A Self-Applicable Partial Evaluator for the Lambda Calculus: Correctness and Pragmatics

CARSTEN K. GOMARD
University of Copenhagen

We describe theoretical and a few practical aspects of an implemented self-applicable partial evaluator for the untyped lambda calculus with constants, conditionals, and a fixed point operator.

The purpose of this paper is first to announce the existence of (and to describe) a partial evaluator that is both higher-order and self-applicable; second to describe a surprisingly simple solution to the central problem of binding time analysis, and third to prove that the partial evaluator yields correct answers.

While λ-mix (the name of our system) seems to have been the first higher-order self-applicable partial evaluator to run on a computer, it was developed mainly for research purposes. Two recently developed systems are much more powerful for practical use, but also much more complex: Similix [3, 5] and Schism [7].

Our partial evaluator is surprisingly simple, completely automatic, and has been implemented in a side effect-free subset of Scheme. It has been used to compile, generate compilers and generate a compiler generator.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theories; D.3.4 [**Programming Languages**]: Processors; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic

General Terms: Languages, Theory

Additional Key Words and Phrases: Compiler generation, lambda calculus, partial evaluation, self-application

1. INTRODUCTION

1.1 Overview

This paper describes a self-applicable partial evaluator (the first, to our knowledge) for the untyped lambda calculus with constants, conditionals, and a fixed point operator. It takes a program in the form of a lambda expression

A preliminary version of this paper appeared in the proceedings of the IEEE 1990 International Conference of Computer Languages.

Author's address: DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen Ø, Denmark.

and some but not all of its arguments and yields a *specialized* or *residual* lambda expression, which, if applied to the values of the remaining arguments, evaluates to the same result that the original program would have when applied to all its arguments.[1]

The partial evaluator is surprisingly simple, and has been implemented in a sideeffect-free subset of Scheme. Since it is self-applicable, it can be used both to compile and generate a compiler when given a programming language definition written in the lambda calculus, as for example, a denotational semantics. Both target programs and the generated compiler are in the form of lambda expressions. A simple example is given of the partial evaluator's application to semantics-directed compiler generation; some engineering work would need to be done, however, to apply it to really large language definitions.

As an example of compilation by partial evaluation: given the denotational semantics for a small imperative language called *Tiny* (more details in Section 5) and the *Tiny*-version of the factorial program (Figure 1), our system automatically produces the target program of Figure 2. Note that the residual program is *single-threaded* in its store parameters [33]. This means that they can all be replaced by a single global variable. With the store parameters removed and the access to the store sequentialized, the residual program looks like assembly code. Figure 2 displays a target program obtained by partially evaluating the *Tiny* interpreter with respect to the *Tiny* factorial program of Figure 1; the corresponding generated compiler is given in Section 5.1.

Earlier semantics-directed compiler generators [1, 25, 29, 39] involved numerous program components and intermediate languages. These are obviated in a self-applicable partial evaluator, since self-application allows us to generate compilers from language definitions automatically, and even to generate a compiler generator. In contrast with much other work, our partial evaluator is completely automatic. Target programs, compilers, and the compiler generator are efficient and have natural structures. Furthermore, the correctness of the partial evaluator implies (by definition) that these programs are faithful to the programs from which they were derived.

What is new in this paper. We make three points. First, we report that the open problem of specializing higher-order programs is solved; second, describe a surprisingly simple solution to the central problem of binding time analysis using types; and third, prove that the partial evaluator produces correct residual programs.

Achieving a higher order, self-applicable partial evaluator has been a goal for researchers since the first successful self-application of a first-order partial evaluator [21]. There were three breakthroughs in the spring of 1989, using three significantly distinct methods. This paper describes the simplest and easiest to understand of the three: a self-applicable partial evaluator for the lambda calculus. The two other systems treat Scheme programs and

[1] In other words, a partial evaluator is a (nontrivial) implementation of Kleene's S_n^m theorem.

```
variables: result x;
result := 1;
x      := 6;
while x
      { result := result * x;
        x      := x - 1 }
```

Fig. 1. Source *Tiny* factorial program.

```
((fix (λ fac (λ store-1
            (if ((=? 0) ((access 1) store-1))
                store-1
              (fac ((λ store-3
                          (((update 1)
                            ((- ((access 1) store-3)) 1))
                           store-3))
                    ((λ store-2
                          (((update 0)
                            ((* ((access 0) store-2)) ((access 1) store-2)))
                           store-2))
                     store-1)))))))
      (((update 1) 6) (((update 0) 1) i-store)))
```

Fig. 2. Factorial residual program—the *Tiny* interpreter has been specialized with respect to Factorial.

extend existing self-applicable partial evaluators. One is based on collecting closures at their site of application [3], and the other represents closures as partially static structures [7].

The initial pragmatic success was reported in the conference paper [20], which this paper extends with a correctness proof. More examples and discussion may be found in [16].

Primary goals of this work. Denotational semantics was intended to be a mathematical description of *what* programs do rather than *how* they do it, but since the lambda calculus may be implemented on a machine, a denotational semantics is also an executable specification. It is, however, not a very efficient one, and this led us to consider partial evaluation of the lambda calculus. A denotational semantics for a programming language may be regarded as an interpreter for that language written in the lambda calculus. This approach to language implementation has some severe sources of inefficiency: for example, each time a construct in the interpreted program is examined by the interpreter, syntactic dispatch and various bookkeeping tasks (lookups in environments, symbol-tables) are performed by the interpreter. These costs are usually referred to as *interpretational overhead*, and the standard solution is to compile programs instead of interpreting them.

Partial evaluation of a denotational description with respect to a concrete program can yield an efficient lambda expression to compute the program's meaning. Partial evaluation can thus automatically compile programs to the lambda calculus, given a denotational semantics. In this paper we investigate partial evaluation of the lambda calculus and its application to compilation of programming languages.

We have been able to *self-apply* a partial evaluator for the lambda calculus. This makes it possible not only to compile from an interpretive language specification, but also to generate standalone compilers from language definitions, and even a standalone compiler generator. Since we have a correctness proof for our partial evaluator, the generated target programs and compilers are known to be faithful to the source programs and language specifications from which they were derived.

We should also mention that, though this paper focuses on the application of partial evaluation to compilation and compiler generation, there are other interesting applications. An example is the generation of an efficient pattern matcher from a naive hand-written algorithm [9].

What is not done in this article. Our system does not yield production-quality compilers or target code near machine level. As in SIS [25], the target code consists of lambda expressions (these could, however, be translated further into machine code, as in [29, 30]). Also, we have not yet tried to implement a large programming language from a formal definition. Among many relevant optimizations to apply to the target lambda expressions is the replacement of function parameters by global variables [33, 35].

Binding time analysis and program annotation. As in all previous self-applicable partial evaluators, an essential component is the use of a *binding time analysis* of the program to be specialized. Its effect is to *annotate* the program by marking as "eliminable" those parts which may be computed during partial evaluation and by marking the remaining parts as "residual." Partial evaluation then proceeds by nonstandard execution. The nonresidual part of the program is interpreted as usual (by beta-reducing applications, unfolding fixed point operators, etc.), whereas the residual applications and abstractions are regarded as base functions whose effect is to generate code (lambda expressions) that is to appear in the residual program.

A nonresidual application has the form operator@operand, and is executed at *partial evaluation time*. A residual application is of form operator$\underline{@}$operand, and this application is *not* executed at partial evaluation time. Instead, code is generated for the operator and the operand. Calling the resulting code pieces operator-code and operand-code, respectively, the result of partially evaluating the residual application operator$\underline{@}$operand will be an application operator-code@operand-code, to appear in the residual program. One could thus regard $\underline{@}$ as a base function with type *Code* × *Code* → *Code*.

For this scheme to succeed, the annotations must be clearly placed in a consistent fashion. It does not, for instance, make sense to have the partial evaluator apply a piece of residual code as if it were a function. In Section 3 we set up a *type system* to ensure error freedom at partial evaluation time. A program that is well typed according to this type system is called *well annotated*. To paraphrase Milner, well-annotated programs cannot cause the partial evaluator to "go wrong." The effect of the type system is to clarify the boundary between compile-time and run-time types in partial evaluation, and the task of binding time analysis is to determine this boundary before partial evaluation begins.

Only static computations must be well typed. Using the partial evaluator to compile from an interpreter requires the interpreter's static computations to be well typed, but its dynamic computations need not be. This makes it possible to partially evaluate interpreters for both strongly typed and untyped languages, and still guarantee absence of type errors during compilation, without the need for "type tags." Since the target program is a lambda expression, run-time type security for strongly typed languages can be obtained by type checking as usual.

We use a two-level lambda calculus which differs in several respects from that of [28]. First, we use the second level (our "residual," their "run-time") for code generation, not execution. Second, we do not require programs to be strongly typed, and so definitions of untyped languages can be handled easily. Third, our version has led naturally to building a self-applicable partial evaluator.

The task of binding time analysis (BTA) is the following: given program p and information as to which arguments will be available during partial evaluation, annotate enough parts of p as residual so the result is well annotated. Binding time analysis is thus a sort of type inference, in which type conflicts are resolved not by reporting type errors but by *changing the conflicting parts*, by relabelling them as residual; that is, to be evaluated at run time. This paper only states *what* BTA must do; *how* it is actually done is described in [15].

For efficiency, as few parts of p as possible should be marked as residual, consistent with the well typedness of its annotated version. The annotations also have other uses, which are important but not discussed in this paper—to avoid generating infinitely large residual programs or duplicating code, that is, generating multiple copies of the same target operations.

1.2 Lambda Calculus

A very simple language, the classical lambda calculus, is used here in order to achieve simplicity and allow a more complete description than would be possible for a larger and more practical language. One intention is to give useful background for reports on the more advanced, complex and practically oriented higher-order partial evaluators recently developed by Bondorf, Danvy, and Consel [3, 5, 7].

A lambda calculus program is an *expression*, exp, together with an initial *environment*, ρ, which is a function from identifiers to values. The program takes its input through its free variables whose values are supplied by the environment. The initial environment is also expected to map base function names, such as cons, to the corresponding functions. The expression abstract syntax is given below and the semantic function $\mathscr{E}$ is defined in Figure 3.

Semantic Domains

$$\begin{aligned} \mathit{Val} &= \mathit{Const} + \mathit{Funval} \\ \mathit{Funval} &= \mathit{Val} \rightarrow \mathit{Val} \\ \mathit{Env} &= \mathit{Var} \rightarrow \mathit{Val} \end{aligned}$$

$$\begin{aligned} &\mathcal{E}\text{: } \mathit{Expression} \rightarrow \mathit{Env} \rightarrow \mathit{Val} \\ &\mathcal{E}[\![\mathsf{var}]\!]\rho &&= \rho(\mathsf{var}) \\ &\mathcal{E}[\![\lambda\mathsf{var}.\,\mathsf{exp}]\!]\rho &&= \lambda \mathit{value}.(\mathcal{E}[\![\mathsf{exp}]\!]\rho[\mathsf{var} \mapsto \mathit{value}])\uparrow \mathit{Funval} \\ &\mathcal{E}[\![\mathsf{exp}_1 \text{ @ } \mathsf{exp}_2]\!]\rho &&= (\mathcal{E}[\![\mathsf{exp}_1]\!]\rho\downarrow \mathit{Funval})\ (\mathcal{E}[\![\mathsf{exp}_2]\!]\rho) \\ &\mathcal{E}[\![\mathsf{fix}\ \mathsf{exp}]\!]\rho &&= \mathit{fix}\ (\mathcal{E}[\![\mathsf{exp}]\!]\rho\downarrow \mathit{Funval}) \\ &\mathcal{E}[\![\mathsf{if}\ \mathsf{exp}_1\ \mathsf{exp}_2\ \mathsf{exp}_3]\!]\rho &&= (\mathcal{E}[\![\mathsf{exp}_1]\!]\rho\downarrow \mathit{Const}) \rightarrow \mathcal{E}[\![\mathsf{exp}_2]\!]\rho,\ \mathcal{E}[\![\mathsf{exp}_3]\!]\rho \\ &\mathcal{E}[\![\mathsf{const}\ \mathsf{c}]\!]\rho &&= \mathsf{c}\uparrow \mathit{Const} \end{aligned}$$

Fig. 3. Lambda calculus semantics.

Below we describe the notational conventions used in Figure 3 and in the remainder this paper.

exp ::= var | exp@exp | λvar.exp | if exp exp exp | fix exp | const constant

Notation. $\mathit{Val} = \mathit{Const} + \mathit{Funval}$ constructs the separated sum of domains *Const* and *Funval*. Given an element $b \in \mathit{Const}$, $v = b\uparrow \mathit{Const} \in \mathit{Val}$ is tagged as originating from *Const*. Conversely, if v is tagged as originating from *Const*, $v\downarrow \mathit{Const}$ strips off the tag yielding an element in *Const*; if v has any other tag, $v\downarrow \mathit{Const}$ produces an error. (We assume that all operations are strict in the error value but omit details.) $\mathit{Funval} = \mathit{Val} \rightarrow \mathit{Val}$ constructs the Scott domain of partial functions from *Val* to *Val*. The notation $\rho[\mathsf{var} \mapsto \mathit{value}]$ is a shorthand for λx.if (x = var) *value* (ρ x) and is used to update environments. $v_1 \rightarrow v_2, v_3$ has the value v_2 if v_1 equals *true* and value v_3 if v_1 equals *false*.

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

Informal semantics. The value of a variable var is found by applying the environment ρ. The value of an abstraction λvar.exp is a function which when given the argument value, *value*, evaluates the body of the function in an extended environment. The value of an application $\mathsf{exp}_1@\mathsf{exp}_2$ is found by applying the value of exp_1, which must be a function, to the value of exp_2. The value of if exp_1 exp_2 exp_3 is the value of either exp_2 or exp_3 depending on whether exp_1 is *true* or *false*. The value of fix exp is the least fixpoint of the function(al) to which exp evaluates. The value of const c is just c with a *Const* tag. We have identifiers denoting base functions such as cons mapped to the corresponding functions in the initial environment, and we could have assigned meaning to first-order constants in the same way. We have, however, found it useful in practice to have an explicit const operator.

For example, consider the exponential function computing x to the nth where x and n are free (input) variables. (In the concrete syntax, we omit some of the explicit application nodes, e.g., for testing whether n′ equals zero).

```
(fix λp.λn′.λx′.if (= n′ 0) 1 (*x′(p@(− n′ 1)@x′)))@n@x
```

Note that the semantic function $\mathscr{E}$ in Figure 3 may be regarded as a self-interpreter for the lambda calculus, since it is easily written as a lambda expression of form fix $\lambda\mathscr{E}$.λexp.λρ.if . . . .

1.3 Partial Evaluation and the Futamura Projections

Following earlier papers on partial evaluation (e.g., [22]), we take **L** *power* [*2*, *3*] to denote the result of running the **L**-program *power* on its two input data, *2* and *3*. The idea is to consider a language as mapping a program text into the function it computes, which has proven valuable in this framework where programs are data objects. Here **L** is the lambda calculus. Thus it holds that **L** *power* [*2*, *3*] = *9*. Given an **L**-program p, a *residual* program for p with respect to partial data $d1$ is a program p_{d1} such that

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

Suppose p is the power program shown above and $d1$ is n = *2*, then a residual program p_2 is

```
(*x(*x 1))
```

with free variable x. The knowledge that n = *2* is incorporated in the residual program.

A *partial evaluator* is a program, which we call *mix*, that has the property that when given p and the partial data $d1$, produces the residual program p_{d1}. This is captured by the *mix equation*:

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

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 **S** *pgm data* = **L** *int*[*pgm*, *data*] and an **S**-to-**T**-compiler *comp* written in **L** is a program that fulfills **S** *pgm data* = **T**(**L** *comp pgm*)*data*.

The *Futamura projections* [13, 10] state that given a partial evaluator *mix* and an interpreter *int* it is possible to compile programs, and even to generate standalone compilers and compiler generators by self-applying *mix*. The three Futamura projections are

L *mix*[*int*, *pgm*] = *target*
L *mix*[*mix*, *int*] = *compiler*
L *mix*[*mix*, *mix*] = *compiler generator* or for short: *cogen*

Program *target* is a specialized version of **L**-program *int*, and so is 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:

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

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

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

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

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

The equations are easily verified using the definitions of interpreters, compilers, and the mix equation. These proofs and a more detailed discussion are presented by Jones et al. [22]. Jones et al. reported their first nontrivial computer realization in [21], where **L** was a first-order language of recursive equations. The system we describe here realizes all three Futamura projections with unlimited use of higher-order functions.

2. PARTIAL EVALUATION USING A 2-LEVEL LAMBDA CALCULUS

The result of applying the BTA (binding time analysis) to an expression is an *annotated* expression where the parts that are *not* to be evaluated at partial evaluation time are marked. We achieve this with a two-level lambda calculus, similar in syntax but different in semantics from that presented by Nielson [26]. In the second phase we blindly obey the annotations: we reduce all redexes not marked as residual and generate residual target code (also a lambda expression) for the marked operations. The focus of this section is the syntax and semantics of our two-level lambda calculus.

The two-level lambda calculus contains two versions of each operator in the ordinary lambda calculus: for each of the "normal" operators: $\lambda, @, \ldots$ there is also a residual version: $\underline{\lambda}, \underline{@}, \ldots$ in the two-level calculus. The abstract syntax of two-level expressions is the follov ing.

$$\begin{aligned} \mathsf{texp} ::= {} & \mathsf{texp@texp} \mid \lambda\mathsf{var.texp} \mid \mathsf{if\ texp\ texp\ texp} \mid \mathsf{fix\ texp} \mid \mathsf{const\ constant} \mid \\ & \mathsf{texp}\underline{@}\mathsf{texp} \mid \underline{\lambda}\mathsf{var.texp} \mid \underline{\mathsf{if}}\ \mathsf{texp\ texp\ texp} \mid \underline{\mathsf{fix}}\ \mathsf{texp} \mid \underline{\mathsf{const}}\ \mathsf{constant} \mid \\ & \mathsf{var} \mid \overline{\mathsf{lift}}\ \mathsf{texp} \end{aligned}$$

Intuitively, for all operators $\lambda, @, \ldots$ the denotations given by the semantic function $\mathscr{T}$ in Figure 4 are the same as they would be for first-level semantic function, and the operators: $\underline{\lambda}, \underline{@}, \ldots$ are suspended, yielding as result a

Semantic Domains

$$\begin{array}{lcl}
\mathit{2Val} & = & \mathit{Const} + \mathit{2Funval} + \mathit{Code} \\
\mathit{2Funval} & = & \mathit{2Val} \rightarrow \mathit{2Val} \\
\mathit{Code} & = & \mathit{Expression} \\
\mathit{2Env} & = & \mathit{Var} \rightarrow \mathit{2Val}
\end{array}$$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{T}\colon \mathit{2Expression} \rightarrow \mathit{2Env} \rightarrow \mathit{2Val}} \\
\mathcal{T}[\![\mathsf{var}]\!]\rho & = & \rho(\mathsf{var}) \\
\\
\mathcal{T}[\![\lambda \mathsf{var}.\, \mathsf{texp}]\!]\rho & = & \lambda \mathit{value}.(\mathcal{T}[\![\mathsf{texp}]\!]\, \rho[\mathsf{var} \mapsto \mathit{value}])\!\uparrow\! \mathit{2Funval} \\
\mathcal{T}[\![\mathsf{texp}_1 \ @\ \mathsf{texp}_2]\!]\rho & = & \mathcal{T}[\![\mathsf{texp}_1]\!]\rho\!\downarrow\! \mathit{2Funval}\ (\mathcal{T}[\![\mathsf{texp}_2]\!]\rho) \\
\mathcal{T}[\![\mathsf{fix}\ \mathsf{texp}]\!]\rho & = & \mathit{fix}\ (\mathcal{T}[\![\mathsf{texp}]\!]\rho\!\downarrow\! \mathit{2Funval}) \\
\mathcal{T}[\![\mathsf{if}\ \mathsf{texp}_1\ \mathsf{texp}_2\ \mathsf{texp}_3]\!]\rho & = & \mathcal{T}[\![\mathsf{texp}_1]\!]\rho\!\downarrow\! \mathit{Const} \rightarrow \mathcal{T}[\![\mathsf{texp}_2]\!]\rho,\ \mathcal{T}[\![\mathsf{texp}_3]\!]\rho \\
\mathcal{T}[\![\mathsf{const}\ \mathsf{c}]\!]\rho & = & \mathsf{c}\!\uparrow\! \mathit{Const} \\
\mathcal{T}[\![\mathsf{lift}\ \mathsf{texp}]\!]\rho & = & \mathit{build\text{-}const}(\mathcal{T}[\![\mathsf{texp}]\!]\rho\!\downarrow\! \mathit{Const})\!\uparrow\! \mathit{Code} \\
\\
\mathcal{T}[\![\underline{\lambda} \mathsf{var}.\, \mathsf{texp}]\!]\rho & = & \text{let } \mathit{nvar} = \mathit{newname}(\mathit{var}) \\
 & & \quad \text{in } \ \mathit{build\text{-}\lambda}(\mathrm{nvar},\ \mathcal{T}[\![\mathsf{texp}]\!]\, \rho[\mathsf{var} \mapsto \mathit{nvar}]\!\downarrow\! \mathit{Code})\!\uparrow\! \mathit{Code} \\
\mathcal{T}[\![\mathsf{texp}_1 \ \underline{@}\ \mathsf{texp}_2]\!]\rho & = & \mathit{build\text{-}@}(\mathcal{T}[\![\mathsf{texp}_1]\!]\rho\!\downarrow\! \mathit{Code},\ \mathcal{T}[\![\mathsf{texp}_2]\!]\rho\!\downarrow\! \mathit{Code})\!\uparrow\! \mathit{Code} \\
\mathcal{T}[\![\underline{\mathsf{fix}}\ \mathsf{texp}]\!]\rho & = & \mathit{build\text{-}fix}(\mathcal{T}[\![\mathsf{texp}]\!]\rho\!\downarrow\! \mathit{Code})\!\uparrow\! \mathit{Code} \\
\mathcal{T}[\![\underline{\mathsf{if}}\ \mathsf{texp}_1\ \mathsf{texp}_2\ \mathsf{texp}_3]\!]\rho & = & \mathit{build\text{-}if}(\mathcal{T}[\![\mathsf{texp}_1]\!]\rho\!\downarrow\! \mathit{Code}, \\
 & & \qquad\qquad \mathcal{T}[\![\mathsf{texp}_2]\!]\rho\!\downarrow\! \mathit{Code}, \\
 & & \qquad\qquad \mathcal{T}[\![\mathsf{texp}_3]\!]\rho\!\downarrow\! \mathit{Code})\!\uparrow\! \mathit{Code} \\
\mathcal{T}[\![\underline{\mathsf{const}}\ \mathsf{c}]\!]\rho & = & \mathit{build\text{-}const}(\mathrm{c})\!\uparrow\! \mathit{Code}
\end{array}$$

Fig. 4. 2-level lambda calculus semantics.

piece of *code* for execution at run time. The lift operator builds a constant expression with the same value as lift's argument. lift is used when a residual expression has a subexpression with a constant value.[2]

A two-level *program* is a second-level expression texp together with an initial mix-time environment ρ_s, which maps the free variables of texp to constants, functions, or code pieces. We assume that free variables whose values are not available at partial evaluation time are mapped to distinct, new variable names. The $\mathcal{T}$-rules then ensure that these new variables become the free variables of the residual program. Note that variables bound by $\underline{\lambda}$, will also (eventually) be bound to fresh variable names, whereas variables bound by λ can be bound to all kinds of values: constants, functions, or (all kinds of) code pieces.

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

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

[2] The introduction of lift renders $\underline{\text{const}}$ superfluous since lift(const c) = $\underline{\text{const}}$ c, but we keep $\underline{\text{const}}$ in the language for symmetry and as a shorthand.

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 texp_1-code@texp_2-code). All the *build*-functions are strict.

The projections check that both operator and operand reduce to code pieces, since it does not make sense to, for example, glue 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 nonresidual λ, and in the rule for $\underline{\lambda}$, the formal parameter is bound to a fresh variable name (which we assume is available whenever needed):

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

Each occurrence of var in texp will then be looked up in ρ, causing var to be replaced by some var_{new}. Since $\underline{\lambda}\text{var.texp}$ 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 sideeffect 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.

The valuation functions for two-level lambda calculus programs are given in Figure 4. The rules contain explicit type checks; Section 3 discusses sufficient criteria for omitting these.

The function *build-@* has type $\mathit{Code} \times \mathit{Code} \to \mathit{Code}$ and is purely syntactic: it builds a residual first-level expression, which is the application of its two arguments. The other *build*-functions have analogous meanings.

3. PROGRAM ANNOTATION

An *annotated* lambda expression exp^{ann} is a two-level expression obtained by replacing some occurrences of @, $\lambda, \ldots$ in exp by the corresponding marked operator: $\underline{@}, \underline{\lambda}, \ldots$. Clearly, the annotations have to be placed consistently so that a summand error is not produced in the rules in Figure 4. Below, we introduce a type system to define *well-annotatedness*, which ensures error freedom. Hence, if exp^{ann} is well annotated, the type checks are superfluous and can be omitted from the partial evaluation algorithm. Our algorithm proceeds like this:

(1) Given an expression exp apply BTA, yielding a well-annotated exp^{ann}

(2) Apply the mix program to exp^{ann} and ρ_s. The mix program is the $\mathcal{T}$-rules written as a lambda expression *without* the type checks. ρ_s maps the free variables of exp^{ann} to either constants, functions, or code pieces.

To define well-annotatedness, we introduce a type system for two-level expressions. The abstract syntax for a two-level type is

$$\textit{type} ::= \textit{const} \mid \textit{type} \rightarrow \textit{type} \mid \textit{code}$$

We take the assertion form $\tau \vdash \text{texp}: \textit{type}$ to mean that when given type assumptions τ on the free variables of texp, the two-level expression texp is well annotated and of type *type*. 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) $t = \textit{const}$ and $v = \text{ct}\uparrow \textit{Const}$ for some ct.
(2) $t = \textit{code}$ and $v = \text{cd}\uparrow \textit{Code}$ for some cd.
(3) (a) $t = t_1 \rightarrow t_2$, $v = \text{f}\uparrow \textit{2Funval}$ for some f, and
(b) $\forall w \in \textit{2Val}$: t_1 suits w implies t_2 suits $\text{f}(w)$.

A type environment τ suits an environment ρ if, for all variables x bound by ρ, $\tau(\text{x})$ suits $\rho(\text{x})$.

The type rules of Figure 5 express the type checking part of the second-level semantics. It is easy to verify that if, for some type t, $\tau \vdash \text{texp}: t$ and τ suits ρ, then t suits $\mathcal{T}[\![\text{texp}]\!]\rho$, and no type error is produced. Note that there do exist two-level expressions texp that are not well annotated, where $\mathcal{T}[\![\text{texp}]\!]\rho$ does not yield an error. The obvious analogy is that in an untyped language such as Scheme there exist perfectly legal programs that would not pass a type system. We are only interested in well-annotated programs. Note also that type *unicity* does not hold: the expression $\lambda\text{x.x}$ is well annotated of type $t \rightarrow t$ for any t.

The type rules for the nonresidual operators are the standard ones. For the residual operators, the rules say that if the subexpressions have type *code*, so has the expression itself. All the rules are derived from the way function $\mathcal{T}$ tests the tags in Figure 4.

The task of BTA. Given a type assumption τ for the free variables of exp, the task of BTA is to annotate it such that $\tau \vdash \text{texp}: \textit{code}$. This is done by marking some parts as residual and inserting lift-operators where necessary. (The lift-operator converts an expression of type *const* into an expression of type *code*, so that the expression appears in the residual program.) This is in fact always possible by making *all* operators residual and inserting *lift*-operators around all free variables of type *const*. Such an annotation is of course not desirable, since partial evaluation would just return the original exp with no computation done at all.

$\mathcal{T}$ can "go wrong" in other ways than by committing type errors. If $\mathcal{T}$ reduces too many redexes, reduction might proceed infinitely or residual code might be duplicated. To avoid this, some redexes should be left in the

$$\frac{\tau[\mathsf{x} \mapsto T_2] \vdash \mathbf{texp}\colon T_1}{\tau \vdash \boldsymbol{\lambda}\mathbf{x}.\,\mathbf{texp}\colon T_2 \to T_1}$$

$$\frac{\tau \vdash \mathsf{texp}\colon (T_1 \to T_2) \to (T_1 \to T_2)}{\tau \vdash \mathsf{fix\ texp}\colon T_1 \to T_2}$$

$$\tau \vdash \mathsf{const\ c}\colon \mathit{const}$$

$$\frac{\tau \vdash \mathsf{texp_1}\colon T_2 \to T_1, \quad \tau \vdash \mathsf{texp_2}\colon T_2}{\tau \vdash \mathsf{texp_1}\ @\ \mathsf{texp_2}\colon T_1}$$

$$\frac{\tau \vdash \mathsf{texp_1}\colon \mathit{const}, \quad \tau \vdash \mathsf{texp_2}\colon T, \quad \tau \vdash \mathsf{texp_3}\colon T}{\tau \vdash \mathsf{if\ texp_1\ texp_2\ texp_3}\colon T}$$

$$\frac{\tau[\mathsf{x} \mapsto \mathit{code}] \vdash \mathsf{texp}\colon \mathit{code}}{\tau \vdash \underline{\lambda}\mathsf{x}.\,\mathsf{texp}\colon \mathit{code}}$$

$$\frac{\tau \vdash \mathsf{texp}\colon \mathit{code}}{\tau \vdash \underline{\mathsf{fix}}\ \mathsf{texp}\colon \mathit{code}}$$

$$\tau \vdash \underline{\mathsf{const}}\ \mathsf{c}\colon \mathit{code}$$

$$\frac{\tau \vdash \mathsf{texp_1}\colon \mathit{code}, \quad \tau \vdash \mathsf{texp_2}\colon \mathit{code}}{\tau \vdash \mathsf{texp_1}\ \underline{@}\ \mathsf{texp_2}\colon \mathit{code}}$$

$$\frac{\tau \vdash \mathsf{texp_1}\colon \mathit{code}, \quad \tau \vdash \mathsf{texp_2}\colon \mathit{code}, \quad \tau \vdash \mathsf{texp_3}\colon \mathit{code}}{\tau \vdash \underline{\mathsf{if}}\ \mathsf{texp_1\ texp_2\ texp_3}\colon \mathit{code}}$$

$$\frac{\tau(\mathsf{x}) = T}{\tau \vdash \mathsf{x}\colon T}$$

$$\frac{\tau \vdash \mathbf{texp}\colon \mathit{const}}{\tau \vdash \mathbf{lift\ texp}\colon \mathit{code}}$$

Fig. 5. Typ ules checking well-annotatedness.

residual program, and it is the job of the BTA to decide which ones. A BTA algorithm would proceed like this:

(1) Given exp with initial type assumptions τ, annotate exp yielding exp^{ann} such that $\tau \vdash \mathsf{exp}^{ann}$: *code.*

(2) Apply finiteness and code duplication analysis. If this step adds any annotations, go back to step 1.

Part 1 of this procedure has been implemented using a modified version of Milner's algorithm W and is described by Gomard [15].

4. CORRECTNESS PROOF FOR MIX

This section is devoted to the formulation and complete proof of a correctness theorem for our partial evaluator. The existence of a correctness theorem guarantees that the mix-generated target programs, compilers (etc.), are all faithful to their specifications. We do not know of any other correctness proof for a self-applicable partial evaluator. (The paper [11] gives a correctness proof of an extremely limited partial evaluation scheme.)

For readability, we consequently omit domain injections and projections in this section. (The equations are hard enough to read without them.) The *annotation-forgetting* function ϕ: *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.

For readers who are not interested in technical details, we now sketch the correctness result before we formally state and prove the "real" theorem. Suppose we are given

(1) a two-level expression texp;
(2) an environment ρ mapping the free variables of texp to values;
(3) an environment ρ_s, mapping the free variables of texp to their mix-time values (constants, functions, or fresh variable names),
(4) an environment ρ_d, mapping these fresh variables to values; and
(5) $\tau \vdash$ texp: *code* if τ suits ρ_s.

Suppose furthermore that for variables x of type *const*: $\rho_s(\mathsf{x}) = \rho(\mathsf{x})$, and for variables y of type *code*: $\rho_d(\rho_s(\mathsf{y})) = \rho(\mathsf{y})$, and that base functions (and other higher-order values) bound in the environments are handled "correctly" (the formalization of this is in Definitions 1 and 2). It then holds that if both $\mathscr{E}[\![\mathscr{T}[\![\mathsf{texp}]\!]\rho_s]\!]\rho_d$ and $\mathscr{E}[\![\phi(\mathsf{texp})]\!]\rho$ are defined, then

$$\mathscr{E}[\![\mathscr{T}[\![\mathsf{texp}]\!]\rho_s]\!]\rho_d = \mathscr{E}[\![\phi(\mathsf{texp})]\!]\rho$$

Thus, what we prove is that our partial evaluator fulfills the mix-equation as stated in Section 1.3.

Preservation of termination properties. All nontrivial partial evaluators so far have had problems with the termination properties of the partial evaluator itself or with the generated residual programs. This partial evaluator is no exception. Consider again the equation

$$\mathscr{E}[\![\mathscr{T}[\![\mathsf{texp}]\!]\rho_s]\!]\rho_d = \mathscr{E}[\![\phi(\mathsf{texp})]\!]\rho$$

There may be two reasons why one side is defined while the other is not.

(1) If a call-by-value strategy is used, then the right side may be undefined while the left side is defined. This is due to the inherent call-by-name nature of partial evaluation. Suppose we have

    (λx.const 2)@bomb

where bomb is a nonterminating expression *made residual*, thus trivially terminating *at partial evaluation time*. Clearly, partial evaluation will discard the bomb and, clearly, evaluation of ϕ((λx.const 2)@bomb) will loop. What has been said elsewhere in the paper does not rely on any specific evaluation strategy, but the correctness result does rely on our lambda calculus being nonstrict. For a strict language, a weaker result holds: *if* both sides are defined, they are equal. For techniques to avoid these problems, see Bondorf and Danvy's paper [5].

(2) Since mix treats *both* branches of *residual* conditional expressions, it is easy to construct an example where $\mathcal{T}$ loops on texp where normal evaluation of ϕ(texp) would terminate. When proving $\mathcal{E}[\![\mathcal{T}[\![\text{texp}]\!]\rho_s]\!]\rho_d = \mathcal{E}[\![\phi(\text{texp})]\!]\rho$, we assume that $\mathcal{T}$ is well defined on all subexpressions of texp. It is always possible to annotate texp such that this condition is fulfilled—a step towards constructing a binding time analyzer that does this is taken in [19]. If we lift this restriction our correctness result will be weakened to: *if* both sides are defined, they are equal.

The rest of the section is devoted to the formalization and proof of the correctness result outlined above. Readers not interested in this may skip to the next section without loss of continuity.

The relation $\mathcal{R}$ to be defined below (*Definition* 1) is vital to the correctness proof. Intuitively, the relation $\mathcal{R}$ expresses that the function $\mathcal{T}$ handles a given two-level expression texp correctly. For an expression of type *const*, let the initial environment ρ be split into a mix-time part ρ_s and a run-time part ρ_d. Relation $\mathcal{R}$ implies that the result of partial evaluation must be the right answer:

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

expressing that if texp has type *const*, then normal evaluation of the unannotated expression yields the same result as partial evaluation of the annotated expression. For an expression of type *code*, relation $\mathcal{R}$ implies that the result of partial evaluation must be an expression, the residual program, which when evaluated yields the right answer:

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

For expressions of a function type, $\mathcal{R}$ expresses that the result of applying the function to a proper argument yields a proper answer.

Definition 1. The relation $\mathcal{R}$ holds for $(\text{texp}, \rho_s, \rho_d, \rho, t) \in 2Exp \times 2Env \times Env \times Env \times Type$ iff

(1) $\tau \vdash \text{texp}: t$ if τ suits ρ_s,
(2) One of the following holds:
  (a) texp has type *const* and $\mathcal{T}[\![\text{texp}]\!]\rho_s = \mathcal{E}[\![\phi(\text{texp})]\!]\rho$
  (b) texp has type *code* and $\mathcal{E}[\![\mathcal{T}[\![\text{texp}]\!]\rho_s]\!]\rho_d = \mathcal{E}[\![\phi(\text{texp})]\!]\rho$
  (c) texp has type $t = t_1 \to t_2$, and
  $\forall \text{texp}_1$: $\mathcal{R}(\text{texp}_1, \rho_s, \rho_d, \rho, t_1)$ implies $\mathcal{R}(\text{texp@texp}_1, \rho_s, \rho_d, \rho, t_2)$

Note that the recursive definition of $\mathcal{R}$ has finite depth, since in the definition of $\mathcal{R}(\text{texp}, \rho_s, \rho_d, \rho, t)$ the recursive applications of $\mathcal{R}$ concern tuples $(\text{texp}', \rho'_s, \rho'_d, \rho', t')$ where t' has fewer type constructors than t.

Since an expression may have free variables, the environments involved ρ_s, ρ_d, ρ must in some sense be well behaved. It turns out that the condition on the environments can also be formulated in terms of $\mathcal{R}$.

Definition 2. Given a set of identifiers, VarSet, and three environments, ρ, ρ_s, ρ_d and a type environment τ that suits ρ_s, we say that ρ_s, ρ_d, ρ agree on VarSet iff $\forall \text{var} \in \text{Varset}$: $\mathcal{R}(\text{var}, \rho_s, \rho_d, \rho, \tau(\text{var}))$.

Suppose ρ_s, ρ_d, ρ agree on VarSet. This means that for all variables of type const: $\rho_s(\mathsf{x}) = \rho(\mathsf{x})$, for variables y of type *code*: $\rho_d(\rho_s(\mathsf{y})) = \rho(\mathsf{y})$. For a higher-order example, suppose ρ_s maps identifier car to a function of type *const* $\rightarrow$ *const*. By expanding the definitions of "agreement" and $\mathscr{R}$, we find that $\forall \mathsf{texp}_1$: $\mathscr{T}[\![\mathsf{texp}_1]\!]\rho_s = \mathscr{E}[\![\mathsf{texp}_1]\!]\rho$ implies $\mathscr{T}[\![\mathsf{car@texp}_1]\!]\rho_s = \mathscr{E}[\![\mathsf{car@texp}_1]\!]\rho$.

THEOREM 3. (*Main Correctness Theorem*) *For all two-level expressions* texp *where* $\mathscr{T}$ *is defined on all subexpressions of* texp, *it holds that* $\forall \rho_s, \rho_d, \rho, \tau$. *The following three conditions*

(1) τ *suits* ρ_s,
(2) ρ_s, ρ_d, ρ *agree on FreeVars*(texp), *and*
(3) $\tau \vdash$ texp: t *for some type* t

imply that $\mathscr{R}(\mathsf{texp}, \rho_s, \rho_d, \rho, t)$ *also holds.*

PROOF. The proof proceeds by induction on the structure of texp. The proofs for the various cases are found in Lemmas 6 to 19. □

COROLLARY 4. *Assume* texp, $\rho_s, \rho_d, \rho, \tau$ *given such that* τ *suits* ρ_s, *and* ρ_s, ρ_d, ρ *agree on FreeVars*(texp).

(1) *If* $\tau \vdash$ texp: *const, then* $\mathscr{T}[\![\mathsf{texp}]\!]\rho_s = \mathscr{E}[\![\phi(\mathsf{texp})]\!]\rho$
(2) *If* $\tau \vdash$ texp: *code, then* $\mathscr{E}[\![\mathscr{T}[\![\mathsf{texp}]\!]\rho_s]\!]\rho_d = \mathscr{E}[\![\phi(\mathsf{texp})]\!]\rho$

We now introduce a name, $\mathscr{H}$, for the property expressed by Theorem 3. $\mathscr{H}$ is also used as an induction hypothesis in the proof. $\mathscr{H}$ expresses that if environments agree on the free variables of a well-annotated two-level expression, then the relation $\mathscr{R}$ will hold for the expression, the environments, and the type.

Definition 5. Given a two-level expression texp, $\mathscr{H}$(texp) holds if $\forall \rho_s, \rho_d, \rho, \tau$. The following three conditions

(1) τ suits ρ_s,
(2) ρ_s, ρ_d, ρ agree on FreeVars(texp),
(3) $\tau \vdash$ texp: t for some type t,

imply that $\mathscr{R}(\mathsf{texp}, \rho_s, \rho_d, \rho, t)$ also holds.

The proofs of the following lemmas all proceed in the same way. Assume texp, $\rho_s, \rho_d, \rho, \tau$ are given such that the three conditions of Definition 5 are fulfilled. The inductive assumption gives that $\mathscr{H}(\mathsf{texp}')$ for the subexpressions of texp. Except in the case of abstraction, the free variables of texp are exactly those of the largest proper subexpressions of texp. Thus ρ_s, ρ_d, ρ agree on the free variables of these expressions, too (in the case of abstraction, we have to construct some new environments ρ'_s, ρ'_d, ρ'). By well annotatedness of texp, the subexpressions are also well annotated, and the inference rules of Figure 5 give us types for the subexpressions. This gives us some facts of the form $\mathscr{R}$(sub-texp, ρ_s, ρ_d, ρ, t'), which then leads (with more or less trouble) to the goal: $\mathscr{R}$(texp, ρ_s, ρ_d, ρ, t).

LEMMA 6. $\forall$var: $\mathscr{H}$(var).

PROOF. {var} = FreeVars(var) □

LEMMA 7. $\forall$c: $\mathscr{H}$(const c).

PROOF. $\mathscr{T}[\![\text{const c}]\!]\rho_s = \text{c} = \mathscr{E}[\![\phi(\text{const c})]\!]\rho$. □

LEMMA 8. $\forall$c: $\mathscr{H}(\underline{\text{const}}\ \text{c})$.

PROOF. $\mathscr{E}[\![\mathscr{T}[\![\underline{\text{const}}\ \text{c}]\!]\rho_s]\!]\rho_d = \mathscr{E}[\![\text{const c}]\!]\rho_d = \text{c} = \mathscr{E}[\![\phi(\underline{\text{const}}\ \text{c})]\!]\rho$. □

LEMMA 9. $\forall$texp: $\mathscr{H}$(texp) *implies* $\mathscr{H}$(λx.texp).

PROOF. Assume ρ_s, ρ_d, ρ, τ are given, satisfying the conditions in Definition 5. It thus holds that $\tau \vdash \lambda$x.texp: t'' where t'' must have the form $t' \to t$. Assume furthermore that texp' is given such that $\mathscr{R}$(texp', ρ_s, ρ_d, ρ, t'). By alpha conversion of λx.texp, we can assume without loss of generality that x does not occur in any expressions but the subexpressions of λx.texp. Define

$$\begin{aligned} \rho'_s &= \rho_s[\text{x} \mapsto \mathscr{T}[\![\text{texp}']\!]\rho_s] \quad \text{and} \\ \rho' &= \rho[\text{x} \mapsto \mathscr{E}[\![\phi(\text{texp}')]\!]\rho] \quad \text{and} \\ \tau' &= \tau[\text{x} \mapsto t'] \end{aligned}$$

and observe that $\forall$id $\in$ FreeVars(texp): $\mathscr{R}$(id, ρ'_s, ρ_d, ρ', τ'(id)), since $\forall$id $\in$ FreeVars(λx.texp): $\mathscr{R}$(id, ρ_s, ρ_d, ρ, τ(id)) and $\mathscr{R}$(x, ρ'_s, ρ_d, ρ', t') where $\mathscr{R}$(x, ρ'_s,ρ_d, ρ', t') follows from the assumption that $\mathscr{R}$(texp', ρ_s, ρ_d, ρ, t'). We now have that ρ'_s, ρ_d, ρ' agree on FreeVars(texp), that τ' suits ρ'_s (clear), and that $\tau' \vdash$ texp: t. Hence $\mathscr{R}$(texp, ρ'_s, ρ_d, ρ', t) by the induction hypothesis.

We are now close to the desired conclusion: $\mathscr{R}$((λx.texp)@texp, ρ_s, ρ_d, ρ, t). The last step is Observation 10. □

Observation 10. Assume, with the above definitions and assumptions, that $\mathscr{R}$(texp, ρ'_s, ρ_d, ρ', t) holds. Then $\mathscr{R}$((λx.texp)@texp, ρ_s, ρ_d, ρ, t) also holds.

PROOF. The type t must either have form $t_1 \to \cdots \to t_n \to$ *const* or $t_1 \to \cdots \to t_n \to$ *code*. We assume that $t = t_1 \to \cdots \to t_n \to$ *const*. (The opposite assumption leads to a very similar development.) Now $\mathscr{R}$(texp, ρ'_s, ρ_d, ρ', t) may be written: $\forall$texp$_1$, . . . , texp$_n$: ($\forall$i $\in$ [1 . . n]: $\tau \vdash$ texp$_i$: t_i and $\mathscr{R}$(texp$_i$, ρ'_s, ρ_d, ρ', t_i)) implies

$$\begin{aligned} &\mathscr{E}[\![\phi(\text{texp@texp}_1\text{@} \cdots \text{@texp}_n)]\!]\rho' \\ &= \mathscr{T}[\![\text{texp@texp}_1\text{@} \cdots \text{@texp}_n]\!]\rho'_s \end{aligned}$$

where the equation may be rewritten to

$$\begin{aligned} &(\mathscr{E}[\![\phi(\text{texp})]\!]\rho')(\mathscr{E}[\![\phi(\text{texp}_1)]\!]\rho') \ldots (\mathscr{E}[\![\phi(\text{texp}_n)]\!]\rho') \\ &= (\mathscr{T}[\![\text{texp}]\!]\rho'_s)(\mathscr{T}[\![\text{texp}_1]\!]\rho'_s) \ldots (\mathscr{T}[\![\text{texp}_n]\!]\rho'_s) \end{aligned}$$

Since x is not free in texp_i, we may again rewrite to get

$$\begin{aligned}&(\mathscr{E}\llbracket\phi(\mathsf{texp})\rrbracket\rho')(\mathscr{E}\llbracket\phi(\mathsf{texp}_1)\rrbracket\rho)\ldots(\mathscr{E}\llbracket\phi(\mathsf{texp}_n)\rrbracket\rho)\\ &=(\mathscr{T}\llbracket\mathsf{texp}\rrbracket\rho'_s)(\mathscr{T}\llbracket\mathsf{texp}_1\rrbracket\rho_s)\ldots(\mathscr{T}\llbracket\mathsf{texp}_n\rrbracket\rho_s)\end{aligned}$$

Now use the definitions of ρ'_s and ρ' and the @-rules for $\mathscr{T}$ and $\mathscr{E}$, to get

$$\begin{aligned}&(\mathscr{E}\llbracket\phi((\lambda \mathsf{x}.\mathsf{texp})@\mathsf{texp}')\rrbracket\rho)(\mathscr{E}\llbracket\phi(\mathsf{texp}_1)\rrbracket\rho)\ldots(\mathscr{E}\llbracket\phi(\mathsf{texp}_n)\rrbracket\rho)\\ &=(\mathscr{T}\llbracket(\lambda \mathsf{x}.\mathsf{texp})@\mathsf{texp}'\rrbracket\rho_s)(\mathscr{T}\llbracket\mathsf{texp}_1\rrbracket\rho_s)\ldots(\mathscr{T}\llbracket\mathsf{texp}_n\rrbracket\rho_s)\end{aligned}$$

More uses of the @-rules yield

$$\begin{aligned}&\mathscr{E}\llbracket\phi((\lambda \mathsf{x}.\mathsf{texp})@\mathsf{texp}'@\mathsf{texp}_1@\cdots@\mathsf{texp}_n)\rrbracket\mathscr{R}_\iota\\ \mathscr{T}\llbracket\ &\mathscr{L}\mathsf{x}.\mathsf{texp})@\mathsf{texp}'@\mathsf{texp}_1@\cdots@\mathsf{texp}_n\rrbracket\rho\end{aligned}$$

Now step back and see that the property that we want to establish, $\mathscr{R}((\lambda\mathsf{x}.\mathsf{texp})@\mathsf{texp}', \rho_s, \rho_d, \rho, t)$, may be written: $\forall \mathsf{texp}_1, \ldots, \mathsf{texp}_n$: $(\forall \mathsf{i} \in [1..\mathrm{n}]$: $\tau \vdash \mathsf{texp}_i$: t_i and $\mathscr{R}(\mathsf{texp}_i, \rho_s, \rho_d, \rho, t_\iota))$ implies

$$\begin{aligned}&\mathscr{E}\llbracket\phi((\lambda \mathsf{x}.\mathsf{texp})@\mathsf{texp}'@\mathsf{texp}_1@\cdots@\mathsf{texp}_n)\rrbracket\rho_s\\ &=\mathscr{T}\llbracket(\lambda \mathsf{x}.\mathsf{texp})@\mathsf{texp}'@\mathsf{texp}_1@\cdots@\mathsf{texp}_n\rrbracket\rho\end{aligned}$$

Since x does not appear free in texp_ι, $\mathscr{R}(\mathsf{texp}_\iota, \rho_s, \rho_d, \rho, t_\iota)$ is equivalent to $\mathscr{R}(\mathsf{texp}_\iota, \rho'_s, \rho_d, \rho', t_\iota)$, and the claim follows from the above development. □

LEMMA 11. $\forall \mathsf{texp}$: $\mathscr{H}(\mathsf{texp})$ *implies* $\mathscr{H}(\underline{\lambda}\mathsf{x}.\mathsf{texp})$.

PROOF. Assume $\rho_s, \rho_d, \rho, \tau$ are given, satisfying the conditions in Definition 5. It thus holds that $\tau \vdash \underline{\lambda}\mathsf{x}.\mathsf{texp}$: *code*, and thus by the inference rules that $\tau[\mathsf{x} \mapsto \mathit{code}] \vdash \mathsf{texp}$: *code*.

We assume that we have at hand an infinite list of variable names which have not previously been used, and when we write $\mathsf{x}_{\mathsf{new}}$, we refer to an arbitrary variable from this list.

Observation 12. For all $w \in \mathit{Val}$, it holds that

$$\forall \mathsf{id} \in \mathrm{FreeVars}(\mathsf{texp}):\quad \mathscr{R}(\mathsf{id}, \rho_s[\mathsf{x} \mapsto \mathsf{x}_{\mathsf{new}}], \rho_d[\mathsf{x}_{\mathsf{new}} \mapsto w], \rho[\mathsf{x} \mapsto w], \tau[\mathsf{x} \mapsto \mathit{code}](\mathsf{id}))$$

since $\forall \mathsf{id} \in \mathrm{FreeVars}(\underline{\lambda}\mathsf{x}.\mathsf{texp})$: $\mathscr{R}(\mathsf{id}, \rho_s, \rho_d, \rho, \tau(\mathsf{id}))$ and $\mathscr{R}(\mathsf{x}, \rho_s[\mathsf{x} \mapsto \mathsf{x}_{\mathsf{new}}], \rho_d[\mathsf{x}_{\mathsf{new}} \mapsto w], \rho[\mathsf{x} \mapsto w], \mathit{code})$.

Since $\tau[\mathsf{x} \mapsto \mathit{code}]$ clearly suits $\rho_s[\mathsf{x} \mapsto \mathsf{x}_{\mathsf{new}}]$ and $\tau[\mathsf{x} \mapsto \mathit{code}] \vdash \mathsf{texp}$: *code*, we may conclude from Observation 12 and the induction hypothesis that $\forall w \in \mathit{Val}$: $\mathscr{R}(\mathsf{texp}, \rho_s[\mathsf{x} \mapsto \mathsf{x}_{\mathsf{new}}], \rho_d[\mathsf{x}_{\mathsf{new}} \mapsto w], \rho[\mathsf{x} \mapsto w], \mathit{code})$. To prove Lemma 11, we must show that

$$\mathscr{E}\llbracket\mathscr{T}\llbracket\underline{\lambda}\mathsf{x}.\mathsf{texp}\llbracket\rho_s\llbracket\rho_d = \mathscr{E}\llbracket\phi(\underline{\lambda}\mathsf{x}.\mathsf{texp})\rrbracket\rho$$

We rewrite the left-hand side of the equation:

$$\begin{aligned}&\mathscr{E}\llbracket\mathscr{T}\llbracket\underline{\lambda}\mathsf{x}.\mathsf{texp}\rrbracket\rho_s\rrbracket\rho_d\\ &=\mathscr{E}\llbracket\lambda\mathsf{x}_{\mathsf{new}}.(\mathscr{T}\rrbracket\mathsf{texp}\llbracket\rho_s[\mathsf{x}\mapsto\mathsf{x}_{\mathsf{new}}])\rrbracket\rho_d\\ &=\lambda v.\mathscr{E}\llbracket\mathscr{T}\rrbracket\mathsf{texp}\llbracket\rho_s[\mathsf{x}\mapsto\mathsf{x}_{\mathsf{new}}]\rrbracket\rho_d[\mathsf{x}_{\mathsf{new}}\mapsto v]\end{aligned}$$

and the right-hand side:

$$\begin{aligned}&\mathscr{E}\llbracket\phi(\underline{\lambda}\mathsf{x}.\mathsf{texp})\rrbracket\rho\\ &=\mathscr{E}\llbracket\lambda\mathsf{x}.\phi(\mathsf{texp})\rrbracket\rho\\ &=\lambda v.\mathscr{E}\llbracket\phi(\mathsf{texp})\rrbracket\rho[\mathsf{x}\mapsto v]\end{aligned}$$

It now remains to show that

$$\lambda v.\mathscr{E}[\![\mathscr{T}[\![\text{texp}]\!]\rho_s[\text{x}\mapsto \text{x}_{\text{new}}]]\!]\rho_d[\text{x}_{\text{new}}\mapsto v] = \lambda v.\mathscr{E}[\![\phi(\text{texp})]\!]\rho[\text{x}\mapsto v]$$

When the two functions are applied to the same (arbitrary) $w \in Val$, the equality to be shown is

$$\mathscr{E}[\![\mathscr{T}[\![\text{texp}]\!]\rho_s[\text{x}\mapsto \text{x}_{\text{new}}]]\!]\rho_d[\text{x}_{\text{new}}\mapsto w] = \mathscr{E}[\![\phi(\text{texp})]\!]\rho[\text{x}\mapsto w]$$

which follows directly from $\forall w \in Val$: $\mathscr{R}(\text{texp}, \rho_s[\text{x}\mapsto \text{x}_{\text{new}}], \rho_d[\text{x}_{\text{new}}\mapsto w], \rho[\text{x}\mapsto w], code)$. □

LEMMA 13. $\forall$texp, texp′: ($\mathscr{H}$(texp) *and* $\mathscr{H}$(texp′)) *implies* $\mathscr{H}$(texp@texp′).

PROOF. Assume $\rho_s, \rho_d, \rho, \tau$ are given, satisfying the conditions in Definition 5. It thus holds that $\tau \vdash$ texp@texp′: t_2 and, by the inference rules, $\tau \vdash$ texp: $t_1 \rightarrow t_2$.

Since FreeVars(texp@texp′) = FreeVars(texp) ∪ FreeVars(texp′), it follows from the induction hypothesis that $\mathscr{R}(\text{texp}, \rho_s, \rho_d, \rho, t_1 \rightarrow t_2)$ and $\mathscr{R}(\text{texp}', \rho_s, \rho_d, \rho, t_1)$. Hence $\mathscr{R}(\text{texp@texp}', \rho_s, \rho_d, \rho, t_2)$. □

LEMMA 14. $\forall$texp, texp′: ($\mathscr{H}$(texp) *and* $\mathscr{H}$(texp′)) *implies* $\mathscr{H}$(texp$\underline{@}$texp′).

PROOF. Assume $\rho_s, \rho_d, \rho, \tau$ are given, satisfying the conditions in Definition 5. It thus holds that $\tau \vdash$ texp$\underline{@}$texp′: *code*, and hence that $\tau \vdash$ texp: *code* and $\tau \vdash$ texp′: *code*. Since FreeVars(texp$\underline{@}$texp′) = FreeVars(texp) ∪FreeVars(texp′), it follows from the induction hypothesis that $\mathscr{R}(\text{texp}, \rho_s, \rho_d, \rho, code)$ and $\mathscr{R}(\text{texp}', \rho_s, \rho_d, \rho, code)$.

$$\begin{aligned}
&\mathscr{E}[\![\mathscr{T}[\![\text{texp}\underline{@}\text{texp}']\!]\rho_s]\!]\rho_d\\
&= \mathscr{E}[\![\mathscr{T}[\![\text{texp}]\!]\rho_s @ \mathscr{T}[\![\text{texp}']\!]\rho_s]\!]\rho_d\\
&= (\mathscr{E}[\![\mathscr{T}[\![\text{texp}]\!]\rho_s]\!]\rho_d)(\mathscr{E}[\![\mathscr{T}[\![\text{texp}']\!]\rho_s]\!]\rho_d)\\
&= (\mathscr{E}[\![\phi(\text{texp})]\!]\rho)(\mathscr{E}[\![\phi(\text{texp}')]\!]\rho)\\
&= \mathscr{E}[\![\phi(\text{texp}\underline{@}\text{texp}')]\!]\rho
\end{aligned}$$

LEMMA 15. $\forall$texp: $\mathscr{H}$(texp) *implies* $\mathscr{H}$(fix texp).

PROOF. We prove Lemma 15 by fixpoint induction. The basic idea is to use the structural induction hypothesis $\mathscr{H}$(texp) to show the induction step in the fixpoint induction.

Assume $\rho_s, \rho_d, \rho, \tau$ are given, satisfying the conditions in Definition 5. It thus holds that $\tau \vdash$ fix texp: t and $\tau \vdash$ texp: $t \rightarrow t$. Since FreeVars(fix texp) = FreeVars(texp), it follows from the induction hypothesis that $\mathscr{R}(\text{texp}, \rho_s, \rho_d, \rho, t \rightarrow t)$.

By the inference rules of Figure 5, t is of form $t_1 \rightarrow \cdots \rightarrow t_n \rightarrow const$, n > 0 or $t_1 \rightarrow \cdots \rightarrow t_n \rightarrow code$, n > 0. For now we assume that $t = t_1 \rightarrow \cdots \rightarrow t_n \rightarrow const$.

We take texp$_{t\perp}$ to be an (arbitrary) closed two-level expression of type t such that

$$\mathscr{T}[\![\text{texp}_{t\perp}]\!]\rho_s = \lambda x_1 \cdots \lambda x_n.\perp = \mathscr{E}[\![\phi(\text{texp}_{t\perp})]\!]\rho$$

Thus $\mathscr{R}(\text{texp}_{t\perp}, \rho_s, \rho_d, \rho, t_1 \rightarrow \cdots \rightarrow t_n \rightarrow const)$ holds. By induction on m, repeatedly using $\mathscr{R}(\text{texp}, \rho_s, \rho_d, \rho, t \rightarrow t)$, we see that $\mathscr{R}(\text{texp@(texp@(}\ldots\text{@texp}_{t\perp})), \rho_s, \rho_d, \rho, t)$ where there are m applications of texp holds for any m.

Since t is of form $t_1 \to \cdots \to t_n \to \mathit{const}$, $\mathscr{R}(\text{fix texp}, \rho_s, \rho_d, \rho, t)$ may also be written as follows: $\forall \text{texp}_1, \ldots, \text{texp}_n$: $(\forall \text{i} \in [1 \,..\, \text{n}]$: $\tau \vdash \text{texp}_i$: t_i and $\mathscr{R}(\text{texp}_i, \rho_s, \rho_d, \rho, t_i))$ implies

$$\mathscr{T}[\![(\text{fix texp})@\text{texp}_1@ \cdots @\text{texp}_n]\!]\rho_s = \mathscr{E}[\![\phi((\text{fix texp})@\text{texp}_1@ \cdots @\text{texp}_n)]\!]\rho$$

This equation is shown by

$$\begin{aligned}
&\mathscr{T}[\![(\text{fix texp})@\text{texp}_1@ \cdots @\text{texp}_n]\!]\rho_s \\
&= (\mathscr{T}[\![\text{fix texp}]\!]\rho_s)(\mathscr{T}[\![\text{texp}_1]\!]\rho_s)\ldots(\mathscr{T}[\![\text{texp}_n]\!]\rho_s) \\
&= \bigsqcup \underbrace{(\mathscr{T}[\![\text{texp}]\!]\rho_s)((\mathscr{T}[\![\text{texp}]\!]\rho_s)\ldots(\mathscr{T}[\![\text{texp}_{t\perp}]\!]\rho_s))}_{\text{m texp's}}(\mathscr{T}[\![\text{texp}_1]\!]\rho_s)\ldots(\mathscr{T}[\![\text{texp}_n]\!]\rho_s)
\end{aligned}$$

Distribute applications over $\bigsqcup$, and use $\mathscr{T}$'s @-rule

$$= \bigsqcup (\mathscr{T}[\![(\text{texp}@(\text{texp}@(\ldots @\text{texp}_{t\perp})))@\text{texp}_1@ \cdots @\text{texp}_n[\![\rho_s)$$

Use that $\mathscr{R}(\text{texp}@(\text{texp}@(\ldots @\text{texp}_{t\perp}))@\text{texp}_1@ \cdots @\text{texp}_n, \rho_s, \rho_d, \rho, t)$ for all m

$$\begin{aligned}
&= \bigsqcup (\mathscr{E}[\![\phi(\text{texp}@(\text{texp}@(\ldots @\text{texp}_{t\perp})))@\text{texp}_1@ \cdots @\text{texp}_n]\!]\rho) \\
&= \bigsqcup (\mathscr{E}[\![\phi(\text{texp})]\!]\rho)((\mathscr{E}[\![\phi(\text{texp})]\!]\rho)\ldots(\mathscr{E}[\![\phi(\text{texp}_{t\perp})]\!]\rho))(\mathscr{E}[\![\phi(\text{texp}_1)]\!]\rho) \\
&\quad \ldots(\mathscr{E}[\![\phi(\text{texp}_n)]\!]\rho) \\
&= (\mathscr{E}[\![\phi(\text{fix texp})]\!]\rho)(\mathscr{E}[\![\phi(\text{texp}_1)]\!]\rho)\ldots(\mathscr{E}[\![\phi(\text{texp}_n)]\!]\rho) \\
&= \mathscr{E}[\![\phi((\text{fix texp})@\text{texp}_1@ \cdots @\text{texp}_n)]\!]\rho
\end{aligned}$$

If t is of form $t_1 \to \cdots \to t_n \to \mathit{code}$, the proof proceeds in a very similar manner, and we omit the calculation.

Lemma 16. $\forall$texp: *$\mathscr{H}$(texp) implies $\mathscr{H}$($\underline{\text{fix}}$ texp).*

Proof. Assume $\rho_s, \rho_d, \rho, \tau$ are given, satisfying the conditions in Definition 5. It thus holds that $\tau \vdash \underline{\text{fix}}\,\text{texp}$: *code* and $\tau \vdash$ texp: *code*. Since FreeVars($\underline{\text{fix}}$ texp) = FreeVars(texp), it follows from the induction hypothesis that $\mathscr{R}(\text{texp}, \rho_s, \rho_d, \rho, \mathit{code})$.

$$\begin{aligned}
&\mathscr{E}[\![\mathscr{T}[\![\underline{\text{fix}}\,\text{texp}]\!]\rho_s]\!]\rho_d \\
&= \mathscr{E}[\![\text{fix}\,\mathscr{T}[\![\text{texp}]\!]\rho_s]\!]\rho_d \\
&= \mathit{fix}\,\mathscr{E}[\![\mathscr{T}[\![\text{texp}]\!]\rho_s]\!]\rho_d \\
&= \mathit{fix}\,\mathscr{E}[\![\phi(\text{texp})]\!]\rho \\
&= \mathscr{E}[\![\phi(\text{fix texp})]\!]\rho \quad \Box
\end{aligned}$$

Lemma 17. $\forall$texp, texp′, texp″: *($\mathscr{H}$(texp) and $\mathscr{H}$(texp′) and $\mathscr{H}$(texp″)) implies $\mathscr{H}$(if texp texp′ texp″).*

Proof. Assume $\rho_s, \rho_d, \rho, \tau$ are given, satisfying the conditions in Definition 5. It thus holds that $\tau \vdash$ if texp texp′ texp″: t for some type t. Since FreeVars(if texp texp′ texp″) = FreeVars(texp) ∪ FreeVars(texp′) ∪ FreeVars(texp″), it follows from the induction hypothesis and the inference rules (Figure 5) that $\mathscr{R}(\text{texp}, \rho_s, \rho_d, \rho, \mathit{const})$, $\mathscr{R}(\text{texp}', \rho_s, \rho_d, \rho, t)$, and $\mathscr{R}(\text{texp}'', \rho_s, \rho_d, \rho, t)$. From $\mathscr{R}(\text{texp}, \rho_s, \rho_d, \rho, \mathit{const})$, we know that $\mathscr{T}[\![\text{texp}[\![\rho_s = \mathscr{E}[\![\phi(\text{texp})[\![\rho$, and that thus the $\mathscr{E}$- and $\mathscr{T}$-rules always choose the same branch. From this fact and the fact that $\mathscr{R}$ holds for texp′ and texp″ $\mathscr{R}$(if texp texp′ texp″, ρ_s, ρ, ρ, t), follows easily no matter whether t is *const*, *code*, or $t_1 \to t_2$. $\Box$

LEMMA 18. ∀texp, texp′, texp″: ($\mathscr{H}$(texp) *and* $\mathscr{H}$(texp′) *and* $\mathscr{H}$(texp″)) *implies* $\mathscr{H}$(<u>if</u> texp texp′ texp″).

PROOF. Assume ρ_s, ρ_d, ρ, τ are given, satisfying the conditions in Definition 5. It thus holds that $\tau \vdash$ <u>if</u> texp texp′ texp″: *code*, and $\tau \vdash$ texp: *code*, $\tau \vdash$ texp′: *code*, *and* $\tau \vdash$ texp″: *code*. Since FreeVars(<u>if</u> texp texp′ texp″) = FreeVars(texp) ∪ FreeVars(texp′) ∪ FreeVars(texp″), it follows from the induction hypothesis that $\mathscr{R}$(texp, ρ_s, ρ_d, ρ, *code*), $\mathscr{R}$(texp′, ρ_s, ρ_d, ρ, *code*), and $\mathscr{R}$(texp″, ρ_s, ρ_d, ρ, *code*).

$$\begin{aligned}
&\mathscr{E}[\![\mathscr{T}[\![\underline{\text{if}}\ \text{texp texp}'\ \text{texp}'']\!]\rho_s]\!]\rho_d\\
&= \mathscr{E}[\![\text{if}\ \mathscr{T}[\![\text{texp}]\!]\rho_s \mathscr{T}[\![\text{texp}']\!]\rho_s \mathscr{T}[\![\text{texp}'']\!]\rho_s]\!]\rho_d\\
&= (\mathscr{E}[\![\mathscr{T}[\![\text{texp}]\!]\rho_s]\!]\rho_d) \rightarrow (\mathscr{E}[\![\mathscr{T}[\![\text{texp}']\!]\rho_s]\!]\rho_d), (\mathscr{E}[\![\mathscr{T}[\![\text{texp}'']\!]\rho_s]\!]\rho_d)
\end{aligned}$$

since $\mathscr{R}$(texp, ρ_s, ρ_d, ρ, *code*), $\mathscr{R}$(texp′, ρ_s, ρ_d, ρ, *code*), and $\mathscr{R}$(texp″, ρ_s, ρ_d, ρ, *code*):

$$\begin{aligned}
&= (\mathscr{E}[\![\phi(\text{texp})]\!]\rho) \rightarrow (\mathscr{E}[\![\phi(\text{texp}')]\!]\rho), (\mathscr{E}[\![\phi(\text{texp}'')]\!]\rho_d)\\
&= \mathscr{E}[\![\phi(\underline{\text{if}}\ \text{texp texp}'\ \text{texp}'')]\!]\rho \quad \square
\end{aligned}$$

LEMMA 19. ∀texp: $\mathscr{H}$(texp) *implies* $\mathscr{H}$(lift texp).

PROOF. Assume ρ_s, ρ_d, ρ, τ are given, satisfying the conditions in Definition 5. It thus holds that $\tau \vdash$ lift texp: *code* and $\tau \vdash$ texp: *const*. Since FreeVars(lift texp) = FreeVars(texp), it follows from the induction hypothesis that $\mathscr{R}$(texp, ρ_s, ρ_d, ρ, *const*).

$$\begin{aligned}
&\mathscr{E}[\![\mathscr{T}[\![\text{lift texp}]\!]\rho_s]\!]\rho_d\\
&= \mathscr{E}[\![\text{const}\ \mathscr{T}[\![\text{texp}]\!]\rho_s]\!]\rho_d\\
&= \mathscr{E}[\![\text{const}\ \mathscr{E}[\![\phi(\text{texp})]\!]\rho]\!]\rho_d\\
&= \mathscr{E}[\![\phi(\text{texp})]\!]\rho \quad \square
\end{aligned}$$

5. COMPILATION AND COMPILER GENERATION

In this section we give an interpreter for an imperative language, *Tiny*, which has while-loops. We give a denotational semantics in lambda calculus form and use it to compile and generate a standalone compiler. The syntax of *Tiny*-programs is

```
program          ::= var-declaration command
var-declaration  ::= variables: variable*
command          ::= while expression command|
                     command, command|
                     variable := expression
expression       := expression operator expression|
                      . . . , −2, −1, 0, 1, 2, . . .
```

The semantic functions are given in Figure 6. We use the notation described in Section 1.2. *Nat* is the flat domain of natural numbers; *Location* is the flat domain of available memory cells, of which *first-location* is assumed to be the first. The function *next-loc* computes the next available location; σ_{init} is the overall undefined store. The environment maps *Tiny* variables to their locations in the store.

Semantic Domains

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

$\mathcal{P}$: *program* $\rightarrow$ *Store* $\rightarrow$ *Store*
$\mathcal{P}[\![\underline{\mathsf{variables:}}\ \mathsf{v}_1,\ldots,\mathsf{v}_n;\ \mathsf{cmd}]\!]\ \sigma_{init} = \mathcal{C}[\![\mathsf{cmd}]\!]\ (\mathcal{D}[\![\mathsf{v}_1,\ldots,\mathsf{v}_n]\!]\ \mathit{first\text{-}location})\ \sigma_{init}$

$\mathcal{D}$: *variable** $\rightarrow$ *Location* $\rightarrow$ *Environment*
$\mathcal{D}[\![\mathsf{v}_1,\ldots,\mathsf{v}_n]\!]\ loc = \lambda id.id{=}\mathsf{v}_1 \rightarrow loc,\ (\mathcal{D}[\![\mathsf{v}_2,\ldots,\mathsf{v}_n]\!]\ \mathit{next\text{-}loc}(loc))(id)$
$\mathcal{D}[\![]\!]\ loc = \lambda x.\mathit{error}_{loc}$

$\mathcal{C}$: *command* $\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$ *Store*
$\mathcal{C}[\![\mathsf{c}_1;\ \mathsf{c}_2]\!]\ \rho\ \sigma = \mathcal{C}[\![\mathsf{c}_2]\!]\ \rho\ (\mathcal{C}[\![\mathsf{c}_1]\!]\ \rho\ \sigma)$
$\mathcal{C}[\![\mathsf{var} := \mathsf{exp}]\!]\ \rho\ \sigma = \sigma[\rho(\mathsf{var}) \mapsto \mathcal{E}[\![\mathsf{exp}]\!]\ \rho\ \sigma]$
$\mathcal{C}[\![\underline{\mathsf{while}}\ \mathsf{exp}\ \mathsf{c}]\!]\ \rho\ \sigma = (\mathit{fix}\ \lambda f.\lambda\sigma_1.\mathcal{E}[\![\mathsf{exp}]\!]{=}0 \rightarrow \sigma_1,\ f(\mathcal{C}[\![\mathsf{c}]\!]\ \rho\ \sigma_1))\ \sigma$

$\mathcal{E}$: *expression* $\rightarrow$ *Environment* $\rightarrow$ *Store* $\rightarrow$ *Nat*
$\mathcal{E}[\![\mathbf{exp_1\ op\ exp_2}]\!]\ \rho\ \sigma = (\mathcal{E}[\![\mathbf{exp_1}]\!]\ \rho\ \sigma)\ op\ (\mathcal{E}[\![\mathbf{exp_2}]\!]\ \rho\ \sigma)$
$\mathcal{E}[\![\mathbf{num}]\!]\ \rho\ \sigma = num$

Fig. 6. *Tiny* semantics.

The semantic functions may easily be written in lambda calculus form to be partially evaluated. 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, but that istore is unknown. In other words, suppose that we have the type assumptions $\tau \vdash$ istore: *code* and $\tau \vdash$ program: *const*. When we apply the $\mathscr{T}$ rules to a well-annotated version of the *Tiny*-interpreter and the factorial program of Figure 1, we get the lambda expression of Figure 2. (Both programs are shown in the Introduction.)

5.1 An Example of Compiler Generation

Self-applying the partial evaluator with respect to the *Tiny*-interpreter yields a *Tiny* to lambda calculus compiler $\mathscr{C}_c$. The compiling function $\mathscr{C}_c$ is essentially a syntactically curried version of the semantic function $\mathscr{C}$. The operators annotated as residual in the *Tiny*-interpreter have been replaced by the corresponding code-generating actions.

To emphasize these striking structural similarities, we have renamed the machine-generated variables into names close to those of $\mathscr{C}$. Figure 7 displays the generated compiling function $\mathscr{C}_c$, syntactically sugared. On the right-hand side of the equations, we use, for brevity, the syntax-font: "@", "λ", . . . where the compilation algorithm builds syntactic expressions, instead of writing *build-@*, *build-*λ. . . . Comparison of Figures 6 and 7 shows

$$\begin{array}{lll}
\mathcal{C}_c[\![\mathsf{c}_1;\ \mathsf{c}_2]\!]\ \rho & = & \text{let } \sigma_1 = \textit{new-name}(\textit{const}\ \sigma) \text{ in} \\
 & & \quad \mathcal{C}_c[\![\mathsf{c}_2]\!]\ \rho\ \text{“@”}\ (\mathcal{C}_c[\![\mathsf{c}_1]\!]\ \rho\ \text{“@”}\ \sigma_1) \\
\mathcal{C}_c[\![\mathsf{var} := \mathsf{exp}]\!]\ \rho & = & \text{let } \sigma_1 = \textit{new-name}(\textit{const}\ \sigma) \text{ in} \\
 & & \quad \text{“update”}\ \text{“@”}\ \rho(\mathsf{var})\ \text{“@”}\ (\mathcal{E}_c[\![\mathsf{exp}]\!]\ \rho\ \sigma_1)\ \text{“@”}\ \sigma_1 \\
\mathcal{C}_c[\![\underline{\mathsf{while}}\ \mathsf{exp}\ \mathsf{c}]\!]\ \rho & = & \text{let } \sigma_1 = \textit{new-name}(\textit{const}\ \sigma) \\
 & & \quad \sigma_2 = \textit{new-name}(\textit{const}\ \sigma) \\
 & & \quad \textit{f1} = \textit{new-name}(\textit{const}\ f) \text{ in} \\
 & & \quad (\text{“fix”}\ \text{“}\lambda\text{”}\ \textit{f1}.\text{“}\lambda\text{”}\ \sigma_2.\ \text{“if”}\ =\ \text{“@”}\ 0\ \text{“@”}\ (\mathcal{E}_c[\![\mathsf{exp}]\!]\ \rho\ \sigma_2) \\
 & & \qquad\qquad\qquad\qquad\qquad \sigma_2 \\
 & & \qquad\qquad\qquad\qquad\qquad \textit{f1}\ \text{“@”}\ (\mathcal{C}_c[\![\mathsf{c}]\!]\ \rho\ \text{“@”}\ \sigma_2))\ \text{“@”}\ \sigma_1
\end{array}$$

Fig. 7. Generated compiling function— $\mathcal{C}_c$ is a syntactically curried version of $\mathcal{C}$.

Table I

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

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

that in the generated compiler the run-time (residual) actions of the interpreter have been replaced by code-building operations.

6. ASSESSMENT

Table I shows the run-times of our example programs. In the following, *fac* denotes the factorial program written in *Tiny* (Figure 1), *target* denotes the factorial residual program (Figure 2), *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. The sizes are measured as the number of cons cells plus the number of atoms in the S-expressions representing the programs.

The interpretative overhead in the mix program is rather large, since all free variables (input variables and function names) are looked up in the initial environment, which is linearly organized in our implementation. Since mix is able to remove interpretational overhead, the speed-ups gained when mix is partially evaluated are accordingly large (perhaps artificially so; in earlier work consistent speedups of 5 to 15 have been reported). The size ratio between *comp* and *tiny* is quite small, since the two programs are very similar in structure. Some operations in *tiny* have been replaced by the corresponding code-generating operations. The same relation holds between *mix* and *cogen*. The absolute sizes of *mix* and *cogen* are large because the initial environments are inlined pieces of code.

7. RELATED WORK

The present work overlaps two areas: *partial evaluation*, which 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.

7.1 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 [24], or as a method to transform imperative Lisp programs [2]. 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 [17, 34], but still not in a self-applicable way. A paper by Hannan and Miller derives partial reduction rules for the lambda calculus, but they do not address the question of when to reduce a redex and when to refrain [18].

The potential of self-application was realized independently in Japan and the Soviet Union [10, 13, 36] in the early 1970's, and experiments were made without conclusive results. The first actual self-application was realized in 1984 [21, 22] for first-order recursive equations. Since then several other self-applicable systems have been developed for programs in the form of term rewriting systems [4], for a simple imperative language [14], for Prolog [12], for a subset of Turchin's Refal language [32], and for stronger systems handling first-order Scheme programs [8, 5].

These systems are reasonably efficient for first-order languages, the generated compilers were typically between three and ten faster than compiling by partial evaluation of an interpreter. In all cases, a binding time analysis was seen as essential for efficient self-application (the reasons for this are detailed by Bondorf et al. [6]). To our knowledge, no nontrivial and fully automatic self-applicable higher-order solutions have been developed prior to the one presented here.

Although it can give dramatic speedups and has much promise, partial evaluation is no panacea. So far, a characteristic is that obtaining good results requires careful attention to programming style (workers must know the strengths and limitations of their tools, even when the tools are very powerful); an alternative is to use a source-to-source, binding-time-oriented program transformations (staging) to change program style in deeper ways [23].

7.2 Semantics-Directed Compiler Generation

The pathbreaking work in this field was SIS: the Semantics Implementation System of Mosses [25]. 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 due to the fact that the reduction strategy is "on-line," and the problem could be eliminated by annotations such as we have used. (Choosing the annotations so as to avoid nontermination is admittedly a challenging problem, but we feel it is one that should be solved *before* doing partial evaluation rather than during it.)

Paulson [29], Weis [39], and Nielson and Nielson [27] present systems based on the pure (typed) lambda calculus. 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 [39] is probably the fastest in this category that has been used on large language definitions. In Nielson's work, the greatest emphasis is put on correctness rather than efficiency.

Systems by Pleban [31] and Appel [1] achieve greater run-time efficiency at the expense of less pure semantic languages—one for each language definition in the former case, and the lambda calculus with dedicated treatment of environments and stores in the latter. Finally, Wand's system and methodology [37, 38] require so much cleverness from the user that it is not clear how it may be automated.

To our knowledge, none of these systems is so powerful that one could consider using the system to construct its own components; and 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.

8. CONCLUSION

We have developed and successfully self-applied a higher-order partial evaluator. The partial evaluator is based on very simple principles and has been proven correct; but it is powerful enough to generate efficient residual programs from a denotational language definition of *Tiny* and to generate a standalone compiler with a very natural structure.

ACKNOWLEDGMENTS

First of all, I want to thank Neil D. Jones without whose collaboration this work would probably not have been done. The three other coauthors of the conference paper [20]: A Bondorf, O. Danvy, and T. Mogensen also deserve thanks, as well as P. Sestoft and the other members and friends of the programming language group at DIKU. Thanks to P. Wadler and J. Hughes for suggesting a simplification of the main induction hypothesis. Last, but not least, thanks to the referees whose thorough reports pointed out several deficiencies and typos in the original manuscript. Special thanks go to the reviewer who found an embarrassing blunder in the correctness proof.

REFERENCES

1. Appel, A. Semantics-directed code generation. In *12th ACM Symposium on Principles of Programming Languages* (New Orleans, Jan. 1985), pp. 315-324.
2. Beckman, L., et al. A partial evaluator, and its use as a programming tool. *Artif. Intell. 7*, 4 (1976), 319-357.
3. Bondorf, A. Automatic autoprojection of higher order recursive equations. In *ESOP '90. 3rd European Symposium on Programming* (Copenhagen, May 1990), N. D. Jones, Ed., Lecture Notes in Computer Science, 432, Springer-Verlag, May 1990, 70-87.
4. Bondorf, A. A self-applicable partial evaluator for term rewriting systems. In *TAPSOFT '89. Proceedings International Conference Theory and Practice of Software Development* (Barcelona, Mar. 1989), J. Diaz and F. Orejas, Eds., Lecture Notes in Computer Science, 352, Springer-Verlag, 1989, 81-95.
5. Bondorf, A., and Danvy, O. Automatic autoprojection of recursive equations with global variables and abstract data types. Tech. Rep. 90/4, DIKU, Univ. of Copenhagen, 1990. To appear in Science of Computer Programming.
6. Bondorf, A., Jones, N. D., Mogensen, T., and Sestoft, P. Binding time analysis and the taming of self-application. Draft, DIKU, Univ. of Copenhagen, Aug. 1988.
7. Consel, C. Analyse de programmes, evaluation partielle et génération de compilateurs. Ph.D. dissertation, Univ. de Paris 6, Paris, June 1989, 109 pp. (in French).
8. Consel, C. New insights into partial evaluation: The Schism experiment. In *ESOP '88, 2nd European Symposium on Programming* (Nancy, France, March 1988). H. Ganzinger, Ed., Lecture Notes in Computer Science, 300, Springer-Verlag, 1988, pp. 236-246.
9. Consel, C., and Danvy, O. Partial evaluation of pattern matching in strings. *Inf. Process. Lett. 30*, 2 (Jan. 1989), 79-86.
10. Ershov, A. P. On the essence of compilation. In *Formal Description of Programming Concepts*, E. J. Neuhold, Ed., North-Holland, Amsterdam, 1978, 391-420.
11. Ershov, A. P., and Itkin, V. E. Correctness of mixed computation in Algol-like programs. In *Mathematical Foundations of Computer Science* (Tatranská Lomnica, Czechoslovakia), J. Gruska, Ed., Lecture Notes in Computer Science, 53, Springer-Verlag, New York, pp. 59-77.
12. Fuller, D. A., and Abramsky, S. Mixed computation of Prolog programs. *New Gen. Comput. 6*, 2 and 3 (1988), 119-141.
13. Futamura, Y. Partial evaluation of computation process—An approach to a compiler-compiler. *Syst. Comput. Controls 2*, 5 (1971), 45-50.
14. Gomard, C. K., and Jones, N. D. Compiler generation by partial evaluation: A case study. *J. Struct. Program. 12*, 3 (July 1991), to appear. Also DIKU Rep. 88/24 and 90/16.
15. Gomard, C. K. Partial type inference for untyped functional programs. In *1990 ACM Conference on Lisp and Functional Programming* (Nice, France, June 1990), ACM, 1990, pp. 282-287.
16. Gomard, C. K., and Jones, N. D. A partial evaluator for the untyped lambda-calculus. *J. Func. Program. 1*, 1 (Jan. 1991), 21-69.
17. Guzowski, M. A. Towards developing a reflexive partial evaluator for an interesting subset of Lisp. Master's Thesis, Dept. of Computer Eng. and Science, Case Western Reserve Univ., Cleveland, Ohio, Jan. 1988.
18. Hannan, J., and Miller, D. Deriving mixed evaluation from standard evaluation for a simple functional language. In *Mathematics of Program Construction* (Groningen, The Netherlands), J. L. A. van de Snepscheut, Ed., Lecture Notes in Computer Science, 375, Springer-Verlag, New York, 1989, pp. 239-255.
19. Jones, N. D. Automatic program specialization: a re-examination from basic principles. In

Partial Evaluation and Mixed Computation, D. Bjørner, A P Ershov, and N D. Jones, Eds., North-Holland, Amsterdam, 1988, pp. 225-282.

20. JONES, N. D , GOMARD, C. K., BONDORF, A., DANVY, O., AND MOGENSEN, T. E. A self-applicable partial evaluator for the lambda calculus. In *1990 International Conference on Computer Languages* (New Orleans, LA, March 1990), IEEE Computer Society, New York, 1990, pp. 49-58.
21. JONES, N. D , SESTOFT, P., AND SØNDERGAARD, H. An experiment in partial evaluation: the generation of a compiler generator. In *Rewriting Techniques and Applications* (Dijon, France), J.-P. Jouannaud, Ed., Lecture Notes in Computer Science, 202, Springer-Verlag, New York, 1985, pp. 124-140.
22. JONES, N. D., SESTOFT, P., AND SØNDERGAARD, H. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp Symbolic Comput. 2*, 1 (1989), 9-50.
23. JØRRING, U., AND SCHERLIS, W. L. Compilers and staging transformations. In *Thirteenth ACM Symposium on Principles of Programming Languages* (St. Petersburg, Fla.), ACM, New York, 1986, pp. 86-96.
24. LOMBARDI, L. A. Incremental computation. In *Advances Comput. 8*, F. L. Alt and M Rubinoff, Eds., 1967, 247-333.
25. MOSSES, P. *SIS—Semantics Implementation System, Reference Manual and User Guide.* DAIMI Rep. MD-30, DAIMI, Univ. of Århus, Denmark, 1979.
26. NIELSON, F. A formal type system for comparing partial evaluators. In *Partial Evaluation and Mixed Computation*, D. Bjørner, A. P. Ershov, and N. D. Jones, Eds , North-Holland, Amsterdam, 1988, 349-384.
27. NIELSON, F., AND NIELSON, H. R. The TML-approach to compiler-compilers. Tech. Rep. 1988-47, Dept. of Computer Science, Technical Univ. of Denmark, 1988.
28. NIELSON, H. R., AND NIELSON, F Automatic binding time analysis for a typed λ-calculus. In *15th ACM Symposium on Principles of Programming Languages* (San Diego, Calif., Jan. 1988), pp. 98-106.
29. PAULSON, L. A semantics-directed compiler generator. In *9th ACM Symposium on Principles of Programming Languages* (Albuquerque, N.M., Jan. 1982), pp. 224-233.
30. JONES, S. L. P. *The Implementation of Functional Programming Languages.* Prentice-Hall, Englewood Cliffs, N.J., 1987.
31. PLEBAN, U Compiler prototyping using formal semantics. *SIGPLAN Not. 19*, 6 (1984), 94-105.
32. ROMANENKO, S. A. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In *Partial Evaluation and Mixed Computation*, D. Bjørner, A. P. Ershov, and N. D. Jones, Eds., North-Holland, Amsterdam, 1988, 445-463.
33. SCHMIDT, D. A. Detecting global variables in denotational specifications. *ACM Trans Program. Lang. Syst. 7*, 2 (Apr. 1985), 299-310.
34. SCHOOLER, R. Partial evaluation as a means of language extensibility. Master's thesis, MIT/LCS/TR-324, Laboratory for Computer Science, MIT, Cambridge, Mass., Aug. 1984.
35. SESTOFT, P. Replacing function parameters by global variables. In *Functional Programming Languages and Computer Architecture* (London, Sept. 1989), ACM Press, New York, and Addison-Wesley, Reading, Mass., 1989, pp. 39-53.
36. TURCHIN, V. F. The use of metasystem transition in theorem proving and program optimization. In *Automata, Languages and Programming. Seventh ICALP* (Noordwijkerhout, The Netherlands), J. De Bakker and J. van Leeuven, Eds., Lecture Notes in Computer Science, 85, Springer-Verlag, New York, 1980, pp. 645-657.
37. WAND, M. A semantic prototyping system. In *SIGPLAN '84 Symposium on Compiler Construction*, (Montreal, Canada, June 1984), pp 213-221.
38. WAND, M. Semantics-directed machine architecture In *9th ACM Symposium on Principles of Programming Languages*, (Albuquerque, N.M., Jan. 1982), pp. 234-241.
39. WEIS, P. Le Systeme SAM: Metacompilation tres efficace a l'aide d'operateur semantiques. Ph.D. dissertation, l'Univ. Paris VII, 1987 (in French).

Received December 1990; revised February 1991; accepted March 1991