Collapsing heterogeneous towers of evaluators

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

April 30, 2019

Declaration

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

Total word count: 0

Signed: Michael Buch

Date: 10th June 2019

Abstract

Intuitively towers of interpreters are a program architecture by which sequences of interpreters interpret each other and a user program is evaluated at the end of this chain. While one can imagine such construct in everyday applications, prior research made use of towers of interpreters as a foundation to model reflection. As such, 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. As a result, the assumptions and constraints that govern tower models make them unapplicable to practical or non-functional settings. Prior formalizations of reflective towers have identified partial evaluation and reflection to harmonize in the development of such towers. We lift several restrictions of reflective towers including reflectivity, meta-circularity and homogeneity of data representation and then construct non-reflective towers of interpreters to explore how partial evaluation techniques can be used to effectively remove levels of interpretation within such systems.

Contents

List of Figures

List of Tables

11102 (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 in the tower. On one hand, it has been used in the formalization of reflection in LISP and serves as a model in the development of reflective languages. On the other hand, towers of interpreters are a frequent occurrence in application development. Examples include interpreters for embedded domain-specific languages (DSLs) or string parsers 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 view interpreters running on virtual hardware as towers of interpreters as well.

The inefficiency of reflective tower models of evaluation has been noted since their inception. CAN COMPILE REFLECTIVE LANGUAGES REMOVE LAYERS OF INTERPRETATION COMPILING WITH MODIFIED COMPILED SEMANTICS

In our study we investigate previous work on collapsing towers of interpreters and aim to take another step towards applying these techniques to real-world settings. 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 contribu-

tions 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.

One of the aims of this study is to explore the effectiveness of compilation of towers under various configurations. The expected outcome follows the Amin et al.'s idea of only staging the user-most interpreters for the most efficient code output. effect of staging at different levels how severe is the difference in generated code? is it worth the increased implementation complexity?

[3]: page 26. Only recently has partial evaluation seen growing adoption with frameworks such as Scala's LMS [4] or Oracle's GraalVM [5]. The design space of partial evaluators is multi-faceted and considerations include:

- use of mixed languages (Futamura's mix for compilation requires same PE input language as mix is written in)
- accept different types of interpreters with different semantics (this is what we partly address) (and what GraalVM addresses?)
- performance
- BTA strategy: automatic or manual? offline or online? how to deal with non-termination?
- Heuristics for guiding PE to produce efficient/desirable code (e.g. you cannot say of a compiler that it doesn't produce more valuable program transformation. But PE's accept that some applications do not warrant it. How to detect such situations?)

semantic gap has been discussed previously as a challenge outstanding in the study of partial evaluation.

Chapter 17: specifcies partial evaluation in the context of general program transformation and outlines PE research areas

TODO: staging vs pe partial evaluatability of SECD (and other linear interpreters/small step semantics vs denotational/big step/recursive descent parsers) retaining info in PE of TDPE style (our strategy

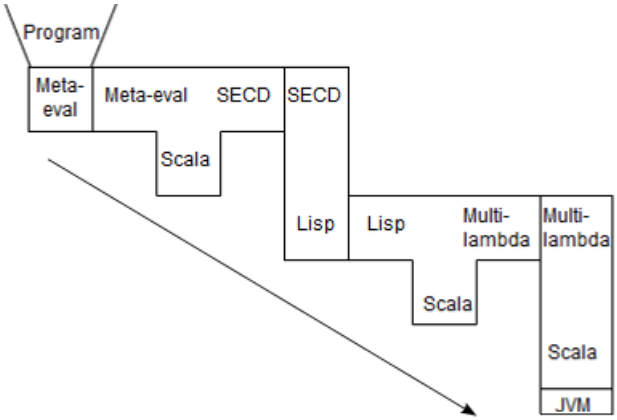

Figure 1: A tombstone diagram representation of our framework

is thunks) contribution: was natural to start with EBASE but uncovered problems BLANK tying the knot in the code instead of data transfer whiteboard diagram are able to stage/lift through compiler/translator structurally similar/equal to javascript tower implementations: 1. regular expression matcher (on metaeval or in place of metaeval) 2. try/catch 3. using cps adding amb (needs set! and store =¿ turn metaeval into cesk machine (possible because SECD supports recursion, lambdas, letrec, etc.)) Our collapsing strategy is based on the fact that we can choose to either evaluate code or generate code which is an inherent property of the multi-level language base $\lambda_{\uparrow\downarrow}$. Thus in a realistic tower base needs to be a multi-level language

Partial evaluators can be thought of as lightweight optimizing compilers targeting specific optimization goals

The aim of our study is to explore the process of turning towers of interpreters into compilers by staging appropriate interpreters and provide a launching point for further research into the optimization of mixed-language systems using partial evaluation.

In section 2 we explain potentially useful background information that cover the fundamental topics we base our framework and experiments starting from the link between interpreters and compilers up to how previous work compiled reflective towers. Section 4 provides an overview of the partial evaluation framework we base our study on called Pink [1]. We systematically describe the process by which we create a heteregeneous tower of interpreters and incrementally collapse in sections 5 through 6.3. Each of these sections discusses the steps we took to create a level in the tower of interpreters as shown in figure 1. We conclude with an evaluation of experimental results followed by

a discussion of potential future work in sections 7 and 8 respectively.

2 Background

2.1 What are Interpreters?

Interpreters vs. compilers Lines are getting blurry: JIT compilation or more recently Graal But the Futamura showed they are fundamentally related in an elegant way by three projections that follow from the SMN theorem in recursive function theory What follows is the study of partial evaluation which lies at the heart of our methodology of collapsing towers of interpreters and is described in more detail in SECTION

2.2 Type-Directed Partial Evluation (TDPE)

Partial evaluation (PE) is a program optimization technique based on the principle of *specialization*. The idea is to produce a program that is specialized to a particular input. 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.

safe vs. congruent division

There are two types of partial evaluation methodologies [3]:

- Offline partial evaluation

- Online partial evaluation [6]

NbE resources:

Alternative techniques include syntax directed partial evaluation and off-line partial evaluation Futamura's second project showed the direct relationship between an interpreter and a compiler -¿ hard to realize Jone's et al. developed MIX which was the first self-applicable PE that operated on a language developed by the same authors called MIXWELL. MIX innovated by making binding-time decisions offline as opposed to during partial evaluation time using source annotations for the PE However as noted in [7], traditionally partial evaluation worked on untyped languages where a single universal datatype could represent all static values PE of typed languages was one of the challenges with PE outlined by Jones [8].

Differences to traditional syntax-directed offline partial evaluators (like MIX): * binding time is automatic (or requires annotations only on inputs to program e.g. like in Pink, which requires knowledege of how the binding-time works. TDPE only requires annotations on base types from which product, sum and function types' binding-times can be deduced. Pink does not necessarily have a binding time analysis step which makes it slightly more convenient for manual annotation) * type-directed means we use the type of a term to guide the normalization (i.e. "extraction function" is parameterized by type) * TDPE reuses underlying evaluator to perform static evaluation vs code generation. Traditionally these are performed by the partial evaluator separately in the form of symbolic computation on the source. Pink takes the former approach

In [9] Danvy builts a language in ML that supports and reflection (section 1) and then shows how partial evaluation of can be achieved using a "normalization function":

page 379 explains role of let-insertion Our framework uses the Eijiro Sumii approach

page 388: actual description of TDPE "We define type-directed partial evaluation as normalization by evaluation over ML values"

page 403: benefits of NbE

[3] page 103: monovariance, polyvariance, congruence in PE

2.3 Staging

Namin et al. [1], propose two languages Pink and Purple. Pink uses a form of online partial evaluation (driven by whether values are ”code?” or not) but requires manual staging facilities. Purple relies on LMS for automatic binding time analysis and staging which limits it to offline partial evaluation and thus relies on further optimization heuristics to achieve the same level of program specialization in the generated code as Pink. at its heart it uses the fact that β-reduction to β-normal form can be viewed as a form of specialization (taken further by NbE [10, 11]) “NbE” extracts the normal form from its meaning (i.e. semantics -¿ syntax or in our case Val -¿ Exp which is done via the “lift” operator in $\lambda_{\uparrow\downarrow}$)

Staging and partial-evaluation are closely related but are often misused interchangeably. Staging is used to split an evaluation into multiple phases, or stages. For instance, a traditional compiler model consists of a compilation pipeline where a program is transformed and passed on to another stage. This is one of the reasons we can call a staged interpreter a compiler, since instead of evaluating source in one go the interpretation is split into multiple stages of program transformation.

A widespread technique for implementing partial evaluators is by staging an interpreter but this need not be the case. COUNTER-EXAMPLE

2.4 Abstract Machines

2.5 Definitional Interpreter

$\lambda_{\uparrow\downarrow}$expects this style of interpreter see [12]

A language is defined by how the interpreter implements it. $\lambda_{\uparrow\downarrow}$uses this idea to produce a staged version of a language through a definitional interpreter of that language with BTA annotations

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

The partial evaluation framework due to Amin et al.’s [1] features a core multi-level language, $\lambda_{\uparrow\downarrow}$, whose evaluator also serves as a partial evaluator. 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. $\lambda_{\uparrow\downarrow}$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 is said to be generating code where code is $\lambda_{\uparrow\downarrow}$'s syntax constructors. The fact that code generation of expressions can be guided using this single operator whose semantics closely resemble a 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.

KEY FEATURES WHY IT SERVES WELL FOR TOWER CONSTRUCTION (ARE THERE ALTERNATIVES)

Stage polymorphism [13]: "abstract over staging decisions" i.e. single program generator can produce code that is specialized in many different ways

Multi-level base evaluator written in $\lambda_{\uparrow\downarrow}$: supports staging operators (**polymorphic Lift**)

Modify other interpreters: make them **stage polymorphic**, i.e. commands either evaluate code (like an interpreter) or generate code (like a translator)

HOW DOES $\lambda_{\uparrow\downarrow}$WORK INTERNALLY =¿ HOW DOES $\lambda_{\uparrow\downarrow}$USE TDPE

Since our work is based on the multi-level language, $\lambda_{\uparrow\downarrow}$, developed by Amin et al. [1] 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. It also features a NbE style lift operator that forms tool by which partial evaluation is achieved. The LISP front-end is described in figure FIGURE. It is a LISP derivative that has support for non-mutable list and cons-cell operations, the lift operator for staging and recursive self-referencing lambdas.

The original Pink implementation assumes we are allowed to make arbitrary changes to evaluators. It effectively adds tags to the emitted representation of a layer above and lets the layer below infer from these tags what tag it itself should pass to the next layer, eventually calling the base-level Lift term.

How are stages encoded? dynamic vs. static (e.g. see MetaOcaml or TDPE's ML extensions)

2.7 Parallels To Towers In the Wild

To provide a real-world analogy of the language towers we are constructing describe some existing arrangements of multi-interpreter systems below:

- Here is a list of languages that are built on top of JavaScript. This is a three-level interpreter system: User-application¡-¿DSL¡-¿JavaScript Interpreter
- Here is a list of languages that compile to Python.
- v86 is a x86 CPU emulator written in JavaScript. This closely resembles our stack machine that is evaluated in both Pink or the Base language's multi-stage evaluator
- 6502asm is a microcontroller emulator in JavaScript

secd -¿ emulator (can PE of machine code help? [14]) meta-eval -¿ python lisp -¿ JavaScript base -¿ VM? GraalVM (could it be adapted)?

2.7.1 Comparison to Other Partial Evaluators

In the Mix partial evaluator [15] interpretative overhead is removed in a similar fashion from a sample interpreter when partially evaluated to terms in the Mixwell language of the same paper. However, the method by which they achieve PE differs ...

2.8 Reflective Towers

The earliest mentions of towers of interpreters appeared in the study of reflection in LISP and its derivatives.

In his proposal for a language extension to Lisp called 3-LISP [16], Smith introduces the notion of a reflective system, a system that is able to reason about itself. 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. Smith argued that there is a distinction between reflection and the ability to reference programs as just ordinary values. Reflection requires a way with which an embedded language can access the structures and state of the process that the embedding lives in. Crucial is the idea of implicit as opposed to explicit

information that an evaluator exposes. Smith's idea of reflection is the capability to explicitly instantiate a language construct that was implicit prior. While environment and continuations, which form the state of a tradidional Lisp process, are implicitly passed around, a 3-LISP program can access both these structures explicitly at any point in time. 3-LISP achieves this by way of a, conceptually infinite, *reflective tower*. Smith divides a process into two parts: a *structural field* that consists of a program with accompanying data structures and an evaluator that acts on the structural field. Replacing the evaluator with a meta-circular one then provides a way to construct an infinite reflective tower. In Smith's original model, meta-circular interpreters each with its own environment and structural field, included a builtin *reflective procedure*, which when called provided access to the state, i.e. environment, of it's interpreter. A meta-interpreter, also referred to as "the ultimate machine" is the upper-most interpreter in a tower and is itself not evaluated but simply a necessesity for the tower to exist in the first place. Questions of performance, potential uses and a complete model separate from implementation details was provided in subsequent work.

Reflection operators take values for expression, environment and continuation and re-install them into the interpreter state. Reification operators provide access to the interpreter state and pass it to the program as values. A packaged up state of ϵ, ρ, and κ that can be treated as regular values is said to be *reified*. In the context of multiple-levels of interpretation in a tower, calling a reflection operator spawns a new level in the tower with the interpreter state being the one at the time of application. Once evaluation was performed in the new level control is passed back to the interpreter that spawned it. Reification operators package up the state of the interpreter at the level of application and pass it to the expression to be evaluated.

A subsequent study due to Danvy et al. [17] provides a systematic approach to constructing reflective towers. The authors provide a denotational semantic account of their reflection model similar to the technique described above and realize these formalizations into a language built with a reflective tower called "Blond". The authors start with a non-reflective tower and non-meta-circular tower. An assumption that the authors carry throughout their paper is that of single-threadedness. This is both to reduce the complexity of designing an implementation and prevents racey side-effects between concurrent towers. The restriction is that the effects of each level in a tower is the interpretation of the level below it. Any non-interpretative work is performed at the last level of the tower, also referred to as its *edge*. Danvy et al.'s key insight was the need for an intensional description of an interpretative tower that relates the interpreter state at different levels of a tower to the reflection and reification operations.

A $DenotableValue_n$ is any valid language construct and its representation as defined by the interpreter at level n. A consequence of this formulation is the fact that domains between levels are distinct but connected via a valuation function and formalizes the earlier notion of an interpreter at level n spawning a new evaluator at $n-1$ through some reflective operation. An even more relevant fact is that according to this denotational model, we are free to choose the representation of denotable values in each level. The authors assume for the rest of their study that levels are identical, however, in our work we assume the exact opposite. None of our levels are identical but can be formulated in the same framework given above. An example would be the denotation of an expression $Exp_0 = (1+2)$ at level 0 in our hypothetical tower of n levels. At level $n = 1$ this can be represented as $Exp_1 = (+\ 1\ 2)_1$ or at level $n = 14$ as $Exp_{14} = (01\ +\ 10)_{14}$, i.e. in binary. In our model we not only keep the notion of non-identical levels and non-metacirularity, but also the concept of a store, which the authors purposefully ommitted to keep the description purely functional.

2.9 Collapsing Towers

Taking a traditional model of interpretation, a conceptually infinite tower of interpreters adds 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. BLOND/STURDY both hint at partial evaluation potentially being a tool capable of removing some of this overhead by specializing individual levels to the interpreters below.

2.9.1 Compiling Reflective Languages

[18]: Language “Black”; has early uses of the act of collapsing modes of interpretation in a reflective setting. Its reflective model is closer to 3-LISP than to Blond or Brown [19]

The Truffle framework due to Whürthinger et al. [20] demonstrate a practical partial evluation framework for interpreters independent of language by providing a language and interpreter specifically designed to partially evaluate and thus collect as much information about a dynamic language at run time as possible.

2.9.2 Examples

Examples drawn from paper on collapsing towers [1]:

- Regular expression matcher ¡- Evaluator ¡- Virtual Machine
  - Generate low-level VM code for a matcher specialized to one regex (through arbitrary number of intermediate interpreters)
- Modified evaluator ¡- Evaluator ¡- Virtual Machine
  - Modified for tracing/counting calls/be in CPS
  - Under modified semantics ”interpreters become program transformers”. E.g. CPS interpreter becomes CPS transformer

Recent work following on from Asai's work has demonstrated the ability to compile an potentially infinite tower of interpreters with dynamically changing semantics of individual levels using novel applications of normalization-by-evaluation [1]. Our work is motivated by following, rephrased, of question posed in the conclusion of their work: is it possible to extend the framework to practical towers and how?

2.10 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 [21], Jones et al.'s Mix partial evaluator [15] 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 2.1. 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 2.2. 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.

2.10.1 Absence of: Meta-circularity

The first constraint imposed by definition 2.10 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 [16], Blond [17], Black [18] or Pink [1].

2.10.2 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.

2.10.3 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 [17]. 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 [21] 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 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.

The mix parial evluation framework [15], Jones et al. demonstrate the PE of a simple interpreter into a language called Mixwell developed by the authors. This is similar in spirit to our framework except it is smaller in height. (section 5 of the paper [15]). also mentions removal of layers of metainterpretation in its conclusion

Recent work due to Sampson et al. [22] 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.

Partial Evaluation of Machine Code

Put together, the three properties imposed by definition 2.10 encourage a generalized solution irregardless of the language or structure of the tower at hand.

One of the earliest serious mentions of collapsing levels of interpretation using partial evaluation was [21]

3 Problems

To put our work and motivation into context consider following program architecture (originally described in [1]): some user script is executed by a Python interpreter running on a JavaScript emulator of a x86 CPU, all of which is run within a browser that eventually is executed on hardware. This construction resembles a practical realization of our defintion of heterogeneity in towers of interpreters, or in this case a tower of languages. Although such scenario might seem far-fetched, other forms of towers such as domain-specific languages embedded within a host are a form of tower of interpreters. Our study presents a systematic construction of a tower that resembles the structure of the tower we describe above: each level's language is different from the one it interprets and strict interpretation, where implementation is infeasible, is replaced with compilation. In addition to the issues outlined in previous work specifically on towers of interpreters, we also touch on and evaluate our results against the collection of open challenges described by Jones [8] some of which have been tackled since but some of which remain open questions, such as the extent to which partial evaluators can perform the work by optimizing compilers and code generation by partial evaluators whose target language is different that its source.

What we envision (with reference to this hypothetical setting) is handling the two following cases:

1. A one-off run of a python script on top of this stack should be collapsed by bypassing the emulator interpretation

2. A continuously running emulator evaluating a continuously running python interpreter should collapse individual runs of interpretation while respecting the dynamically changing environment

    - Here a dynamically changing environment also implies effects that are capable of changing the semantics of interpreters within the tower at runtime

    - In literature, the closest to compiling a dynamically changing tower is [23, 1] (for a *reflective* language Black) and GraalVM [5]

To tackle the first of these problems we construct a similar yet condensed form of the setting as shown in 1

3.1 Why do we want to collapse towers?

The main reason is performance. The key realization of partial evaluation that lead to its development is that interpreters do redundant work but we can make it so they don't. Program specialization is simple and attractive on paper but poses significant engineering challenges and has not seen widespread adoption (until recent increasingly successful work on interpreter virtualization [5]).

Binding time analysis is one of the obstacles of program specialization. The program specializer needs to decide, either automatically or with assistance from the programmer, which data to treat as static and which as dynamic. Simple divisions can lead to code explosion or inefficient code generation, or worse, to non-termination of the specializer. This problem is known as *division* and is one of the key differences between offline and online PE techniques [3].

An interesting consequence of collapsing towers of interpreters demonstrated in [1] paper is the ability to derive translators in the process of collapsing.

A trivial but useful example is logging. Given the tower in figure FIGURE we want to keep the added *useful* behaviour of I_3 while removing the *unuseful* other work of interpreting an intermediate representation. The interpretation of the IR of the level above is a mere accidental consequence of design instead of a necessity. We claim this work to be *interpretative overhead* and defer its quantification by benchmarks to a later section.

Collapsing the tower achieves exactly what we wanted, base-language (here the compilation target language) expressions including logging specialized to the user-level program.

A restriction with this method is its reliance on meta-circularity and reflection and other unsafe techniques:

- we are able "inject" the logging evaluator into tower because of its meta-circularity
- we expose staging operators throughout the tower through simple string manipulation
- instrumentation relies on meta-circularity since we simply redefine how constructs are evaluated before injecting the evaluator
- modification of semantics of the tower are done via reflection (ELABORATE)

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

Stage only user-most interpreter (we investigate configurations in this thesis): *wire tower* such that the **staging commands in L_n are interpreted directly in terms of staging commands in L_0** i.e. staging commands pass through all other layers handing down commands to layers below without performing any staging commands

Non-reflective method: meta-circular evaluator **Pink** =¿ collapse arbitrary levels of "self-interpretation"

By abstracting over staging decisions one can write the same program to both perform staging or evaluate directly [1] (maybe-lift)

$\lambda_{\uparrow\downarrow}$features:

- *run residual code*
- binding-time/stage polymorphism [24]
- preserves execution order of future-stage expressions
- does not require type system or static analysis
  - TDPE [9] (great explanation also at [7]): **polymorphic Lift** operator turns static values into dynamic (future-stage) expressions

In partial evaluation terms, the configurations, i.e. set of run-time computational states, is stored in stBlock while the division, i.e. denotation of static vs dynamic values

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

Since SECD is a stack-based virtual machine all operations and data storage is performed on one of the four stack registers. By the principle of *division* in partial evaluators CITE, we want to be able to separate static and dynamic values to prevent undesired behaviour in our partial evaluator (EXAMPLE OF NON-TERMINATION). A minor extension we add to the original framework is the ability to lift nested tuples as opposed to only two-element tuples. This addition to the small-step semantics is shown in FIGURE.

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.

SHOW OPERATIONAL SEMANTICS

5 Level 2: SECD

The SECD machine due to Landin [25] 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 [25]. Since then it has thoroughly been studied and documented [26, 27, 28] 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 [2], 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 eval-

uator 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 [2]) 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. We make use of the notion of *stage-polymorphism* introduced by Offenbeck et al. [13] 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 2.

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

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

Figure 2: "Stage-polymorphism" [1] in our definitional SECD interpreter

execution in our framework makes use of the partial evaluator used in Pink [1] 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:

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

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 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.

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

```
(let machine (lambda _ stack (lambda _ dump (lambda _ control (lambda _ environment
    (if (eq? 'LDC (car control))
        (((((machine (cons (cadr control) stack)) dump) (cdr control)) environment)
    (if (eq? 'DUM (car control))
        (((((machine stack) dump) (cdr control)) (cons '() environment))
    (if (eq? 'WRITEC (car control))
        (car s)
    ...
    ...)))))))
    (let initStack '()
    (let initDump '()
        (lambda _ ops (((((machine initStack) initDump) ops))))
```

Figure 3: 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.

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.

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 3. 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 3. 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 3 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 [1] 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 2. 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 4 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 4 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))
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 4: Annotated version of the SECD interpreter in figure 3 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. [29], 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 [2] 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 [30] 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 5 which shows a recursive function that decrements a user provided number down to zero.

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

```
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
```

Figure 5: An example recursive function application annotated to show the issue with partially evaluating this type of construct.

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 6 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)) envir
6     (((((machine (cdr stack)) (cons (cdddr control) dump)) fns) (caddr control)) envir
```

Figure 6: 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 7 shows the modifications to the operational semantics of Landin's SECD machine [25] 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

$$\text{s e (}\mathbf{LDF}\text{ ops.c) d fns} \longrightarrow ((\lambda\text{e'.(machine '() e' ops 'ret fns)) ops.e).s e c d fns} \quad (1)$$

$$\begin{aligned}\text{(entryClo recClo.s) e (}\mathbf{RAP}\text{.c) d fns} \longrightarrow\ & \text{'() e entryOps (s e c fns.d) (rec mem entryEnv '()).fns} \quad (2)\\ \text{where entryClo} :=\ & \text{(entryFn (entryOps entryEnv))}\\ \text{recClo} :=\ & \text{(recFn (recOps recEnv))}\\ \text{rec} :=\ & \lambda\text{env.(machine '() env recOps 'ret (rec mem recEnv.'()).fns)}\\ \text{mem} :=\ & \text{((s e c fns.d) (recOps recEnv).fns)}\end{aligned}$$

$$\text{s e (}\mathbf{LDR}\text{ (i j).c) d fns} \longrightarrow ((\lambda\text{.(locate i j fns)).s) e c 'fromldr.d fns} \quad (3)$$

$$\begin{aligned}\text{v.s e (}\mathbf{RTN}\text{.c) (s' e' c' d' fns'.d) fns} \longrightarrow\ & \text{(lift-all v)} \quad \text{if d-register is tagged with 'ret} \quad (4)\\ & (\lambda\text{x.((car s') (cons (car x) (cddr s'))) (cddr s1)}\\ & \qquad \text{if d-register is tagged with 'fromldr}\\ & \qquad \text{where s1 := v@\_}\\ & \text{(v.s') e'c'd'fns'} \qquad \text{otherwise}\end{aligned}$$

$$\begin{aligned}\text{(fn env.v).s e (}\mathbf{AP}\text{.c) ('() env') fns} \longrightarrow\ & \text{res.s env c d fns} \quad (5)\\ \text{where res} :=\ & \text{fn@(lift-all (v s.env'))}\end{aligned}$$

$$\begin{aligned}\text{(v m.s) e (}\mathbf{LIFT}\text{.c) d fns} \longrightarrow\ & \text{res.s e c d fns} \quad (6)\\ \text{where res} :=\ & \text{(lift v)} \quad \text{if (num? v) or (sym? v)}\\ & \text{(lift rec)} \quad \text{if (lambda? v)}\\ & \quad \text{where rec :=}\\ & \qquad (\lambda\text{env.(machine '() env.recEnv c'}\\ & \qquad\quad \text{'ret (rec (recOps recEnv).fns)))}\\ & \text{(lift (}\lambda\text{env.(machine '() env.m c' 'ret fns)))} \quad \text{otherwise}\end{aligned}$$

Figure 7: Modifications to the SECD operational semantics as presented by Kogge [2]. These modifications permit us to stage our SECD machine interpreter and enable residualization of recursive function applications.

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. [31] 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 *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 5 with the new semantics we load the recursive function using the new **LDR** instead of the **LD** instruction as highlighted in figure 8.

```
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 8: Recursive countdown example from figure 5 rewritten with the SECD operational semantics in figure 7

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

5.2 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 [2] though with modifications (see figure 9) 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.

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` ⟨id_0⟩ `LDC` ⟨id_1⟩ `CONS ...` `LDC` ⟨id_{n-1}⟩ `LDC` ⟨id_n⟩ `CONS`
where ⟨id_n⟩ is the nth identifier in the string representing ⟨expr⟩

Figure 9: Modifications to the SECD compiler described by Kogge [2]

Given a source program in SecdLisp we invoke the compiler as shown in figure 10. As line 3 suggests 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 10: Compilation and execution of a program in SecdLisp on our PE framework.

5.3 Example

Figure 11a 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 11c 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 12. The language resembles Jones et al.'s Mixwell and M languages in their demonstration of the Mix partial evaluator [15] 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

```
(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 11: 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 12: Syntax of $M_e$which gets compiled into SECD instructions for interpretation by the SECD machine

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 13. 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 6.3.

Figure 14 shows the $M_e$interpreter running a program computing factorial using the Y-combinator for recursion (figure 14a) 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 15). As the programmer we know this control flow can be reduced even further since the $M_e$source program is static data.

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

Figure 13: 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 14 we stage the $M_e$interpreter. As indicated by Amin et al. during their demonstration of collapsing towers written in Pink [1], 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 [1] 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 14: 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 15: Generated code running the example in 14 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 14) is shown in figure 17 and the corresponding SECD instructions that $M_e$compiled down to in figure 16. 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 l
  ...
```

Figure 16: SECD instructions for example an factorial on a staged $M_e$interpreter

6.2 String Matcher

To evaluate our extended tower to the one used in the original Pink implementation we wrote a string matcher KNUTH to be executed on our staged SECD machine. Running the SECD in evalms's compilation mode we see SECD instructions as literals being included in the generated code. What's worse is that these literals are never used. Their presence is an example of a trade-off we make between implementation complexity and intrusiveness versus removal of interpretative overhead. Although we removed most of the evaluation overhead of the SECD machine itself, these instructions are present in the generated code because our implementation converts anything in the *env* and *stack* registers into dynamic values in the whenever we use lambdas as firt-class citizens for instance when using them as return values. If we wanted to reduce generated code further we could either reuse the *fns* register for non-recursive functions as well and prevent instructions loaded as part of the *LDF* instruction to be lifted.

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

6.3 Experiments

Figure 1 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.

- Able to achieve compilation of stack-machine on top the Pink evaluator (including tracing evaluator etc.)
- Compilation i.e. collapsing through explicit staging annotations requires intricate knowledge of infrastructure and does not support all data structures e.g. stacks

6.4 Benchmarks

We extended the benchmarks provided as part of the original framework [1] with timings for staging the stack machine with respect to a user factorial program and timings for evaluating said program.

Generalization: because we sacrifice the fact input is static and mark them as dynamic (code) values PE technique is more like a translation then evaluation. The result of evluation is a new IR in terms of the base language

Varying degrees of generalization:

1. Interpreter: VM, Static input: Instructions, Dynamic inpute: Generalized to be the numbers or specially tagged value =¿ here we benchmark interpretative overhead of SECD machine for various generalization points
    - Treat all input as static =¿ equivalent to full evaluation
    - Treat all input as dynamic =¿ Generate a recursive loop in base-language terms but doesn't require case checking against non-existent instructions
    - Treat small part of input as dynamic

- Treat part of input as dynamic =¿ Evaluates most of program

2. Interpreter: Pink, Static input: VM, further input: instructions =¿ need to decide where to stage
3. Interpreter: VM, Static input: Evaluator, further input: User program =¿ need to decide where to stage

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)

```
(lambda f0 x1
  (let x2 (eq? x1 0)
  (if x2 1
        (let x3 (- x1 1)
        (let x4 (f0 x3) (* x1 x4))))))
```

Figure 18: Generated code without the SECD machine

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.

7 Conclusion

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

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) [32] 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 [2] 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 [33], 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.

8 Future Work

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

can we provide a well-annotatedness property as in [34] 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. [31].

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 [8] 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?

```
(let x0
        (lambda f0 x1
          (let x2 (cons 1 '.)
          (let x3 (cons 10 x2)
          (let x4 (cons x3 x1)
          (let x5
                (lambda f5 x6
                  (let x7 (car x6)
                  (let x8 (car x7)
                  (let x9 (eq? x8 0)
                  (if x9
                        (let x10 (car x6)
                        (let x11 (cdr x10) (car x11)))
                        (let x10 (car x6)
                        (let x11 (cdr x10)
                        (let x12 (car x11)
                        (let x13 (car x10)
                        (let x14 (* x13 x12)
                        (let x15 (cdr x6)
                        (let x16 (cdr x15)
                        (let x17 (car x16)
                        (let x18 (cdr x17)
                        (let x19 (car x18)
                        (let x20 (- x13 x19)
                        (let x21 (cons x14 '.)
                        (let x22 (cons x20 x21)
                        (let x23 (cons '. x4)
                        (let x24 (cons x22 x23) (f5 x24))))))))))))))))))))))
          (let x6 (car x4)
          (let x7 (cdr x6)
          (let x8 (car x7)
          (let x9 (car x6)
          (let x10 (cons x8 '.)
          (let x11 (cons x9 x10)
          (let x12 (cons '. x4)
          (let x13 (cons x11 x12) (x5 x13)))))))))))) (x0 '.))
```

Figure 19: Generated code with the SECD machine and $\lambda_{\uparrow\downarrow}$smart-constructors 4

Bibliography

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

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

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

[4] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," in *Acm Sigplan Notices*, vol. 46, no. 2. ACM, 2010, pp. 127–136.

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

[6] W. R. Cook and R. Lämmel, "Tutorial on online partial evaluation," *arXiv preprint arXiv:1109.0781*, 2011.

[7] B. Grobauer and Z. Yang, "The second futamura projection for type-directed partial evaluation," *Higher-Order and Symbolic Computation*, vol. 14, no. 2-3, pp. 173–219, 2001.

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

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

[10] U. Berger and H. Schwichtenberg, "An inverse of the evaluation functional for typed lambda-calculus," in *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. IEEE, 1991, pp. 203–211.

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

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

[13] 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.

[14] V. Srinivasan and T. Reps, "Partial evaluation of machine code," in *ACM SIGPLAN Notices*, vol. 50, no. 10. ACM, 2015, pp. 860–879.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer, "Practical partial evaluation for high-performance dynamic language runtimes," in *ACM SIGPLAN Notices*, vol. 52, no. 6. ACM, 2017, pp. 662–676.

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

[22] 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.

[23] K. Asai, H. Masuhara, and A. Yonezawa, *Partial evaluation of call-by-value λ-calculus with side-effects*. ACM, 1997, vol. 32, no. 12.

[24] F. Henglein and C. Mossin, "Polymorphic binding-time analysis," in *European Symposium on Programming*. Springer, 1994, pp. 287–301.

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

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

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

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

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

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.