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

June 1, 2019

Declaration

I Michael Buch of Queens' College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 0

Signed: Michael Buch

Date: 10th June 2019

Acknowledgements

I would like to thank my supervisors, Dr. Nada Amin and Prof. Alan Mycroft, for their advice and continuous feedback throughout this work. In my regular discussions with Prof. Mycroft he helped me contextualize my thoughts and provided out-of-the-box ideas that formed the basis for much of the work I chose to pursue during the course of my thesis. His meticulous attention to detail and emphasis on clear and concise explanations were of great assistance in structuring and writing my dissertation.

My project progress and understanding of fundamental topics in partial evaluation would not have been possible without the great advice and patience of Dr. Amin. Not only has she helped me with tracking down bugs with incredible speed but shaped the direction of research that the project ended up taking. She assisted with the implementations and design of the tower of interpreters I describe in the report and her ideas, including the staging of a string matcher and the decision on which interpreters the tower consists of, helped keep my project, despite its experimental nature, on the right track.

Finally I would like to thank friends, family and the staff at the computer laboratory for the great support and fruitful discussions throughout the course of my studies and help broaden my horizon in the field of computer science.

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

13765 (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 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

[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

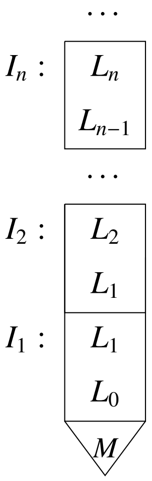


Figure 1: A tower of interpreters where each interpreter I_n is written in language L_{n-1} and interprets a language L_n, for some $n \geq 0$. In literature the tower often grows downwards, however, in our study we refer to I_0 as the base interpreter and grow the tower upwards for convenience. M is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

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 (JVM) [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 2 depicts two versions of our experimental tower. Traditionally reflective towers are thought of as completely vertical like the one on the left. However, details such as how a tower grows, shrinks and collapses while executing user programs worked rather mysteriously. We decided to implement our tower using occasional layers of compilation (as shown on the right). The two versions of our tower are extensionally equal since they yield the same output for a given program to evaluate. Part of our study is devoted to evaluating the effect of the intensional structure of towers on the act of collapsing them.

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 that are available at run-time versus 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 2.6, we can partially evaluate individual interpreters in a heterogeneous tower 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 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. We systematically describe the process by which we create a heterogeneous tower of interpreters and incrementally collapse it in sections 5.1 through 5.4. We conclude with an evaluation of our findings followed by a discussion of potential future work in section 6.

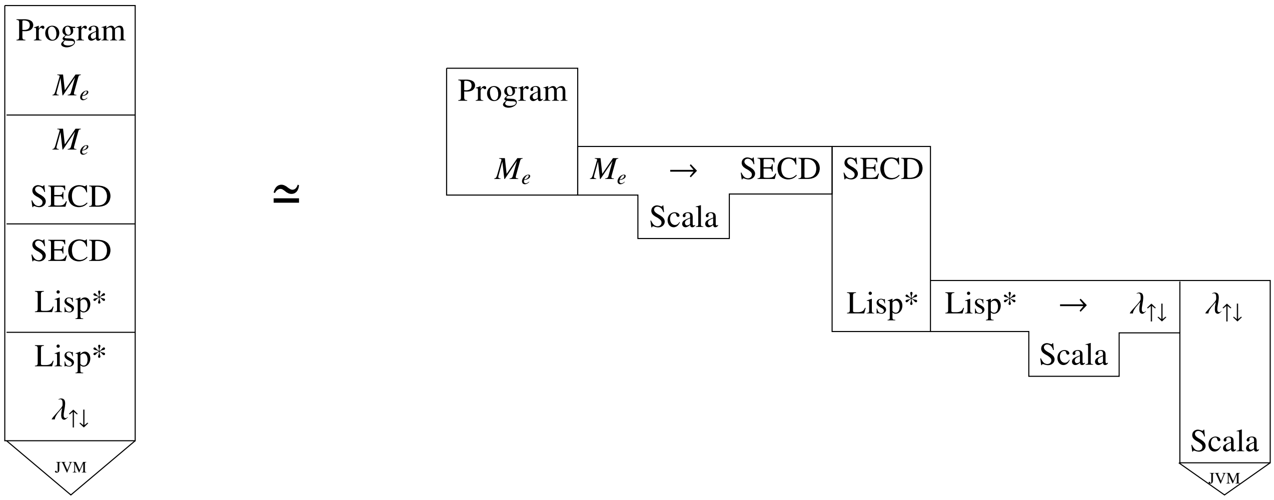

Program
M_e
M_e
SECD
SECD
Lisp*
Lisp*
$\lambda_{\uparrow\downarrow}$
JVM
≃
Program
M_e
M_e → SECD
Scala
SECD
Lisp*
Lisp* → $\lambda_{\uparrow\downarrow}$
Scala
$\lambda_{\uparrow\downarrow}$
Scala
JVM

Figure 2: Tombstone diagrams that represent two 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* in our diagram also encompasses any underlying machinery necessary to run it. 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 power of the *lift* operator described in section 2.6 and its vital role in collapsing heterogeneous towers.

2 Background

2.1 λ-Calculus and de Bruijn Indices

Terms in the untyped lambda calculus consist of variables, lambda abstractions and applications 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. Replacing occurrences of x with y brings the free variable, y, into a scope where a lambda already bound identifier y:

$$\lambda y.yy$$

Typically one would perform an α-conversion (i.e., rename variables) in the lambda appropriately before substitution to prevent a clash of variable names:

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

De Bruijn introduced a canonical lambda notation that prevents such variable name collisions and eliminates the need for α-conversions during β-reductions [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 2 for free variable y, the above example in de Bruijn index notation is:

$$(\lambda\lambda.1\;0)\;2$$

Variables and lambdas in $\lambda_{\uparrow\downarrow}$ (see section 2.7) follow the de Bruijn indexing scheme to avoid the complexity of managing variable names and their scopes.

Free variables in the SECD machine (section 2.3) are *2D de Bruijn indices*. An environment that maps variables to values is represented as a list of stack frames. Thus two indices (i, j) refer to a variable whose value is in environment, E, at the jth position within the ith stack frame. The above example with 2D indices is:

$$(\lambda\lambda.(0,1)\ \ (0,0))\ \ (1,2)$$

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*). Implementing letrecs is not as straightforward as the derivation of *let*-expressions from the λ-calculus and varies between language implementations. 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:

$$x = \mathit{fix}(f)$$

By definition of a fixed point we get an equation that resembles recursive function application:

$$\mathit{fix}(f) = f(\mathit{fix}(f))$$

An implementation of *fix* in the λ-calculus is called the *Y-combinator* and is defined as:

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

This combinator is often used to demonstrate recursion even in languages that do not support it. For some term, g, we can write a recursive application as follows:

$$Y(g) = g(Y(g))$$

We can now define a letrec using the Y-combinator:

```
  letrec f = λf.λx.e in e'
≡ let f = Y(λf.λx.e) in e'
```

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*. We can create a circular list, `start`, representing a stream of ones and zeroes as follows [2]:

```
let start = 0 :  end
and end = 1 :  start
```

Here, we set (i.e., tie together) the `end` of the cyclic list to point back to its `start` and lazy evaluation ensures we do not expand the definition indefinitely.

One can utilize this technique to efficiently implement recursion in a language where recursive definitions are retrieved from some environment upon application. The SECD machine creates a knot in its environment data structure for recursive function application using the **RAP** instruction [11]. For some function (`f`) SECD creates a closure taking an argument (`arg`), an expression (`exp`) and an environment (a list of free-variable values, `env`) where the first value in the environment is set to the address of `f` (denoted `&f`):

[2] https://wiki.haskell.org/Tying_the_Knot

```
Clo(arg, exp, &f.env)
```

Recursive applications of `f` now reuse its definition repeatedly with only a single definition residing in memory regardless of recursion depth. Despite being more memory efficient, this technique requires mutable data structures or call-by-name semantics, neither of which are supported in $\lambda_{\uparrow\downarrow}$.

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'
let f = λ_fx.e in e'
```

Each lambda implicitly has a reference to itself available for use from within its definition. Pink (section 2.7) implements this type of recursion by reserving its first argument for a self-reference.

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 to 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 during branching and on function call/return. A function we call *step* explicates the machine progresses by reading next instructions and operands from the remaining entries in the control register and terminates at a **STOP** instruction, at which point the machine returns all values or a single value from the S-register respectively. As described in section 5.2 we add an additional register labelled R that holds recursive function definitions for convenience of implementation and partial evaluation.

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 immediate 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 **LD** instruction which loads from a stack frame and offset (recall the 2D de Bruijn indices from section 2.1). **RTN** places the value computed in the function onto the stack and returns to the caller.

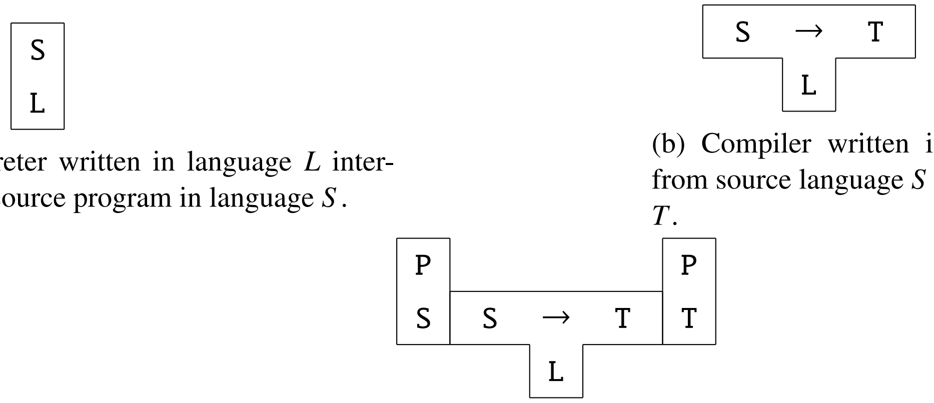
  ```
  LDC 10 LDF
      (LD (1 1) LDC 20 MPY RTN)
  AP
  STOP

  Result: (200)
  ```

- A recursive call assumes the stack to hold two function bodies (which in SECD are simply lists of instructions): a recursive function and a function that initiates the first recursive call. **RAP** ties a knot in its environment such that a definition of the closure is accessible during its application; this enables recursion (see section 2.2.2). 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)
```



(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

2.4 Interpretation and Compilation

An interpreter reads and directly executes instructions based on an input source program (depicted in figure 3a as a *tombstone diagram* for a pictorial view of composition). *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 and 3c. Here a program P written in language S is compiled to a program P written in T; that compiler executes in language L. 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.

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 two-argument function $f(x, y)$, one can produce a specialized one-argument 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 a program 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 that is a faster than executing p with both x and y directly:

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

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 can 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 split interpretation into two stages: one that generates a residual program and another that executes it [19].

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].

2.5 Partial Evaluation

The output of a partial evaluator is a *residual program* which, in the ideal case, is a 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). In the program to-be-specialized we refer to the *binding-time* of a variable as static if the data it holds during the lifetime of the program is static and otherwise 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.

2.6 Type-Directed Partial Evaluation

Danvy devised a remarkably simple method of implementing a partial evaluator solely based on normalization in the simply-typed λ-calculus called *Type-Directed Partial Evaluation* (TDPE) [9]. This PE technique is completely driven by the *type* of expressions being specialized (hence it is *type-directed*). TDPE 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). Thus such partial evaluators are typically implemented in a multi-level language capable of distinguishing dynamic and static data. Static reduction (i.e., reduction of expressions annotated as static) is performed through evaluation of the multi-level language's interpreter and well-formedness of the generated code is guaranteed by the language's type system. During residualization TDPE generates code that includes *binding-time coercions* which are type coercions that help to convert between dynamic and static values for cases where doing so benefits the residualization process. The two classes of coercion operators are *reification* (also called *lift*) and *reflection*, both of which are defined on product, function and literal types [22].

Reification converts a static expression of type T to a dynamic one and is denoted with a $\downarrow^T$. The function *lift* is syntactic sugar for $\downarrow^T$ where the type parameter is omitted for convenience. Reflection is simply the dual operation of reification. The exact coercion rules are shown in figure 4.

Given an expression and a separate object representing its and each of its subterm's type, *residualization* in TDPE is finally defined as annotation (and automatic insertion of binding-time coercions if necessary) followed by static reduction of said expression. The example below (section 2.6.1) provides a detailed demonstration of this process.

As dicussed in section 5.1, the partial evaluator embedded in $\lambda_{\uparrow\downarrow}$ [7] implements most of the binding-time coercion rules according to figure 4's definitions with the exception of *reflect*, which uses an algorithm attributed to Eijiro Sumii [23] to ensure correct order of evaluation of side-effects. In constrast to Danvy's TDPE, $\lambda_{\uparrow\downarrow}$ does not automatically insert *lift* annotations but instead relies on them being manually provided by the user.

2.6.1 Example

Consider 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.[3]

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

A correct binding-time could be that shown in (2) because after static reduction we obtain (3) which is a completely dynamic expression. However, even after reduction the multiple occurrences of $\underline{\lambda} a.a$ and the 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.

[3] We omit the type annotations of terms for simplicity but the actual TDPE algorithm relies on them to guide its residualization

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

$\triangleright$ *(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 (6) 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]. Using the coercion operator, TDPE would yield the annotation in (5).

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

$$\equiv \underline{\lambda} g.\underline{\lambda} d.(\overline{\lambda} f.g \mathbin{\underline{@}} (f \mathbin{\overline{@}} d) \mathbin{\underline{@}} (\mathit{lift}\ f)) \mathbin{\overline{@}} \overline{\lambda} a.a \tag{5}$$

$\triangleright$ *(via static reduction)*

$$\underline{\lambda} g.\underline{\lambda} d.g \mathbin{\underline{@}} d \mathbin{\underline{@}} \underline{\lambda} a.a \tag{6}$$

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

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

Reflection

$$\uparrow_{t} e = e \tag{10}$$

$$\uparrow_{t_1 \to t_2} e = \overline{\lambda} v.\ \uparrow_{t_2} (e \underline{@} (\downarrow^{t_1} v)) \tag{11}$$

$$\uparrow_{t_1 \times t_2} e = \overline{cons}(\uparrow_{t_1} \underline{car}\, e, \uparrow_{t_2} \underline{cdr}\, e) \tag{12}$$

Figure 4: Coercion 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 expressions and e denotes dynamic expressions. Overlines represent static terms and underlines dynamic ones. 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.

2.7 $\lambda_{\uparrow\downarrow}$ Overview

We now provide an overview of the partial evaluation framework developed for Pink [7] that we use throughout our experiments. At its core is the multi-level language $\lambda_{\uparrow\downarrow}$ (see figure 2). 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 (which is why we refer to it as *stage-polymorphic*). 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 [25]. For example a residualized addition of two literals is,

```
Code(
    Let(Plus(Lit(5),Lit(5)),
         Var(0)) // de Bruijn index
)
```

In $\lambda_{\uparrow\downarrow}$ the TDPE *reify* operation is called *lift* and converts static expressions of type `Val` into dynamic expressions of type `Exp`. The fact that residualization 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 (see example 2.7.1) by annotating its source (i.e., wrapping expressions in calls to $\lambda_{\uparrow\downarrow}$'s *lift* operator) 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*. Firstly, binding time annotations are not performed automatically by the PE but rely on a user annotating the source with *lift*. Secondly, the PE does not require the type of the input expression to be known statically. The ability to manually guide binding-time decisions without the need for a separate type object as input allows greater flexibility in which expressions we want to residualize. The residualization algorithm of $\lambda_{\uparrow\downarrow}$ only ever converts static into dynamic expressions. Thus the *reflect* operator serves a different purpose to TDPE's specification. Instead of coercing dynamic into static expressions, the implementation adds an expression to the global accumulator of residual terms, `stBlock`, that is used to generate the specialized code through let-insertion (see line 52 in listing 1). Danvy introduced *let-insertion* into an implementation of TDPE [23] to ensure side-effects in expressions, possibly duplicated by TDPE, are perfromed with the same frequencey and in the same order as they occur in a source program.

```
// Scala implementation of λ↑↓
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
...
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          // lifted data, including
                                            // residual programs, are wrapped in
                                            // this Code(...) constructor
...
// Core evaluator for λ↑↓
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))))
      }
    ...
}
// Auxiliary state
var stBlock: List[(Int, Exp)] = Nil
var stFresh = 0
var stFun: List[(Int,Env,Exp)] = Nil

// Helper functions
def run[A](f: => A): A = {
    val sF = stFresh
    val sB = stBlock
    val sN = stFun
    try f finally { stFresh = sF; stBlock = sB; stFun = sN }
}
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()
}
def reifyc(f: => Val) = reify { val Code(e) = f; e }
def reflectc(s: Exp) = Code(reflect(s))

// TDPE-style 'reify' operator
def lift(v: Val): Exp = v match {
    case Cst(n) => Lit(n)                          // number
    case Str(s) => Sym(s)                          // string
    case Tup(Code(u),Code(v)) => reflect(Cons(u,v)) // pair
    case Code(e) => reflect(Lift(e))               // dynamic expression
    case Clo(env2,e2) =>                           // closure
      stFun collectFirst { case (n,`env2`,`e2`) => n } match {
        case Some(n) =>
          Var(n)                                   // de Bruijn idx if was function memoized
        case None =>
          stFun :+= (stFresh,env2,e2)              // memoize for performance
          reflect(Lam(reify{ val Code(r) = evalms(env2:+Code(fresh()):+Code(fresh()),e2); r }))
      }
}
...
```

Listing 1: Main points of interest of the $\lambda_{\uparrow\downarrow}$ interpreter written in Scala [26].

```
(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]

Pink uses the notion of *stage-polymorphism* introduced by Offenbeck et al. [27] 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 and is implemented as shown in figure 5. Whenever the *lift* operator is now used in the *interpreter* or *compiler* it will cause the single evaluator *evalms* to either execute or generate code.

Now we can combine meta-circular interpreters or compilers in a single invocation of an input program as follows:

```
(let int_src       (quote interpreter)
(let comp_src      (quote compiler)
    ((((interpreter int_src) int_src) comp_src) <input quoted program>))))
```

The function *quote* is the LISP function that prevents evaluation of its argument. The above example builds a meta-circular tower of three ordinary interpreters and a staged interpreter (essentially a compiler) that run some program (similar to the tower represented in figure 8b).

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 that an interpreter returns. This will dynamize and hence residualize any operation that includes them, a process more thoroughly described in the example below.

2.7.1 Example Staged Interpreter

The excerpt in figure 6 demonstrates how *lift* annotations are used in staging Pink’s interpreter. Any product types (*cons*), function types (*lambda*) and literal types (*num*/*quote*) that are interpreted will be residualized (since they are wrapped in *maybe-lift*). In turn any operation on them will also residualize. Effectively we now generate another program that includes only the operations and values of the original expressions (`exp`) without the interpreter’s if-statement tree that matches redundant terms (i.e. without any interpretative overhead). Our interpreter can be said to perform compilation since it generates residual terms in $\lambda_{\uparrow\downarrow}$.

```
(lambda _ maybe-lift (lambda _ eval (lambda _ exp (lambda _ env
  (if (num?                exp)    (maybe-lift exp)
  (if (sym?                exp)    (env exp)
  (if (sym?           (car exp))
    ...
    (if (eq?  'lambda (car exp))
         (maybe-lift (lambda f x ...
    ...
    ;Meta-circular lift
    (if (eq?  'lift   (car exp))    (lift ((eval (cadr exp)) env))
    ...
    (if (eq?  'cons   (car exp))
         (maybe-lift (cons ((eval (cadr exp)) env)
                                              ((eval (caddr exp)) env)))
    (if (eq?  'quote  (car exp))    (maybe-lift (cadr exp))
    ...
    ((env (car exp)) ((eval (cadr exp)) env))))))))))))))))))))))))))
  (((eval (car exp)) env) ((eval (cadr exp)) env)))))))))
```

Figure 6: Snippet from Pink's staged interpreter source. Highlighted in green are the polymorphic *lift* expressions we use for binding-time annotations.

3 Heterogeneous Towers

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 [28] 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 off, namely the question of how one might achieve the effect of compiling multiple interpreters in heterogeneous settings. We view heterogeneous towers as a generalization of reflective towers and define *heterogeneous* as follows:

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

Observation 3.1. Heterogeneous towers of interpreters are towers which generalize homogeneous towers by:

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 and the representation of data can be different between the two even if the languages coincide; this gives us a way of addressing the difference in intensional structure of towers such as the Python-x86-JavaScript tower (figure 7)

3.1 Absence of: Meta-circularity

The first generalization described by observation 3.1 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, for example, 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 tools. 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 their 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 an inspiration for the second point of definition 3.1. We call the difference in operational semantics or data representation between two interpreters a *semantic gap*.

Another motivation of ours 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 7[4]. As a comparison our tower is shown in 2. 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 so 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 more closely resembles practice and adds some insightful challenges to our experiments.

[4] We only present a high-level view of the Python-x86-JavaScript tower. The actual realization is of course more complex and requires treatment of side-effects, which we leave out of our study.

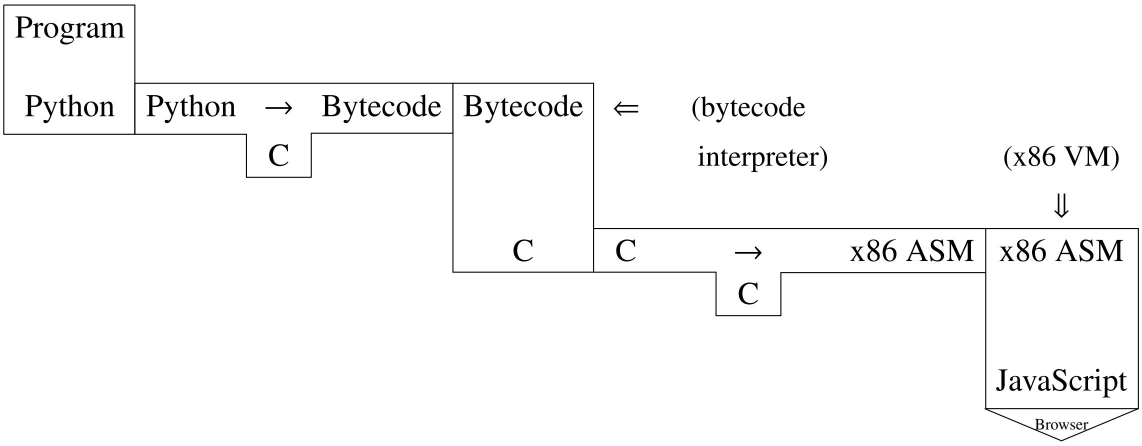


Figure 7: A hypothetical tower of interpreters that serves as the model for the tower we built (figure 2). The diagram depicts a x86 virtual machine (VM) written in JavaScript running a Python [29] interpreter that in turn executes some Python program. In our model, Python is first translated to bytecode which is then interpreted by some bytecode interpreter (written in the C language [30]). *Browser* encompasses the JavaScript interpreter within a browser and any underlying technologies required to host the browser.

4 General Recipe for Collapsing Towers

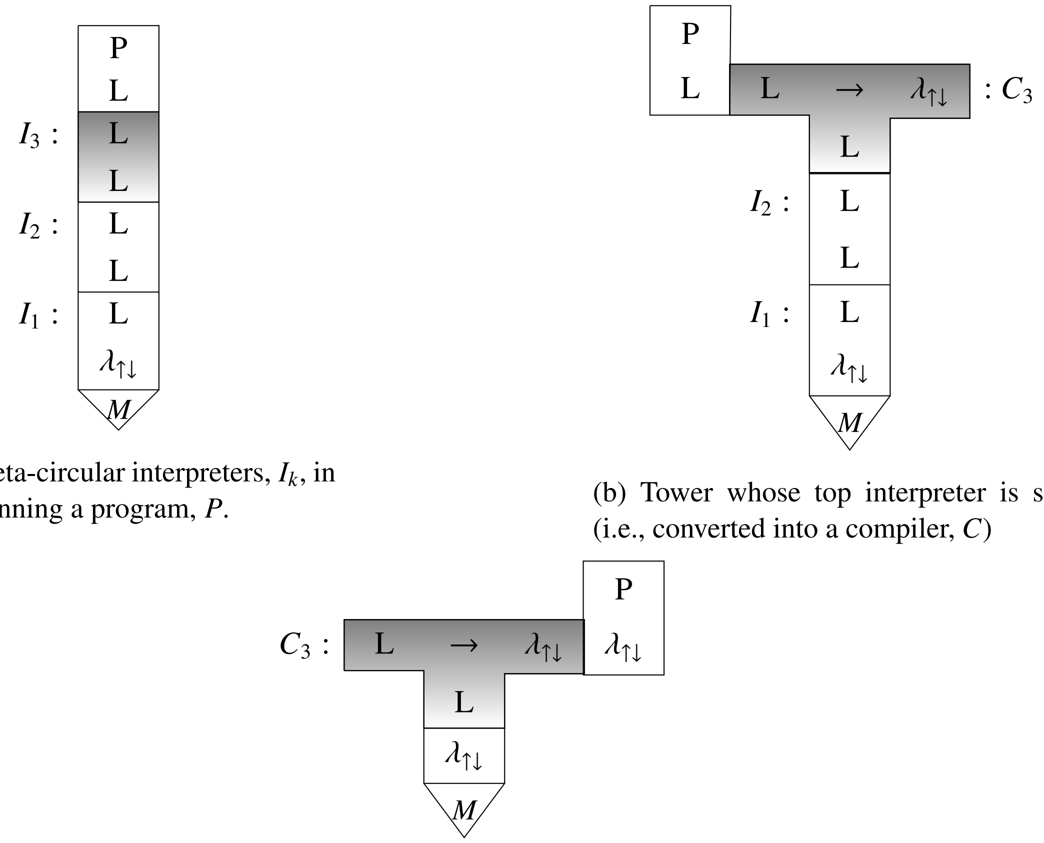
In this section we describe what it means to construct and *collapse* a tower of interpreter and the methodology Pink used to do so in previous work [7]. Then we discuss changes that have to be considered when applying these techniques to a heterogeneous setting.

Collapsing a tower means removing overhead from multiple interpretation. To do so we specialize our tower with respect to a user program and produce a residual program with as much interpretation removed as possible. The three key ingredients to collapsing a tower using Pink's technique are:

1. A multi-level language
2. A stage-polymorphic base evaluator for the multi-level language
3. TDPE-style *reify* operator

Amin et al.'s multi-level language, $\lambda_{\uparrow\downarrow}$, differentiates between static and dynamic expressions using types; this allows $\lambda_{\uparrow\downarrow}$ to express binding-time information. It also defines a *lift* (i.e., TDPE's reify) operator such that the PE can coerce static to dynamic values. A single evaluator for $\lambda_{\uparrow\downarrow}$ operates on both dynamic and static expressions which enables so called *binding-time agnostic* staging [7]. This means to *stage an interpreter* we can annotate it with the polymorphic lift instantiated such that it invokes $\lambda_{\uparrow\downarrow}$'s *lift* (such as *compiler* in figure 5).

4.1 Construction and Collapse of a Tower


P
L
I_3 :
L
L
I_2 :
L
L
I_1 :
L
$\lambda_{\uparrow\downarrow}$
M
P
L
L
→
$\lambda_{\uparrow\downarrow}$
: C_3
L
I_2 :
L
L
I_1 :
L
$\lambda_{\uparrow\downarrow}$
M
C_3 :
L
→
$\lambda_{\uparrow\downarrow}$
P
$\lambda_{\uparrow\downarrow}$
L
$\lambda_{\uparrow\downarrow}$
M

(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 8a 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 8: Tombstone diagrams representing the process of collapsing a tower using $\lambda_{\uparrow\downarrow}$

Figures 8a to 8c depict the process of collapsing a tower 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. The key benefit of meta-circularity is that the *lift* operator defined in $\lambda_{\uparrow\downarrow}$ is accessible to each interpreter. We can now stage some interpreter in the tower, in this example the user-most one I_3, by annotating it with *maybe-lift* expressions as described in section 2.7; this interpreter is now equivalent to a compiler from L to $\lambda_{\uparrow\downarrow}$ (C_3 in figure 8b). When we execute the tower (i.e., invoke $\lambda_{\uparrow\downarrow}$'s partial evaluator) C_3 residualizes while all other levels in the tower evaluate (essentially *propagating* binding-time information of program P from the top to the base of the tower). At I_1 a call to *lift* now invokes $\lambda_{\uparrow\downarrow}$'s TDPE-style reify. 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 the polymorphic lift's were instantiated to be the identity function (i.e., *interpreter* in figure 5). The result is a collapsed tower in figure 8c where all intermediate interpreters, I_1 through I_3, have been removed from the tower (assuming the absence of side-effects at individual levels).

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 sections 5.1 to 5.4 we provide evidence to support this claim on our experimental tower.

4.2 Effect of Heterogeneity

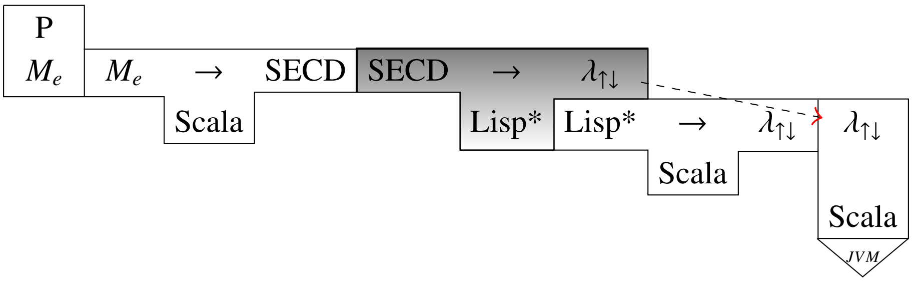

P
M_e
M_e → SECD
Scala
SECD → $\lambda_{\uparrow\downarrow}$
Lisp*
Lisp* → $\lambda_{\uparrow\downarrow}$
Scala
$\lambda_{\uparrow\downarrow}$
Scala
JVM

Figure 9: Our heterogeneous tower of interpreters (2) after staging at the SECD level (shaded tombstone). All computation to the left of the staged interpreter is carried into the residual program. The collapse essentially moved the code from above the SECD machine to the base.

We chose the SECD machine to create semantic gaps within our tower through the low-level instruction set that comes with it. For convenience we chose to implement the M_e level as a translator from M_e to SECD instructions instead of writing the interpreter in the instructions directly. Is is noteworthy to note that the addition of translation layers into the tower revealed insights into the process of collapsing heterogeneous towers that are not immediately apparent from the composition of tombstones. We describe our observations and further effects of heterogeneity on collapsing towers in the remainder of this section.

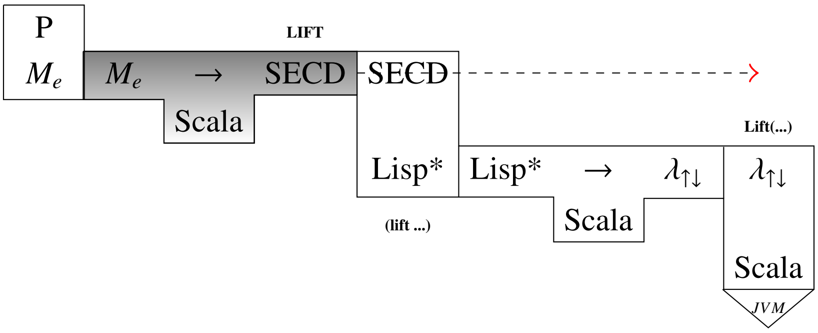

P
M_e
LIFT
M_e → SECD
Scala
SECD
Lisp*
(lift ...)
Lisp* → $\lambda_{\uparrow\downarrow}$
Scala
Lift(...)
$\lambda_{\uparrow\downarrow}$
Scala
JVM

Figure 10: Our heterogeneous tower of interpreters after staging at the M_e level (shaded tombstone). At each level, starting from M_e, the *lift* operator is implemented differently but together they achive the effect of moving code from the M_e interpreter to the base.

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 section 5.2.1, requires careful design of its division rules and consideration of which locations to lift to avoid non-termination at PE-time. Figure 9 depicts the effect of staging the SECD interpreter in our tower. After staging the SECD machine (highlighted in grey) it now operates as a compiler from SECD instructions to $\lambda_{\uparrow\downarrow}$ terms. Collapsing the tower in this configuration essentially means moving code from the SECD level to the base interpreter across the Lisp* level. Consistent with the composition rules of tombstones, the levels above SECD are residualized and present in the output as well.

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 is to implement a built-in *lift* at all levels below the interpreter that is to be staged. As we explain in more detail in section 5.2.1, the implementation of *lift* may require us to reverse engineer and transform the representation of closures, pairs or other constructs which the *lift* at the base expects.

A more subtle collapse of our tower occurs when we stage the M_e level (see figure 10). In this case staging has a slightly different effect which is not obvious from the tombstones. Since the M_e level is actually a translator already, staging the translator will simply yield another translator. Instead, staging M_e means we generate SECD instructions that *lift* (i.e., signal the PE to residualize) expressions of M_e using a new **LIFT** instruction (added in section 5.3.1). As we can see from the annotated tombstone diagram, we use the *lift* operator as a mechanism to move code through each level to the base where it is residualized. Whereas passing binding-times between interpreters of a homogeneous towers is implicit in *lift* due to meta-circularity, heterogeneity makes it explicit.

5 Construction of an Experimental Heterogeneous Tower

In this section we explore the construction of our heterogeneous tower of interpreters from figure 2. We start with a brief overview of $\lambda_{\uparrow\downarrow}$ and Lisp*. Each subsequent subsection explains how we design and implement an interpreter in the tower. Additionally we describe the process of staging each level and show generated code after collapsing the tower on example programs.

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 2) form the lowest two levels of the tower, which we mostly keep unchanged from Pink's implementation.

In order to improve optimality of the generated code we add logic that reduces purely static expressions to $\lambda_{\uparrow\downarrow}$'s *reflect* (additions to original source are highlighted in green below). 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:

```
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()
        }
}
```

5.2 Level 3: SECD

We chose the SECD abstract machine as one of our levels for following reasons:

1. **Maturity**: SECD was the first abstract machine of its kind developed. Since then it has thoroughly been studied and documented [31, 32, 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 explicit stacks to perform all computation deviates from the semantics of $\lambda_{\uparrow\downarrow}$ and its LISP front-end and thus satisfies the *semantic gap* property well.

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

An additional benefit of using SECD is that the Pink framework 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 (from the small-step semantics and compiler in 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** instruction is encountered.

5.2.1 Staging a SECD Machine

We design our SECD machine so that it can be staged using $\lambda_{\uparrow\downarrow}$'s *lift*; this enables us to experiment with 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 translates 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 from 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 the binding-time of a SECD register to be the combination of all possible binding-times of elements of a register. A register that only holds dynamic or static data is classified as dynamic or static respectively. 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 or mostly static. 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 most elements in S are 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.

We stage our SECD interpreter by annotating its source with $\lambda_{\uparrow\downarrow}$'s *lift* according to the division in table 1.

SECD Register	**Classification**	**Reason**
S (Stack)	Mixed (mostly dynamic)	We want to specialize the SECD machine with respect to some static program. The residual output should thus include only the operations and literals that occur in said program. Since all operations utilize S we mark values added onto it as dynamic (with the exception of closures 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. It can also temporarily hold static values transferred from S and thus is only classified as *mostly dynamic*
C (Control)	Static	We make sure the register only receives static values and is thus static (we ensure this through η-expansion in section 5.2.1)
D (Dump)	Mixed	Used for saving state of any other register and thus elements can be both dynamic or static
R (recursive functions) *newly added register*	Dynamic	A specialized version of E to hold recursive function closures which is always static

Table 1: Division rules for our approach to staging a SECD machine. We added the R-register in section 5.2.3 to make partial evaluation of recursive SECD functions calls more convenient.

<table>
<tr><th>Step</th><th>Register Contents</th></tr>
<tr><td>0</td><td>s: ()
e: ()
c: (LDC 10 LDC 20 ADD STOP)
d: ()</td></tr>
<tr><td>1</td><td>s: (10)
e: ()
c: (LDC 20 ADD STOP)
d: ()</td></tr>
<tr><td>2</td><td>s: (20 10)
e: ()
c: (ADD STOP)
d: ()</td></tr>
<tr><td>3</td><td>s: (30)
e: ()
c: (STOP)
d: ()</td></tr>
<tr><td colspan="2">Generated Code (without smart constructor):
(lambda f0 x1 (+ 20 10))</td></tr>
<tr><td colspan="2">Generated Code (with smart constructor):
(lambda f0 x1 30)</td></tr>
</table>

Table 2: Example of SECD evaluation and $\lambda_{\uparrow\downarrow}$code generated using our PE framework. The division follows that of table 1. The lambda abstraction, $f0$, is a consequence of our SECD interpreter architecture (see section 5.2.2) and is used to accept user input.

5.2.2 The Interpreter

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

Figure 11: Structure of tail-recursive SECD interpreter (this is the unstaged version and thus we omit the R-register). 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 and the choice of curried argument list order 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. 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

```
((SECDMachine '(LDC 10 LDC 20 ADD STOP)) '())
```

where the arguments to the machine are a list of SECD instructions performing an addition of two integers and an empty environment (registers **c** and **e** in example of table 2 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* (line 12) as well because we want to specialize the machine without the dynamic input of the environment provided yet.

```
(let SECDMachine (lambda _ stack (lambda _ dump (lambda _ control (lambda _ env
    (if (eq? 'LDC (car control))
        (((((SECDMachine (cons (maybe-lift (cadr control)) stack)) dump) (cdr control)) env)
    (if (eq? 'DUM (car control))
        (((((SECDMachine stack) dump) (cdr control)) (cons (maybe-lift '()) env))
    (if (eq? 'STOP (car control))
        s
    ...
    ...)))))))
    (let initStack '()
    (let initDump '()
        (lambda _ ops (maybe-lift (((SECDMachine 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 the division of table 1.

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 unnecessary instruction dispatch logic from the specialized code, whose effect on interpreter efficiency was studied extensively by Ertl et al. [33]. 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. The coding of a specializable SECD machine must avoid putting values into the control register that are dynamic since this could suddenly cause comparisons between dynamic and static values which is a specialization time error at best and non-termination of the PE at worst.

Another issue we addressed when 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 by arranging that when the recursive function looks up the first argument in

```
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: Example recursive function application

the environment it finds the recursive closure (section 2.2.2). According to Kogge's [13] description of the SECD operational semantics this requires an instruction that is able to mutate variables. Given the choice between adding support for an underlying `set-car!` instruction in $\lambda_{\uparrow\downarrow}$ or extending the SECD machine such that recursive function 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. Without the ability to tie a knot in the environment we need a different way of achieving recursion in SECD. Additionally we need to design the internals of the machine such that we can stage it using Pink's TDPE.

Following example, in which a recursive function decrements a user provided number down to zero, demonstrates the issue of partially evaluating a recursive call in using SECD small-step semantics:

Our PE would not terminate were we to specialize this program by simply evaluating the machine. 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.

Following example 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
    (((((SECDMachine (cdr stack)) (cons (cdddr control) dump)) fns) (cadr control)) env)
    (((((SECDMachine (cdr stack)) (cons (cdddr control) dump)) fns) (caddr control)) env))
```

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.

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 14 shows the modifications to the operational semantics of Kogge's SECD description [13] which allow it to be partially evaluatable with a TDPE and does not require a `set-car!` in the underlying language. The idea is to wrap the recursive SECD instructions in a closure at the LISP-level and use Pink lambda's ability to self-reference to achieve recursion. Closures also allows us to produce residual code for SECD function bodies, which previously was not possible since they were simply represented as lists of instructions.

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

1. Functions bodies in the SECD machine are now lambdas that wrap a call to the implicit state transition which is our way of achieving recursion without tying a knot. This also helps our aim of staging the machine since Pink's *lift* operates on lambdas and allows us to residualize entire function bodies.
2. The **RTN** instruction aids termination of TDPE by preventing it to evaluate the machine (i.e., perform state transitions) further upon detection of special markers that we push onto D (we detail this technique later in this section). This ensures that we can terminate the unfolding of recursive calls.
3. Instructions **AP**/**RTN**/**LIFT** all feature calls to a *lift* operator that follow the division established earlier in table 1 to stage the SECD interpreter.

$$\text{s e (}\textbf{LDF}\text{ ops.c) d r} \longrightarrow \text{((}\lambda\text{e'.(run@('() e' ops 'ret r))) ops.e).s e c d r} \tag{13}$$

$$\begin{aligned}
\text{(entryClo recClo.s) e (}\textbf{RAP}\text{.c) d r} &\longrightarrow \text{'() e entryOps (s e c r.d) (lift@rec recEnv).r}\\
\text{where (entryFn (entryOps entryEnv))} &:= \ \text{entryClo}\\
\text{(recFn (recOps recEnv))} &:= \ \text{recClo}\\
\text{rec} &:= \ \lambda\text{env.(run@('() env recOps 'ret.d (rec recEnv).r))}
\end{aligned} \tag{14}$$

$$\text{s e (}\textbf{LDR}\text{ (i j).c) d r} \longrightarrow \text{(locate@(i j r)).s e c 'ldr.d r} \tag{15}$$

$$\begin{aligned}
\text{v.s e (}\textbf{RTN}\text{.c) (s' e' c' d' r'.d) r} \longrightarrow\ & \text{lift@v} && \textit{if}\ \text{s' = 'ret}\\
& \text{lift@(}\lambda\text{x.(fn@(lift@(x.env))))} && \textit{if}\ \text{s' = 'ldr}\\
& \qquad \text{where (fn (ops env)) := v} &&\\
& \text{(v.s') e' c' d r'} && \textit{otherwise}
\end{aligned} \tag{16}$$

$$\begin{aligned}
\text{s' e (}\textbf{AP}\text{.c) env' r} \longrightarrow\ & \text{(fn@(lift@(x.env))).s e c d r}\\
& \quad \text{where ((fn (ops.env)) x.s) := s'}
\end{aligned} \tag{17}$$

$$\begin{aligned}
\text{(v.s) e (}\textbf{LIFT}\text{.c) d r} &\longrightarrow \text{res.s e c d r}\\
\text{where res} &:= \ \text{lift@v} && \textit{if}\ \text{(num? v)}\ \textit{or}\ \text{(sym? v)}\\
&\ \ \text{lift@(}\lambda\text{x.(run@('() x.env ops 'ret r)))} && \textit{if}\ \text{(clo? v)}\\
&\ \quad \text{where (fn (ops env)) := v} &&
\end{aligned} \tag{18}$$

Figure 14: 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 the machine 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, R* are represented by *s, e, c, d, r* respectively. The syntax *num?/sym?/clo?* are conditions satisfied when their argument is a number, string or a closure respectively. The function *lift* is $\lambda_{\uparrow\downarrow}$'s TDPE-style *reify* which is used to mark a term as dynamic.

```
(if (eq? 'RAP (car ops)) ;SECD recursive function application
  (let recClo (cadr s)
  (let entryClo (car s)
  ...
  (let rec
    (maybe-lift (lambda rec env
        ((((((SECDMachine '()) (cons 'ret d)) ;lift recursive SECD function body
                    (cons (cons (cons rec recEnv) oldEnv) fns)) recOps) env)
        )
  ...
  (if (eq? 'RTN (car ops)) ;SECD function return
      (if (eq? 'ret (car d))
        (mla (car s)))
  ...
```

Figure 15: A simplified excerpt from our staged SECD machine showing the augmented **RAP** instruction. We residualize a call to the `SECDMachine` state transition function with recursive function body `recOps` in *C*. To stop unfolding this call we push a marker, `'ret`, onto *D* and test for it when we encounter a **RTN**.

Firstly, equation 13 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 means we can safely lift all elements in *S* when we try following our division. 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 *R-register*, that is responsible for holding our closures that wrap recursive instructions of a **RAP** call. In the traditional SECD machine both the recursive and the calling function are kept in the environment, loaded onto the stack using **LD** and subsequently called using **AP**. However, for simplicity we keep recursive functions in *R* to distinguish them from free variables in *E* and aid debuggability. Thus we introduce a new **LDR** instruction that returns the contents of the R-register by index, just as **LD** does for the E-register.

Modification to the **RAP** instruction are described by (14). 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 lifts and appends a newly constructed recursive closure to *R*. The *lift* is necessary because we want the definition of the recursive function to appear in the residual program. The new closure when applied to an environment, runs the machine with the *C* containing instructions of the recursive function body. As depicted in figure 15, applying the closure places a *'ret* tag onto *D* 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 push 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 by returning the top-most value on the stack if the first element of *D* is the sentinel string *'ret*. This works because function definitions now reside in lambdas in the interpreter and SECD function invocation is lambda application. 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 **LDF**'s instructions from *S* into *C*, now **AP** initiates a call to the lambda that we wrapped the instructions in. We lift the extended environment, `x.env`, in case the function we pass it to is dynamic.

To rewrite the example from figure 13 on our extended SECD machine we load the recursive function using the new **LDR** instead of the **LD** instruction:

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

The above changes to the machine show that to permit TDPE 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 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 a 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 16) to support our modified calling conventions and additional register described in section 5.2.3. Since we hold recursive function definitions in the *R*, we want to index into it instead of the regular environment register that holds variable values. Additionally, we need to make sure our compiler supports passing values from the user through the environment. To that end, we keep track of and increment an offset into *E* during compilation whenever a free variable is detected via a missed look-up in the environment. The **quote** built-in from (21) 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⟩ (19)
Code : `(LDR (i, j))` if lookup is in a **letrec**
where `(i,j)` is an index into the **R-register**
`(LD (i, j))` otherwise
where `(i,j)` is an index into the **E-register**

Syntax : (**lift** ⟨expr⟩) (20)
Code : ⟨expr⟩ `LIFT`

Syntax : (**quote** ⟨expr⟩) (21)
Code : `LDC` $\langle id_0 \rangle$ `LDC` $\langle id_1 \rangle$ `CONS ...` `LDC` $\langle id_{n-1} \rangle$ `LDC` $\langle id_n \rangle$ `CONS`
where $\langle id_n \rangle$ is the nth identifier in the string representing ⟨expr⟩

Figure 16: Modifications to the SECD compiler described by Kogge [13]

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

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

Line 3 feeds the compiled SECD instructions to the SECD machine interpreter source described in section 5.2.2 and begins partial evaluation through a call to `ev` which is the entry point to $\lambda_{\uparrow\downarrow}$.

5.2.5 Example

Figure 17a shows a program to compute factorial numbers recursively written in SecdLisp. The program is translated into SECD instructions by our compiler and then input to our staged machine. Figure 17c 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 our augmented closure representation. 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 SECD's stack-based semantics are left in the generated code and cannot be reduced further without changing the architecture of the underlying machine.

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

(a) Example Factorial in SecdLisp.

```
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) Generated SECD instructions when compiling SecdLisp factorial.

```
(let x0
  (lambda f0 x1 <=== Takes user input
    (let x2
      (lambda f2 x3 <=== Definition of factorial
        (let x4 (car x3)
        (let x5 (car x4)
        (let x6 (eq? x5 0)
        (if x6
          (let x7 (car x3)
          (let x8 (cdr x7) (car x8)))
          (let x7 (car x3)
          (let x8 (cdr x7)
          (let x9 (car x8)
          (let x10 (car x7)
          (let x11 (* x10 x9)
          (let x12 (- x10 1)
          (let x13 (cons x11 '.)
          (let x14 (cons x12 x13)
          (let x15 (cons '. x1)
          (let x16 (cons x14 x15) (f2 x16)))))))))))))))) <=== Recursive Call
    (let x3 (cons 1 '.)
    (let x4 (cons 10 x3)
    (let x5 (cons '. x1)
    (let x6 (cons x4 x5)
    (let x7 (x2 x6) (cons x7 '.)))))))) (x0 '.))
```

(c) Residual factorial program

Figure 17: Example factorial program running on our staged SECD machine.

5.3 Level 4: M_e

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

Figure 18: Syntax of M_e which gets compiled into SECD instructions for interpretation by the SECD machine

Armed with a staged SECD machine and a language to target it with (i.e., SecdLisp), we build the next interpreter in the tower that gets compiled into SECD instructions. The interpreter defines a language called M_e (from the tombstone diagram in figure 2). Its syntax is described in figure 18. The language is based on Jones et al.'s toy language M in their demonstration of the Mix partial evaluator [28] 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 Lisp*'s parsing infrastructure. Further work could investigate changing representation of data structures like closures to increase the semantic gaps between $\lambda_{\uparrow\downarrow}$ and M_e to experiment with even more heterogeneity than in the tower we built.

M_e supports traditional functional features such as recursion, first-class functions, currying but also LISP-like quotation. Note that unlike in $\lambda_{\uparrow\downarrow}$ and Lisp*, variables in M_e do not use de Bruijn notation and lambdas are not self-referencing. We implement the language as a case-based interpreter shown in figure 19. 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. The latter mimics Pink's polymorphic **maybe-lift**.

```
1  def meta_eval(program: String, lift: String = "(lambda (x) (lift x))") = s"
2      (letrec (eval) ((lambda (exp env)
3          (if (sym? exp)
4              (env exp)
5          (if (num? exp)
6              ($lift exp)
7          (if (eq? (car exp) '+)
8              (+ (eval (cadr exp) env) (eval (caddr exp) env))
9          ...
10         (if (eq? (car exp) 'lambda)
11             ($lift (lambda (x)
12                 (eval (caddr exp)
13                     (lambda (y) (if (eq? y (car (cadr exp)))
14                                         x
15                                         (env y))))))
16                     ((eval (car exp) env) (eval (cadr exp) env))))))))))))))
17         (eval (quote $program) '()))
```

Figure 19: Staged interpreter for M_e

Figure 20 shows the M_e interpreter running a program computing factorial using the Y-combinator for recursion (figure 20a) 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 21). As the programmer we know this control flow can be reduced even further since the M_e source program is static data.

```
((lambda (fun)              ;Definition of Y-combinator
          ((lambda (F)
             (F F))
           (lambda (F)
             (fun (lambda (x) ((F F) x))))))

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

(a) Factorial written in Lisp* using the Y-combinator for recursion.

```
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) Generated instructions after compiling (a)

Figure 20: Example factorial written in M_e. Here our tower is collapsed while staging the SECD machine.

```
(let x0
  (lambda f0 x1
    (let x2
      (lambda f2 x3
        (let x4 (car x3)
        (let x5 (car x4)
        (let x6 (sym? x5)
        (if x6
          ...               ;Do something
          (let x8 (car x7)  ;Else
          (let x9 (num? x8)
          (if x9
            ...
            (let x12 (car x11)
            (let x13 (eq? x12 '+)
            (if x13
              ...
              (let x29 (f2 x28) (+ x29 x25)))))))  ;Perform addition
                                                   ;and start next interpreter loop

              (let x17 (eq? x16 '-)
              (if x17
                ...
```

Figure 21: Residual code running after collapsing the example in figure 20 with a staged SECD machine. Traces of M_e's dispatch logic are highlighted in green.

5.3.1 Staging M_e and Collapsing the Tower

In an effort to further optimize our generated code from the example in figure 20 we stage the M_e interpreter. Amin et al., during their demonstration of collapsing towers written in Pink [7], mention that staging at the user-most level should yield the optimal residual 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 built-in *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 this type of binding-time information through interpreter boundaries.

The state transitions for the **LIFT** instruction in the staged SECD machine are shown in (18). The intended use of the instruction is to signal $\lambda_{\uparrow\downarrow}$ to lift the top of the stack. Thus running following on our SECD machine,

```
LDC 10 LIFT STOP
```

would generate a residual $\lambda_{\uparrow\downarrow}$ term 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 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 R and D; the lambda takes an environment and runs the instructions that it wrapped to completion.

Through the addition of a *lift* built-in into SecdLisp we can now stage the M_e interpreter and run it on our SECD interpreter (figure 19). The residual program for the factorial example (figure 20) is shown in figure 23 and the corresponding SECD instructions that M_e compiled down to in figure 22. The generated SECD instructions are the same as in the unstaged M_e interpreter with the exception of the newly inserted **LIFT** instructions following our *maybe-lift* annotations. 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 22: SECD instructions for example an factorial on a staged M_e interpreter

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

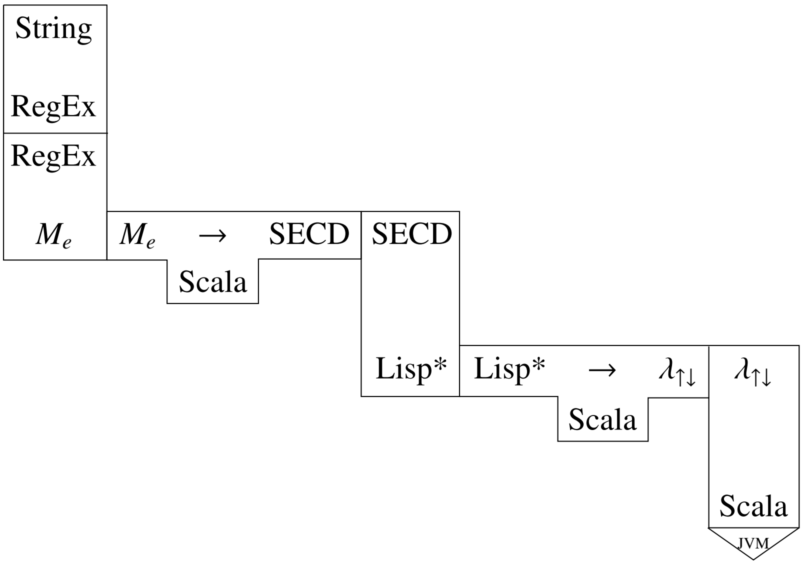

String
RegEx
RegEx
M_e
M_e → SECD
Scala
SECD
Lisp*
Lisp* → $\lambda_{\uparrow\downarrow}$
Scala
$\lambda_{\uparrow\downarrow}$
Scala
JVM

Figure 24: Tombstone diagram of our 5-level experimental tower. It is the tower of figure 2 extended by one level: a regular expression matcher that checks whether an input string matches some regular expression pattern, *RegEx*.

5.4 Level 5: String Matcher

Following the experiments performed in Amin et al.'s study [7], we extend our tower one last time and implement a regular expression matcher proposed by Kernighan et al. [34] in M_e(see figure 24). 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" followed by any character sequence. 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 26a. It is far from the most efficient version and we can see clear traces of the matcher logic in

```
(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.

the generated code such as a check for an “_” character on line 21 while our pattern against which we specialize does not contain any.

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:

```
(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))
```

As desired, the specialized matcher in figure 26b will succeed on any input string. This supports our hypothesis further that to generate optimal residual code during a collapse one should stage the user-most interpreter (in this case the string matcher). With this experiment we also demonstrated a collapse of a 5-level tower of interpreters even in the presence compilation layers (M_e-to-SECD and Lisp*-to-$\lambda_{\uparrow\downarrow}$).

```
(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 homogeneous 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 some form of interpretation are glued together. To the best of our knowledge, our work is one of a handful, together with Amin et al.'s previous explorations [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 and collapse 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). We started by generalizing the concept of reflective towers to ones that consisted of non-meta-circular interpreters. To model a tower that could potentially contain layers of translation between languages, such as the one in figure 7, we also introduced the notion of *semantic gaps*, which dictate the extent to which two adjacent interpreters differ in operational semantics or representation of data. A combination of these two properties (non-meta-circularity and semantic gaps) form the new class of towers of interpreters that we call *heterogeneous* (as opposed to the *homogeneous* reflective towers). The theoretical implications from heterogeneity on a tower's structure and procedure to collapse them was a focal point for our experiments. We envisioned two problems with collapsing heterogeneous towers: (1) we needed to devise a strategy to signal to the PE at the base of the tower which expressions to residualize without a meta-circular *lift* (even through levels of compilation) (2) semantic gaps required us to perform complex transformations on program constructs in one interpreter to adhere to their representation in adjacent interpreters. We constructed and then collapsed a 4-level heterogeneous tower with a SECD machine as one of its levels to provide evidence for these hypotheses and gain more insights into heterogeneity.

Staging an interpreter amounts to reifying literals, lambdas and product types it returns. An abstract machine is not guaranteed to distinguish these types by data structures or a type-system but can instead rely on dedicated instructions to differentiate data of these types. 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 Pink's reflect operator to achieve optimal residualization. In order to propagate the decision of whether to generate or evaluate an expression through levels in the tower, we implemented a **lift** operator (or abstract machine instruction in the case of our SECD compiler) at the level which previously did not support such an operation.

The type of overhead our study concerned itself with was 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 top-most staged level. We used our experimental tower to investigate the effect of staging interpreters at various points. As expected, the interpretative overhead of all levels up to the one being staged was completely reduced during specialization time. More notably, the structure of the generated code followed that of the interpreter that was staged. In our case, staging at the SECD machine level generated code that contained traces of the SECD semantics such as stack-based operations and as a result still optimizable to the programmer. In comparison, staging at a level above yielded a residual program without any signs of SECD and more optimal for our example programs.

With our experiments we demonstrated the successful collapse of a heterogeneous tower. We also showed the ability of a TDPE-style *reify* operation to essentially move code across levels of a tower, which also worked across levels of compilation. However, 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 involved the implementation of a reification operator in each interpreter that is missing it. This then required the reverse-engineering and conversion of types in an interpreter that TDPE's *reify* operates on to the representation that the interpreter below in the tower expects. In practice these changes would require intimate knowledge of and intrusive changes to an interpreter. Additionally, in our experimental tower we did not consider the residualization of side-effects which a useful collapse procedure would need for wider applicability. Our methodology could, however, help the optimization of smaller-scale systems in practice where towers consist of embedded DSL interpreters or regular expression matchers (even in the absence of meta-circularity and presence of translation layers).

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 a 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. 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. For example, call/cc allows 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 others like the Warren Abstract Machine (WAM) [35] as a SECD replacement. This would allow us to investigate the applicability of our method to collapse towers to other programming paradigms such as 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 certain types of towers are collapsible.

A major subject of focus in PE is the ability 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 technique to collapse towers less intrusive. Instead of reimplementing a *lift* 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 communication between languages that target the Graal Virtual Machine and could prove useful in further experimenting with heterogeneous towers where multiple interpreters pass, e.g., binding-times to each other.

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.

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] 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.

[23] 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.

[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] 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.

[26] 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.

[27] 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.

[28] 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.

[29] G. Van Rossum and F. L. Drake, *The Python language reference manual*. Network Theory Ltd., 2011.

[30] B. W. Kernighan, D. M. Ritchie, C. L. Tondo, and S. E. Gimpel, *The C programming language*. prentice-Hall Englewood Cliffs, NJ, 1988, vol. 2.

[31] O. Danvy, “A rational deconstruction of Landin’s SECD machine,” in *Symposium on Implementation and Application of Functional Languages*. Springer, 2004, pp. 52–71.

[32] J. D. Ramsdell, “The tail-recursive SECD machine,” *Journal of Automated Reasoning*, vol. 23, pp. 43–62, 1999.

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

[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.