Contents

8218 (errors:5) 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 in the tower. One of the earliest mentions of such architectures in literature is a language extension to Lisp called 3-LISP [1] introduced by Smith. The treatment of programs as data is a core concept in the Lisp family of languages and enables convenient implementations of Lisp interpreters written in Lisp themselves. These are known as *meta-circular* interpreters. 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. By way of a, conceptually infinite, *reflective tower* 3-LISP allows interpreters within the tower access to the internal data structures and state of the interpreter adjacent to it. A subsequent study due to Danvy et al. [2] provides a systematic approach to constructing reflective towers. The authors provide a denotational semantic account of reflection and develop a language based on the reflective tower model called *Blond*.

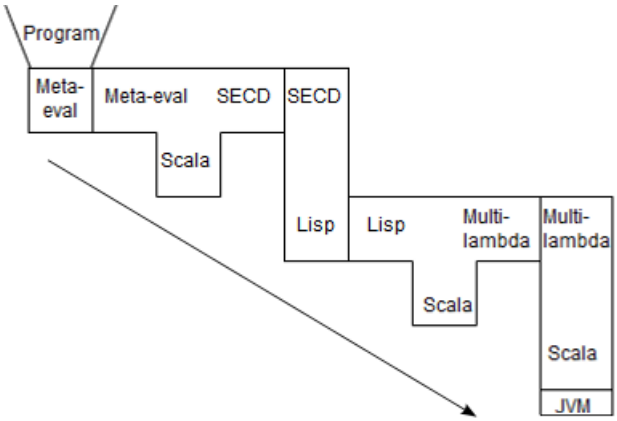
However, with respect to a traditional model of interpretation, a conceptually infinite tower of interpreters adds excessive evaluation overhead solely for the purpose of achieving reflection. In the original proposals of the reflective tower models only minimal attention was given to the imposed cost of performing new interpretation at each level of a tower. Works dating back to Sturdy's thesis on reflection [3] and Danvy et al.'s language Blond [2] hint 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. In his language *Black* [4], Asai et al. uses hand-crafted partial evaluation, and in a later study MetaOcaml [5], to efficiently implement a reflective tower and is the earliest reference to "collapsing" evaluation overhead in towers of interpreters. Finally, Amin et al. introduced a language Pink [6] to demonstrate specifically the collapse of a tower of interpreters.

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.

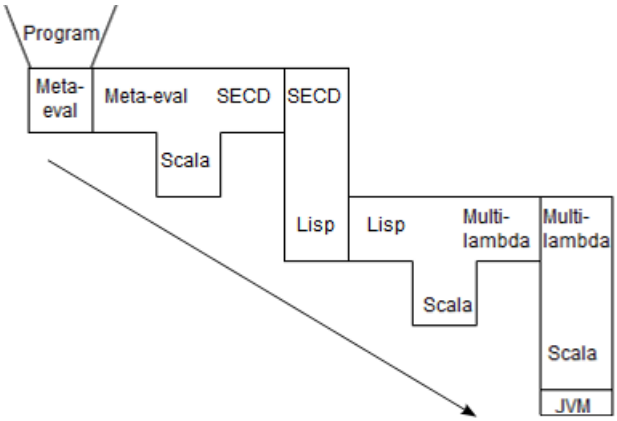
However, these two branches of research do not overlap and work on towers of interpreters rarely studied their counterparts in production systems. It is natural to ask the question of what it would take to apply previous techniques in partial evaluation to a practical setting. This is the question Amin et al. poses in their conclusion after describing Pink [6] and is the starting point to our thesis.

We aim to bring previous work of removing interpretative overhead using partial evaluation to towers in practice. Our study achieves this by constructing a proof-of-concept tower of interpreters that resembles one that could occur in practice and experiment with collapsing it under different configurations and evaluating the resulting optimized programs. We demonstrate that given a multi-level language and a lift operator 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. As part of the development of this framework our contributions include: (1) the development of extensions to the SECD machine that allow it to be staged (2) implementation of a compiler from a minimal LISP-style language to instructions of the staged SECD machine (3) demonstration of collapsing towers of interpreters built on top of the aforementioned SECD machine (4) evaluate the effect of staging at different interpreters within a tower of interpreters (5) demonstrate the ability to use NbE-style lift operators to perform partial evaluation across levels in the tower that are compilers (6) and finally evaluation of the structure of the generated code and possible optimizations.

In section 2 we explain potentially useful background information that cover the fundamental topics we base our framework and experiments on, starting from the link between interpreters and compilers up to how previous work managed to compile reflective towers. Section 4 provides an overview of the partial evaluation framework we base our study on called Pink [6]. We systematically describe the process by which we create a heteregeneous tower of interpreters and incrementally collapse it in sections 5 through 7. Each of these sections discusses the steps we took to create a level in the tower of interpreters as shown in figure 1a. We conclude with an evaluation of our findings followed by a discussion of potential future work in sections 8 and 9 respectively.



(a) A tombstone diagram representation of our framework



(b) A hypothetical tower of interpreters that serves as the model for the tower we built (shown in figure 1a

Figure 1: Comparison between the tower we develop and the one we modelled it after.

2 Background

2.1 Interpretation and Compilation

An interpreter parses and executes instructions based on an input source program. The semantics of a programming language as perceived by its user can be defined by the interpreter itself, in which case we call the interpreter a definitional interpreter [7].

A compiler translates a program into another representation that can subsequently be executed by some underlying machine, or interpreter. This process of translation can occur in a pipeline of an arbitrary number of stages in which a source program is transformed into intermediate representations (IR) to aid it's analysis or further transformation. Modern optimizing compilers consist of stages in which optimization is performed on IR to improve properties of the output code such as execution speed, size or security.

In the 1970s Futamura showed that compilers and interpreters are fundamentally related in elegant way by three equations also known as the Futamura projections [8]. 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 specialized function $f_x(y)$ fixed against the input x. Program specialization considers f to be a program and x and y some inputs to said program:

$$p_x = [\![mix]\!]\ [p, x]$$
$$out = [\![p_x]\!]\ [y]$$

where p is the program being specialized, p_x the specialized program with respect to input x, and mix is the tool that performs the specilization of p. In the above equations 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:

$$\begin{aligned} target &= [\![mix]\!]\ [int_L, src_L] \\ &= [\![comp_L]\!]\ [src_L] \end{aligned}$$

Furthermore, *self-applying mix* and specializing it to an interpreter, int_L, allows one to derive a com-

piler 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. This sparked further development of partial evaluators.

2.2 Type-Directed Partial Evluation (TDPE)

Partial evaluation (PE) is a program optimization technique based on the insight that there is room for statically reducing a program to produce a specialized and hopefully more efficient version. A *partial evaluator* determines whether operations involving inputs to specialize against can be reduced or have to be left in the program. A specialized program, also referred to as *residual program*, is a version of the original program where as much computation has been performed as possible 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 it's *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 result of a *binding-time analysis* is called a *division* and 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 data 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 guarantee termination of a PE but when it can we call it *safe*.

In the literature we distinguish between *online* and *offline* [9] partial evaluation (or more recently a hybrid between the two due to Shali et al. [10]). Offline PE performs a BTA before it begins specialization whereas the online approach makes decisions about whether to residualize expressions once residualization has begun. The PE we use in our framework (section 2.3) is an online PE because it residualizes depending on the type of expressions and does not provide a dedicated binding-time analyzer.

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

Since our work is based on the multi-level language, $\lambda_{\uparrow\downarrow}$, developed by Amin et al. [6] we provide a summary of the core language which is a call-by-value λ-calculus split into two evaluation contexts, one in which expressions are code and the other in which expressions normalize to values. The framework also includes a LISP front-end that translates s-expressions into terms of $\lambda_{\uparrow\downarrow}$. We will be using this framework as the basis for our tower of interpreters. The language distinguishes between *expressions*, which are terms in a $\lambda_{\uparrow\downarrow}$ program, and *values* which represent the static output after evaluating an expression. Thus when we talk about generating code using $\lambda_{\uparrow\downarrow}$, we mean values in the language.

Figure 2 provides an overview of the structure of $\lambda_{\uparrow\downarrow}$ as a partial evaluator. Given an expression the PE evaluates terms to values while differentiating between static values (simply `Val`) and dynamic values (terms wrapped in the `Code` constructor). An expression wrapped in a `Lift` constructor gets evaluated to `Code` and appears in the residual program.

$\lambda_{\uparrow\downarrow}$ is built on the concept of TDPE and implements a powerful construct formally described in [11] called *lift*. The operator converts the semantics of an expression to it's syntax and thus can be said to generate code (i.e., terms of $\lambda_{\uparrow\downarrow}$). 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 it's source provided the possibility of changing the interpreter's internals and enough knowledge of the its semantics.

We make use of the notion of *stage-polymorphism* introduced by Offenbeck et al. [12] to support two modes of operation: (1) regular evaluation (2) generation of $\lambda_{\uparrow\downarrow}$code and its subsequent execution. 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 3. Whenever the *lift* operator is now used in the *interpreter* or *compiler* it will cause *evalms* to either evaluate or generate code respectively.

An advantage of the TDPE and why $\lambda_{\uparrow\downarrow}$ 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 [6] recipe: lift all terminal values an interpreter returns. Once the returned values are dynamic, any operation that includes them is also marked dynamic and will occur in the residual program.

```
// 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)) =>
          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)
}
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))
}
...
```

Figure 2: Main points of interest of the $\lambda_{\uparrow\downarrow}$ evaluator architecture.

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

Figure 3: "Stage-polymorphism" implementation from Amin et al.'s Pink [6]

2.4 Abstract Machines

SECD [13]

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 [3], Jones et al.'s Mix partial evaluator [14] 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, $I_0, I_1, ..., I_n$ where $n \in \mathbb{R}_{\geq 0}$ and I_n determines an interpreter at level n interpreted by I_{n-1}, written in language L such that L_{I_n} is the language interpreter I_n is written in.

A level here is analogous to an instance of an interpreter within the tower and as such level n implies I_n if not mentioned explicitly otherwise.

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

1. For any two levels $n, m \in \mathbb{R}_{\geq 0}, L_{I_n} \not\equiv L_{I_m}$
2. For any two levels $n, m \in \mathbb{R}_{\geq 0}, L_{I_n} \not\blacktriangleleft L_{I_m}$, where $\blacktriangleleft$ implies access to the left-hand side interpreter's state and $m \geq n$
3. For any language used in the tower $L_m \in \Sigma_L, \exists L_a \notin \Sigma_L . L_m \blacktriangleleft L_c \wedge L_c \blacktriangleleft L_c$

A common situation where one find such properties within a system of languages is the embedding of domain-specific languages (DLSs) and we describe the consequence of these properties in the

subsequent sections.

3.1 Absence of: Meta-circularity

The first constraint imposed by definition 3 is that of necessarily mixed languages between levels of an interpretative 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 [4] or Pink [6].

3.2 Absence of: Reflectivity

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. With reflection, however, programs can begin to become difficult to reason about and the extent of control of potentially destructive operations on a running interpreter's semantics introduces overhead. Reflection in reflective towers implies the ability to modify an interpreter's interpreter. Hierarchies of language embeddings as the ones we are interested in rarely provide reflective capabilities at every part of the embedding.

3.3 Mixed Language Systems

An early mention of non-reflective and non-metacircular towers was provided in the first step of Danvy's systematic description of the reflective tower model [2]. However, potential consequences were not further investigated in their study. However, their denotational explanation of general interpretation and description of interpretater state served as a useful foundation for later work and our current study.

An extensive look at mixed languages in reflective towers was performed in chapter 5 of Sturdy's thesis [3] where he highlighted the importance of supporting a mixture of languages within a interpretation framework. Multi-layer systems such as YACC and C or Shell and Make are common practice. 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. Our work is similar in its commmon representation of

```
def reflect(s:Exp) = {
    ...
    case _ =>
        s match {
          case Fst(Cons(a, b)) => a
          case Snd(Cons(a, b)) => a
          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 4: *Smart constructors* (highlighted in red) in $\lambda_{\uparrow\downarrow}$'s *reflect* operator

languages, however, 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 reasonably 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 [6] and is depicted as a tombstone diagram in figure 1b.

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

4.1 Changes to $\lambda_{\uparrow\downarrow}$

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 5.1) does not permit. These are shown in an excerpt of $\lambda_{\uparrow\downarrow}$'s definitional interpreter in figure 4.

5 Level 2: SECD

The SECD machine due to Landin [13] is a well-studied stack-based abstract machine initially developed in order to provide a machine model capable of interpreting LISP programs. 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 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.

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 [13]. Since then it has thoroughly been studied and documented [15, 16, 17] 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. 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 such that SECD satisfies this property well.

3. **Extensibility**: Extensions to the machine, many of which are described by Kogge [18], 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 and supports all the list-processing primitives that it's inventors described the operational semantics with and aided the development complexity. 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 [18]) 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.

5.1 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 to aid this process. This poses a question of what it means for an abstract machine to be staged.

From the architecture of a SECD machine the intended place for free variables 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. By definition, a staged evaluator should have a means of generating some form of intermediate representation, for example residual code, and evaluate in multiple stages. In our case we split the evaluation of the SECD machine into reduction of static values and residualization for dynamic values.

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 [6] 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. Following are our division rules:

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.

We refer to our division as coarse grained since dynamic values pollute whole registers that could serve as either completely static or mixed valued. An example would be a machine that simply performs arithmetic on two integers and returns the result. The state machine transitions would occur

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 5.1.2
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
FNS (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

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 4 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.

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.

Now that we clarified how static and dynamic values are classified in our SECD machine we describe its implementation as a definitional interpreter.

5.1.1 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 _ ops (((((machine initStack) initDump) ops))))
```

Figure 5: 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 5. 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 5. 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 5 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 [6] 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 3. 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 6 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 6 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 6: Annotated version of the SECD interpreter in figure 5 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 5.1.1

.

This recipe is not enough, however, because of the conflicting nature of the small-step semantics of the SECD machine with NbE-style partial 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. [19], 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 spefic state transitions are described in APPENDIX but the instruction essentially works in two steps. First the user creates

two closures on the stack. One which holds the recursive function and another which contains 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 is the recursive closure. According to Kogge's [18] 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 [20] do not impose. Given the choice between adding support for an underlying `set-car!` instruction in $\lambda_{\uparrow\downarrow}$or extend 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 as well.

5.1.2 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 7 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) RAP)) ;Set up initial recursive call. Set counter to 10
```

Figure 7: 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 and the 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 8 highlights this in the internals of the VM.

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

Figure 8: Snippet from the internals of the SECD interpreter from section 5.1.1. 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 9 shows the modifications to the operational semantics of Landin's SECD machine [13] which allow it to be partially evaluatable with a TDPE and do not require a "set-car!" in the underlying language.

Firstly, equation 1 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 defined by **LDF** in control-register, 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. [21] 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 **fns-register** or simply **fns** which is responsible for holding the recursive instructions of a **RAP** call.

In equation 2 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* and

s e (**LDF** ops.c) d fns ⟶ ((λe'.(machine '() e' ops 'ret fns)) ops.e).s e c d fns (1)

(entryClo recClo.s) e (**RAP**.c) d fns ⟶ '() e entryOps (s e c fns.d) (rec mem entryEnv '()).fns (2)
where entryClo := (entryFn (entryOps entryEnv))
recClo := (recFn (recOps recEnv))
rec := λenv.(machine '() env recOps 'ret (rec mem recEnv.'()).fns)
mem := ((s e c fns.d) (recOps recEnv).fns)

s e (**LDR** (i j).c) d fns ⟶ ((λ.(locate i j fns)).s) e c 'fromldr.d fns (3)

v.s e (**RTN**.c) (s' e' c' d' fns'.d) fns ⟶ (lift-all v) if d-register is tagged with 'ret (4)
(λx.((car s') (cons (car x) (cddr s'))) (cddr s1)
if d-register is tagged with 'fromldr
where s1 := v@_
(v.s') e'c'd'fns' otherwise

(fn env.v).s e (**AP**.c) ('() env') fns ⟶ res.s env c d fns (5)
where res := fn@(lift-all (v s.env'))

(v m.s) e (**LIFT**.c) d fns ⟶ res.s e c d fns (6)
where res := (lift v) if (num? v) or (sym? v)
(lift rec) if (lambda? v)
where rec :=
(λenv.(machine '() env.recEnv c'
'ret (rec (recOps recEnv).fns)))
(lift (λenv.(machine '() env.m c' 'ret fns))) otherwise

Figure 9: Modifications to the SECD operational semantics as presented by Kogge [18]. These modifications permit us to stage our SECD machine interpreter and enable residualization of recursive function applications.

recFn, the SECD instructions these functions execute and an environment. **RAP** then saves the current 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, 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 fns 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 7 with the new semantics we load the recursive function using the new **LDR** instead of the **LD** instruction as highlighted in figure 10.

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

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

Figure 10: Recursive countdown example from figure 7 rewritten with the SECD operational semantics in figure 9

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

5.2 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 [18] though with modifications (see figure 11) to support our modified calling conventions and additional registers described in section 5.1.2. 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 an missed look-up in the environment. The **quote** built-in (equation 9) 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 12. As line 3 suggests

Syntax : ⟨identifier⟩ (7)
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⟩) (8)
Code : ⟨expr⟩ `LIFT`

Syntax : (**quote** ⟨expr⟩) (9)
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 11: Modifications to the SECD compiler described by Kogge [18]

we feed the compiled SECD instructions to the SECD machine interpreter source described in section 5.1.1 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.

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

Figure 12: Compilation and execution of a program in SecdLisp on our PE framework.

5.3 Example

Figure 13a shows a program to compute factorial numbers recursively written in our SECD LISP front-end. The program is translated into SECD instructions by our compiler (see section 5.2) and then input to our staged machine. Figure 13c 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.

6 Level 3: Meta-Eval

Armed with a staged SECD machine and a language to target it with we build to next interpreter in the tower that gets compiled into SECD instructions. This evaluator defines a language called Meta-Eval, M_e, whose syntax is described in figure 14. The language resembles Jones et al.'s Mixwell and M languages in their demonstration of the Mix partial evaluator [14] in the sense the toy language is a LISP derivative and $M_e$serves both 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 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 our parsing infrastructure from the $\lambda_{\uparrow\downarrow}$LISP front-end. 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 15. 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 $\lambda_{\uparrow\downarrow}$. We demonstrate examples of running the interpreter on our staged virtual machine in section **??**.

Figure 16 shows the $M_e$interpreter running a program computing factorial using the Y-combinator for recursion (figure 16a) 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 17). As the programmer

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

(a) LISP Front-end

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

(b) SECD Instructions

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

(c) Prettified Generated Code

Figure 13: Example Factorial

⟨program⟩ ::= ⟨expression⟩
⟨expression⟩ ::= ⟨variable⟩
| ⟨literal⟩
| (lambda (⟨variable⟩) ⟨expression⟩)
| (⟨expression⟩ ⟨expression⟩)
| (op_2 ⟨expression⟩ ⟨expression⟩)
| (if ⟨$\text{expression}_{\text{condition}}$⟩ ⟨$\text{expression}_{\text{consequence}}$⟩ ⟨$\text{expression}_{\text{alternative}}$⟩)
| (let (⟨variable⟩) (⟨expression⟩) ⟨$\text{expression}_{\text{body}}$⟩)
| (letrec (⟨variable⟩) (⟨$\text{expression}_{\text{recursive}}$⟩) ⟨$\text{expression}_{\text{body}}$⟩)
| (quote ⟨expression⟩)
⟨variable⟩ ::= ID
⟨literal⟩ ::= NUM | 'ID
op_2 ::= and | or | $-$ | $+$ | $*$ | $<$ | eq?

Figure 14: Syntax of $M_e$which gets compiled into SECD instructions for interpretation by the SECD machine

we know this control flow can be reduced even further since the $M_e$source program is static data.

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

Figure 15: Staged interpreter for M_e

6.1 Staging M_e and Collapsing the Tower

In an effort to further optimize our generated code from the example in figure 16 we stage the $M_e$interpreter. As indicated by Amin et al. during their demonstration of collapsing towers written in Pink [6], 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 [6] by lifting all literals and closures returned by the interpreter and let $\lambda_{\uparrow\downarrow}$'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.

The state transitions for the **LIFT** operation in the staged SECD machine is shown in equation 6.

```
((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 16: 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 17: Generated code running the example in 16 on a staged SECD machine. Traces of the M_e's dispatch logic is highlighted in green.

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 the LISP front-end to $\lambda_{\uparrow\downarrow}$. Thus,

```
LDC 10 LIFT STOP
```

would generate $\lambda_{\uparrow\downarrow}$ code representing the constant 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 **fns** and **dump**. 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 execute wrapped in a lambda and auxiliary state information such as the environment. In this case we simply construct a lambda that 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 16) is shown in figure 19 and the corresponding SECD instructions that $M_e$compiled down to in figure 18. 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 <=== M_e Dispatch Logic
         (NIL LD (1 1) CONS LD (1 2) AP JOIN )
    (LD (1 1) NUM? SEL
        (LD (1 1) LIFT JOIN ) <=== Lift literals
  ...
 (LDC letrec LD (1 1) CAR EQ SEL
     (NIL NIL LDF
         (LD (2 1) CADR CAR LD (1 1) EQ SEL
             (LD (12 1) LIFT JOIN) <=== Lift recursive lambdas
  ...
  (LDC lambda LD (1 1) CAR EQ SEL
         (LDF (NIL LDF
             (LD (3 1) CADR CAR LD (1 1) EQ SEL
                 (LD (2 1) JOIN) (NIL LD (1 1) CONS LD (3 2) AP JOIN) RTN)
                    CONS LD (2 1) CADDR CONS LDR (1 1) AP RTN) LIFT JOIN) <=== Lift lambdas
  ...
```

Figure 18: SECD instructions for example an factorial on a staged $M_e$interpreter

7 Level 4: String Matcher

Following the experiments performed in Amin et al.'s study of towers [6], we extend our tower one level further and implement a regular expression matcher proposed by Kernighan et al. [22] in M_e.

8 Conclusion

Figure 1a shows a representation of our tower in terms of "sideways-stacked" tombstone diagrams. Although here the tower grows upwards and to the left this does not necessarily be. The compilers, or *translators* labelled LABEL and LABEL have been implemented in scala for convenience. To realize a vertical tower structure our Lisp-to-$\lambda_{\uparrow\downarrow}$translator, which converts Lisp source to Scala ASTs, could be ommitted letting the base-evaluator evaluate s-expressions directly. Similarly, the μ-to-SECD translator could be implemented in SECD instructions itself, possibly through generating such instructions using LABEL in a bootstrapping fashion.

Demonstrated a recipe for constructing heterogeneous towers Motivate the experimentation of other abstract machines and potentially target real-world systems Method of staging use exposed LIFT construct applies across evaluators, as opposed to only interpreters, as well if you instrument the compiler appropriately

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

However, it is not feasible in real-world stacks to simply stage or even just instrument interpreters such as JavaScript or Python. What is potentially applicable though is the application of this strategy to domain-specific languages where control of the internals of a language is realistic.

Cost of Emulation (or interpretation) [23] Turing Tax Turing Tax specific slides

The goal of our study was to investigate the extent to which previous work on collapsing towers of interpreters applies to heterogeneous settings and the interpretative overhead that can be eliminated in the process. We demonstrate the ability to turn a sequence of evaluators into a code generator by through a proof of concept heterogeneous tower built on a SECD virtual machine. We show that the trace of any intermediate interpreter can mostly be eliminated from compiled code when staging at the user-most interpreter level. We show a recipe for staging an stack-based abstract machine based on small-step semantics by converting it into an annotated definitional interpreter and using eta-expansion to prevent pollution of binding-times between segments of the machine and prevent non-termination of the type-directed partial evaluator.

Kogge's extensive study of machines for symbolic computation [18] and Diehl et al.'s survey of abstract machines provide several attractive opportunities for future investigations. The widely used Warren Abstract Machine (WAM) in logic programming, PCKS machine [24], MultiLisp

A key strategy in our methodology for collapsing towers of interpreters is the exposure of a *lift* operator that propagates binding-time information to lower levels of the tower. More specifically, once we reach a level with a significant semantic gap, such as the SECD machine in our tower, the reimplementation of *lift* becomes necessary and more intrusive. Thus a consequence of heterogeneity and thus the absence of meta-circularity is the availability of binding time information of an interpreter. Two follow-up questions arise from this: 1. Is there a way to unify the binding-time metadata across levels in a tower then remove the need for reimplementing staging operators 2. Since we essentially *lift* in the same places of an interpreter throughout all levels, can we identify places in an interpreter where BT annotations can be added or inferred automatically. (IS THE DIFFERENCE ONLY SYNTACTIC)

The lower in the tower one stages the more noise the generated code outputs and traces of the structure of the underlying machine are noticeable although reducing further would require changing its semantics which is what staging at upper levels essentially does Main challenge in collapsing in presence of heterogeneity is the propagating of the *lift* operator throughout each level. "lifting of closures particularly needs to be rewritten at every level which would make applying our methodology to real-world

towers infeasible.

Technical difficulties: implementation of letrec/multi-arg lambdas, implementation of mutable cells, decision on how to stage (i.e. where to annotate) but is essential to performance, leaking implementations between layers, base language getting bloated with features, non-termination when performing recursive calls from within SECD machine (no differentiation between a function being recursive or not) A partial evaluator is referred to as *strong* if, when the result of partially evaluating to an interpreter with respect to a target program yields the target proram as output. This is called Jones' optimality directly implies the removal of a layer of interpretation. Our experiments show that depending on which level of a tower we apply our partial evaluator on we can have this strong property of partial evaluators hold. When staging the user-most interpreter we are able to generate ANF code that is equivalent to the subject program the user wrote. If we stage at the level of the SECD machine we generate code that resembles some the structure of the SECD machine and are not able to get completely rid of the stack-based operations of the SECD machine.

9 Future Work

Future work: abstract machine for partial evaluation? multi-level language abstract machine

can we provide a well-annotatedness property as in [25] is there a property such as partial evaluatability?

Another avenue to explore is the efficiency of binding time analysis. In our staged SECD machine we dynamize all of the stack and all of the environment because of pollution of single dynamic variables. Binding-time improvements could be explored using online partial evaluators to decide on the binding time at specialization time. Additionally one could explore polyvariant analysis to reduce inefficient dynamization of actually static values. [21].

Our study lives in the field of deriving compilers from interpreters and presents techniques of realizing such transformations using previous work on TDPE in the context of towers of interpreters. The crucial difference between modern compilers and interpreters, though the line got blurrier in the last years through JIT compilation in interpreted languages like Python and interpreter virtualization such as GraalVM, are the architecture dependent optimizations and complex pipeline of optimization passes present in compilers. Even in our small-scale study of partial evaluators we encountered

questions of optimizing size and structure of generated code and proposed solutions specific to the interpreters we are partially evaluating. We echo the question posed in Jones [26] discussion of challenges in PE of whether we can generate compilers or in our case even simply generate code that rivals modern optimiing compilers. -¿ need more testing, all optimizations we introduced are very specific to the wiring of the tower, which is maybe how it is supposed to be, although compilers do not require optimizations based specifically on the program source they operate on

extending heterogeneity to mean targeting a different (e.g. lower-level language) than pink

stratification of towers how can changing semantics of individual levels in real world towers work soundly?

Recent work due to Sampson et al. [27] differentiates between value splicing and materialization. Materialization and cross-stage references are used to persist information across stages. This provides a possible solution to pass information about staging decisions across levels.

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

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

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

[6] N. Amin and T. Rompf, "Collapsing towers of interpreters," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 52, 2017.

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

[8] 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.

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

[10] A. Shali and W. R. Cook, "Hybrid partial evaluation," *ACM SIGPLAN Notices*, vol. 46, no. 10, pp. 375–390, 2011.

[11] U. Berger, M. Eberl, and H. Schwichtenberg, "Normalization by evaluation," in *Prospects for Hardware Foundations*. Springer, 1998, pp. 117–137.

[12] G. Ofenbeck, T. Rompf, and M. Püschel, "Staging for generic programming in space and time," in *ACM SIGPLAN Notices*, vol. 52, no. 12. ACM, 2017, pp. 15–28.

[13] P. J. Landin, "The mechanical evaluation of expressions," *The computer journal*, vol. 6, no. 4, pp. 308–320, 1964.

[14] 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, no. 1, pp. 9–50, 1989.

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

[16] J. D. Ramsdell, "The tail-recursive secd machine," *Journal of Automated Reasoning*, vol. 23, no. 1, pp. 43–62, 1999.

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

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

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

[20] 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.

[21] O. Danvy, K. Malmkjær, and J. Palsberg, "The essence of eta-expansion in partial evaluation," *Lisp and Symbolic Computation*, vol. 8, no. 3, pp. 209–227, 1995.

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

[23] M. Steil, "Dynamic re-compilation of binary risc code for cisc architectures," *Technische Universitat Munchen*, 2004.

[24] L. Moreau, "The pcks-machine: An abstract machine for sound evaluation of parallel functional programs with first-class continuations," in *European Symposium on Programming*. Springer, 1994, pp. 424–438.

[25] C. K. Gomard and N. D. Jones, "A partial evaluator for the untyped lambda-calculus," *Journal of functional programming*, vol. 1, no. 1, pp. 21–69, 1991.

[26] N. D. Jones, "Challenging problems in partial evaluation and mixed computation," *New generation computing*, vol. 6, no. 2-3, pp. 291–302, 1988.

[27] A. Sampson, K. S. McKinley, and T. Mytkowicz, "Static stages for heterogeneous programming," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 71, 2017.