Collapsing heterogeneous towers of interpreters

Michael Buch
Queens' College

UNIVERSITY OF CAMBRIDGE

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

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

Email: mb2244@cam.ac.uk

May 10, 2019

Abstract

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

To collapse a tower means to *stage* an interpreter in the tower (i.e., convert the interpreter into a compiler by splitting its execution into several stages) and statically reduce all the evaluation performed preceeding 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 framework that was 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

11156 (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. In his proposal 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 interpreters within the tower to access and modify internal state of interpreters adjacent to each other. 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. [3] and Danvy et al. [2] provide systematic approaches to constructing reflective towers. The authors provide denotational semantic accounts of reflection and develop a 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 [2] hinted at partial evaluation potentially being a tool capable of removing some of this overhead by 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 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

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

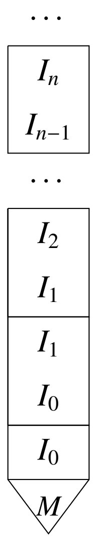


Figure 1: A tower of interpreters where each interpreter, I_n, is interpreted by the one below it in the diagram, I_{n-1}, for some $n \geq 0$. In literature the tower often grows downwards, however, in our study we refer to I_0 as the base interpreter and grow the tower upwards for convenience. M is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

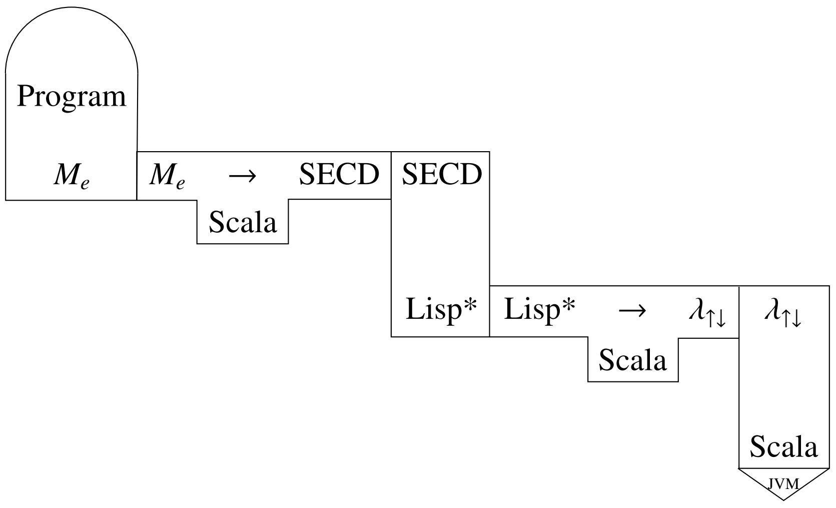
modified. Essentially this is achieved by residualizing and executing interpreters at run-time of the tower to remove the cost of multiple interpretation; this effectively *collapses* a tower.

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

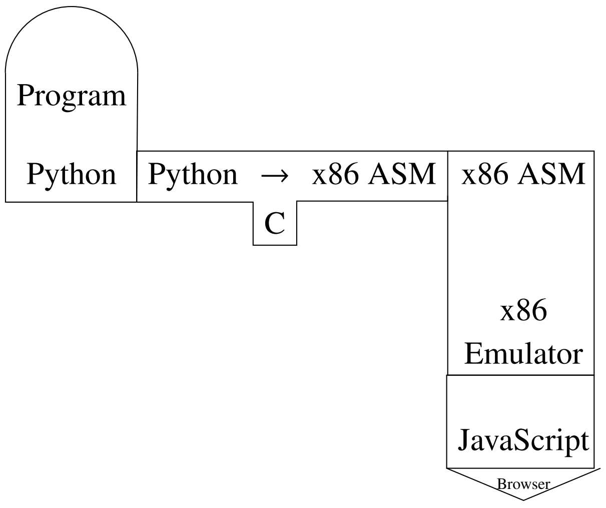
We aim to bring previous work of removing interpretative overhead using partial evaluation to towers into practice. Our study achieves this by constructing a proof-of-concept tower of interpreters that resembles one more similar to those in real-world systems. We then collapse it under different configurations and evaluate the resulting optimized programs. We demonstrate that given a multi-level language and a type-directed PE we can stage individual interpreters in a sequence of non-metacircular interpreters and effectively generate code specialized for a given program eliminating interpretative overhead in the process. Our work's contributions are: (1) develop an experimental heterogeneous tower of interpreters and a strategy for collapsing it (2) evaluate the effect that staging at different levels within our tower has on the residual programs (3) discuss the effects that heterogeneity in towers imposes on TDPE (4) demonstrate issues with and potential approaches to staging abstract machines, specifically a SECD machine, using TDPE.

In section 2 we explain background information that covers the fundamental topics we base our experiments and discussions on. We then define *heterogeneity* in towers of interpreters in section 3. In section 4 we describe the recipe that the partial evaluation framework Pink [7] used to construct and collapse meta-circular towers and then show how this recipe changes as a result of heterogeneity. We present the implementation and evaluation of our experimental tower in section 5. Section 5.1 provides an overview of Pink. We systematically describe the process by which we create a heterogeneous tower of interpreters and incrementally collapse it in sections 5.2 through 5.4. Each of these sections discusses the steps we took to create a level in the tower of interpreters as shown in figure 2a. We conclude with an evaluation of our findings followed by a discussion of potential future work in section 6.



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

(a) A tombstone diagram representation of our experimental tower where 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 and Lisp* is its Lisp based front-end [7]. *JVM* is the Java Virtual Machine [8] and in our diagram also encompasses a CPU which executes it.

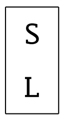


(b) A hypothetical tower of interpreters that serves as the model for the tower we built (shown in figure 2a). The diagram depicts a JavaScript x86 emulator running a Python interpreter that in turn executes a Python script. In this diagram *Browser* encompasses the JavaScript interpreter and the underlying CPU that hosts the browser application.

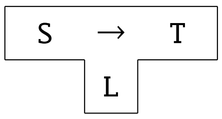
Figure 2: Comparison between our experimental tower (2a) and the one we modelled it on (2b).

2 Background

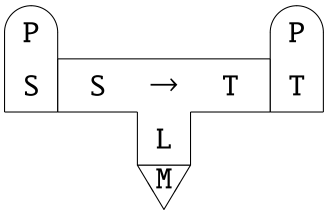
2.1 Interpretation and Compilation



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



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



(c) Example compilation of a program P from language S to a language T. M represents the underlying machine that the compiler runs on.

An interpreter reads and directly executes instructions based on an input source program (depicted in figure 3a as a *tombstone diagram*). An interpreter can define possibly multiple semantics of a language, in which case we call it a *definitional interpreter* [9].

A compiler translates a program into another representation that can subsequently be executed by some underlying machine, or interpreter. A tombstone representation of a compiler is shown in figure 3b. The translation process can occur in a pipeline of an arbitrary number of stages in which a source program is transformed into intermediate representations (IR) to aid its analysis or further transformation.

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 [10]. At its core, the three projections are based on the theory of function *specialization* (or in mathematical terms *projection*). Given a function $f(x, y)$, one can produce a new unique specialized function $f_x(y)$ for a fixed value of x. In program specialization, we consider f to be a program and the inputs to said program are sets of *static* data x (known before the program's runtime) and *dynamic* data y (only known once a program starts).

A *partial evaluator* (also called *residualizer*) is usually denoted by *mix* and takes as input 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 x:

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

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

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

We could now go a step further and instead of specializing an interpreter specialize *mix* itself and consider the interpreter to be its static input. We pass to $[\![mix]\!]$ its source, *mix*, and an interpreter, int^L, a process that 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):

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

A practical realization of the Futamura projections has since been an active area of research. The difficulty in their implementation is the question of *how* one can specialize an interpreter and meanwhile also generate the most efficient and correct code, a question that is being explored in the design of partial evaluators to this day [11, 12].

2.2 Type-Directed Partial Evaluation

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

The *binding-time analysis (BTA)* stage inside a PE determines whether expressions can be reduced at specialization time or should be preserved in the residual program. A BTA produces a *division* [12]; 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 a regular 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 [12] (or more recently a hybrid between the two due to Shali et al. [13]). Offline PE performs a BTA before it begins specialization whereas the online approach makes decisions about whether to residualize expressions once partial evaluation has begun.

Danvy devised a method of implementing a partial evaluator solely based on the ideas of normalization in the simply-typed λ-calculus called *Type-Directed Partial Evaluation (TDPE)* [14]. The result is a remarkably simple methodology of implementing residualizers with a binding-time analysis that is completely driven by the *types* of the expressions being specialized. TDPE is built on three core concepts [15, 16]:

1. BTA produces an expression annotated with static or dynamic binding-times such that reducing static terms yields a *completely dynamic* residual expression (i.e., an expression containing only dynamic variables)
2. BTA should be able to generate code that includes *binding-time coercions* which are type corrections that help to convert between dynamic and static values for cases where doing so benefits the residualization process
3. Static reduction is performed by evaluation of an expression and well-formedness of the generated code is guaranteed by the implementing language's type system

Consider the example from Danvy's description of TDPE [14] in (1). The aim is to annotate the function applications (denoted by @) and definitions with a static (overline) or dynamic (underline) binding-time.

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

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

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

$\triangleright$ *(via static reduction)*

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

Instead we can use an η-expansion to turn instances of f as a higher-order value into static function applications. The resulting annotations are shown in (4) and the η-expansion is highlighted in green. After static reduction we obtain the optimal expression in (5) that only contains dynamic values, only a single unfolding of f and no β-redexes. Danvy then generalizes the η-expansion into a class of coercions that permit residualization of static values in dynamic contexts. We represent such coercions with a $\downarrow^t$ (or *lift* as we use in later sections and as defined in figure 4) which represents the conversion from a static value of type t to a dynamic value. Using the coercion operator, TDPE would yield the annotation in (6).

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

$\triangleright$ *(via static reduction)*

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

To summarize, TDPE makes use of the type system of a two-level language to direct the residualization of expressions. The output of this lightweight BTA is an expression whose terms are annotated with dynamic or static binding times which when statically reduced yield a purely dynamic expression. Static reduction is performed through regular evaluation of the two-level language's interpreter.

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

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

TDPE includes a set of operators to convert between dynamic and static terms to aid the optimality of generated code. The two classes of coercion operators are reification (also called lift) and reflection both of which are defined on product, function and literal types. Residualization is finally defined as performing annotations and binding-time coercions followed by static reduction.

As dicussed in section 5.1, the Pink [7] partial evaluator implements most of the binding time coercions according to figure 4's definitions with the exception of *reflect*, which is implemented using an algorithm attributed to Eijiro Sumii [17] to handle order of evaluation of side-effects.

Reification (Lifting)

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

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

where x is not a free variable in v

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

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

Reflection

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

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

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

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

Figure 4: Reduction rules for reification (static to dynamic) and reflection (dynamic to static) in TDPE as defined by Danvy [14] where t denotes types, v denotes static values, e denotes dynamic expressions. The syntax $cons/car/cdr$ corresponds to the LISP functions of the same name that create a pair, extract the first element of a pair and extract the second element of a pair respectively.

2.3 $\lambda_{\uparrow\downarrow}$ Overview

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

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

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

```
// expressions (i.e., dynamic data)
abstract class Exp
case class Lit(n:Int) extends Exp       // Integers
case class Sym(s:String) extends Exp    // Strings
case class Var(n:Int) extends Exp       // Variables
case class Lam(e:Exp) extends Exp       // Lambdas
...
// values (i.e., static data)
abstract class Val
type Env = List[Val]                        // Environment
case class Cst(n:Int) extends Val           // Integers
case class Str(s:String) extends Val        // Strings
case class Clo(env:Env,e:Exp) extends Val   // Closures
case class Code(e:Exp) extends Val          // Residual dynamic data
...

// Converts Expressions (Exp) into Values (Val)
def evalms(env: Env, e: Exp): Val = e match {
    case Lit(n) => Cst(n)
    case Sym(s) => Str(s)
    case Var(n) => env(n)
    case Lam(e) => Clo(env,e)
    case Lift(e) =>
      Code(lift(evalms(env,e)))
    ...
    case If(c,a,b) =>
      evalms(env,c) match {
        case Cst(n) =>
          if (n != 0) evalms(env,a) else evalms(env,b)
        case (Code(c1)) => // Generate an if-statement if conditional is dynamic
          reflectc(If(c1, reifyc(evalms(env,a)), reifyc(evalms(env,b))))
      }
    ...
}
...
var stBlock: List[(Int, Exp)] = Nil
def reify(f: => Exp) = run {
    stBlock = Nil
    val last = f
    (stBlock.map(_._2) foldRight last)(Let) // Let-insertion occurs here
}
def reflect(s:Exp) = {
    stBlock :+= (stFresh, s)
    fresh()
}
// NBE-style 'reify' operator (semantics -> syntax)
def lift(v: Val): Exp = v match {
    case Cst(n) => // number
      Lit(n)
    case Str(s) => // string
      Sym(s)
    case Tup(Code(u),Code(v)) => reflect(Cons(u,v))
    case Clo(env2,e2) => // function
      stFun collectFirst { case (n,`env2`,`e2`) => n } match {
        case Some(n) =>
          Var(n)
        case None =>
```

```
            stFun :+= (stFresh,env2,e2)
            reflect(Lam(reify{ val Code(r) = evalms(env2:+Code(fresh()):+Code(fresh()),e2); r }))
        }
    case Code(e) => reflect(Lift(e))
}
...
```

Listing 1: Main points of interest of the $\lambda_{\uparrow\downarrow}$ evaluator architecture.

For convenience the Pink framework additionally provides a LISP front-end that translates LISP s-expressions into $\lambda_{\uparrow\downarrow}$ expressions. We make use of the notion of *stage-polymorphism* introduced by Offenbeck et al. [18] 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 with. This is achieved by operators that are polymorphic over what stage they operate on and is simply implemented as shown in figure 5. Whenever the *lift* operator is now used in the *interpreter* or *compiler* it will cause *evalms* to either evaluate or generate code respectively.

```
(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]

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

2.4 The SECD Machine and Instruction Set (ISA)

The SECD machine due to Landin [19] is a well-studied stack-based abstract machine initially developed in order to provide a model for evaluation of terms in the λ-calculus. All operations on the original SECD machine are performed on four registers: stack (S), environment (E), control (C), dump (D). *C* holds a list of instructions that should be executed. *E* stores values of free variables, function arguments, function return values and functions themselves. The *S* register stores results of function-local operations and the *D* register is used to save and restore state in the machine when performing control flow operations. A step function makes sure the machine progresses by reading next instructions and operands from the remaining entries in the control register and terminates at a **STOP** or **WRITEC** instruction, at which point the machine returns all values or a single value from the S-register respectively.

We now describe the instruction set and implementation details described by Kogge [20], which itself is a followup to Henderson's LispKit SECD machine [21]. 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.

2.4.1 Examples

- **LDC** is used to load an operand (a string or constant) onto the stack (i.e., S-register). **SEL** is used to branch 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 to resume the rest of the program while placing the value computed in a branch on top of the stack.

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

Result: (done -5)
```

- **LDF** defines a SECD function (i.e., a list of instructions stored as an element in *S*). **AP** applies a SECD function to the second element on the stack which in the example below is NIL (i.e., the LISP empty list '()). **RTN** places the value computed in the function onto the top of the stack and resumes the rest of the program.

```
NIL LDF
    (LDC 10 LDC 20 MPY RTN)
AP
STOP
```

```
    Result: (200)
```

- A recursive call consists of two SECD functions: a recursive function and a function that initiates the first recursive call. **RAP** will set the argument to the recursive function such that it will always find itself in the environment, thus allowing recursion. Additionally, it will initiate the first recursive call. In the below example the recursive function does not have an exit condition and will continuously call itself with the closure found through **LD** in the environment. 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
      (NIL LD (2 1) AP RTN)
  CONS LDF
      (NIL LD (2 1) AP RTN) RAP STOP

      Result: does not terminate
  ```

3 Heterogeneity

A central part of our study revolves around the notion of heterogeneous towers. Prior work on towers of interpreters that inspired some these concepts includes Sturdy's work on the Platypus language framework that provided a mixed-language interpreter built from a reflective tower [4], Jones et al.'s Mix partial evaluator [22] 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 systems of meta-circular interpreters into one-pass compilers. We continue from where the latter left of, namely the question of how one might achieve the effect of compiling multiple interpreters in heterogeneous settings. Our definition of *heterogeneous* is as follows:

Definition 3.1. Towers of interpreters are systems of interpreters, $I_0^L, I_1^L, ..., I_n^L$ where $n \in \mathbb{R}_{\geq 0}$ and I_n^L determines an interpreter at level n interpreted by I_{n-1}^L, written in language L. The language at interpreter level n is denoted L_n

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

1. For any two adjacent interpreters I_n and I_{n-1} where $n \in \mathbb{R}_{\geq 0} : L_n \not\equiv L_{n-1}$

2. For any two adjacent interpreters used in the tower, I_n and I_{n-1}, the operational semantics *or* the representation of data is different between the two

3.1 Absence of: Meta-circularity

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

3.2 Absence of: Reflection

The ability to introspect and change the state of an interpreter during execution is a tool reflective languages use for implementation of debuggers, tracers or even language features. Reflection in reflective towers implies the ability to modify an interpreter's interpreter. With reflection, however, programs can begin to become difficult to reason about and preventive measures for potentially destructive operations on a running interpreter's semantics introduces overhead. Towers that we are interested in rarely provide reflective capabilities in every, or even a single, of its interpreters.

3.3 Semantic Gap and Mixed Language Systems

An early mention of non-reflective and non-metacircular towers was provided in the first step of Danvy et al.'s denotational description of the reflective tower model [2]. 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 serves as the motivation for the second point of definition 3.2. We call the difference in operational semantics or data representation between two interpreters a *semantic gap*.

An extensive look at mixed languages in reflective towers was performed in chapter 5 of Sturdy's thesis [4] where he highlighted the importance of supporting a mixture of languages within his interpretation framework and provided examples of multi-layer systems such as invocation of the UNIX shell through Make a script. Sturdy goes on to introduce into his framework support for mixed languages that transform to a LISP parse tree to fit the reflective tower model. We remove the requirement of reflectivity and argue that this provides a convenient way of collapsing, through partial evaluation a mixed level tower of interpreters. While Sturdy's framework *Platypus* is a reflective interpretation of mixed languages, we construct a non-reflective tower consisting of mixed languages.

One of our motivations stems from the realization that systems consisting of several layers of interpretation can feasibly be constructed. A hypothetical tower of interpreters that served as a model for the one we built throughout our work was described in Amin et al.'s paper on collapsing towers [7] and is depicted as a tombstone diagram in figure 2b. As a comparison our diagram is shown in 2a. We replace the x86 emulator with a SECD abstract machine interpreter and Python with our own functional toy language, *Meta-eval*. Multi-lambda is the partial evaluator and multi-level core language from Pink [7]. Although here the tower grows upwards and to the left, this need not be. The compilers, or *translators*, from Meta-eval to SECD and from Lisp to Multi-lambda have been implemented in Scala purely for simplicity. To realize a completely vertical tower (i.e., consisting of interpreters only), our Lisp to Multi-lambda translator, which converts LISP source to Scala ASTs, could be ommitted letting the base evaluator evaluate s-expressions directly. Similarly, the Meta-eval to SECD compiler could be implemented in SECD instructions itself. However, we argue that the presence of compilation layers in our experimental tower resembles a setting in practice more closely and adds some insightful challenges to our experiments.

4 Recipe to Collapsing Towers

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

4.1 TDPE and Staging a Definitional Interpreter

Pink defines a multi-level language that differentiates between static and dynamic values. This is essential to express binding-time information. Then the lift and reflect operators are implemented such that BTA can coerce static to dynamic values or vice versa respectively.

In his original description of TDPE Danvy [14] showed that we can use above tools to residualize a definitional interpreter with respect to a source program. Pink demonstrates how to stage such an interpreter by wrapping all literal, function and product types in calls to a TDPE-style lift (i.e., reify) operator. Additionally, Pink introduces the concept of *binding-time agnostic staging*. Here, the same definition of an interpreter can be used to residualize or simply evaluate a program. In reference to Pink, *staging an interpreter* means lifting types as described above but also activating *code generation mode*, a detail we use in the next section.

4.2 Construction and Collapse of a Tower

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

When we now execute the tower (i.e., invoking the partial evaluator at the base) only the top-most evaluator residualizes while all others evaluate. Because of meta-circularity, instead of evaluating a user-value it is now being lifted at all stages essentially propagating binding-time information from top-most interpreter to the base PE. At the last interpreter above the base evaluator, the lift now dispatches to the actual reify operation as specified by TDPE. Effectively after residualization the generated program will only include the values staged at the top-most interpreter while the rest of the tower was reduced at specialization time since they were not evaluated in code-generation mode.

4.3 Effect of Heterogeneity

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

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

5 Construction of an Experimental Heterogeneous Tower

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

The first level in our tower is the evaluator for $\lambda_{\uparrow\downarrow}$ which we mostly take from the way it was defined in Pink. It serves as our partial evaluator and generates $\lambda_{\uparrow\downarrow}$ terms, in A-Normal Form [23], given LISP expressions through its front-end.

To reduce the amount of generated code we add logic within $\lambda_{\uparrow\downarrow}$'s *reflect* that reduces purely static expressions. Reducible expressions include arithmetic and list access operations. We refer to these in later sections as *smart constructors* and they aid in normalizing static expressions that the division (described in section 5.2.1) does not permit. These are shown in an excerpt of $\lambda_{\uparrow\downarrow}$'s definitional interpreter in figure 6.

```
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 _ =>
            stBlock :+= (stFresh, s)
            fresh()
        }
}
```

Figure 6: *Smart constructors* (highlighted in green) in $\lambda_{\uparrow\downarrow}$'s *reflect*

5.2 Level 2: SECD

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

1. **Maturity**: SECD was the first abstract machine of its kind developed by Landin in 1964 [19]. Since then it has thoroughly been studied and documented [24, 25, 21] making it a strong foundation to build on.

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

3. **Extensibility**: Extensions to the machine, many of which are described by Kogge [20], have been developed to support richer features then the ones available in its most basic form including parallel computation, lazy evaluation and call/cc semantics.

An additional benefit of using a LISP machine is that the $\lambda_{\uparrow\downarrow}$ framework we use as our partial evaluator also features a LISP front-end that supports all the list-processing primitives which Kogge used to describe the operational semantics of SECD [20] with. Our first step in constructing the heterogeneous tower is implementing the standard SECD machine (described by the small-step semantics and compiler from Kogge's book on symbolic computation [20]) using $\lambda_{\uparrow\downarrow}$'s LISP front-end. We model the machine through a case-based evaluator with a *step* function at its core that advances the state of the machine until a **STOP** or **WRITEC** instruction is encountered.

5.2.1 Staging a SECD Machine

Since a part of our experiments of collapsing towers is concerned with the effect of staging at different levels in the tower, we want to design the SECD machine such that it can later be staged. By definition, a staged evaluator should have a means of generating some form of intermediate representation, for example residual code, followed by a way to execute it (directly or through further stages). We split the evaluation of the SECD machine into two stages: reduction of static values and residualization of dynamic values.

From the architecture of a SECD machine the intended place for free variables and user input to live in is the environment register. A simple example in terms of SECD instructions is as follows:

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

Here we load only a single value of 10 into the environment and omit the second argument that the LDF expects and uses inside its body. Instead it simply loads at a location not yet available and trusts the user to provide the missing value at run-time. Rewriting the above in LISP-like syntax we have the following:

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

where y is a free variable.

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. Staged execution in our framework makes use of the partial evaluator used in Pink [7] to generate code in $\lambda_{\uparrow\downarrow}$. Before being able to stage a SECD machine we define our division by where static values can be transferred from. If a dynamic value can be transferred from a register, A, to another register, B, we classify register B as dynamic. 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

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

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

serve as either completely static or mixed valued. An example would be a machine that simply performs arithmetic on two integers and returns the result. The state machine transitions would occur as shown in table 2. As the programmer we know there is no unknown input and the expression can simply be reduced to the value 30 following the SECD small-step semantics. However, by default our division assumes the S-register to be dynamic and thus generates code for the addition of 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. As such we keep this division as the basis for our staged SECD machine since it is less intrusive to the machine's definitional interpreter and still allows us to residualize efficiently.

Through its LISP front-end the $\lambda_{\uparrow\downarrow}$ evaluator can operate as a partial evaluator by exposing its *lift* operator. Using this operator we can then annotate the interpreter we want to stage according to a pre-defined division. Given the division in table 1 we implement the SECD machine as a definitional interpreter.

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

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

5.2.2 The Interpreter

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

Figure 7: Structure of interpreter for SECD machine (unstaged). Lambdas take two arguments, a self-reference for recursion which is ignored through a "_" sentinel and a caller supplied argument. All of SECD's stack registers are represented as LISP lists and initialized to empty lists. "..." indicate omitted implementation details. The full interpreter is provided in APPENDIX.

Our staged machine is written in $\lambda_{\uparrow\downarrow}$'s LISP front-end as a traditional case-based virtual machine that dispatches on SECD instructions stored in the C-register. The structure of our interpreter without annotations to stage it is shown in figure 7. Of note are the single-argument self-referential lambdas due to the LISP-frontend and the out-of-order argument list to the machine. To allow a user to supply instructions to the machine we return a lambda that accepts input to the control register (C) in line 12 of figure 7. Once a SECD program is provided we curry the machine with respect to the last *environment* register which is where user-supplied arguments go. An example invocation is

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

where `secd` is the source of figure 7 and the arguments to the machine are the arithmetic example of table 2 and an empty environment respectively.

Similar to how the Pink interpreter is staged [7] we annotate the expressions of the language that our interpreter defines for which we want to be able to generate code for with the stage-polymorphic *maybe-lift* operators defined in 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 8 shows these preliminary annotations. We wrap the initializing call to the SECD machine in *maybe-lift* as well because we want to specialize the machine without the dynamic input of the environment provided yet. Hence line 12 in figure 8 simply signals the PE to generate code for the curried SECD machine.

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

Figure 8: Annotated version of the SECD interpreter in figure 7 with differences highlighted in green. *maybe-lift* is used to signal to the PE that we want to generate code for the wrapped expression. Here we follow exactly the division of table 1. These changes are not enough to fully stage the machine as discussed in section 5.2.2

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

Key to us removing interpretative overhead of the SECD machine is the elimination of the unnecessary instruction dispatch logic, whose effect on interpreter efficiency was studied extensively by Ertl et al. [26], in the specialized code. Since the SECD program is known at PE time and thus has static binding time, we do not want to lift the constants against which we compare the control register. However, if we put something into the control register that is dynamic we are suddenly comparing dynamic and static values which is a specialization time error at best and non-termination of the PE at worst.

Another issue we dealt with in the process of writing the staged SECD interpreter is the implementation of the RAP instruction which is responsible for recursive applications. The instruction essentially works in two steps. First the user creates two closures on the stack. One which holds the recursive function definition and another which contains a function that initiates the recursive call and prepares any necessary arguments. **RAP** calls the latter and performs the subtle but crucial next step. It forms a knot in the environment such that when the recursive function looks up the first argument in the environment it finds the recursive closure. According to Kogge's [20] description of the SECD operational semantics this requires an instruction that is able to mutate variables, a restriction that subsequent abstract machines such as the CESK machine [27] do not impose. Given the choice between adding support for an underlying `set-car!` instruction in $\lambda_{\uparrow\downarrow}$ or extending the SECD machine such that recursive functions applications do not require mutation in the underlying language we decided to experiment on both methodologies. We first discuss our initial implementation of the latter but evaluate our findings in designing the former in section 5.4 as well.

5.2.3 Tying the Knot

We now provide a substantial redesign to the internal `RAP` calling convention while keeping the semantics of the instruction in-tact in order to allow SECD style recursive function calls without the need for mutable variables and more crucially enable their partial evaluation. The idea is to wrap the recursive SECD instructions in a closure at the LISP-level, perform residualization on the closure and distinguish between returning from a regular as opposed to recursive function to ensure termination of our specializer. We demonstrate the issue of partially evaluating a recursive call in the standard SECD semantics on the example in figure 9 which shows a recursive function that decrements a user provided number down to zero.

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

Figure 9: An example recursive function application annotated to show the issue with partially evaluating this type of construct.

Were we to simply reduce this program by evaluating the machine we would hit non-termination of our PE. Our 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 (`LD (1 1)`) with a constant

0. By virtue of 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 in the actual machine we hit this choice again repeatedly without stopping because we have no way of signalling to stop partially evaluating. Figure 10 highlights this in the internals of the VM.

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

Figure 10: Snippet from the internals of the SECD interpreter from section 5.2.2. Highlighted are the locations at which our partial evaluator does not terminate. TDPE attempts to evaluate both branches because we cannot determine the outcome of the conditional.

Instead of evaluating the recursive call we want to instead generate the call and function 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 11 shows the modifications to the operational semantics of Landin's SECD machine [19] which allow it to be partially evaluatable with a TDPE and do not require a `set-car!` in the underlying language.

Firstly, equation 15 changes the representation of functions in the SECD machine from simply lists of instructions to a function that accepts an environment and upon invocation takes a step in the abstract machine with the instructions put into control-register by **LDF**, essentially performing the role that **AP** previously did. Working with thunks makes the necessary changes to stage the machine less intrusive and effectively prevents the elements of the control register being marked as dynamic. This is in line with the ideas of Danvy et al. [28] which showed that eta-expansion can enable partial evaluation by hiding dynamic values from static contexts. Note also that we add a new functions register, which we refer to as the **F-register** or simply **fns** which is responsible for holding the recursive instructions of a **RAP** call.

In equation 16 the **RAP** instruction 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 and an environment. **RAP** then saves the current

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

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

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

$$\begin{aligned}
\text{v.s e (}\mathbf{RTN}\text{.c) (s' e' c' d' f'.d) f} &\longrightarrow \text{(lift-all v)}\quad \text{if d-register is tagged with 'ret}\\
&\ \text{(}\lambda\text{x.((car s') (cons (car x) (cddr s'))) (cddr s1)}\\
&\qquad \text{if d-register is tagged with 'fromldr}\\
&\qquad \text{where s1 := v@}\_\\
&\ \text{(v.s') e'c'd'f'}\qquad \text{otherwise}
\end{aligned} \tag{18}$$

$$\begin{aligned}
\text{(fn env.v).s e (}\mathbf{AP}\text{.c) ('() env') f} &\longrightarrow \text{res.s env c d f}\\
\text{where res} &:=\ \text{fn@(lift-all (v s.env'))}
\end{aligned} \tag{19}$$

$$\begin{aligned}
\text{(v m.s) e (}\mathbf{LIFT}\text{.c) d f} &\longrightarrow \text{res.s e c d f}\\
\text{where res} &:=\ \text{(lift v)}\quad \text{if (num? v) or (sym? v)}\\
&\ \text{(lift rec)}\quad \text{if (lambda? v)}\\
&\quad \text{where rec :=}\\
&\qquad \text{(}\lambda\text{env.(step '() env.recEnv c'}\\
&\qquad\quad \text{'ret (rec (recOps recEnv).f)))}\\
&\ \text{(lift (}\lambda\text{env.(step '() env.m c' 'ret f)))}\quad \text{otherwise}
\end{aligned} \tag{20}$$

Figure 11: Modifications to the SECD operational semantics as presented by Kogge [20]. These modifications permit us to stage our SECD machine interpreter and enable residualization of recursive function applications.

contents of all registers on the D-register and appends a closure on the fns-register. This closure when applied to an environment, takes a step in the machine with the control-register containing instructions of the recursive function body and a self-reference to the closure. Additionally, applying the closure tags the dump-register with a "ret" tag later used as an indicator to stop evaluating the current call, which is crucial to aid termination during specialization time.

In the traditional SECD machine both the recursive and the calling function are kept in the environment and then loaded on the stack using **LD**, subsequently called using **AP**. However, for simplicity we keep recursive functions on the fns-register. Thus we introduce a new **LDR** instruction that returns the contents of the fns-register by index, just as **LD** does for the E-register. However, we wrap the action of finding a function in F in a thunk to be able to generate code for this operation during partial evaluation.

The **RTN** instruction works on the state set up by the modified instructions above to decide the interpreter's behaviour upon returning from a SECD function. In the original semantics of SECD, **RTN** would reinstall the state of all registers from the dump and add the top most value of the current stack-register back onto the restored stack-register. This modelled the return from a function application. As we previously showed, taking another step in the machine when specializing a recursive function will lead to non-termination of our specializer. Thus, we simply stop evaluation when returning from a function by tagging the register with a *'ret* symbol and returning the top-most value on the stack to the call site of a lambda. This works because function definitions reside in lambdas in the interpreter now and SECD function application is lambda invocation. The last case we are concerned with is the currying of SECD functions. This occurs when we invoke a **RTN** immediately after an **LDR** which loaded a thunk into the return location on the stack. To properly return a lambda we unpack the closure from **LDR**'s thunk, construct a **LDF**-style closure, lift and then return it.

To rewrite the example from figure 9 with the new semantics we load the recursive function using the new **LDR** instead of the **LD** instruction as highlighted in figure 12.

The above changes to the machine show that to permit partial evaluation of the original SECD semantics, an intrusive set of changes which necessitate knowledge of the inner workings of the machine are required. The complexity partially arises from the fact that the stack-based semantics do not lend themselves well to TDPE through $\lambda_{\uparrow\downarrow}$. We have to convert representations of program constructs, particularly closures, from how SECD stores them to what the underlying PE expects and is able to lift. Since $\lambda_{\uparrow\downarrow}$is built around lifting closures, literals and cons-pairs we have to wrap function definitions in thunks which complicates calling conventions within the machine. Additionally, deciding on and implementing a congruent division for a SECD-style abstract machine, where values can move be-

```
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))
```

Figure 12: Recursive countdown example from figure 9 rewritten with the SECD operational semantics in figure 11

tween a set of stack registers, requires careful bookkeeping of non-recursive versus recursive function applications and online binding-time analysis checks. On one hand, the most efficient code is generated by allowing as much of the register contents to be static. On the other hand, the finer-grained the division the more difficult to reason about and potentially less extensible a division becomes.

5.2.4 SECD Compiler

To continue the construction of a tower where each level is performing actual interpretation of the level above we would have to implement an interpreter written in SECD instructions as the next level in the tower. To speed up the development process and aid debuggability we implement compiler that parses a LISP-like language, which we refer to as *SecdLisp*, and generates SECD instructions. It is based on the compiler described by Kogge [20] though with modifications (see figure 13) to support our modified calling conventions and additional registers described in section 5.2.3. Since we hold recursive function definitions in the fns-register 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. We keep track of and increment an offset into the E-register during compilation whenever a free variable is detected via a missed look-up in the environment. The **quote** built-in (equation 23) is used to build lists of identifiers from s-expressions. This is useful when we extend the tower in later sections and want to pass SecdLisp programs as static data to the machine.

Given a source program in SecdLisp we invoke the compiler as shown in figure 14. As line 3 suggests we feed the compiled SECD instructions to the SECD machine interpreter source described in section

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

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

Syntax : (**quote** ⟨expr⟩) (23)
Code : `LDC` $\langle id_0 \rangle$ `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 13: Modifications to the SECD compiler described by Kogge [20]

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

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

Figure 14: Compilation and execution of a program in SecdLisp on our PE framework.

5.2.5 Example

Figure 15a shows a program to compute factorial numbers recursively written in SecdLisp. The program is translated into SECD instructions by our compiler (see section 5.2.4) and then input to our staged machine. Figure 15c is the corresponding residualized program generated by $\lambda_{\uparrow\downarrow}$(and prettified to LISP syntax). An immediate observation we can make is that the dispatch logic of the SECD interpreter has been reduced away successfully. Additionally, we see the body of the recursive function being generated in the output code thanks to the modifications to **RAP**, **AP** and **LDF**. The residual program contains two lambdas, one that executes factorial and another that takes input from the user in form of the environment (line 25). In the function body itself (lines 4 to 20), however, the numerous *cons* calls and repeated list access operations (*car*, *cdr*) indicate that traces of the underlying SECD semantics are left in the generated code and cannot be reduced further without changing the architecture of the underlying machine.

5.3 Level 3: Meta-Eval

Armed with a staged SECD machine and a language to target it with, we build the next interpreter in the tower that gets compiled into SECD instructions. The interpreter defines a language called Meta-Eval, M_e, whose syntax is described in figure 16. The language is based on Jones et al.'s Mixwell and M languages in their demonstration of the Mix partial evaluator [22] in the sense that the toy language is a LISP derivative and $M_e$serves as a demonstration of evaluating a non-trivial program through our extended SECD machine. However, we omit a built-in operator such as Mixwell's **read** that helps identify binding times of a user program and stage our interpreter in section 5.3.1 instead. M_e also enables the possibility of implementing substantial user-level programs and further levels in the tower. The reason for choosing a LISP-like language syntax again is that it allows us to reuse $\lambda_{\uparrow\downarrow}$ LISP front-end's parsing infrastructure. Further work would benefit from changing representation of data structures like closures to increase the semantic gaps between $\lambda_{\uparrow\downarrow}$ and M_e and demonstrate even more heterogeneity than in the tower we built.

M_e supports the traditional functional features such as recursion, first-class functions, currying but also LISP-like quotation. We implement the language as a case-based interpreter shown in figure 17. Note that to reduce complexity in our implementation we define our interpreter within a Scala string. Line 1 starts the definition of a function, `meta_eval`, that allows us to inject a string representing the M_e program and another representing the implementation of a **lift** operator. This mimics the polymorphic **maybe-lift** we define in $\lambda_{\uparrow\downarrow}$.

Figure 18 shows the M_e interpreter running a program computing factorial using the Y-combinator for recursion (figure 18a) on our staged VM. As opposed to producing an optimal residual program we now see the dispatch logic of our M_e interpreter in the generated code (figure 19). As the programmer we know this control flow can be reduced even further since the M_e source program is static data.

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

(a) LISP Front-end

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

(b) SECD Instructions

```
(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) Prettified Generated Code

Figure 15: Example Factorial

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

Figure 16: Syntax of $M_e$which gets compiled into SECD instructions for interpretation by the SECD machine

5.3.1 Staging M_e and Collapsing the Tower

In an effort to further optimize our generated code from the example in figure 18 we stage the M_e interpreter. As indicated by Amin et al. during their demonstration of collapsing towers written in Pink [7], staging at the user-most level of a tower of interpreters should yield the most optimal code. In this section we aim to demonstrate that staging at other levels than the top-most interpreter does indeed generate less efficient residual programs.

Staging the M_e interpreter is performed just as in Pink [7] by lifting all literals and closures returned by the interpreter and letting $\lambda_{\uparrow\downarrow}$'s evaluator generate code of operations performed on the lifted values. The main caveat unique to M_e's interpreter is a consequence of heterogeneity: M_e does not have access to a builtin *lift* operator. This poses the crucial question of how one can propagate the concept of lifting expressions through levels of the tower without having to expose it at all levels. We take the route of making a *lift* operator available to the levels above the SECD machine which requires the implementation of a new SECD **LIFT** instruction.

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

Figure 17: Staged interpreter for M_e

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

```
LDC 10 LIFT STOP
```

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

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

The other two cases that our **LIFT** semantics permit are regular functions constructed via **LDF** and closures set up by **RAP**. Behind the apparent complexity again lies the same recipe for staging an interpreter as we identified before but in this case operating on the top most value of the stack. We make sure to lift the operand if it is a number or a string. In the case that the operand is a closure or function we construct, lift and return a new lambda using the state we stored in registers **F** and **D**. Note the subtle difference in behaviour between lifting a SECD closure or a lambda. The former is defined by **LDF** or **RAP** and includes instructions waiting to be executed wrapped in a lambda and auxiliary state information such as the environment. In this case we simply construct a lambda that

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

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

(a) LISP Front-end

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

(b) SECD Instructions

Figure 18: Example factorial on M_e

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

Figure 19: Generated code running the example in 18 on a staged SECD machine. Traces of the M_e's dispatch logic is highlighted in green.

takes an environment and performs a step in the machine with the instructions that were wrapped. However, a lambda on the stack only occurs as a result of a call to **LDR** in which case we unpack the instructions and state from the thunk, *recOps* and *recEnv* respectively, and again wrap a call to the step function in a lambda.

Through the addition of a *lift* built-in into SecdLisp we can now residualize the M_e interpreter and run it on our SECD interpreter. The residual program for the factorial example (figure 18) is shown in figure 21 and the corresponding SECD instructions that M_e compiled down to in figure 20. The generated SECD instructions are the same as in the unstaged M_e interpreter with the exception of the newly inserted **LIFT** instructions as we have specified in the interpreter definition. This has the effect that the residual program resembles exactly the M_e definition of our program but now in terms of $\lambda_{\uparrow\downarrow}$ and all traces of the SECD machine have vanished. This demonstrates that we successfully removed all layers of interpretation between the base evaluator ($\lambda_{\uparrow\downarrow}$) and the user-most interpreter (M_e). Comparing this configuration to running our example on the staged machine (and unstaged M_e) we can see that the structure of the generated code resembles the structure of the interpreter that we staged. When staging at the SECD level we could see traces of stack-like operations that to the programmer seemed to be 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 <=== Me 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 20: 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 21: Prettified Residual Program in $\lambda_{\uparrow\downarrow}$for an example factorial on a staged $M_e$interpreter

5.4 Level 4: String Matcher

Following the experiments performed in Amin et al.'s study of towers [7], we extend our tower further one last time and implement a regular expression matcher proposed by Kernighan et al. [29] in M_e. The source is shown in figure 22.

```
(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 22: 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.

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" where "done" serves as a terminal marker for the matcher. 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 24a. It is far from the most efficient version and we can see clear traces of the matcher logic in the generated code such as a check for an “_” character on line 21 while our pattern against which we specialize does not contain any.

Now we stage the matcher according to the implementation provided in the Pink experiments [7] by simply lifting all symbols on return from the matcher and the initial recursive call to begin matching (see figure 23).

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

Figure 23: Staged regular expression matcher by wrapping returned symbols in M_e’s *lift*.

Continuing the trend, the generated code when staging the user-most interpreter (in this case the string matcher) yields the optimal residual program. As we wanted, the specialized matcher in figure 24b will succeed on any input string.

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

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

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

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

6 Conclusions and Future Work

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

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

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

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

We first chose a SECD interpreter to be the level that adds heterogeneity to the tower. The lack of a built-in lift operator and the difference in how it represents program constructs such as closures required us to adapt the internals of the machine to aid the termination and efficient residualization of our TDPE. A shortcoming of TDPE that we encountered when staging our SECD interpreter is the lack of a general recipe for staging an abstract machine. Staging an interpreter amounts to reifying literals, lambdas and product types it returns. In an abstract machine the semantics are not guaranteed to distinguish these types by data structures or a type-system but can instead rely on dedicated instructions for each. Hence, the points to reify at are dictated by the architecture of the underlying machine. In a stack-based machine we created a conservative division tailored to the SECD stack-reigsters and reduced static expressions in the TDPE reflect operator to achieve optimal residualization.

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

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

Although we showed the successful collapse of a heterogeneous tower of interpreters, realizing our methodology on a practical setting such as the Python-x86-JavaScript tower will require additional work. Our approach to propagating the TDPE binding-time information involves the implementation of a reification operator in each interpreter that is missing it. This requires the deconstruction of the types that TDPE's reify operates on and conversion to the representation that the interpreter below in the tower expects. These changes would require intimate knowledge of and intrusive changes to an interpreter. Additionally, in our experimental tower we do not consider the residualization of side-effects which a useful collapse procedure would need for wider applicability.

In smaller scale settings in which towers consist of embedded DSL interpreters or regular expression matchers, however, our experiments could help the optimization of such systems using the simple to implement TDPE even in the absence of meta-circularity.

6.1 Future Work

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] 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.

[3] 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.

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

[10] 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.

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

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

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

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

[15] ——, “Online type-directed partial evaluation,” BRICS, Technical Report RS-97-53, 1997.

[16] 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.

[17] 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.

[18] 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.

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

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

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

[22] 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.

[23] 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.

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

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

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

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

[28] 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.

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

Appendices

Appendix A

SECD

s e (**NIL**.c) d $\longrightarrow$ (nil.s) e c d

s e (**LDC** x.c) d $\longrightarrow$ (x.s) e c d

s e (**LD** (i.j).c) d $\longrightarrow$ (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 $\longrightarrow$ ((*OP* a).s) e c d)

  where *OP* is one of CAR, CDR, ...

(a b.s) e (***OP***.c) d $\longrightarrow$ ((a *OP* b).s) e c d

  where *OP* is one of CONS, ADD, SUB, MPY, ...

(x.s) e (**SEL** ct cf.c) d $\longrightarrow$ s e c? (c.d)

  where c? = ct if x $\neq$ 0, and cf if x = 0

s e (**JOIN**.c) (cr.d) $\longrightarrow$ s e cr d

s e (**LDF** f.c) d $\longrightarrow$ ((f.e).s) e c d

((f.e') v.s) e (**AP**.c) d $\longrightarrow$ NIL (v.e') f (s e c.d)

(x.z) e' (**RTN**.q) (s e c.d) $\longrightarrow$ (x.s) e c d

s e (**DUM**.c) d $\longrightarrow$ s (nil.e) c d

((f.(nil.e)) v.s) (nil.e) (**RAP**.c) d $\longrightarrow$ 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 $\longrightarrow$ halt the machine and return *s*

(x.s) e (**WRITEC**.c) d $\longrightarrow$ halt the machine and return *x*

Figure A.1: SECD Machine instruction transitions mostly according to Kogge’s description [20]. 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 [20] explanation of **RAP**'s semantics. A **letrec** gets translated into a series of SECD function definitions where the last one initiates a recursive call.