Collapsing heterogeneous towers of interpreters

Michael Buch
Queens' College

UNIVERSITY OF
CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: mb2244@cam.ac.uk

May 22, 2019

Abstract

A tower of interpreters is a program architecture that consists of a sequence of interpreters each interpreting the one adjacent to it. The overhead induced by multiple layers of evaluation can be optimized away using a program specialization technique called *partial evaluation*, a process referred to as *collapsing of towers of interpreters*. Towers of interpreters in literature are synonymous with reflective towers and provide a tractable method with which to reason about reflection and design reflective languages. Reflective towers studied thus far are *homogeneous*, meaning individual interpreters are meta-circular and have a common data representation between each other. Research into homogeneous towers rarely considered the applicability of associated optimization techniques in practical settings where multiple interpretation layers are commonplace but the towers are *heterogeneous* (i.e., interpreters lack meta-circularity, reflection and data homogeneity). The aim of our study was to investigate the extent to which previous methodologies for collapsing reflective towers apply to heterogeneous configurations.

To collapse a tower means to *stage* an interpreter in the tower (i.e., convert the interpreter into a compiler by splitting its execution into several stages) and statically reduce all the evaluation performed by preceding interpreters. Where the procedure to collapse homogeneous towers is trivial because computation performed in one interpreter can be represented in terms of its interpreter and information of which operations to partially evaluate can be propagated using the same built-in operators, this is not the case in a heterogeneous setting. There, one would need to convert representations of program constructs at each interpreter boundary and find a way to pass information needed by the partial evaluator through the tower. Our contributions include: (1) we construct and collapse an experimental heterogeneous tower using Pink, a language that was previously used to collapse reflective towers through a modified variant of partial evaluation called *type-directed partial evaluation (TDPE)* (2) we stage a SECD abstract machine using TDPE which required modification of its operational semantics to ensure termination in the presence of recursive calls (3) we investigate the hypothesis that staging at different levels in the tower affects its optimality after collapse.

Contents

12585 (errors:1) words (out of 15000)

1 Introduction

Towers of interpreters are a program architecture which consists of sequences of interpreters where each interpreter is interpreted by an adjacent interpreter (depicted as a tombstone diagram in figure 1). Each additional *level* (i.e., interpreter) in the tower adds a constant factor of interpretative overhead to the run-time of the system. One of the earliest mentions of such architectures in literature is a language extension to LISP called 3-LISP [1] introduced by Smith. Smith describes the notion of a reflective system, a system that is able to reason about itself, as a tower of meta-circular interpreters, also referred to as a *reflective tower* [1]. Using this architecture 3-LISP enables an interpreter within the tower to access and modify internal state of its neighbouring interpreters. An interpreter is *meta-circular* when the language the interpreter is written in and the language it is interpreting are the same. Meta-circularity and the common data representation between interpreters are core properties of reflective towers studied in previous work. We refer to towers with such properties as *homogeneous*. Subsequent studies due to Wand et al. [2] and Danvy et al. [3] show systematic approaches for constructing reflective towers. The authors provide denotational semantic accounts of reflection and develop languages based on the reflective tower model called *Brown* and *Blond* respectively.

In the original reflective tower models only minimal attention was given to the imposed cost of performing new interpretation at each level of a tower. Then works by Sturdy [4] and Danvy et al.'s language Blond [3] hinted at the possibility of removing some of this overhead by partially evaluating (i.e., specializing) interpreters with respect to the interpreters below in the tower. Asai et al.'s language *Black* [5] is a reflective language implemented through a reflective tower. The authors use

[1] Reflective towers in theory are considered to be potentially infinite. Given enough computing resources one can create towers consisting of an unbounded number of interpreters. In Wand et al.'s reflective tower model [2], for instance, new interpreters in a tower are spawned through a built-in *reflect* operator

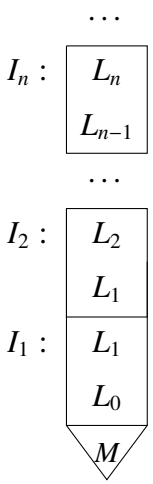


Figure 1: A tower of interpreters where each interpreter I_n is written in language L_{n-1} and interprets a language L_n, for some $n \geq 0$. In literature the tower often grows downwards, however, in our study we refer to I_0 as the base interpreter and grow the tower upwards for convenience. M is the underlying machine (e.g. CPU) on which the base interpreter is executed.

a hand-crafted partial evaluator, and in a later study use MetaOCaml [6], to efficiently implement the language. Asai and then, using the language Pink [7], Amin et al. demonstrate the ability to compile a reflective language while the semantics of individual interpreters in the underlying tower can be modified. Essentially this is achieved by specializing and executing functions of an interpreter at run-time to remove the cost of multiple interpretation; this effectively *collapses* a tower.

Parallel to all the above theoretical research into reflective towers, practical programmers have been working with towers of interpreters to some extent dating back to the idea of language parsers. Writing a parser in an interpreted language already implies two levels of interpretation: one running the parser and another the parser itself. Other examples include interpreters for embedded domain-specific languages (DSLs) or string matchers embedded in a language both of which form towers of two levels. Advances in virtualization technology has driven increasing interest in software emulation. Viewing emulation as a form of interpretation we can consider interpreters running on virtual hardware, such as the bytecode interpreter in the Java Virtual Machine [8], as towers of interpreters as well.

However, these two branches of research do not overlap and work on towers of interpreters rarely studied their counterparts in production systems. It is natural to ask the question of what it would take to apply previous techniques in partial evaluation to a practical setting. This is the question Amin et

al. pose in their conclusion after describing Pink [7] and is the starting point for this thesis.

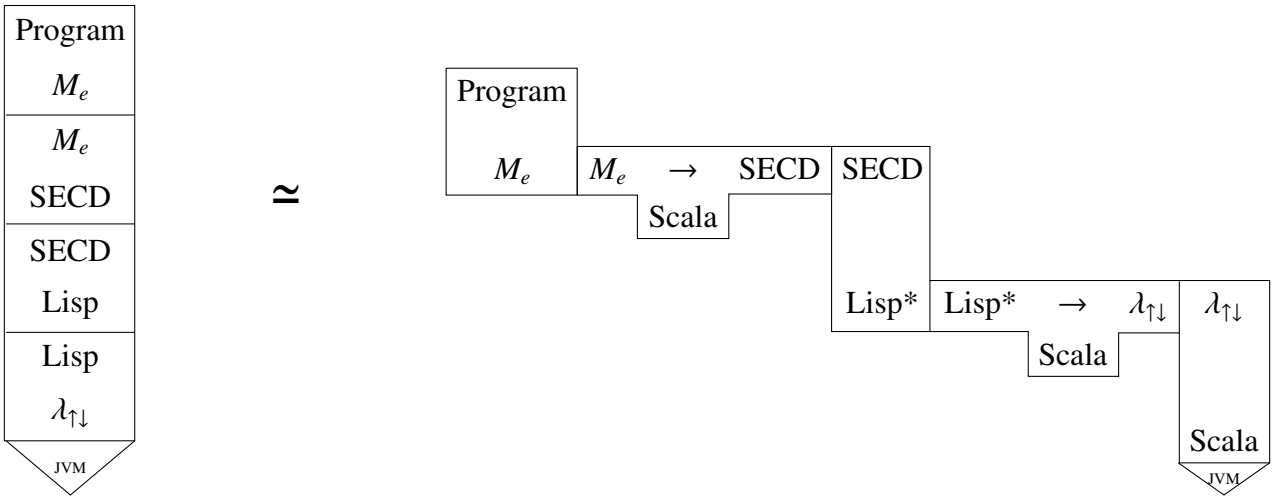
We aim to bring previous work of removing interpretative overhead in towers using partial evaluation into practice. Our study achieves this by constructing a proof-of-concept tower of interpreters that more-closely resembles those in real-world systems. Figure 2a describes two versions of our experimental tower. Traditionally reflective towers can be thought of as completely vertical as the one depicted on the left. However, details such as how a tower grows and shrinks or how exactly execution of user programs gets performed worked rather mysteriously, in part also due to meta-circularity. We decided to implement our tower using occasional layers of compilation (as shown on the right). On one hand it is more convenient to implement such a tower because compilation helps translation from higher-level to less intuitive lower-level languages (such as the SECD instruction set). On the other hand, we hope to improve the understanding of the importance of the intensional structure of towers, particularly how it affects the process of collapsing towers. We can assume that extensionally the computation they perform is equivalent since they both execute a user program yielding the same output.

We then collapse the experimental tower under different configurations and evaluate the resulting optimized programs. We demonstrate that given a language capable of expressing types of variables available at run-time and compile-time (i.e., a *multi-level language*) and a type-directed PE (TDPE), a lightweight partial evaluator due to Danvy [9] described in section 2.5, we can partially evaluate individual interpreters in a sequence of non-metacircular interpreters and effectively generate code specialized for a user program (hopefully eliminating interpretative overhead in the process). Our work's contributions are:

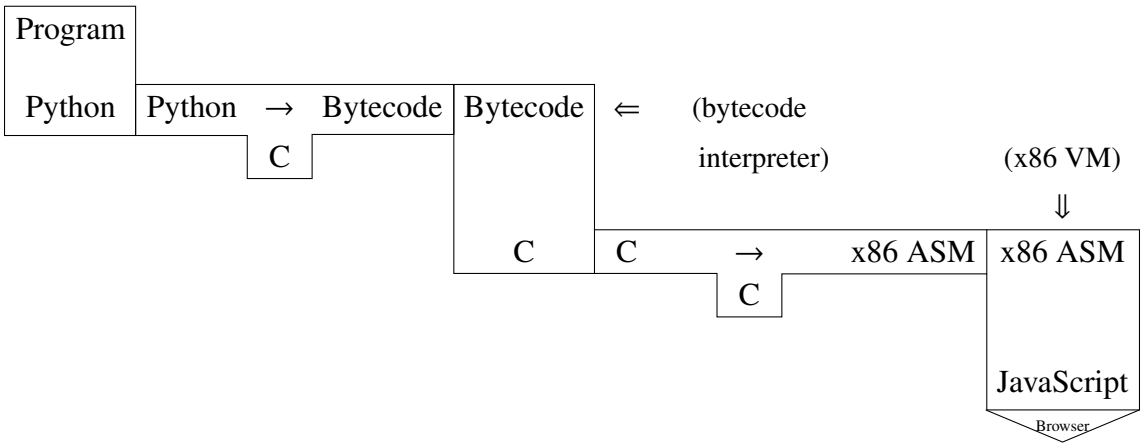
1. Develop an experimental heterogeneous tower of interpreters and a strategy for collapsing it
2. Evaluate the effect that staging at different levels within our tower has on the residual programs
3. Discuss the effects that heterogeneity in towers imposes on TDPE
4. Demonstrate issues with and potential approaches to staging abstract machines, specifically a SECD machine, using TDPE

In section 2 we explain background information that covers the fundamental topics we base our experiments and discussions on. We then define *heterogeneity* in towers of interpreters in section 3. In section 4 we describe the recipe that the partial evaluation framework Pink [7] used to construct and collapse meta-circular towers and then show how this recipe changes as a result of heterogeneity.

We present the implementation and evaluation of our experimental tower in section 5. Section 5.1 provides an overview of Pink. We systematically describe the process by which we create a heterogeneous tower of interpreters and incrementally collapse it in sections 5.2 through 5.4. We conclude with an evaluation of our findings followed by a discussion of potential future work in section 6.



(a) Tombstone diagrams that represent our two extensionally equal versions of our experimental tower of interpreters. M_e is our toy language described in section 5.3, $\lambda_{\uparrow\downarrow}$ refers to the multi-level language introduced as part of Pink [7] and Lisp* is $\lambda_{\uparrow\downarrow}$'s Lisp based front-end. *JVM* is the Java Virtual Machine [8] and in our diagram also encompasses any underlying machinery necessary to run the JVM. While the left depicts the intuitive view of a tower, we actually implement it using the architecture on the right. Not only is the tower on the right simpler to construct but it also highlights the effect collapsing of towers has on the computation of individual levels and the power of the *lift* operator described in section 2.5



(b) A hypothetical tower of interpreters that serves as the model for the tower we built (figure 2a). The diagram depicts a JavaScript x86 virtual machine (VM) running a Python interpreter that in turn executes a Python script. *Browser* encompasses the JavaScript interpreter within a browser and any underlying technologies required to host the browser.

Figure 2: Comparison between our experimental tower (2a) and the one we modelled it on (2b).

2 Background

2.1 λ-Calculus and de Bruijn Indices

Terms in the untyped lambda calculus consist of variables, lambda abstractions and application of terms. An identifier denotes each variable and determines which lambda binds which variables. Unbound variables are called *free*. Consider now the application of a lambda to a free variable y:

$$(\lambda x.\lambda y.xy)y$$

A β-reduction of the above term involves an invalid substitution of occurrences of x with y because we bring the free variable, y, into a scope where a lambda already bound the identifier y:

$$\lambda y.yy$$

To prevent a clash of variable names typically one would perform an α-conversion to rename variables in the lambda appropriately before substitution:

$$\begin{aligned} & (\lambda y.xy)[x := y] \\ \equiv_\alpha\ & (\lambda z.xz)[x := y] \\ \equiv\ & \lambda z.yz \end{aligned}$$

De Bruijn introduced a canonical lambda notation that prevents such variable name collisions and eliminates the need for α-conversions during substitutions [10]. *De Bruijn indices* denote each variable with an integer that is the number of lambda abstractions between a variable's occurrence and the lambda binding it. Assuming an initial index of 1 for free variable y, the above example in de Bruijn index notation is:

$$(\lambda\lambda.1\ 0)\ 1$$

Variables and lambdas in $\lambda_{\uparrow\downarrow}$ (see section 2.6) follow the de Bruijn indexing scheme to avoid the complexity of managing variable names and their scopes.

2.2 Difficulties in Recursion

Let-expressions in functional languages are typically syntactic sugar for λ-abstraction and application. Consider following term:

$$(\lambda f.e')(\lambda x.e)$$

As long as f does not occur in e we can rewrite the above in let-expression form:

```
let f = λx.e in e'
```

Recursion in programming languages is the ability to reference an expression from within its definition. Permitting the occurrence of f in e in the let-expression above would leave us with a recursive let-expression (also *letrec*). However, translating recursion from λ-calculus terms to implement letrecs is not as straightforward. Three possible approaches are outlined below.

2.2.1 Fixed-Point Combinators

The fixed point of function f is some value x such that $f(x) = x$. The function that determines the fixed point of a function is typically labelled *fix* and defined as $x = fix(f)$. By the definition of a fixed point we get an equation that resembles recursive function application $fix(f) = f(fix(f))$. Church developed the *Y-combinator*, a fixed-point combinator in the lambda calculus: $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ Notably Church showed that the *Y-combinator* can be used to implement recursion even in languages that do not support it. For some term g: $Yg = g(Yg)$

A letrec can now be implemented by using the *Y-combinator* as follows:

```
letrec f = Y(λf.λx.e) in e'
```

However, this can be inefficient compared to other solutions because of the pressure on the stack.

2.2.2 Tying the Knot

In languages that permit lazy evaluation or mutation one can create circular data structure definitions using the principle of *tying a knot*. In a language with lazy evaluation one can create a circular efficiently by sharing parts of its definition: x = 0:y y = 1:x Creates a stream of ones and zeroes on-demand.

One can utilize this technique to efficiently implement recursion in functions that seek the definition of a closure from an environment. On a recursive call a location within the environment is set to the definition of the calling closure such that it can find and reuse it during repeated calls. SECD uses these concepts to implement its **RAP** instruction for recursive function application (more details in 2.3) [11].

2.2.3 Self-referencing Lambdas

A final solution is to add self-references to the definition of lambdas. In such a language a lambda named f with argument x and body e is written as: $\lambda_f x.e$. Thus let-expressions and letrecs can simply be expressed as:

```
let f = λ_x.e in e'
letrec f = λ_f x.e in e'
```

Each lambda implicitly has a reference to itself available for use from within its definition. Pink (section 2.6) implements this type of recursion (also *open recursion*) using self-referencing lambdas.

2.3 The SECD Machine and Instruction Set (ISA)

The SECD machine due to Landin [12] is a well-studied stack-based abstract machine initially developed in order to provide a model for evaluation of terms in the λ-calculus. All operations on the original SECD machine are performed on four registers: stack (S), environment (E), control (C), dump (D). *C* holds a list of instructions that should be executed. *E* stores values of free variables (including functions), function arguments and function return values. The *S* register stores results of function-local operations and the *D* register is used to save and restore state in the machine when performing control flow operations. A function we call *step* makes sure the machine progresses by reading next instructions and operands from the remaining entries in the control register and terminates at a **STOP** or **WRITEC** instruction, at which point the machine returns all values or a single value from the S-register respectively.

We now describe the instruction set and implementation details described by Kogge [13], which itself is a followup to Henderson's LispKit SECD machine [14]. The three types of SECD instructions are: (1) function definition and application (2) special forms including if-statements (3) anything else such

as arithmetic. While a table describing all instructions and their transitions is available in figure A.1, below we present examples that demonstrate the instructions needed to comprehend later sections.

2.3.1 Examples

- **LDC** loads an operand (a string or constant) onto the stack (i.e., S-register). Arithmetic instructions such as **SUB** operate on the top two items of S. **SEL** branches to two different sets of instructions depending on whether the top of S is 0 (i.e., false) or not. **JOIN** jumps back from a branch and resumes the rest of the program while placing the value computed in that branch on top of the stack, e.g.

```
LDC 10 LDC 20 SUB
LDC 0 GT SEL
    (LDC 5 JOIN)
    (LDC -5 JOIN)
LDC done
STOP

Result: (done -5)
```

- **LDF** loads a SECD function (i.e., a list of instructions) onto S. The argument to the function is the second element in S. **AP** applies the function on top of S to its argument which in the example below is 10. A function can access its argument through the environment register via a lookup using **LD**. **RTN** places the value computed in the function onto the stack and returns to the caller.

```
LDC 10 LDF
    (LD (1 1) LDC 20 MPY RTN)
AP
STOP

Result: (200)
```

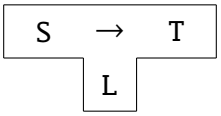
- A recursive call consists of two SECD functions: a recursive function and a function that initiates the first recursive call. **RAP** will set the argument to the recursive function such that it will

always find itself in the environment, thus allowing recursion. Additionally, it will initiate the first recursive call. In the below example the recursive function will call itself and increment its argument by 1 and return the argument once it reaches 5. The function finds its definition through `LD (2 1)` in the environment and subsequently calls itself. For a better understanding of recursive SECD function application see the **RAP/letrec** analogy in figure A.2 and the transition table in figure A.1.

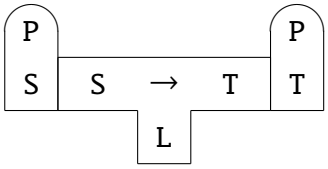
```
DUM NIL LDF
    (LD (1 1) LDC 5 EQ SEL                ;argument == 5?
        (LD (1 1) JOIN)                   ;return argument
        (NIL LDC 1 LD (1 1) ADD           ;increment argument
            CONS LD (2 1) AP JOIN) RTN) ;Recursive call occurs here
CONS LDF
     (NIL LDC 0 CONS LD (1 1) AP RTN) RAP STOP

    Result: (5)
```

2.4 Interpretation and Compilation

(a) Interpreter written in language L interpreting a source program in language S.



(b) Compiler written in L that translates from source language S to a target language T.



(c) Example compilation of a program P from language S to a language T

Figure 3: Tombstone diagrams representing interpretation and compilation

An interpreter reads and directly executes instructions based on an input source program (depicted in figure 3a as a *tombstone diagram*). *definitional interpreters* define the language they implement but not necessarily uniquely [15].

A compiler translates a program into another representation that can subsequently be executed by some underlying machine, or interpreter. A tombstone representation of a compiler is shown in figure 3b. The translation process can occur in a pipeline of an arbitrary number of stages in which a source program is transformed into intermediate representations (IR) to aid its analysis or further transformation. Here a program P written in language S is compiled to a program P written in T; that compiler executes in language L. We use tombstone diagrams throughout our study to represent towers of interpreters because they provide a pictorial view of composition and an intensional perspective of a tower's structure. Previous work on reflective towers study their extensional structure forced by their meta-circularity. A tombstone view gives a way to highlight the effect of heterogeneity to their intensional structure.

In the 1970s Futamura showed that compilers and interpreters are fundamentally related in an elegant way by three equations also known as the Futamura projections [16]. At its core, the three projections are based on the theory of function *specialization* (or in mathematical terms *projection*). Given a

function $f(x, y)$, one can produce a new unique specialized function $f_x(y)$ for a fixed value of x. In program specialization, we consider f to be a program and the inputs to said program are sets of *static* data x (known before the program's runtime) and *dynamic* data y (only known once a program starts). A *partial evaluator* (also called *residualizer*) is usually denoted by *mix* and takes two inputs: a program and static data to specialize against. Evaluating *mix* (i.e, $[\![mix]\!]$) against some program p using static data x yields one of possibly many residual programs, p_x, for a fixed value of x:

$$\begin{aligned} p_x &= [\![mix]\!]\ (p, x) \\ out &= [\![p_x]\!]\ (y) \end{aligned}$$

In the above equations p is said to have been *partially evaluated*. Futamura's first projection showed that a compiler for a language L, $comp^L$, is functionally equivalent to a specializer, *mix*, for an interpreter for language L, int^L. In other words, partially evaluating int^L given the source of a L-program, src^L, achieves compilation:

$$\begin{aligned} target &= [\![mix]\!]\ (int^L, src^L) \\ &= [\![comp^L]\!]\ (src^L) \end{aligned}$$

We could now go a step further and instead of specializing an interpreter specialize *mix* itself and consider the interpreter to be its static input. We pass to $[\![mix]\!]$ its source, *mix*, and an interpreter, int^L; this process is referred to as *self-application*. This yields Futamura's second projection which says that by self-applying a partial evaluator one is able to derive a compiler from an interpreter (i.e., just a semantic description of a language):

$$comp^L = [\![mix]\!]\ (mix, int^L)$$

In literature a *staging transformation* [17, 18] converts an interpreter into a compiler by splitting the interpreter's direct execution into several stages. The Futamura projections above imply that partial evaluation can be used to introduce two stages to interpretation: one that generates a residual program and another that executes it [19].

A practical realization of the Futamura projections has since been an active area of research. The difficulty in their implementation is the question of how one can best specialize an interpreter and meanwhile also generate the most efficient and correct code; this question is still being explored in

the design of partial evaluators to this day [20, 19].

2.5 Type-Directed Partial Evaluation

Partial evaluation (PE) is a program-optimization technique based on the insight that there is room in programs for statically reducing and producing a specialized (and thus hopefully more efficient) version of itself. The output of a partial evaluator is a *residual program* which, in the ideal case, is version of the original program where as much computation as possible has been performed with the data that was available at specialization time (i.e., during run-time of the partial evaluator). The portion of data that is known at specialization time is called static and otherwise dynamic. For variables in the program to-be-specialized we refer to its *binding-time* as static if the data it holds during the lifetime of the program is static. Otherwise a variable's binding-time is dynamic.

Partial evaluators generally contain a *binding-time analysis (BTA)* stage which determines whether expressions can be reduced at specialization time or should be preserved in the residual program. A BTA produces a *division* [19]; this assigns to each function and variable in a program a binding-time. A division is said to be *congruent* if it assures that every expression that involves dynamic data is marked as dynamic and otherwise as static. A partial evaluator being just an ordinary program, a problem one can run into is non-termination. A congruent division does not always guarantee termination of a PE but when it does we call the division *safe*.

In the literature we distinguish between *online* and *offline* partial evaluation [19] (or more recently a hybrid between the two due to Shali et al. [21]). Offline PE performs a BTA before it begins specialization whereas the online approach makes decisions about whether to residualize expressions once partial evaluation has begun.

Danvy devised a method of implementing a partial evaluator solely based on the ideas of normalization in the simply-typed λ-calculus called *Type-Directed Partial Evaluation* [9]. The result is a remarkably simple methodology of implementing residualizers with a binding-time analysis that is completely driven by the *types* of the expressions being specialized. TDPE is built on three core concepts [22, 23]:

1. BTA produces an expression annotated with static or dynamic binding-times such that reducing static terms yields a *completely dynamic* residual expression (i.e., an expression containing only dynamic variables)

2. BTA should be able to generate code that includes *binding-time coercions* which are type corrections that help to convert between dynamic and static values for cases where doing so benefits the residualization process
3. Static reduction (i.e., reduction of expressions annotated as static) is performed by evaluation of an expression and well-formedness of the generated code is guaranteed by the implementing language's type system

Consider the example (1) from Danvy's description of TDPE [9]. The aim is to annotate the function applications (denoted by @) and definitions with a static (overline) or dynamic (underline) binding-time.

$$\lambda g.\lambda d.(\lambda f.g \mathbin{@} (f \mathbin{@} d) \mathbin{@} f) \mathbin{@} \lambda a.a \tag{1}$$

A correct binding-time could be that shown in (2) because after static reduction we obtain (3) which is a completely dynamic expression. However, the duplication of f after reduction and a dynamic redex, $(\underline{\lambda} a.a) \mathbin{\underline{@}} d$, could be optimized away further with a modification to the source expression (i.e., a binding-time coercion). The challenge we are faced with is that f is applied statically but cannot be completely reduced because we also pass it as an argument to a dynamic expression, prohibiting us from residualizing the expression to its fullest.

$$\underline{\lambda} g.\underline{\lambda} d.(\overline{\lambda} f.g \mathbin{\underline{@}} (f \mathbin{\underline{@}} d) \mathbin{\underline{@}} f) \mathbin{\overline{@}} \underline{\lambda} a.a) \tag{2}$$

▷ *(via static reduction)*

$$\underline{\lambda} g.\underline{\lambda} d.(g \mathbin{\underline{@}} ((\underline{\lambda} a.a) \mathbin{\underline{@}} d) \mathbin{\underline{@}} \underline{\lambda} a.a \tag{3}$$

We can apply η-expansion to turn instances of f as a higher-order value into static function applications; TDPE uses this operation during specialization time to increase the number of static expressions it can reduce. The resulting annotations are shown in (4) and the η-expansion is highlighted in green. After static reduction we obtain the optimal expression in (5) that only contains dynamic values, only a single unfolding of f and no β-redexes. Danvy then generalizes the η-expansion into a class of coercions that permit residualization of static values in dynamic contexts, an expression with a dynamic hole [24]. We represent such coercions with a $\downarrow^T$ (or *lift* as we use in later sections and as defined in figure 4) which represents the conversion from a static value of type T to a dynamic value. Using the

coercion operator, TDPE would yield the annotation in (6).

$$\underline{\lambda}g.\underline{\lambda}d.(\overline{\lambda}f.g\ \underline{@}\ (f\ \overline{@}\ d)\ \underline{@}\ \underline{\lambda}x.f\ \overline{@}x)\ \overline{@}\ \overline{\lambda}a.a \tag{4}$$

▷ *(via static reduction)*

$$\underline{\lambda}g.\underline{\lambda}d.g\ \underline{@}\ d\ \underline{@}\ \underline{\lambda}a.a \tag{5}$$

$$\underline{\lambda}g.\underline{\lambda}d.(\overline{\lambda}f.g\ \underline{@}\ (f\ \overline{@}\ d)\ \underline{@}\ (\mathit{lift}\ f)\ )\ \overline{@}\ \overline{\lambda}a.a \tag{6}$$

where we omit the type parameter on *lift* for simplicity

The traditional TDPE *residualize* operation takes as input an expression and a separate object representing the expressions type. Based on the latter argument the residualization algorithm will perform reification according to the rules in figure 4.

To summarize, TDPE makes use of the type system of a two-level language to direct the residualization of expressions. The output of this lightweight BTA is an expression whose terms are annotated with dynamic or static binding times which when statically reduced yield a purely dynamic expression. Static reduction is performed through regular evaluation of the two-level language's interpreter. TDPE includes a set of operators to convert between dynamic and static terms to aid the optimality of generated code. The two classes of coercion operators are reification (also called lift) and reflection both of which are defined on product, function and literal types. *Residualization* is finally defined as performing annotations and binding-time coercions followed by static reduction.

As dicussed in section 5.1, the Pink [7] partial evaluator implements most of the binding time coercions according to figure 4's definitions with the exception of *reflect*, which is implemented using an algorithm attributed to Eijiro Sumii [25] to ensure correct order of evaluation of side-effects.

Reification (Lifting)

$$\downarrow^{t} v = v \tag{7}$$

$$\downarrow^{t_1 \to t_2} v = \underline{\lambda} x.\ \downarrow^{t_2} (v \overline{@} (\uparrow_{t_1} x)) \tag{8}$$

where x is not a free variable in v

$$\downarrow^{t_1 \times t_2} v = \underline{cons}(\downarrow^{t_1} \overline{car}\, v, \downarrow^{t_2} \overline{cdr}\, v) \tag{9}$$

$$lift = \lambda t.\lambda v.\ \downarrow^{t} v \tag{10}$$

Reflection

$$\uparrow_{t} e = e \tag{11}$$

$$\uparrow_{t_1 \to t_2} e = \overline{\lambda} v.\ \uparrow_{t_2} (e \underline{@} (\downarrow^{t_1} v)) \tag{12}$$

$$\uparrow_{t_1 \times t_2} e = \overline{cons}(\uparrow_{t_1} \underline{car}\, e, \uparrow_{t_2} \underline{cdr}\, e) \tag{13}$$

$$reflect = \lambda t.\lambda e.\ \uparrow_{t} e \tag{14}$$

Figure 4: Reduction rules for reification (static to dynamic) and reflection (dynamic to static) in TDPE as defined by Danvy [9] where t denotes types, v denotes static values, e denotes dynamic expressions. The syntax $cons/car/cdr$ corresponds to the LISP functions of the same name that create a pair, extract the first element of a pair and extract the second element of a pair respectively.

2.6 $\lambda_{\uparrow\downarrow}$ Overview

We now provide an overview of the partial evaluation framework developed for Pink [7] which forms the base of our experimental tower (Lisp* and $\lambda_{\uparrow\downarrow}$ in figure 2a). At Pink's core is the multi-stage language, $\lambda_{\uparrow\downarrow}$. The language distinguishes between static values and dynamic values using type constructors `Val` and `Exp` respectively. The core evaluator will either residualize, or statically reduce an expression based on the binding-times on its individual terms. The core evaluator (*evalms* in listing 1) serves as our PE that produces residual programs (i.e., generates code) that are represented by dynamic expressions wrapped in a `Code` constructor and are in A-normal form [26]. For example a residualized addition of two literals is,

```
Code(Plus(Lit(5),Lit(5)))
```

In $\lambda_{\uparrow\downarrow}$ the TDPE *reify* operation is called *lift* and converts static `Val`s into dynamic `Exp`s. The fact that code generation of expressions can be guided using this single operator, whose semantics closely resemble expression annotation, is attractive for converting interpreters into translators. A user of $\lambda_{\uparrow\downarrow}$ can stage an interpreter by annotating its source provided the possibility of changing the interpreter's internals and enough knowledge of its semantics.

Despite being based on TDPE, the partial evaluation scheme used in Pink is a modified variant which we refer to as *dynamic TDPE*. The binding time analysis is done during run-time of the evaluator and does not necessarily require a type-system. The PE operates purely on static or dynamic values of $\lambda_{\uparrow\downarrow}$. The ability to dynamically decide on binding-time decisions and not needing a type-system also allows more flexibility in which expressions we want to residualize. Additionally, the *reflect* operator serves a different purpose to TDPE's specification. Instead of coercing static into dynamic expressions, *reflect* adds an expression to the global accumulator of residual terms, `stBlock`, that generates the specialized code through let-insertion (see line 43 in listing 1). Danvy introduced *let-insertion* into an implementation of TDPE [25] to ensure non-idempotent side-effects in expressions duplicated by TDPE are themselves not performed more often than they occur in a source program.

```
// Scala implementation of λ↑↓
// expressions (i.e., dynamic data)
abstract class Exp                          // The type of dynamic expressions
case class Lit(n:Int) extends Exp           // Integers
case class Sym(s:String) extends Exp        // Strings
case class Var(n:Int) extends Exp           // Variables (represented as de Bruijn indices)
case class Lam(e:Exp) extends Exp           // Lambdas (no need for argument list due
                                            // to de Bruijn variables)
case class App(e1:Exp, e2:Exp) extends Exp  // Function application
...
// values (i.e., static data)
abstract class Val                          // The type of static expressions
type Env = List[Val]                        // Environment
case class Cst(n:Int) extends Val           // Integers
case class Str(s:String) extends Val        // Strings
case class Clo(env:Env,e:Exp) extends Val   // Closures
case class Code(e:Exp) extends Val          // Residual dynamic data
...

// Converts Expressions (Exp) into Values (Val)
def evalms(env: Env, e: Exp): Val = e match {
    case Lit(n) => Cst(n)
    case Sym(s) => Str(s)
    case Var(n) => env(n)
    case Lam(e) => Clo(env,e)
    case Lift(e) =>
      Code(lift(evalms(env,e)))
    ...
    case If(c,a,b) =>
      evalms(env,c) match {
        case Cst(n) =>
          if (n != 0) evalms(env,a) else evalms(env,b)
        case (Code(c1)) => // Generate an if-statement if conditional is dynamic
          reflectc(If(c1, reifyc(evalms(env,a)), reifyc(evalms(env,b))))
      }
    ...
}
...
var stBlock: List[(Int, Exp)] = Nil
```

```
def reify(f: => Exp) = run {
    stBlock = Nil
    val last = f
    (stBlock.map(_._2) foldRight last)(Let) // Let-insertion occurs here
}
def reflect(s:Exp) = {
    stBlock :+= (stFresh, s)
    fresh()
}
// TDPE-style 'reify' operator (semantics -> syntax)
def lift(v: Val): Exp = v match {
    case Cst(n) => // number
      Lit(n)
    case Str(s) => // string
      Sym(s)
    case Tup(Code(u),Code(v)) => reflect(Cons(u,v))
    case Clo(env2,e2) => // function
      stFun collectFirst { case (n,`env2`,`e2`) => n } match {
        case Some(n) =>
          Var(n)
        case None =>
          stFun :+= (stFresh,env2,e2)
          reflect(Lam(reify{ val Code(r) = evalms(env2:+Code(fresh()):+Code(fresh()),e2); r }))
      }
    case Code(e) => reflect(Lift(e))
}
...
```

Listing 1: Main points of interest of the $\lambda_{\uparrow\downarrow}$ interpreter written in Scala [27].

We make use of the notion of *stage-polymorphism* introduced by Offenbeck et al. [28] to support two modes of operation: (1) ordinary evaluation (2) generation and subsequent execution of $\lambda_{\uparrow\downarrow}$ terms. Stage-polymorphism allows abstraction over how many stages an evaluation is performed over. This is achieved by operators that are polymorphic over what stage they operate on and is simply implemented as shown in figure 5. Whenever the *lift* operator is now used in the *interpreter* or *compiler* it will cause *evalms* to either evaluate or generate code respectively.

Now we can combine meta-circular interpreters or compilers in a single invocation of a program like in the following example:

```
(let interpreter (let maybe-lift (lambda (x) x) (...)))
(let compiler (let maybe-lift (lambda (x) (lift x)) (...)))
```

Figure 5: "Stage-polymorphism" implementation from Amin et al.'s Pink [7]

```
(let int_src       (quote interpreter)
    (let comp_src      (quote compiler)
        ((((interpreter int_src) int_src) comp_src) program))))
```

Figure 6: Use of stage-polymorphic definitions of *lift* defined in figure 5. The function *quote* is the LISP function that prevents evaluation of its argument and *program* is a quoted input program provided by the caller.

An advantage of TDPE and why the Pink framework serves as an appropriate candidate PE in our experiments is that it requires no additional dedicated static analysis tools to perform its residualization, keeping complexity at a minimum. Given an interpreter we can stage it by following Amin et al.'s [7] recipe: lift all terminal values an interpreter returns. Marking returned values as dynamic will dynamize and residualize any operation that includes them.

3 Heterogeneity

A central part of our study revolves around the notion of heterogeneous towers. Prior work on towers of interpreters that inspired some these concepts includes Sturdy's work on the Platypus language framework that provided a mixed-language interpreter built from a reflective tower [4], Jones et al.'s Mix partial evaluator [29] in which systems consisting of multiple levels of interpreters could be partially evaluated and Amin et al.'s study of collapsing towers of interpreters in which the authors present a technique for turning towers of meta-circular interpreters into one-pass compilers. We continue from where the latter left of, namely the question of how one might achieve the effect of compiling multiple interpreters in heterogeneous settings. Our definition of *heterogeneous* is as follows:

Definition 3.1. Towers of interpreters are systems of interpreters, $I_1^L, I_2^L, ..., I_n^L$ where $n \in \mathbb{N}_{\geq 1}$ and I_k^L determines an interpreter at level k written in language L_{k-1} and interprets programs in L_k. The language at interpreter level k is denoted L_k

Definition 3.2. Heterogeneous towers of interpreters are towers which exhibit following properties:

1. For any two adjacent interpreters I_k and I_{k-1} where $k \in \mathbb{N}_{\geq 1} : L_k \not\equiv L_{k-1}$ can hold
2. For any two adjacent interpreters used in the tower, I_k and I_{k-1}, the operational semantics *or* the representation of data can be different between the two

3.1 Absence of: Meta-circularity

The first constraint imposed by definition 3.2 is that of mixed languages between levels of a tower. A practical challenge this poses for partial evaluators is the inability to reuse language facilities across interpreters. This also implies that one cannot in general define reflection and reification procedures as in 3-LISP [1], Brown [2], Blond [3], Black [5] or Pink [7].

3.2 Absence of: Reflection

Reflection in an interpreter enables the introspection and modification of its state during execution. It is a tool reflective languages can use to embed tools such as debuggers or run-time instrumentation into programs. Reflection in reflective towers implies the ability to modify an interpreter's interpreter which can be beneficial in the implementation of said tool. However, it also allows potentially destructive operations on a running interpreter's semantics which can become difficult to reason about or debug. Towers that we are interested in rarely provide reflective capabilities in every, or even a single, of its interpreters. Thus, we do not support or experiment with reflection in our study.

3.3 Semantic Gap and Mixed Language Systems

Danvy et. al mentioned the possibility of non-reflective non-meta-circular towers early on in his denontational description of the reflective tower model [3]. The authors explored the idea of having different denotations for data at every level of the tower. However, since it was not the focus of their study, the potential consequences were not further investigated but serve as the motivation for the second point of definition 3.2. We call the difference in operational semantics or data representation between two interpreters a *semantic gap*.

One of our motivations stems from the realization that systems consisting of several layers of interpretation can feasibly be constructed. A hypothetical tower of interpreters that served as a model

for the one we built throughout our work was described in Amin et al.'s paper on collapsing towers [7] and is depicted as a tombstone diagram in figure 2b. As a comparison our tower is shown in 2a. We replace the x86 emulator with a SECD abstract machine interpreter and Python with our own functional toy language, M_e. The label $\lambda_{\uparrow\downarrow}$ represents the multi-level core language from Pink [7] and Lisp* is the LISP-like front-end to $\lambda_{\uparrow\downarrow}$. Although here the tower grows upwards and to the left, this need not be. The compilers, or *translators*, from M_e to SECD and from Lisp* to $\lambda_{\uparrow\downarrow}$ have been implemented in Scala purely for simplicity. To realize a completely vertical tower (i.e., consisting of interpreters only), the Lisp*-$\lambda_{\uparrow\downarrow}$ translator could be omitted such that the $\lambda_{\uparrow\downarrow}$ interpreter evaluates s-expressions directly. Similarly, the M_e-SECD compiler could be implemented in SECD instructions itself. However, we argue that the presence of compilation layers in our experimental tower resembles a setting in practice more closely and adds some insightful challenges to our experiments.

4 General Recipe for Collapsing Towers

In this section we describe the methodology that Pink uses to construct and collapse towers and then discuss changes that have to be considered when applying it to a heterogeneous setting.

4.1 TDPE and Staging a Definitional Interpreter

Pink defines a multi-level language that differentiates between static and dynamic values. This is essential to express binding-time information. The TDPE-style *lift* (i.e., reify) operator is implemented such that the PE can coerce static to dynamic values.

In his original description of TDPE Danvy [9] showed that we can use above tools to residualize a definitional interpreter with respect to a source program. Pink demonstrates how to stage such an interpreter by wrapping all literal, function and product types in calls to its *lift* operator. Additionally, Pink introduces the concept of *binding-time agnostic staging*. Here, the a single interpreter can be used to residualize or simply evaluate a program. In reference to Pink, *staging an interpreter* means lifting types as described above but also activating *code generation mode*, a detail we use in the next section. Code generation mode means an interpreter's polymorphic lift (defined in 5) is instantiated such that it invokes $\lambda_{\uparrow\downarrow}$'s built-in *lift*.

4.2 Construction and Collapse of a Tower

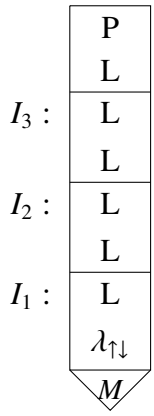
A tower can then be constructed using a set of meta-circular interpreters each interpreting the next level in the tower. The key benefit of meta-circularity and the basis of collapsing the tower is that the *lift* operator defined in the base evaluator is accessible to each interpreter. We can then stage the user-most interpreter (i.e., the interpreter, I_n, running the program provided by the user), as described in section 4.1. Although previous work did not provide an insight into the exact effect of staging at different levels in the tower, an intuitive reason we stage at the top-most level is that we want to eliminate as much interpretative overhead as possible which is achieved by collapsing the maximal set of interpreters in the tower. In our experimental tower in sections 5.1 to 5.4 we provide evidence to support this claim.

When we now execute the tower (i.e., invoking the partial evaluator at the base) only the top-most evaluator residualizes while all others evaluate. Because of meta-circularity, instead of evaluating a user-value it is now being lifted at all stages essentially propagating binding-time information from top-most interpreter to the base PE. At the last interpreter above the base evaluator, the *lift* now dispatches to the actual reify implementation as specified by TDPE. Effectively after residualization the generated program will only include the values staged at the top-most interpreter while the rest of the tower was reduced at specialization time since it was not evaluated in code-generation mode. Figures 7a-7c depict the above process through tombstone diagrams. We start with a meta-circular tower of interpreters all written in the same language, L, and Pink's $\lambda_{\uparrow\downarrow}$ at the base. After we stage the top-most interpreter (figure 7b) we can residualize a user program by generating $\lambda_{\uparrow\downarrow}$ terms; this is essentially the same as removing the intermediate interpreters as they do not get residualized which yields the collapsed tower in figure 7c (assuming the absence of side-effects at individual levels).

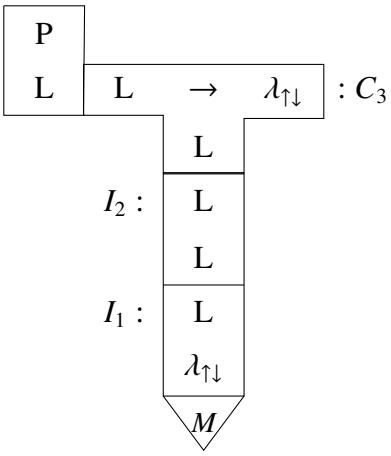
4.3 Effect of Heterogeneity

In mixed-language towers a *lift* operation is not necessarily available to all interpreters unless explicitly provided at a level. Hence, one approach to propagating binding-time information in heterogeneous towers is to implement a built-in *lift* at all levels below the interpreter that should be staged. As we explain in more detail in section 5.2.1, the implementation of *lift* may require us to transform the representation of closures, pairs or other constructs which the lift at the base expects.

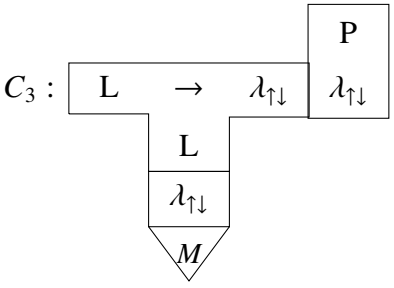
Additionally, the TDPE residualization algorithm uses evaluation of expressions to drive its BTA and



(a) Tower of meta-circular interpreters, I_k, in language, L, running a program, P.



(b) Tower whose top interpreter is staged (i.e., converted into a compiler, C)



(c) Final representation of the tower in 7a after collapsing it. All intermediate interpretation (levels I_1 to I_3) has been eliminated (by evaluating it during PE time) and P has been specialized with respect to the top-most staged interpreter, C_3. The residual program P consists of $\lambda_{\uparrow\downarrow}$ terms in ANF-normal form.

Figure 7: Tombstones representing the process of collapsing a tower using $\lambda_{\uparrow\downarrow}$

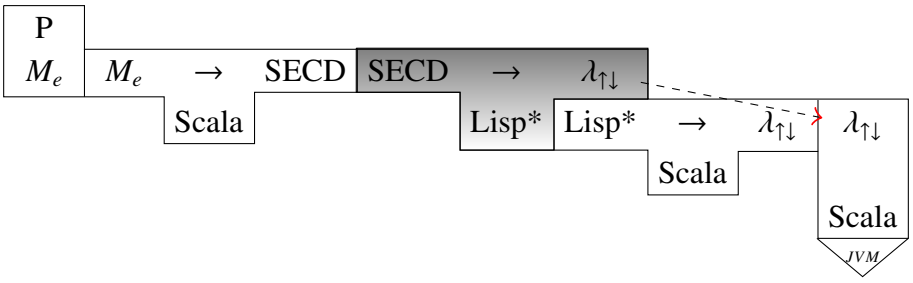


Figure 8: Our heterogeneous tower of interpreters after staging at the SECD level (shaded tombstone). Residualization of program visible through the process of staging the SECD interpreter (since it now outputs $\lambda_{\uparrow\downarrow}$ terms).

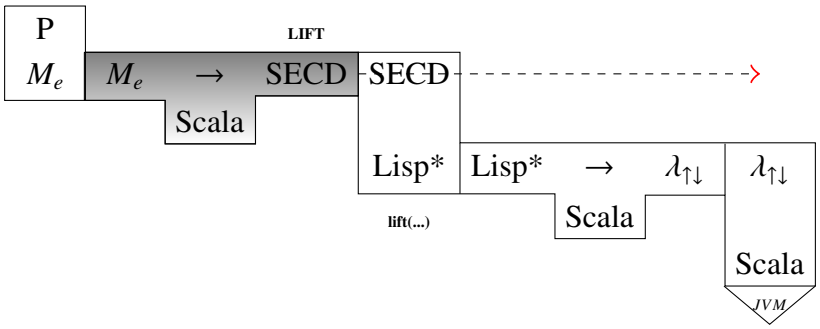


Figure 9: Our heterogeneous tower of interpreters after staging at the M_e level (shaded tombstone). Residualization of program (i.e., collapse of tower) is hidden. All computation to the left of the staged interpreter is carried into the residual program.

static reductions. While a definitional interpreter can be staged using TDPE by simply lifting values it returns, the process of staging an abstract machine, as we show in 5.2.1, requires careful design of its division rules and consideration of which locations to lift. Representation of closures can cause issues with non-termination if there is no inherent distinction between ordinary and recursive function definitions (an example of which is shown in section 5.2.1). To avoid these issues one can transform closures to fit the underlying lift operation's interface and use tags on them that signal the PE when to terminate in the unfolding of recursive function calls.

```
def reflect(s:Exp) = {
    ...
    case _ =>
        s match {
          case Fst(Cons(a, b)) => a
          case Snd(Cons(a, b)) => b
          case Plus(Lit(a), Lit(b)) => Lit(a + b)
          case Minus(Lit(a), Lit(b)) => Lit(a - b)
          case Times(Lit(a), Lit(b)) => Lit(a * b)
          ...
          case _ => // all other cases
            stBlock :+= (stFresh, s)
            fresh()
        }
}
```

Figure 10: *Smart constructors* (highlighted in green) in $\lambda_{\uparrow\downarrow}$'s *reflect*

5 Construction of an Experimental Heterogeneous Tower

5.1 Level 1 & 2: $\lambda_{\uparrow\downarrow}$

The $\lambda_{\uparrow\downarrow}$ interpreter (running on the JVM) and its Lisp front-end (Lisp* in figure 2a) form the first two levels of the tower. We keep Pink's implementation of the two levels mostly unchanged.

To reduce the amount of generated code we add logic within $\lambda_{\uparrow\downarrow}$'s *reflect* that reduces purely static expressions. Reducible expressions include arithmetic and list access operations. We refer to these in later sections as *smart constructors* and they aid in normalizing static expressions that the division (described in section 5.2.1) does not permit. These are shown in an excerpt Pink's source in figure 10.

5.2 Level 3: SECD

Our reasoning behind choosing the SECD abstract machine as one of our levels is three-fold:

1. **Maturity**: SECD was the first abstract machine of its kind developed by Landin in 1964 [12]. Since then it has thoroughly been studied and documented [30, 31, 14] making it a strong foundation to build on.

2. **Large Semantic Gap**: A central part of our definition of heterogeneity is that languages that adjacent interpreters interpret are significantly different from each other (see section 3). In the case of SECD's operational semantics, the representation of program constructs such as closures and also the use of stacks to perform all computation deviates from the semantics of $\lambda_{\uparrow\downarrow}$ and it's LISP front-end and thus satisfies the *semantic gap* property well.

3. **Extensibility**: Extensions to the machine, many of which are described by Kogge [13], have been developed to support richer features than the ones available in its most basic form including parallel computation, lazy evaluation and call/cc semantics.

An additional benefit of using a LISP machine is that the $\lambda_{\uparrow\downarrow}$ framework we use as our partial evaluator also features a LISP front-end that supports all the list-processing primitives which were used to describe the operational semantics of SECD that we implement (described by the small-step semantics and compiler from Kogge's book on symbolic computation [13]). We model the machine through a case-based evaluator with a *step* function at its core that repeatedly advances the state of the machine until a **STOP** or **WRITEC** instruction is encountered.

5.2.1 Staging a SECD Machine

We design our SECD machine such that it can be staged using Pink's PE; this will enable us to experiment the effect of staging at different levels of a tower. By definition, a staged evaluator should have a means of generating an intermediate representation, for example residual code, followed by a way to execute it (directly or through further stages). Partial evaluation allows us to split the SECD interpreter's execution into two stages: (1) reduce static values and residualize dynamic values (2) execute the residual expressions.

From the architecture of a SECD machine the intended place for free variables and user input to live in is the environment register. For example an expression as the following uses a free-variable, y, which is unknown and a residualizer would classify as dynamic:

```
((lambda (x) (x + y)) 10)
```

This would translate to following SECD instructions:

```
NIL LDC 10 CONS LDF (LD (1 1) LD (2 1) ADD RTN) AP STOP
```

We load only a single value of 10 into the environment and omit the second argument that **LDF** expects and uses inside its body. Instead **LDF** simply loads at a location not yet available (i.e., `LD (2 1)`) and trusts the user to provide the missing value at run-time.

Prior to deciding on the methodology for code generation we need to outline what stages one can add to the evaluation of the SECD machine and how the binding-time division is chosen. We define our division by where static values can be transferred from. If a dynamic value can be transferred from a register, A, to another register, B, we classify register B as dynamic. We define the binding-time of a SECD register to be the combination of all possible binding-times of elements of a register. When a register can hold both dynamic and static values, the register's binding time is referred to as *mixed*. The full division for each SECD register is provided in table 1.

We refer to our division as coarse grained since dynamic values pollute whole registers that could serve as either completely static or mixed valued. An example would be a machine that simply performs arithmetic on two integers and returns the result. The state machine transitions would occur as shown in table 2. As the programmer we know there is no unknown input and the expression can simply be reduced to the value 30 following the SECD small-step semantics. However, by default our division assumes the S-register to be dynamic and thus generates code for adding two constants. In such cases the smart constructors discussed in section 5.1 allow as to reduce constant expressions that a conservative division would otherwise not. We keep this division as the basis for our staged SECD machine since it is less intrusive to its interpreter and still allows us to residualize efficiently.

Through its LISP front-end the $\lambda_{\uparrow\downarrow}$ evaluator can operate as a partial evaluator by exposing its *lift* operator. We stage our SECD machine interpreter by annotating its source with said *lift* according to the division in table 1.

SECD Register	Classification	Reason
S (Stack)	Mixed (mostly dynamic)	Function arguments and return values operate on the stack *and* dynamic environment and thus are mostly dynamic. Elements of the stack can, however, be static in the case of thunks described in section 5.2.3
E (Environment)	Mixed (mostly dynamic)	Most elements in this register are dynamic because they are passed from the user or represent values transferred from the stack. Since the stack can transfer static values on occasion the environment can contain static values as well.
C (Control)	Static	We make sure the register only receives static values and is thus static (we ensure this through eta-expansion in section 5.2.1)
D (Dump)	Mixed	Used for saving state of any other register and thus elements can be both dynamic, static or a combination of both
F (Functions)	Static	Since it resembles a *control* register just for recursively called instructions we also classify it as static

Table 1: Division rules for our approach to staging a SECD machine

Step	Register Contents
0	s: () e: () c: (LDC 10 LDC 20 ADD WRITEC) d: ()
1	s: (10) e: () c: (LDC 20 ADD WRITEC) d: ()
2	s: (20 10) e: () c: (ADD WRITEC) d: ()
3	s: (30) e: () c: (WRITEC) d: ()
4	s: () e: () c: () d: ()
Generated Code (without smart constructor): (lambda f0 x1 (+ 20 10))	
Generated Code (with smart constructor): (lambda f0 x1 30)	

Table 2: Example of SECD evaluation and $\lambda_{\uparrow\downarrow}$code generated using our PE framework. The division follows that of table 1.

5.2.2 The Interpreter

```
1  (let SECDMachine (lambda _ stack (lambda _ dump (lambda _ control (lambda _ environment
2      (if (eq? 'LDC (car control))
3          (((((SECDMachine (cons (cadr control) stack)) dump) (cdr control)) environment)
4      (if (eq? 'DUM (car control))
5          (((((SECDMachine stack) dump) (cdr control)) (cons '() environment))
6      (if (eq? 'WRITEC (car control))
7          (car s)
8      ...
9      ...)))))))
10     (let initStack '()
11     (let initDump '()
12         (lambda _ control (((((SECDMachine initStack) initDump) control))))
```

Figure 11: Structure of interpreter for SECD machine (unstaged). Lambdas take two arguments, a self-reference for recursion (which is ignored through a "_" sentinel) and a caller supplied argument. All of SECD's stack registers are represented as LISP lists and initialized to empty lists. "..." indicate omitted implementation details

Our staged machine is written in $\lambda_{\uparrow\downarrow}$'s LISP front-end as a traditional case-based interpreter that dispatches on SECD instructions stored in the C-register. The structure of our SECD interpreter, *SECDMachine*, without annotations to stage it is shown in figure 11. Of note are the single-argument self-referential lambdas due to the LISP-frontend and the out-of-order argument list to the machine. To allow a user to supply instructions to the machine we return a lambda that accepts input to the control register (C) in line 12 of figure 11. Once a SECD program is provided we curry SECDMachine with respect to the *environment* which is where user-supplied arguments go. An example invocation is

```
((machine '(LDC 10 LDC 20 ADD WRITEC)) '())
```

where the arguments to the machine are the arithmetic example of table 2 and an empty environment respectively.

To stage our interpreter we annotate terms that we want to be able to generate code for with the stage-polymorphic *maybe-lift* operators (defined in figure 5). With our division in place (see table 1) we simply wrap in calls to *maybe-lift* all constants that potentially interact with dynamic values and all expressions that add elements to the stack, environment or dump. Figure 12 shows these preliminary annotations. We wrap the initializing call to the SECD machine in *maybe-lift* as well because we want

to specialize the machine without the dynamic input of the environment provided yet. Hence line 12 in figure 12 simply signals the PE to generate code for the curried SECD machine.

```
1  (let machine (lambda _ stack (lambda _ dump (lambda _ control (lambda _ environment
2      (if (eq? 'LDC (car control))
3          (((((machine (cons (maybe-lift (cadr control)) stack)) dump) (cdr control)) environment)
4      (if (eq? 'DUM (car control))
5          (((((machine stack) dump) (cdr control)) (cons (maybe-lift '()) environment))
6      (if (eq? 'WRITEC (car control))
7          (car s)
8      ...
9      ...)))))))
10     (let initStack '()
11     (let initDump '()
12         (lambda _ ops (maybe-lift (((machine initStack) initDump) ops)))))))
```

Figure 12: Annotated version of the SECD interpreter in figure 11 with differences highlighted in green. The function *maybe-lift* is used to signal to the PE that we want to generate code for the wrapped expression. Here we follow exactly the division of table 1. As discussed in section 5.2.2 these changes are not enough to fully stage the machine

This recipe is not enough, however, because of the conflicting nature of our SECD machine's stepwise evaluation with TDPE's static reduction by evaluation. To progress in partially evaluating the machine we must take state-transition steps and essentially execute it at PE time. A consequence of this is that the PE can get into a situation where dynamic values are evaluated in static contexts potentially leading to undesired behaviour such as non-termination at specialization time (see 5.2.3 for more details). Where we encountered this particularly often is the accidental lifting of SECD instructions or specialization of recursive SECD function calls.

Key to us removing interpretative overhead of the SECD machine is the elimination of its unnecessary instruction dispatch logic from the specialized code, whose effect on interpreter efficiency was studied extensively by Ertl et al. [32]. Since the SECD program is known at PE time and thus has static binding time, we do not want to lift the constants against which we compare the control register. However, if we put something into the control register that is dynamic we are suddenly comparing dynamic and static values which is a specialization time error at best and non-termination of the PE at worst.

Another issue we dealt with in the process of writing the staged SECD interpreter is the implementation of the **RAP** instruction which is responsible for recursive applications. The instruction essentially works in two steps. First the user creates two closures on the stack. One which holds the recursive

function definition and another which contains a function that initiates the recursive call and prepares any necessary arguments. **RAP** calls the latter and performs the subtle but crucial next step. It forms a knot in the environment such that when the recursive function looks up the first argument in the environment it finds the recursive closure. According to Kogge's [13] description of the SECD operational semantics this requires an instruction that is able to mutate variables, a requirement that subsequent abstract machines such as the CESK machine [33] do not impose. Given the choice between adding support for an underlying `set-car!` instruction in $\lambda_{\uparrow\downarrow}$ or extending the SECD machine such that recursive functions applications do not require mutation in the underlying language we decided to experiment on the latter.

5.2.3 Tying the Knot

We now provide a substantial redesign to the internal `RAP` calling convention while keeping the semantics of the instruction in tact in order to allow SECD style recursive function calls without the need for mutable variables and more crucially enable their partial evaluation. The idea is to wrap the recursive SECD instructions in a closure at the LISP-level, perform residualization on the closure and distinguish between returning from a regular as opposed to recursive function to ensure termination of our specializer. We demonstrate the issue of partially evaluating a recursive call in the standard SECD semantics on the example in figure 13 which shows a recursive function that decrements a user provided number down to zero.

```
1  DUM NIL LDF ;Definition of recursive function starts here
2      (LD (1 1)
3       LDC 0 EQ ;counter == 0?
4       SEL
5         (LDC done STOP) ;Base case: Push "done" and halt
6         (NIL LDC 1 LD (1 1) SUB CONS LD (2 1) AP JOIN) ;Recursive Case: Decrement counter
7       RTN)
8      CONS LDF
9      (NIL LD (3 1) CONS LD (2 1) AP RTN) ;Set up initial recursive call
10     RAP
```

Figure 13: An example recursive function application annotated to show the issue with partially evaluating this type of construct.

Were we to specialize this program by simply evaluating the machine our PE would not terminate. The exit out of the recursive function (defined on line 1) occurs on line 5 but is guarded by a conditional check on line 3. This conditional compares a dynamic value (i.e., `LD (1 1)`) with a constant 0. By

virtue of our division's congruence the 0-literal and whole if-statement are classified as dynamic. However, for TDPE this dynamic check does not terminate the PE but instead attempts to reduce both branches of the statement. Since both branches are simply a recursive call of the *step* function we hit this choice again repeatedly without terminating because we have no way of signalling to stop partially evaluating. Figure 14 highlights this in the internals of the machine.

```
(if (eq? 'SEL (car control))
  (if (car stack) ;Do not know the result because value on stack is dynamic
    ;Make another step in machine. Will eventually hit this condition again
    ;because we are evaluating a recursive program
    (((((machine (cdr stack)) (cons (cdddr control) dump)) fns) (cadr control)) environment)
    (((((machine (cdr stack)) (cons (cdddr control) dump)) fns) (caddr control)) environment))
```

Figure 14: Snippet from the internals of the SECD interpreter from section 5.2.2. Highlighted are the locations at which our partial evaluator does not terminate. TDPE attempts to evaluate both branches because we cannot determine the outcome of the conditional. The variable *fns* denotes the F-register.

Instead of evaluating the recursive call, we want to instead generate the function definition and call in our residual program. What we now need to solve is how one can produce residual code for these SECD instructions that are to-be-called recursively. The key to our approach is to reuse $\lambda_{\uparrow\downarrow}$'s ability to lift closures. Figure 15 shows the modifications to the operational semantics of Landin's SECD machine [12] which allow it to be partially evaluatable with a TDPE and do not require a `set-car!` in the underlying language.

Before explaining our modifications to the SECD instruction set in detail we briefly outline four significant changes to note:

1. SECD functions are now lambdas that wrap a call to the implicit state transition
2. **RAP** semantics are adapted to an alternative way to achieve recursion since *tying the knot* in the environment is not possible due to a lack of a *set-car!* operation
3. The **RTN** instruction aids termination of the specializer by preventing state transitions upon detection of appropriate tags on D
4. Instructions **AP/RTN/LIFT** all feature calls to a *lift* operator that follows the division established earlier in table 1

$$\text{s e (}\mathbf{LDF}\text{ ops.c) d f} \longrightarrow ((\lambda\text{e'.(run@('() e' ops 'ret f))) ops.e).s e c d f} \tag{15}$$

$$\begin{aligned}
\text{(entryClo recClo.s) e (}\mathbf{RAP}\text{.c) d f} &\longrightarrow \text{'() e entryOps (s e c f.d) (rec (mem.recEnv)).f} \qquad (16)\\
\text{where (entryFn (entryOps entryEnv))} &:= \text{entryClo}\\
\text{(recFn (recOps recEnv))} &:= \text{recClo}\\
\text{rec} &:= \lambda\text{env.(run@('() env recOps 'ret (rec (mem.recEnv)).f))}\\
\text{mem} &:= \text{((s e c f.d) (recOps.recEnv).f)}
\end{aligned}$$

$$\text{s e (}\mathbf{LDR}\text{ (i j).c) d f} \longrightarrow \text{(locate@(i j f)).s e c 'fromldr.d f} \tag{17}$$

$$\begin{aligned}
\text{v.s e (}\mathbf{RTN}\text{.c) (s' e' c' d' f'.d) f} \longrightarrow\ & \text{lift@v} \quad \textit{if d-register is tagged with 'ret} \qquad (18)\\
& \lambda\text{x.(fn@(lift@(x.env)))}\\
& \qquad \textit{if d-register is tagged with 'fromldr}\\
& \qquad \text{where (fn (ops env)) := v}\\
& \text{(v.s') e' c' d' f'} \qquad \textit{otherwise}
\end{aligned}$$

$$\begin{aligned}
\text{s' e (}\mathbf{AP}\text{.c) env' f} \longrightarrow\ & \text{(fn@(lift@(x.env))).s e c d f} \qquad (19)\\
& \text{where ((fn (ops.env)) x.s) := s'}
\end{aligned}$$

$$\begin{aligned}
\text{(v.s) e (}\mathbf{LIFT}\text{.c) d f} \longrightarrow\ & \text{res.s e c d f} \qquad (20)\\
\text{where res} := \ & \text{lift@v} \quad \textit{if (num? v) or (sym? v)}\\
& \text{lift@(}\lambda\text{x.(run@('() x.env ops 'ret f)))} \quad \textit{if (clo? v)}\\
& \text{where (fn (ops env)) := v}
\end{aligned}$$

Figure 15: Modifications to the SECD operational semantics by Kogge [13]. The original transitions are shown in figure A.1. The function *run* takes state-transition steps according to *C* until it halts. The function *locate* returns an element at index (i,j) from a multi-dimensional list. An "@" denotes function application and registers *S, E, C, D, F* are represented by *s, e, c, d, f* respectively. The syntax *num?/sym?/clo?* are conditions satisfied when their argument is a number, string or a closure respectively. An "_" is a void argument to a function.

Firstly, equation 15 augments the representation of functions in the SECD machine (simply lists of instructions) with a thunk that accepts an environment and upon invocation runs the abstract machine with the instructions put into the C-register by **LDF**. Working with thunks makes the necessary changes to stage the machine less intrusive and effectively prevents the elements of the control register being marked as dynamic. This is in line with the ideas of Danvy et al. [24] which showed that eta-expansion can enable partial evaluation by hiding dynamic values from static contexts.

Note also that we add a new functions register, which we refer to as the **F-register**[2] which is responsible for holding the recursive instructions of a **RAP** call. In the traditional SECD machine both the recursive and the calling function are kept in the environment and then loaded on the stack using **LD**, subsequently called using **AP**. However, for simplicity we keep recursive functions in F to distinguish them from free variables in E and aid debuggability. Thus we introduce a new **LDR** instruction that returns the contents of the F-register by index, just as **LD** does for the E-register.

Modification to the **RAP** instruction are described by (16). As in the original semantics it still expects two closures on top of the stack: one that performs the initial recursive call which we refer to as *entryClo* and another that represents the actual set of instructions that get called recursively, *recClo*. Each closure consists of a function (*entryFn* or *recFn*), the SECD instructions these functions execute (*entryOps* or *recOps*) and an environment (*entryEnv* or *recEnv*). **RAP** then saves the current contents of all registers on D and appends a closure to F. This closure when applied to an environment, runs the machine with the control-register containing instructions of the recursive function body and a self-reference to the closure. Additionally, applying the closure places a *'ret* tag onto the dump-register which is later used as an indicator to stop evaluating the current call; this is crucial to aid termination during specialization time.

In the original semantics of SECD, **RTN** would restore the state of all registers from the dump and add the top most value of the current S-register back onto the restored S-register. This modelled the return from a function application. As we showed earlier in this section, taking another step in the machine when specializing a recursive function will lead to non-termination of our specializer. Thus, we simply stop evaluation when returning from a function by tagging the register with a *'ret* symbol and returning the top-most value on the stack to the call site of a lambda. This works because function definitions reside in lambdas in the interpreter now and SECD function application is lambda

[2] In his description of the SECD machine Kogge [13] uses F to label the register that holds a list of free memory locations in the machine. We do not require such register in our implementation and thus repurpose it to store recursive function definitions

invocation. The last case we are concerned with is the currying of SECD functions. This occurs when we invoke a **RTN** immediately after an **LDR**. To properly return a lambda we construct a **LDF**-style closure, lift and then return it.

Finally, we modify **AP** to adhere to the new calling convention of SECD functions required by the thunks that **RAP/LDF/LDR** add onto the stack. Where previously **AP** would call a function by simply reinstating instructions from S into C, now **AP** initiates a call to the lambda that we wrapped **LDF**'s instructions in. We pass a lifted environment and top of the stack to the function in case it is dynamic.

To rewrite the example from figure 13 with the new semantics we load the recursive function using the new **LDR** instead of the **LD** instruction as highlighted in figure 16.

```
1 DUM NIL LDF
2     (LD (1 1)
3      LDC 0 EQ
4      SEL
5        (LDC done STOP)
6        (NIL LDC 1 LD (1 1) SUB CONS LDR (1 1) AP JOIN)
7      RTN)
8     CONS LDF
9     (NIL LD (2 1) CONS LDR (1 1) AP RTN) RAP))
```

Figure 16: Recursive countdown example from figure 13 rewritten with the SECD operational semantics in figure 15

The above changes to the machine show that to permit partial evaluation of the original SECD semantics, an intrusive set of changes which necessitate knowledge of the inner workings of the machine are required. The complexity partially arises from the fact that the stack-based semantics do not lend themselves well to TDPE through $\lambda_{\uparrow\downarrow}$. We have to convert representations of program constructs, particularly closures, from how SECD stores them to what the underlying PE expects and is able to lift. Since $\lambda_{\uparrow\downarrow}$ is built around lifting closures, literals and cons-pairs we have to wrap function definitions in thunks which complicates calling conventions within the machine. Additionally, deciding on and implementing a congruent division for a SECD-style abstract machine, where values can move between a set of stack registers, requires careful bookkeeping of non-recursive versus recursive function applications and online binding-time analysis checks. On one hand, the most efficient code is generated by allowing as much of the register contents to be static. On the other hand, the finer-grained the division the more difficult to reason about and potentially less extensible a division becomes.

5.2.4 SECD Compiler

To continue the construction of a tower where each level is performing actual interpretation of the level above we would have to implement an interpreter written in SECD instructions as the next level in the tower. To speed up the development process and aid debuggability we implement compiler that parses a LISP-like language, which we refer to as *SecdLisp*, and generates SECD instructions. It is based on the compiler described by Kogge [13] though with modifications (see figure 17) to support our modified calling conventions and additional registers described in section 5.2.3. Since we hold recursive function definitions in the F-register we want to index into it instead of the regular environment register that holds variable values. We keep track of and increment an offset into the E-register during compilation whenever a free variable is detected via a missed look-up in the environment. Additionally, we need to make sure our compiler supports passing values from the user through the environment. The **quote** built-in (equation 23) is used to build lists of identifiers from s-expressions. This is useful when we extend the tower in later sections and want to pass SecdLisp programs as static data to the machine.

Syntax : ⟨identifier⟩ (21)
Code : `(LDR (i, j))` if lookup is in a **letrec**
where `(i,j)` is an index into the **F-register**
`(LD (i, j))` otherwise
where `(i,j)` is an index into the **E-register**

Syntax : (**lift** ⟨expr⟩) (22)
Code : ⟨expr⟩ `LIFT`

Syntax : (**quote** ⟨expr⟩) (23)
Code : `LDC` ⟨id_0⟩ `LDC` ⟨id_1⟩ `CONS ... LDC` ⟨id_{n-1}⟩ `LDC` ⟨id_n⟩ `CONS`
where ⟨id_n⟩ is the nth identifier in the string representing ⟨expr⟩

Figure 17: Modifications to the SECD compiler described by Kogge [13]

Given a source program in SecdLisp we invoke the compiler as shown in figure:

```
1         val instrs = compile(parseExp(src))
2         val instrSrc = instrsToString(instrs, Nil, Tup(Str("STOP"), N)))
3         ev(s"(\$secd_source '(\$instrSrc))")
```

As line 3 suggests we feed the compiled SECD instructions to the SECD machine interpreter source described in section 5.2.2 and begin interpretation or partial evaluation through a call to `ev` which is the entry point to $\lambda_{\uparrow\downarrow}$. Thus we still effectively maintain our tower and simply use the SecdLisp compilation step as a tool to generate the actual level in the tower in terms of SECD instructions more conveniently.

5.2.5 Example

Figure 18a shows a program to compute factorial numbers recursively written in SecdLisp. The program is translated into SECD instructions by our compiler (see section 5.2.4) and then input to our staged machine. Figure 18c is the corresponding residualized program generated by $\lambda_{\uparrow\downarrow}$(and prettified to LISP syntax). An immediate observation we can make is that the dispatch logic of the SECD interpreter has been reduced away successfully. Additionally, we see the body of the recursive function being generated in the output code thanks to the modifications to **RAP**, **AP** and **LDF**. The residual program contains two lambdas, one that executes factorial and another that takes input from the user through the environment (line 25). In the function body itself (lines 4 to 20), however, the numerous *cons* calls and repeated list access operations (*car*, *cdr*) indicate that traces of the underlying SECD semantics are left in the generated code and cannot be reduced further without changing the architecture of the underlying machine.

5.3 Level 4: M_e

Armed with a staged SECD machine and a language to target it with, we build the next interpreter in the tower that gets compiled into SECD instructions. The interpreter defines a language called M_e. Its syntax is described in figure 19. The language is based on Jones et al.'s toy language M in their demonstration of the Mix partial evaluator [29] in the sense that it is a LISP derivative and serves as a demonstration of evaluating a non-trivial program through our staged SECD machine. The main difference is that we support higher-order functions. M_e also enables the possibility of implementing substantial user-level programs and further levels in the tower. The reason for choosing a LISP-like language syntax again is that it allows us to reuse $\lambda_{\uparrow\downarrow}$ LISP front-end's parsing infrastructure. Further work would benefit from changing representation of data structures like closures to increase the semantic gaps between $\lambda_{\uparrow\downarrow}$ and M_e and demonstrate even more heterogeneity than in the tower we built.

M_e supports the traditional functional features such as recursion, first-class functions, currying but also LISP-like quotation. We implement the language as a case-based interpreter shown in figure 20. Note that to reduce complexity in our implementation we define our interpreter within a Scala string. Line 1 starts the definition of a function, `meta_eval`, that allows us to inject a string representing the M_e program and another representing the implementation of a **lift** operator. This mimics the

```
(letrec (fact)
    ((lambda (n m)
        (if (eq? n 0)
            m
            (fact (- n 1) (* n m)))))
        (fact 10 1))
```

(a) LISP Front-end

```
DUM NIL LDF
    (LDC 0 LD (1 1) EQ SEL
        (LD (1 2) JOIN)
        (NIL LD (1 2) LD (1 1) MPY CONS
            LDC 1 LD (1 1) SUB CONS LDR (1 1) AP
            JOIN)
        RTN)
    CONS LDF
    (NIL LDC 1 CONS LDC 10 CONS
        LDR (1 1) AP RTN) RAP STOP
```

(b) SECD Instructions

```
1  (let x0
2    (lambda f0 x1 <=== Takes user input
3      (let x2
4        (lambda f2 x3 <=== Definition of factorial
5          (let x4 (car x3)
6          (let x5 (car x4)
7          (let x6 (eq? x5 0)
8          (if x6
9            (let x7 (car x3)
10           (let x8 (cdr x7) (car x8)))
11           (let x7 (car x3)
12           (let x8 (cdr x7)
13           (let x9 (car x8)
14           (let x10 (car x7)
15           (let x11 (* x10 x9)
16           (let x12 (- x10 1)
17           (let x13 (cons x11 '.)
18           (let x14 (cons x12 x13)
19           (let x15 (cons '. x1)
20           (let x16 (cons x14 x15) (f2 x16)))))))))))))))) <=== Recursive Call
21     (let x3 (cons 1 '.)
22     (let x4 (cons 10 x3)
23     (let x5 (cons '. x1)
24     (let x6 (cons x4 x5)
25     (let x7 (x2 x6) (cons x7 '.)))))))) (x0 '.))
```

(c) Prettified Generated Code

Figure 18: Example Factorial

$$
\begin{aligned}
\langle \text{program} \rangle &::= \langle \text{exp} \rangle \\
\langle \text{exp} \rangle &::= \langle \text{variable} \rangle \\
&\mid \langle \text{literal} \rangle \\
&\mid (\text{lambda}\ \ (\langle \text{variable} \rangle)\ \langle \text{exp} \rangle) \\
&\mid (\langle \text{exp} \rangle\ \langle \text{exp} \rangle) \\
&\mid (op_2\ \langle \text{exp} \rangle\ \langle \text{exp} \rangle) \\
&\mid (\text{if}\ \langle \text{exp}_{\text{cond}} \rangle\ \langle \text{exp}_{\text{conseq}} \rangle\ \langle \text{exp}_{\text{alt}} \rangle) \\
&\mid (\text{let}\ (\langle \text{variable} \rangle)\ (\langle \text{exp} \rangle)\ \langle \text{exp}_{\text{body}} \rangle) \\
&\mid (\text{letrec}\ (\langle \text{variable} \rangle)\ (\langle \text{exp}_{\text{recursive}} \rangle)\ \langle \text{exp}_{\text{body}} \rangle) \\
&\mid (\text{quote}\ \langle \text{exp} \rangle) \\
\langle \text{variable} \rangle &::= \text{ID} \\
\langle \text{literal} \rangle &::= \text{NUM} \mid \text{'ID} \\
op_2 &::= \text{and} \mid \text{or} \mid - \mid + \mid * \mid < \mid \text{eq?}
\end{aligned}
$$

Figure 19: Syntax of M_e which gets compiled into SECD instructions for interpretation by the SECD machine

polymorphic **maybe-lift** we define in $\lambda_{\uparrow\downarrow}$.

Figure 21 shows the M_e interpreter running a program computing factorial using the Y-combinator for recursion (figure 21a) on our staged SECD machine. As opposed to producing an optimal residual program we now see the dispatch logic of our M_e interpreter in the generated code (figure 22). As the programmer we know this control flow can be reduced even further since the M_e source program is static data.

```
def meta_eval(program: String, lift: String = "(lambda (x) x)") =   s"
    (letrec (eval) ((lambda (exp env)
        (if (sym? exp)
            (env exp)
        (if (num? exp)
            ($lift exp)
        (if (eq? (car exp) '+)
            (+ (eval (cadr exp) env) (eval (caddr exp) env))
        ...
        ...
        (if (eq? (car exp) 'lambda)
            ($lift (lambda (x)
                (eval (caddr exp)
                    (lambda (y) (if (eq? y (car (cadr exp)))
                                    x
                                    (env y))))))
                ((eval (car exp) env) (eval (cadr exp) env)))))))))))))
        (eval (quote $program) '()))
```

Figure 20: Staged interpreter for M_e

5.3.1 Staging M_e and Collapsing the Tower

In an effort to further optimize our generated code from the example in figure 21 we stage the M_e interpreter. As indicated by Amin et al. during their demonstration of collapsing towers written in Pink [7], staging at the user-most level of a tower of interpreters should yield the most optimal code. In this section we aim to demonstrate that staging at other levels than the top-most interpreter does indeed generate less efficient residual programs.

Staging the M_e interpreter is performed just as in Pink by lifting all literals and closures returned by the interpreter and letting $\lambda_{\uparrow\downarrow}$'s evaluator generate code of operations performed on them. The main caveat unique to M_e's interpreter is a consequence of heterogeneity: M_e does not have access to a builtin *lift* operator. This poses the crucial question of how one can propagate the concept of *lifting expressions* through levels of the tower without having to expose it at all levels. We take the route of making a *lift* operator available to the levels above the SECD machine which requires the implementation of a new SECD **LIFT** instruction. Further work could investigate other possibilities of passing binding-time information through interpreter boundaries.

```
((lambda (fun)
          ((lambda (F)
             (F F))
           (lambda (F)
             (fun (lambda (x) ((F F) x))))))

        (lambda (factorial)
          (lambda (n)
            (if (eq? n 0)
                1
                (* n (factorial (- n 1)))))))
```

(a) LISP Front-end

```
DUM NIL LDF
    (LD (1 1) SYM? SEL
            (NIL LD (1 1) CONS LD (1 2) AP JOIN) (LD (1 1) NUM? SEL
    (NIL LD (1 1) CONS LDF
            (LD (1 1) RTN) AP JOIN) (LDC + LD (1 1) CAR EQ SEL
    (NIL LD (1 2) CONS LD (1 1) CADDR CONS LDR (1 1) AP NIL LD (1 2) CONS LD (1 1)
       CADR CONS LDR (1 1) AP ADD JOIN) (LDC - LD (1 1) CAR EQ SEL
    (NIL LD (1 2) CONS LD (1 1) CADDR CONS LDR (1 1) AP NIL LD (1 2) CONS LD (1 1)
       CADR CONS LDR (1 1) AP SUB JOIN) (LDC * LD (1 1) CAR EQ SEL
       ...
       JOIN) JOIN) JOIN) JOIN) RTN) CONS LDF
       ...
       LDC 1 CONS LDC n CONS LDC - CONS CONS LDC factorial CONS CONS
       LDC n CONS LDC * CONS CONS LDC 1 CONS LDC . LDC 0 CONS LDC n CONS
       LDC eq? CONS CONS LDC if CONS CONS LDC . LDC n CONS CONS LDC lambda
       ...
       LDR (1 1) AP RTN ) RAP STOP
```

(b) SECD Instructions

Figure 21: Example factorial on M_e

```
(let x0
  (lambda f0 x1
    (let x2
      (lambda f2 x3
        (let x4 (car x3)
        (let x5 (car x4)
        (let x6 (sym? x5)
        (if x6
          ...
          (let x8 (car x7)
          (let x9 (num? x8)
          (if x9
            ...
            (let x12 (car x11)
            (let x13 (eq? x12 '+)
            (if x13
              (let x14 (car x3)
              ...
              (let x28 (cons x27 x23)
              (let x29 (f2 x28) (+ x29 x25))))))))))))))))))
              ...
              (let x17 (eq? x16 '-)
              (if x17
                (let x18 (car x3)
                ...
```

Figure 22: Generated code running the example in 21 on a staged SECD machine. Traces of the M_e's dispatch logic is highlighted in green.

The state transitions for the **LIFT** operation in the staged SECD machine are shown in (20). The intended use of the instruction is to signal $\lambda_{\uparrow\downarrow}$ to lift the top of the stack. We do this by dispatching to the builtin *lift* operator provided by SecdLisp. Thus running following on our SECD machine,

```
LDC 10 LIFT STOP
```

would generate a $\lambda_{\uparrow\downarrow}$ code expression representing the constant 10,

```
Code(Lit(10))
```

The other case that our **LIFT** operates on are closures constructed via **LDF** or **RAP**. Behind the apparent complexity again lies the same recipe for staging an interpreter as we identified before but in this case operating on the top most value of the stack. We make sure to lift the operand if it is a number or a string. In the case that the operand is a closure we construct, lift and return a new lambda using the state we stored in registers F and D. We simply construct a lambda that takes an environment and runs the instructions that were wrapped to completion.

Through the addition of a *lift* built-in into SecdLisp we can now residualize the M_e interpreter and run it on our SECD interpreter. The residual program for the factorial example (figure 21) is shown in figure 24 and the corresponding SECD instructions that M_e compiled down to in figure 23. The generated SECD instructions are the same as in the unstaged M_e interpreter with the exception of the newly inserted **LIFT** instructions as we have specified in the interpreter definition. This has the effect that the residual program resembles exactly the M_e definition of our program but now in terms of $\lambda_{\uparrow\downarrow}$ and all traces of the SECD machine have vanished. This demonstrates that we successfully removed all layers of interpretation between the base evaluator ($\lambda_{\uparrow\downarrow}$) and the user-most interpreter (M_e). Comparing this configuration to running our example on the staged machine (and unstaged M_e) we can see that the structure of the generated code resembles the structure of the interpreter that we staged. When staging at the SECD level we could see traces of stack-like operations that to the programmer seemed optimizable. When we stage at the M_e layer these operations are gone and we are left with LISP-like semantics of M_e.

```
DUM NIL LDF
    (LD (1 1) SYM? SEL <=== M_e Dispatch Logic
         (NIL LD (1 1) CONS LD (1 2) AP JOIN )
    (LD (1 1) NUM? SEL
        (LD (1 1) LIFT JOIN ) <=== Lift literals
  ...
 (LDC letrec LD (1 1) CAR EQ SEL
     (NIL NIL LDF
         (LD (2 1) CADR CAR LD (1 1) EQ SEL
             (LD (12 1) LIFT JOIN) <=== Lift recursive lambdas
  ...
  (LDC lambda LD (1 1) CAR EQ SEL
         (LDF (NIL LDF
             (LD (3 1) CADR CAR LD (1 1) EQ SEL
                 (LD (2 1) JOIN) (NIL LD (1 1) CONS LD (3 2) AP JOIN) RTN)
                    CONS LD (2 1) CADDR CONS LDR (1 1) AP RTN) LIFT JOIN) <=== Lift lambdas
  ...
```

Figure 23: SECD instructions for example an factorial on a staged M_e interpreter

5.4 Level 5: String Matcher

Following the experiments performed in Amin et al.'s study of towers [7], we extend our tower further one last time and implement a regular expression matcher proposed by Kernighan et al. [34] in M_e. The source is shown in figure 25. It returns the string *'yes* on a successful match and *'no* otherwise. The string *done* marks the end of a pattern or input string to help the matcher terminate.

We then collapse two different configurations of the tower: (1) Staged M_e interpreter running the plain matcher (2) Unstaged M_e interpreter running a staged version of the matcher. The pattern we specialize against is

```
'(a * done)
```

which should match zero or more occurrences of character "a". Logically, this pattern will match any string and thus the optimal specialized version of the matcher should simply return a "yes" on any input indicating a successful match.

The residualized program when we collapse the tower while staging the M_e interpreter is presented in figure 27a. It is far from the most efficient version and we can see clear traces of the matcher logic in the generated code such as a check for an "_" character on line 21 while our pattern against which we specialize does not contain any.

```
(lambda f0 x1
  (let x2
    (lambda f2 x3
      (let x4
        (lambda f4 x5 <=== Definition of factorial
          (let x6 (eq? x3 0)
          (let x7
            (if f4 1
              (let x7 (- x3 1)
              (let x8 (x1 x5)
              (let x9 (* x3 x6) x7)))) x5))) f2))
  (let x3
    (lambda f3 x4 <=== Definition of Y-combinator
      (let x5
        (lambda f5 x6
          (let x7
            (lambda f7 x8
              (let x9 (x4 x4)
              (let x10 (f7 x6) x8)))
          (let x8 (x2 f5) x6)))
      (let x6
        (lambda f6 x7
          (let x8 (x5 x5) f6))
      (let x7 (x4 f3) x5))))
  (let x4 (x1 f0)
  (let x5 (x2 6) x3)))))
```

Figure 24: Prettified Residual Program in $\lambda_{\uparrow\downarrow}$for an example factorial on a staged M_e interpreter

```
(letrec (star_loop) ((lambda (m) (lambda (c) (letrec (inner_loop)
                        ((lambda (s)
                            (if (eq? 'yes (m s)) 'yes
                            (if (eq? 'done (car s)) 'no
                            (if (eq? '_ c) (inner_loop (cdr s))
                            (if (eq? c (car s)) (inner_loop (cdr s)) 'no))))))
                        inner_loop))))
    (letrec (match_here) ((lambda (r) (lambda (s)
                (if (eq? 'done (car r))
                    'yes
                    (let (m) ((lambda (s)
                        (if (eq? '_ (car r))
                            (if (eq? 'done (car s))
                                'no
                                ((match_here (cdr r)) (cdr s)))
                            (if (eq? 'done (car s)) 'no
                            (if (eq? (car r) (car s))
                                ((match_here (cdr r)) (cdr s))
                                'no)))))
                        (if (eq? 'done (car (cdr r))) (m s)
                        (if (eq? '* (car (cdr r)))
                            (((star_loop (match_here (cdr (cdr r)))) (car r)) s)
                            (m s))))))))
                (let (match) ((lambda (r)
                    (if (eq? 'done (car r))
                        (lambda (s) 'yes)
                        (match_here r))))
                        match))
```

Figure 25: Unstaged regular expression (RE) matcher written in M_e. The matcher checks whether a string satisfies a given RE pattern containing letters, underscores or wildcards.

Now we stage the matcher according to the implementation provided in the Pink experiments [7] by simply lifting all symbols on return from the matcher and the initial recursive call to begin matching (see figure 26).

```
(letrec (star_loop) ((lambda (m) (lambda (c) (letrec (inner_loop)
                        ((lambda (s)
                            (if (eq? 'yes (m s)) (lift 'yes)
                            (if (eq? 'done (car s)) (lift 'no)
                            ...
    (letrec (match_here) ((lambda (r) (lambda (s)
                (if (eq? 'done (car r))
                    (lift 'yes)
                    ...
                    (lift (lambda (s) 'yes))
                    (lift (match_here r)))))
                    match))
```

Figure 26: Staged regular expression matcher by wrapping returned symbols in M_e's *lift*.

Continuing the trend, the generated code when staging the user-most interpreter (in this case the string matcher) yields the optimal residual program. As we wanted, the specialized matcher in figure 27b will succeed on any input string.

```
(lambda f0 x1
  (let x2
    (lambda f2 x3
      (let x4 (car x3)
      (let x5 (car x4)
      (let x6 (eq? 'done x5)
      (if x6
        (lambda f7 x8 'yes)
        (let x7 (car x3)
        (let x8
          (lambda f8 x9
            (lambda f10 x11
              (let x12 (car x9)
              (let x13 (car x12)
              (let x14 (eq? 'done x13)
              (if x14 'yes
                (let x15
                  (lambda f15 x16
                    (let x17 (car x9)
                    (let x18 (car x17)
                    (let x19 (eq? '_ x18)
                    (if x19
                      (let x20 (car x16)
                      (let x21 (car x20)
                      (let x22 (eq? 'done x21)
                      (if x22 'no
                      ...
```

(a) Residual program when collapsing our experimental tower while staging at the M_e level.

```
(lambda f0 x1
  (let x2 (car x1)
  (let x3
    (lambda f3 x4
      (let x5 (car x4) 'yes))
  (let x4 (cons x2 '.) (x3 x4)))))
```

(b) Residual program when collapsing our experimental tower while staging at the regular expression matcher level.

6 Conclusions and Future Work

6.1 Conclusions

The aim of our study was to connect the extensive collection of work on reflective towers with their counterparts in more practical settings. Collapsing of towers of interpreters encompasses the techniques to remove interpretative overhead that is present in such systems. The construction of towers of interpreters has previously been either limited to reflective towers, in which each interpreter is meta-circular and exposes its internals for the purpose of reflection, or a consequence of modular systems design where layers of tools that perform interpretation of some form are glued together.

To the best of our knowledge, our work is one of a handful, together with Amin et al.'s previous explorations of heterogeneous towers [7], that explicitly focus on the overheads and optimization of towers of interpreters that are not meta-circular. We built on the ideas from the Pink framework and re-used its TDPE-based partial evaluator to construct our own experimental tower.

A tower of meta-circular interpreters can be collapsed into a residual program in a single pass by only staging a single interpreter in the tower and relying on the meta-circular definitions of *lift* to propagate binding-time information to the multi-level base evaluator which handles the actual code generation (in Pink through an embedded partial evaluator). This work started by asking the question of how a collapse of a tower can be achieved without a readily available *lift* and what difficulties could arise when an interpreter in the tower differs in its semantics from interpreters adjacent to it; we label towers that exhibit a combination of these properties as heterogeneous.

In figure 2b we imagined a hypothetical tower where an emulator written in JavaScript interpreting a Python interpreter finally runs some user-supplied program. In our proof-of-concept tower, depicted in figure 2a, we take the emulated machine to be a SECD machine for simplicity and the Python interpreter to be an interpreter for our toy functional language, M_e. Of course the individual levels of the tower differ substantially in complexity and ability to perform side-effects. However, in this study we mainly focused on the process of constructing and collapsing a tower with a simple but important property: heterogeneity.

We first chose a SECD interpreter to be the level that adds heterogeneity to the tower. The lack of a built-in lift operator and the difference in how it represents program constructs such as closures required us to adapt the internals of the machine to aid the termination and efficient residualization of

our TDPE. A challenge of TDPE that we addressed when staging our SECD interpreter is the lack of a general recipe for staging an abstract machine. Staging an interpreter amounts to reifying literals, lambdas and product types it returns. In an abstract machine the semantics are not guaranteed to distinguish these types by data structures or a type-system but can instead rely on dedicated instructions for each. Hence, the points to reify at are dictated by the architecture of the underlying machine. In our experiments we created a conservative division tailored to the SECD stack-registers and reduced static expressions in the TDPE reflect operator to achieve optimal residualization.

Removing a layer of interpretation requires a way of propagating the decision of whether to generate or evaluate an expression through levels in the tower which we tackle by implementing a **lift** operator at the level which previously did not support such an operation (in our case the SECD machine level). This required the implementation of a transformation from a SECD-style closure to the one that the level below it expects. Representation of closures and the semantics of recursive function applications in SECD proved problematic during the process of staging the machine. A lack of distinction between recursive and non-recursive function definitions meant that we had to devise a strategy to stop unfolding recursive calls to avoid non-termination of our PE. We tackled this by tagging recursive closures and signal the SECD return instruction to avoid another state transition.

We mainly focus on the overhead of the dispatch logic in an interpreter that decides which operation to perform based on the current term being evaluated. The interpretative cost we removed in a tower is that between the base evaluator and the last level that was staged. We used our experimental tower to investigate the effect of staging interpreters at various heights. Our results showed that the interpretative overhead of all levels up to the one being staged is completely reduced during specialization time. More notably, the structure of the generated code follows that of the interpreter that was staged. In our case the staging at the SECD machine level yielded generated code that contained traces of the SECD semantics including stack-based operations. Despite being the optimal output when specializing the SECD interpreter, the residual program could be reduced even further if it were not for the rigid architecture of the SECD machine.

Although we showed the successful collapse of a heterogeneous tower of interpreters, realizing our methodology on a practical setting such as the Python-x86-JavaScript tower will require additional work. Our approach to propagating the TDPE binding-time information involves the implementation of a reification operator in each interpreter that is missing it. This requires the deconstruction of the types that TDPE's *reify* operates on and conversion to the representation that the interpreter below

in the tower expects. These changes would require intimate knowledge of and intrusive changes to an interpreter. Additionally, in our experimental tower we do not consider the residualization of side-effects which a useful collapse procedure would need for wider applicability.

In smaller scale practical settings where towers consist of embedded DSL interpreters or regular expression matchers our experiments could help the optimization of such systems using the simple to implement TDPE even in the absence of meta-circularity and in the presence of layers of translators.

6.2 Future Work

We hope our study provided a platform and the necessary techniques to eventually make collapsing towers in practice a reality. The next step is to extend our definition of heterogeneity to investigate ways of dealing with side-effects at various levels of tower. The ability to perform side-effects such as destructive data structure changes are essential in real-world programs regardless of their domain but were not considered in our study. One of the considerations is whether side-effects should be residualized, removed or executed during PE time. More broadly a next step would be to devise a method of dealing with situations where a level does not have a necessary feature that an interpreter in a different level requires. Currently any feature, including side-effects, is implemented from the base up to the interpreter that uses it. Furthermore, Kogge presents various extensions to the SECD machine such as a call/cc operator, lazy evaluation or even concurrency (through MultiLisp) [13]. Implementing such extensions could aid the experimentation with features not being available at adjacent levels. Semantics such as call/cc allow us to emulate side-effects such as such as exceptions and non-determinism.

Nothing restricts our heterogeneous tower to using a SECD abstract machine. Instead further work could experiment with other abstract machine such as the Warren Abstract Machine (WAM) [35] as an alternative level. This would allow us to investigate the applicability of our method to collapse towers to other programming paradigms, in this case logic programming. Even in the presence of the SECD machine we could replace the interpreters running on it, in our case M_e, with higher-level logic programming interpreters instead of the lower-level WAM. This could lead into a study of stratifications of towers and the extent to which they are collapsible.

A major subject of focus in PE is the ability of a PE to output residual programs in a language different to the subject language or the one the PE was written in. This could prove useful when

staging between a fixed set of levels that is not the whole tower. Such a feature would need to be supported by the underlying PE methodology (i.e., TDPE in our case).

Ongoing work involves generalizing and making our TDPE methodology less intrusive. Instead of reimplementing a reify operation at the levels that need it, feasible techniques could, at least for particular domains or languages, pass the TDPE binding-time information in the form of data through each level. Whürthinger's GraalVM [36] allows the inter-communication of languages that target the Graal Virtual Machine and could prove useful in further experimenting with heterogeneous towers where different interpreters pass such information between each other.

Bibliography

[1] B. C. Smith, "Reflection and semantics in lisp," in *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1984, pp. 23–35.

[2] M. Wand and D. P. Friedman, "The mystery of the tower revealed: A nonreflective description of the reflective tower," *Lisp and Symbolic Computation*, vol. 1, pp. 11–38, 1988.

[3] O. Danvy and K. Malmkjaer, "Intensions and extensions in a reflective tower," in *Proceedings of the 1988 ACM conference on LISP and functional programming*. ACM, 1988, pp. 327–341.

[4] J. C. Sturdy, "A lisp through the looking glass." Ph.D. dissertation, University of Bath, 1993.

[5] K. Asai, S. Matsuoka, and A. Yonezawa, "Duplication and partial evaluation," *Lisp and Symbolic Computation*, vol. 9, no. 2-3, pp. 203–241, 1996.

[6] K. Asai, "Compiling a reflective language using MetaOCaml," *ACM SIGPLAN Notices*, vol. 50, no. 3, pp. 113–122, 2015.

[7] N. Amin and T. Rompf, "Collapsing towers of interpreters," *Proceedings of the ACM on Programming Languages*, vol. 2, p. 52, 2017.

[8] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification*. Pearson Education, 2014.

[9] O. Danvy, "Type-directed partial evaluation," in *Partial Evaluation: Practice and Theory*. Springer, 1999, pp. 367–411.

[10] N. G. De Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem," in *Indagationes Mathematicae (Proceedings)*, vol. 75, no. 5. Elsevier, 1972, pp. 381–392.

[11] L. C. Paulson, "Foundations of functional programming," 1995.

[12] P. J. Landin, "The mechanical evaluation of expressions," *The computer journal*, vol. 6, pp. 308–320, 1964.

[13] P. M. Kogge, *The architecture of symbolic computers*. McGraw-Hill, Inc., 1990.

[14] P. Henderson, *Functional programming: application and implementation*. Prentice-Hall, 1980.

[15] J. C. Reynolds, "Definitional interpreters for higher-order programming languages," in *Proceedings of the ACM annual conference-Volume 2*. ACM, 1972, pp. 717–740.

[16] Y. Futamura, "Partial evaluation of computation process–an approach to a compiler-compiler," *Higher-Order and Symbolic Computation*, vol. 12, no. 4, pp. 381–391, 1999.

[17] U. Jørring and W. L. Scherlis, "Compilers and staging transformations," in *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1986, pp. 86–96.

[18] E. Brady and K. Hammond, "A verified staged interpreter is a verified compiler," in *Proceedings of the 5th international conference on Generative programming and component engineering*. ACM, 2006, pp. 111–120.

[19] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[20] N. D. Jones, "Challenging problems in partial evaluation and mixed computation," *New generation computing*, vol. 6, pp. 291–302, 1988.

[21] A. Shali and W. R. Cook, "Hybrid partial evaluation," *ACM SIGPLAN Notices*, vol. 46, pp. 375–390, 2011.

[22] O. Danvy, "Online type-directed partial evaluation," BRICS, Technical Report RS-97-53, 1997.

[23] B. Grobauer and Z. Yang, "The second futamura projection for type-directed partial evaluation," *Higher-Order and Symbolic Computation*, vol. 14, pp. 173–219, 2001.

[24] O. Danvy, K. Malmkjær, and J. Palsberg, "The essence of eta-expansion in partial evaluation," *Lisp and Symbolic Computation*, vol. 8, pp. 209–227, 1995.

[25] J. Hatcliff, T. Mogensen, and P. Thiemann, *Partial Evaluation: Practice and Theory: DIKU 1998 International Summer School, Copenhagen, Denmark, June 29-July 10, 1998*. Springer, 2007.

[26] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, "The essence of compiling with continuations," in *ACM Sigplan Notices*, vol. 28. ACM, 1993, pp. 237–247.

[27] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," EPFL, Tech. Rep., 2004.

[28] G. Ofenbeck, T. Rompf, and M. Püschel, "Staging for generic programming in space and time," in *ACM SIGPLAN Notices*, vol. 52. ACM, 2017, pp. 15–28.

[29] N. D. Jones, P. Sestoft, and H. Søndergaard, "Mix: a self-applicable partial evaluator for experiments in compiler generation," *Lisp and Symbolic computation*, vol. 2, pp. 9–50, 1989.

[30] O. Danvy, "A rational deconstruction of landins secd machine," in *Symposium on Implementation and Application of Functional Languages*. Springer, 2004, pp. 52–71.

[31] J. D. Ramsdell, "The tail-recursive secd machine," *Journal of Automated Reasoning*, vol. 23, pp. 43–62, 1999.

[32] M. A. Ertl and D. Gregg, "The structure and performance of efficient interpreters," *Journal of Instruction-Level Parallelism*, vol. 5, pp. 1–25, 2003.

[33] M. Felleisen, "The calculi of lambda-v-cs conversion: A syntactic theory of control and state in imperative higher-order programming languages," Ph.D. dissertation, Indiana University, 1987.

[34] B. W. Kernighan, "A regular expression matcher," *Oram and Wilson [OW07]*, pp. 1–8, 2007.

[35] D. H. Warren, "An abstract prolog instruction set," *Technical note 309*, 1983.

[36] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One vm to rule them all," in *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, 2013, pp. 187–204.

Appendices

Appendix A

SECD

s e (**NIL**.c) d ⟶ (nil.s) e c d
s e (**LDC** x.c) d ⟶ (x.s) e c d
s e (**LD** (i.j).c) d ⟶ (locate((i.j),e).s) e c d
where locate((i.j), lst) returns the element at
the ith row and jth column in the multi-dimensional list "lst"

(a.s) e (***OP***.c) d ⟶ ((*OP* a).s) e c d)
where *OP* is one of CAR, CDR, ...
(a b.s) e (***OP***.c) d ⟶ ((a *OP* b).s) e c d
where *OP* is one of CONS, ADD, SUB, MPY, ...

(x.s) e (**SEL** ct cf.c) d ⟶ s e c? (c.d)
where c? = ct if x ≠ 0, and cf if x = 0
s e (**JOIN**.c) (cr.d) ⟶ s e cr d

s e (**LDF** f.c) d ⟶ ((f.e).s) e c d
((f.e') v.s) e (**AP**.c) d ⟶ NIL (v.e') f (s e c.d)
(x.z) e' (**RTN**.q) (s e c.d) ⟶ (x.s) e c d

s e (**DUM**.c) d ⟶ s (nil.e) c d
((f.(nil.e)) v.s) (nil.e) (**RAP**.c) d ⟶ nil (set-car!((nil.e),v).e) f (s e c.d)
where set-car!(x, y) sets the first element of "x" to "y" and returns "x"

s e (**STOP**.c) d ⟶ halt the machine and return *s*
(x.s) e (**WRITEC**.c) d ⟶ halt the machine and return *x*

Figure A.1: SECD Machine instruction transitions mostly according to Kogge's description [13]. The instruction that causes a transition is in **bold**.

Assume: letrec f1 = A1 ... fn = An in E
= (λ f1 ... fn | E) A1 ... An

Code = (DUM NIL LDF (..code for An... RTN) CONS
LDF (..code for A1.. RTN) CONS
LDF (..code for E.. RTN) RAP)

Figure A.2: Kogge's [13] explanation of **RAP**'s semantics. A **letrec** gets translated into a series of SECD function definitions where the last one initiates a recursive call.