp))
                (if ...
T-ann

> (define mix-ann '(@ (@ ,T-ann x1)
    (lam id

```

```

        (if (@ (@ eq? id) (const x1))
            x2
            (if (@ (@ eq? id) (const x2))
                (const-r x2)
                (@ ,i-env-ann id))))))
mix-ann

> (define self-compiler (L mix mix-ann sint-ann))
self-compiler

; The self-compiler has one free variable, x2. The program to compile is
; bound to this variable.

> (define power-target1 (L self-compiler 'dummy power))
power-target1

> (equal? power-target power-target1)
#t

> (define cogen (L mix mix-ann mix-ann))
cogen

> (define self-compiler1 (L cogen 'dummy sint-ann))
self-compiler1

> (equal? self-compiler self-compiler1)
#t

>

```

B T Annotated

```
(fix (lam T
(lam exp (lam env
  (if (atom? exp)
    (env exp)
    (((lam head-exp (lam tail-exp
      (if ((eq? head-exp) (const const))
        (lift (car tail-exp))
        (if ((eq? head-exp) (const const-r))
          (@-r build-const-r (lift (car tail-exp)))
          (if ((eq? head-exp) (const lam))
            (lam-r value ((T (take-body exp))
                          (lam y (if ((eq? y)
                                      (take-var exp))
                                      value
                                      (env y))))))
            (if ((eq? head-exp) (const lam-r))
              (@-r (lam-r par
                    (@-r (@-r build-lam-r par)
                        ((T (cadr tail-exp))
                          (lam x
                            (if ((eq? x)
                                (car tail-exp))
                                par
                                (env x))))))
                    (@-r new-name-r (const-r nil)))
              (if ((eq? head-exp) (const @))
                (@-r ((T (car tail-exp)) env)
                    ((T (cadr tail-exp)) env))
                (if ((eq? head-exp) (const @-r))
                  (@-r (@-r build-app-r ((T (car tail-exp)) env))
                      ((T (cadr tail-exp)) env))
                  (if ((eq? head-exp) (const if))
                    (if-r ((T (car tail-exp)) env)
                        ((T (cadr tail-exp)) env)
                        ((T (caddr tail-exp)) env))
                    (if ((eq? head-exp) (const if-r))
                      (@-r (@-r (@-r build-if-r
                                ((T (car tail-exp)) env))
                            ((T (cadr tail-exp)) env))
                          ((T (caddr tail-exp)) env))
                      (if ((eq? head-exp) (const fix))
                        (fix-r ((T (car tail-exp)) env))
```

```

                                (if ((eq? head-exp) (const fix-r))
                                    (@-r build-fix-r ((T (car tail-exp)) env))
                                (@-r (@-r error-r exp)
                                    (const-r "Wrong syntax"))
                                ))))))))
    (car exp))
  (cdr exp))))))

```

C The Generated Self-Compiler

Below is a printing of the generated self-compiler. We have done very little editing as you can see, the only major thing was to remove most of the endless case-analysis of predefined functions. The compiler takes its input through the free variable `x2`.

```

(((fix (lam v-33 (lam v-34 (lam v-35
(if (atom? v-34)
    (v-35 v-34)
    ((lam v-36
      (lam v-37
        (if ((eq? v-36) (const const))
            (lift (car v-37))
            (if ((eq? v-36)
                (const lam))
                ((lam par-38
                  ((build-lam par-38)
                   ((v-33 (take-body v-34))
                    (lam v-39
                     (if ((eq? v-39)
                         (take-var v-34))
                         par-38
                         (v-35 v-39))))))
                  (new-name (const nil)))
            (if ((eq? v-36) (const @))
                ((build-app
                  ((v-33 (car v-37)) v-35))
                  ((v-33 (cadr v-37)) v-35))
                (if ((eq? v-36) (const if))
                    (((build-if
                      ((v-33 (car v-37)) v-35))
                      ((v-33 (cadr v-37)) v-35))
                      ((v-33 (caddr v-37)) v-35))
                    (if ((eq? v-36) (const fix))

```

```

                                (build-fix
                                ((v-33 (car v-37)) v-35))
                                ((build-app
                                ((build-app (const error)) v-34))
                                (const-r "Wrong syntax"))))))))
    (car v-34))
  (cdr v-34)))))
x2)
(lam v-32
  (if ((eq? v-32) (const assoc))
    (const assoc)
    (if ((eq? v-32) (const atom?))
      (const atom?)
      (if ((eq? v-32) (const car))
        (const car)
        (if ((eq? v-32) (const cdr))
          (const cdr)
          (if ((eq? v-32) (const cadr))
            (const cadr)
            (if ((eq? v-32) (const caddr))
              (const caddr)
              (if ((eq? v-32) (const cons))
                (const cons)
                (if ((eq? v-32)
                    (const eq?))
                  (const eq?)
                  (if ((eq? v-32)
                      (const build-lam))
                    (const build-lam)
                    (if ((eq? v-32)
                        (const build-app))
                      (const build-app))
                    (if ....

```

and so on *ad nauseam* ... (539 lines more.)

D The Generated Compiler Generator

Below is a printing of the generated compiler generator. We have done very little editing (as you can see), the only major thing being that we have removed most of the endless case-analysis of predefined functions. The compiler generator takes its input through the free variable `x2`.

```
((fix
(lam v-165 (lam v-166 (lam v-167
(if (atom? v-166)
(v-167 v-166)
((lam v-168
(lam v-169
(if ((eq? v-168) (const const))
.lift (car v-169))
(if ((eq? v-168) (const const-r))
(build-app (const build-const))
.lift (car v-169)))
(if ((eq? v-168)
(const lam))
((lam par-172
((build-lam par-172)
((v-165 (take-body v-166))
(lam v-173
(if ((eq? v-173)
(take-var v-166))
par-172
(v-167 v-173))))))
(new-name (const nil)))
(if ((eq? v-168) (const lam-r))
(build-app
((lam par-170
((build-lam par-170)
((build-app
(build-app
(const build-lam))
par-170))
((v-165
(cadr v-169))
(lam v-171
(if ((eq? v-171)
(car v-169))
par-170
(v-167 v-171)))))))))
```

```

        (new-name (const nil))))
      ((build-app (const new-name))
       (build-const nil)))
    (if ((eq? v-168)
        (const @))
        ((build-app
          ((v-165 (car v-169)) v-167))
          ((v-165 (cadr v-169)) v-167))
        (if ((eq? v-168) (const @-r))
            ((build-app
              ((build-app (const build-app))
               ((v-165 (car v-169)) v-167)))
              ((v-165 (cadr v-169)) v-167))
            (if ((eq? v-168) (const if))
                (((build-if ((v-165 (car v-169))
                             v-167))
                       ((v-165 (cadr v-169)) v-167))
                 ((v-165 (caddr v-169)) v-167))
                (if ((eq? v-168) (const if-r))
                    ((build-app
                      ((build-app
                        ((build-app
                          (const build-if))
                          ((v-165 (car v-169))
                           v-167)))
                        ((v-165 (cadr v-169))
                         v-167)))
                      ((v-165 (caddrv-169))
                       v-167))
                    (if ((eq? v-168) (const fix))
                        (build-fix
                         ((v-165 (car v-169)) v-167))
                        (if ((eq? v-168) (const fix-r))
                            ((build-app
                              (const build-fix))
                             ((v-165 (car v-169)) v-167))
                            ((build-app
                              ((build-app (const error))
                               v-166))
                               (const-r "Wrong syntax"))))))))))))
      (car v-166))
    (cdr v-166))))))
  x2)
(lam v-164

```



```

(if ((eq? v-164) (const assoc))
  (const assoc)
  (if ((eq? v-164) (const atom?))
    (const atom?)
    (if ((eq? v-164) (const car))
      (const car)
      (if ((eq? v-164) (const cdr))
        (const cdr)
        (if ((eq? v-164) (const cadr))
          (const cadr)
          (if ((eq? v-164) (const caddr))
            (const caddr)
            (if ...

```

and so on *ad nauseam* ... (538 lines more.)

E Tiny Interpreter

Below is the listing of the annotated *Tiny* interpreter that we have used to compile and generate compilers.

```

(lam program
  ((lam e ((lam d ((lam c
    ((lam var-decl
      ((lam statement-part
        ((lam var-env
          (@-r ((c statement-part) var-env)
            (const-r new-store)))
          ((d (cdr var-decl)) (const 0))))
        (take-statement-part program)))
      (take-var-decl program)))

  (fix (lam c (lam com (lam var-env (lam-r store
    (if (seq? com)
      (@-r ((c (second-com com)) var-env)
        (@-r ((c (first-com com)) var-env) store))
      (if (assign? com)
        (@-r (@-r (@-r update-r
          (var-env (take-id com)))
          (((e (take-ass-exp com)) var-env) store))
          store)
        (if (while? com)
          (@-r (fix-r (lam-r f

```

```

(lam-r store
  (if-r (@-r (@-r eq?-r
              (const-r 0))
            (((e (take-cond-exp
                  com))
              var-env)
            store)))
  store
  (@-r f (@-r
          ((c
            (take-while-body com))
          var-env)
        store)))) store)
(@-r (@-r error "Illegal command") com)))))))))

(fix (lam d (lam var-list (lam location
  (if (null? var-list)
    (lam x (const no-location))
    (lam y
      (if ((eq? y) (car var-list))
        location
        (((d (cdr var-list)) (1+ location)) y))))))))))

(fix (lam e (lam exp (lam var-env (lam store
  (if (number? exp)
    exp
    (if (is-var? exp)
      (@-r (@-r (const-r access) (var-env exp)) store)
      (@-r (@-r ((lam id (if ((eq? id) (const *))
                            *-r
                            (if ((eq? id) (const -))
                                --r
                                error-r)))
          (op-name exp))
      (((e (arg1 exp)) var-env) store))
      (((e (arg2 exp)) var-env) store))))))))))

```

F The Generated Tiny Compiler

Below is a listing of the generated *Tiny* to lambda calculus compiler.

```
(lam v-62
  ((lam v-68
    ((lam v-74
      ((lam v-81
        ((lam v-82
          ((lam v-83
            ((lam v-84
              (((v-81
                v-83)
                v-84)
                (build-const
                  (const
                    new-store))))))
              ((v-74
                (cdr v-82))
                (const 0))))
              (take-statement-part v-62)))
            (take-var-decl v-62)))

(fix (lam v-75 (lam v-76 (lam v-77 (lam v-78
(if (seq? v-76)
  (((v-75 (second-com v-76)) v-77)
   (((v-75 (first-com v-76)) v-77) v-78)))
(if (assign? v-76)
  ((build-app
    ((build-app
      ((build-app (const update))
        (v-77 (take-id v-76))))
      (((v-68 (take-ass-exp v-76)) v-77)
        v-78)))
    v-78)
  (if (while? v-76)
    ((build-app (build-fix
      ((lam par-79
        ((build-lam par-79)
        ((lam par-80
          ((build-lam par-80)
          (((build-if
            ((build-app
              ((build-app (const eq?))
```

```

                                (build-const (const 0)))
                                (((v-68 (take-cond-exp v-76))
                                   v-77)
                                 par-80)))
                                par-80)
                                ((build-app par-79)
                                 (((v-75 (take-while-body v-76))
                                   v-77)
                                 par-80))))
                                (new-name (const nil))))
                                (new-name (const nil))))
                                v-78)
    ((build-app
      ((build-app error)
        (error "Illegal command"))) v-76)))))))))

(fix (lam v-69 (lam v-70 (lam v-71
  (if (null? v-70)
    (lam v-73 (const no-location))
    (lam v-72
      (if ((eq? v-72) (car v-70))
        v-71
        (((v-69 (cdr v-70))
          (1+ v-71))
         v-72))))))))))

(fix (lam v-63 (lam v-64 (lam v-65 (lam v-66
  (if (number? v-64)
    (lift v-64)
    (if (is-var? v-64)
      ((build-app
        ((build-app (const access))
          (lift (v-65 v-64))))
        v-66)
      ((build-app
        ((build-app
          ((lam v-67
            (if ((eq? v-67) (const *))
              (const mult)
              (if ((eq? v-67) (const -))
                (const minus)
                (const error))))
            (op-name v-64)))
          (((v-63 (arg1 v-64)) v-65) v-66)))

```

((v-63 (arg2 v-64)) v-65) v-66)))))))))

$\frac{\tau(x) = T}{\tau \vdash x: T}$
$\frac{\tau[x \mapsto T_2] \vdash \text{texp}: T_1}{\tau \vdash \lambda x. \text{texp}: T_2 \rightarrow T_1}$
$\frac{\tau \vdash \text{texp}_1: T_2 \rightarrow T_1, \quad \tau \vdash \text{texp}_2: T_2}{\tau \vdash \text{texp}_1 @ \text{texp}_2: T_1}$
$\frac{\tau \vdash \text{texp}: (T_1 \rightarrow T_2) \rightarrow (T_1 \rightarrow T_2)}{\tau \vdash \text{fix texp}: (T_1 \rightarrow T_2)}$
$\frac{\tau \vdash \text{texp}_1: \text{base}, \quad \tau \vdash \text{texp}_2: T, \quad \tau \vdash \text{texp}_3: T}{\tau \vdash \text{if texp}_1 \text{ texp}_2 \text{ texp}_3: T}$
$\tau \vdash \text{const } c: \text{base}$
$\frac{\tau[x \mapsto \text{code}] \vdash \text{texp}: \text{code}}{\tau \vdash \underline{\lambda} x. \text{texp}: \text{code}}$
$\frac{\tau \vdash \text{texp}_1: \text{code}, \quad \tau \vdash \text{texp}_2: \text{code}}{\tau \vdash \text{texp}_1 @ \text{texp}_2: \text{code}}$
$\frac{\tau \vdash \text{texp}: \text{code}}{\tau \vdash \underline{\text{fix}} \text{ texp}: \text{code}}$
$\frac{\tau \vdash \text{texp}_1: \text{code}, \quad \tau \vdash \text{texp}_2: \text{code}, \quad \tau \vdash \text{texp}_3: \text{code}}{\tau \vdash \underline{\text{if}} \text{ texp}_1 \text{ texp}_2 \text{ texp}_3: \text{code}}$
$\tau \vdash \underline{\text{const}} c: \text{code}$
$\frac{\tau \vdash \text{texp}: \text{base}}{\tau \vdash \text{lift texp}: \text{code}}$

Figure 3: Type Rules Checking Well-annotatedness

```

(fix (lam sint
  (lam exp (lam env
    (if (atom? exp)
      (env exp)
      (((lam head-of-exp (lam tail-of-exp
        (if ((eq? head-of-exp) (const const))
          (lift (car tail-of-exp))
          (if ((eq? head-of-exp) (const lam))
            (lam-r value ((sint (take-body exp))
                          (lam y (if ((eq? y) (take-var exp))
                                      value
                                      (env y))))))
          (if ((eq? head-of-exp) (const @))
            (@-r ((sint (car tail-of-exp)) env)
                  ((sint (cadr tail-of-exp)) env))
            (if ((eq? head-of-exp) (const if))
              (if-r ((sint (car tail-of-exp)) env)
                     ((sint (cadr tail-of-exp)) env)
                     ((sint (caddr tail-of-exp)) env))
              (if ((eq? head-of-exp) (const fix))
                (fix-r ((sint (car tail-of-exp)) env))
                (@-r (@-r error-r exp)
                      (lift (const "Wrong syntax"))))))))
    (car exp))
    (cdr exp))))))

```

Figure 4: Well-Annotated Lambda Calculus Self-Interpreter

Program p :

```

(fix (lam fib (lam x
  (if ((< x) (const 2))
    (const 1)
    ((+ (fib ((- x) (const 1)))
        (fib ((- x) (const 2)))))))

```

Program $r = \mathbf{L} \text{ mix } [sint, p]$:

```

(fix (lam value-6 (lam value-7
  (if ((< value-7) (const 2))
    (const 1)
    ((+ (value-6 ((- value-7) (const 1)))
        (value-6 ((- value-7) (const 2)))))))

```

Figure 5: Fibonacci And Fibonacci

```

(if ((eq? head-of-exp) (const lam))
  ((lam name
    ((build-lam name)
      ((compile (take-body exp))
        (lam id (if ((eq? id) (take-var exp))
                     name
                     (env id))))))
    (new-name (const nil)))
  (if ((eq? head-of-exp) (const @))
    ((build-app ((compile (car tail-of-exp)) env))
      ((compile (cadr tail-of-exp)) env))))

```

Figure 6: Compilation of Abstractions and Applications

```

(if ((eq? head-of-exp) (const @))
  ((build-app ((cogen (car tail-of-exp)) env))
    ((cogen (cadr tail-of-exp)) env))
  (if ((eq? head-of-exp) (const @-r))
    ((build-app
      ((build-app (const build-app))
        ((cogen (car tail-of-exp)) env)))
      ((cogen (cadr tail-of-exp)) env))))

```

Figure 7: Application Treatment in the Compiler Generator

Semantic Domains

$$\begin{aligned} \text{Store} &= \text{Location} \rightarrow \text{Nat} \\ \text{Environment} &= \text{Variable} \rightarrow \text{Location} \end{aligned}$$

\mathcal{P} : $\text{program} \rightarrow \text{Store} \rightarrow \text{Store}$

$$\mathcal{P}[\underline{\text{variables}} \ v_1, \dots, v_n; \text{cmd}] \ \sigma_{\text{init}} = \mathcal{C}[\text{cmd}] \ (\mathcal{D}[v_1, \dots, v_n] \ \text{first-location}) \ \sigma_{\text{init}}$$

\mathcal{D} : $\text{variable}^* \rightarrow \text{Location} \rightarrow \text{Environment}$

$$\begin{aligned} \mathcal{D}[v_1, \dots, v_n] \ \text{loc} &= \lambda \text{id}. \text{id} = v_1 \rightarrow \text{loc}, \mathcal{D}[v_2, \dots, v_n] \ (\text{next-loc loc}) \ \text{id} \\ \mathcal{D}[] \ \text{loc} &= \lambda x. \text{error}_{\text{loc}} \end{aligned}$$

\mathcal{C} : $\text{command} \rightarrow \text{Environment} \rightarrow \text{Store} \rightarrow \text{Store}$

$$\mathcal{C}[c_1; c_2] \ \rho \ \sigma = \mathcal{C}[c_2] \ \rho \ (\mathcal{C}[c_1] \ \rho \ \sigma)$$

$$\mathcal{C}[\text{var} := \text{exp}] \ \rho \ \sigma = \sigma[\rho(\text{var}) \mapsto \mathcal{E}[\text{exp}] \ \rho \ \sigma]$$

$$\mathcal{C}[\underline{\text{while}} \ \text{exp} \ \underline{\text{do}} \ c] \ \rho \ \sigma = (\text{fix } \lambda f. \lambda \sigma_1. \mathcal{E}[\text{exp}] \ \rho \ \sigma = 0 \rightarrow \sigma \ f(\mathcal{C}[c] \ \rho \ \sigma_1)) \ \sigma$$

\mathcal{E} : $\text{expression} \rightarrow \text{Environment} \rightarrow \text{Store} \rightarrow \text{Nat}$

... as usual ...

Figure 8: Tiny Semantics

```

(fix (lam c
  (lam com (lam rho (lam-r store
    (if (seq? com)
      (@-r ((c (second-com com)) rho)
        (@-r ((c (first-com com)) rho) store))
    (if (assign? com)
      (@-r (@-r (@-r update-r (rho (take-id com)))
        (((e (take-exp com)) rho) store))
        store)
    (if (while? com)
      (@-r (fix-r (lam-r f
        (lam-r store
          (if-r (@-r (@-r eq?-r (const-r 0))
            (((e (take-cond com)) rho) store))
            store
            (@-r f (@-r ((c (take-body com)) rho)
              store)))) v))
        store)
      (@-r (@-r error "Illegal command") (lift com))))))))))

```

Figure 9: The semantic function \mathcal{C} in two-level lambda calculus

```

((fix (lam fac
  (lam store-1
    (if ((eq? 0) ((access 1) store-1))
      store-1
      (fac
        ((lam store-3
          (((update 1) ((- ((access 1) store-3)) 1)) store-3))
          ((update 0)
            ((* ((access 0) store-1)) ((access 1) store-1)))
            store-1))))))
    (((update 1) 6)
      ((update 0) 1)
      (const new-store))))

```

Figure 10: Factorial Residual Program

$\mathcal{C}_c[\mathbf{c}_1; \mathbf{c}_2] \rho$	$= \text{let } nn_1 = \text{new-name in}$ $\mathcal{C}_c[\mathbf{c}_2] \rho \text{ “@” } (\mathcal{C}_c[\mathbf{c}_1] \rho \text{ “@” } nn_1)$
$\mathcal{C}_c[\mathbf{var} := \text{exp}] \rho$	$= \text{let } nn_1 = \text{new-name in}$ $\text{“update” “@” } \rho(\mathbf{var}) \text{ “@” } (\mathcal{E}_c[\text{exp}] \rho nn_1) \text{ “@” } nn_1$
$\mathcal{C}_c[\mathbf{while} \text{ exp } \mathbf{c}] \rho$	$= \text{let } nn_1 = \text{new-name}$ $nn_2 = \text{new-name}$ $nn_3 = \text{new-name in}$ $(\text{“fix” “}\lambda\text{” } nn_3. \text{“}\lambda\text{” } nn_2. \text{ “if = @ 0” “@” } (\mathcal{E}_c[\text{exp}] \rho nn_2)$ nn_2 $nn_3 \text{ “@” } (\mathcal{C}_c[\mathbf{c}] \rho \text{ “@” } nn_2)) \text{ “@” } nn_1$

Figure 11: \mathcal{C}_c Syntactically Sugared

```

(fix (lam c
  (lam com (lam rho
    ((lam new-name1
      ((build-lam new-name1)
        (if (seq? com)
          ((build-app ((c (second-com com)) rho))
                    ((build-app ((c (first-com com)) rho))
                                new-name1)))
        (if (assign? com)
          ((build-app
            ((build-app
              ((build-app (const update))
                (rho (take-id com))))
              (((e (take-ass-exp com)) rho) new-name1))))
            new-name1)
          (if (while? com)
            ((build-app (build-fix
              ((lam new-name2
                ((build-lam new-name2)
                  ((lam new-name3
                    ((build-lam new-name3)
                      ((build-if
                        ((build-app
                          ((build-app (const eq?))
                            (build-const (const 0))))
                          (((e (take-cond-exp com)) rho)
                            new-name3)))
                        new-name3)
                        ((build-app new-name2)
                          ((build-app
                            ((c (take-while-body com)) rho))
                            new-name3))))))
                        (new-name (const nil))))))
              (new-name (const nil))))
            new-name1)
            ((build-app ((build-app error)
              (error "Illegal command"))) com))))))
    (new-name (const store))))))

```

Figure 12: The Generated Compiling Function \mathcal{C}_c