Ongoing Thesis Writeup

Michael Buch

January 21, 2019

Abstract

Intuitively towers of interpreters are a program architecture by which sequences of interpreters interpret each other with a user program beinge evaluated at the end of the 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, assumptions and constraints govern tower models which 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 restriction of reflective towers including reflectivity, meta-circularity, homogeneity of data representation and construct non-reflective towers of interpreters and extend formalisms to such setting and go on to generalize previous techniques on partially evaluating towers of interpreters.

1 About this document

This is a collection of references and summaries to research in the field of meta-programming/supercompilation/partial evaluation

2 An Introduction to Towers of Interpreters

2.1 LISP-3, Metacircularity and Reflection

In his proposal for a language extension to Lisp called LISP-3 [1], Smith introduces the notion of a reflective system, a system that is able to reason about itself. He argues that reflection is not a property that metacircular interpreters languages provide, but additionally requires a way with which an embedded language can access structures of the system it was described in in its own terms. LISP-3 achieves this by way of a, conceptually infinite, reflective tower. Crucial is the idea of implicit as opposed to explicit information about a system during computation. Smith's idea of reflection is the ability to explicitly instantiate a language construct that was implicit prior. While environment and continuations, which form the state of the Lisp process, are implicitly passed around, a LISP-3 program can access both these structures explicitly at any point in time. 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 evaluator (MCP in [1]) then provides a way to construct infinite reflective tower. Key to such a reflective tower are "reflective procedures" which when called from an evaluator are run at a level above in the tower.

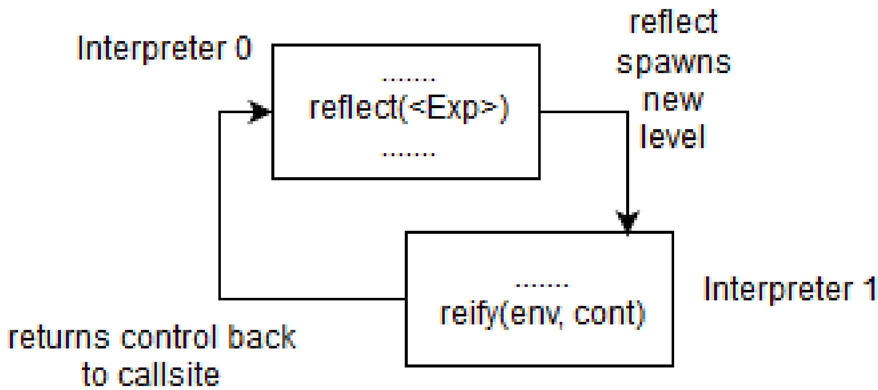

Figure 1: Demonstration of reflection and reification in a two-level tower.

2.2 A Formal Account

Friedman et al. took Smith's account of reflection and decomposed it into distinct operations: reification and reflection [2, 3]. The authors provide one of the early denotational descriptions of reflective towers assuming a model where an interpreter maps an expression (Exp), environment (E) and continuation (K) to a domain of answers (A) and the input values to the interpreter bind to ϵ, ρ, κ respecitvely:

$$\mathcal{E} : Exp \to Env \to K \to A = \lambda \epsilon \rho \kappa.... \tag{1}$$

The need for a formalization of reflection was that earlier work described reflection in terms of reflective towers which themselves were explained in terms of a tower model. To part from this circular reasoning Friedman et al. define levels within a tower and their mutual interaction through operators on the state of the valuation function above. Reflection takes values for expression, environment and continuation and re-installs 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 is said to be "reified". In the context of multiple-levels of interpretation in a tower, calling a reflection operator, `meaning( <Exp> )` 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. Diagramatically this process is shown in figure 1.

A subsequent study due to Danvy et al. [4] provide a systematic approach to constructing reflective towers, 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 without the assumption that individual levels are meta-circular. 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 ecah levels 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 in terms of the domains of interpreter state with respect to levels of a tower. Adding supscripts to the valuation function in equation 1 and providing domains for components of the interpreter's state, we get:

$$\rho_n \in Environment_n = Identifier_n \rightarrow DenotableValue_n$$
$$\kappa_n \in Continuation_n = DenotableValue_n \rightarrow Answer_n$$
$$\mathcal{E}_n : Expression_n \times Environment_n \times Continuation_n \rightarrow Answer_n$$

$$\mathcal{E}_n[\![interpreter_n]\!]\rho_n\kappa_n \simeq \mathcal{E}_{n-1} \tag{2}$$

A consequence of this formulation is the fact that domains between levels are distinct but connected via the valuation function in equation 2. 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, non 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 $Exp_1 = (v_1, (1 + 2)_0)$ or at level $n = 10$ as $Exp_{10} = (v_{10}, (01 + 10))$. In our model do we not only add 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 funcitonal.

$$\begin{aligned}
&\mathcal{E}_n[\![((reify\ (e\ r\ k)\ body)\ E*)]\!]\rho_n\kappa_n \\
&\qquad = \mathcal{E}_{n+1}[\![body]\!]([[\![e]\!] \mapsto (map_n\ \hat{exp}_n[\![E*]\!]) \\
&\qquad\qquad [\![r]\!] \mapsto (\hat{env}_n\rho_n) \\
&\qquad\qquad [\![k]\!] \mapsto (\hat{cont}_n\kappa_n)]\rho_{n+1})\kappa_{n+1}
\end{aligned} \tag{3}$$

To gain a a better intuition for the above, we now summarize Danvy et al.'s denotational description of reflection and reification operators in their tower model. Equation 3 describes the effect of a reification operation, here *reify(e r k)* where e, r and k are the variables that are bound to the expression, environment and continuation of the upper level respectively, between a level n and the level above (i.e. its interpreter) $n + 1$.

$$\begin{aligned}
&\mathcal{E}_n[\![(reflect(E\ R\ K))]\!]\rho_n\kappa_n \\
&\qquad = \mathcal{E}_n[\![E]\!]\rho_n(\lambda\, a.\mathcal{E}_n[\![R]\!]\rho_n \\
&\qquad\qquad (\lambda\, b.\mathcal{E}_n[\![K]\!]\rho_n \\
&\qquad\qquad (\lambda\, c.\mathcal{E}_{n-1}(\check{exp}_n\, a)(\check{env}_n\, b)(\check{cont}_n\, c))))
\end{aligned} \tag{4}$$

The reflection operation in relation between two levels, n and the interpreter it interprets $n - 1$, is shown in equation 4. The domains for individual reflection and reification operations (superscripted with ˆ and ˇ respectively), are given equation 5

$$\begin{aligned}
exp_n^{\wedge} &: DenotableValue_n \rightarrow Expression_{n-1} \cup error \\
env_n^{\wedge} &: DenotableValue_n \rightarrow Environment_{n-1} \cup error \\
cont_n^{\wedge} &: DenotableValue_n \rightarrow Continuation_{n-1} \cup error \\
exp\check{}_n &: Expression_n \rightarrow DenotableValue_{n+1} \\
env\check{}_n &: Environment_n \rightarrow DenotableValue_{n+1} \\
cont\check{}_n &: Continuation_n \rightarrow DenotableValue_{n+1}
\end{aligned} \tag{5}$$

An important observation is that in this model, reification and reflection operators are not commutative. As an example reifying a continuation at level n, $cont_n$, followed by reflecting the continuation in level $n+1$ does not yield the same domain when called in reversed order: $\check{cont}_{n+1} \circ \hat{cont}_n \neq \hat{cont}_n \circ \check{cont}_{n+1}$. The expression types given by equations 5 let us explain this trivially by rewriting the compositions as:

$$\begin{multline}(Continuation_n \rightarrow DenotableValue_{n+1}) \rightarrow (DenotableValue_{n+1} \\ \rightarrow Continuation_{n+1-1} \cup error) = Continuation_n \tag{6}\end{multline}$$

$$\begin{multline}(DenotableValue_{n+1} \rightarrow Continuation_n \cup error) \\ \rightarrow (Continuation_n \rightarrow DenotableValue_{n+1}) = DenotableValue_{n+1} \cup error \tag{7}\end{multline}$$

A less formal explanation of this statement is in terms of the possible values reflection versus reification can result in. Reflection spawns a new interpreter that can yield any result, including an error. Then reifying an error would not yield a valid interpreter state. If one reflects a reified expression it by definition corresponds to simple evaluation in the current interpreter. The importance of this is that this restricts us from being able to fully explain a reflective tower model and valuation functions that act on a level below, $\mathcal{E}^{-}1_n$. If we are not able to provide a definition for reflection and reification at interpreters below any level n we will not have a full description of a reflective tower. This gives a denotational account for the metacontinuations that 3-LISP originally introduced. It was to deal with exaclty this discrepancy in the compositionality of reflection and reification operations. Danvy et al. then add meta-continuations into the equations previously described and their purpose is to describe the continuation that accepts the result of the interpreter it interprets.

2.3 Heterogeneity

A central part of our study revolves around the notion of heterogeneous towers. Prior work on towers of interpreters that inspired some these concepts includes Sturdy's work on the Plathypus language framework that provided a mixed-language interpreter built from a reflective tower [5], Jones et al.'s Mix partial evaluator [6] 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 turning multiple interpreters into compilers in heterogeneous settings. Since *heterogeneous* has been mentioned but not explained until now we provide our definition of heterogeneous 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

3. For any language used in the tower $L_m \in \sigma_L$, $\exists L_a \notin \Sigma_L . L_m \blacktriangleleft L_c \wedge L_c \blacktriangleleft L_c$

So far we have only discussed towers that were homogeneous in nature. We define homogeneous as the combination of metacircularity and reflectivity of interpreter embeddings. A consequence of homogeneity are small semantic gaps throughout levels. First part of [4]

[5]: earliest serious mention of collapsing levels of interpretation using partial evaluation: "Neither the program nor the language have any further meaning unless handled by an active processing agent that makes it do something in the structural field [Smith 82] of that agent, that is, the world of things that the agent knows about and manipulates. We call such an agent an evaluator. The evaluator might be the circuitry and microcode of a computer, or perhaps it might be another interpreter. The evaluator, which is a concrete representation of an abstract language, gives meaning to the program, and in turn the program gives meaning to its input data, that is, makes results from them. The language is given its meaning by the evaluator, which is part of the interpreter—an interpreter is a combination of language and evaluator. This gives us: evaluator(program + input) -¿ output where the evaluator provides the meaning, which, in the traditional understanding of computation, it conjures up from nowhere in particular. In this thesis, we examine the way in which each part of a computation involving evaluator, language and program gives meaning to the other parts, and the way in which a meaning can be given to the outermost evaluator so that the evaluator can then give meaning to the rest. To clarify this, consider this sentence. With no-one to read it, it means nothing to anyone."

"A pair of techniques called reification and reflection, not found in most computing systems, begin to build a bridge between languages and the programs which are written in them. They build this bridge by describing the domains of the elements and rules from which the language is built, in terms of the domains of values which can be handled by the program, and allowing the program access to itself and to its interpreter in these terms. Use of such access for inspection is called reification, and for modification is called reflection."

"It is possible to provide a program with some access to its own interpreter without using a tower [Wand and Friedman]—that is, to just one interpreter, which runs conventionally—but this is not a regular structure and does little to model how an interpreter works. The weakness in this results from the lack of a consistent model for interpretation, that is, from its interpreter being an ad-hoc program rather than having the regular structure that is imposed by a tower of levels. In terms of our earlier analogy of the digits of a digital clock (in section 2.3), it explains the hours in terms of the minutes alone, but does not make the conceptual jump of generalization necessary to explain that hours are related to minutes as minutes are to seconds, and so cannot capture the principle behind such a clock.

Thus, flat reflective systems are suitable for implementing reflection, but not for explaining it in all its splendour."

Perhaps the closest study of mixed languages in a reflective tower was performed in in chapter 5 of Sturdy's thesis [5] where he highlighted for the importance of supporting a mixture of languages within a interpretation framework since 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.

Sturdy develops a framework Plathypus for reflective tower interpretation of many languages whereas we construct a non-reflective tower consisting of mixed languages.

The mix parial evluation framework [6], 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 [6]). also mentions removal of layers of metainterpretation in its conclusion

Recent work due to Sampson et al. [7] 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.

2.4 Coming Full Circle: Partial Evaluation and Reflective Towers

2.4.1 Compiling Reflective Languages

[8]: 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 [9]

The Truffle framework due to Whürthinger et al. [10] 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.

3 Examples

Examples drawn from paper on collapsing towers [11]:

- 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

4 Methodologies Background

- Stage polymorphism [12]: "abstract over staging decisions" i.e. single program generator can produce code that is specialized in many different ways (instance of the Fourth Futamura Projection? [13])
- 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)
- Stage only user-most interpreter: *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 [11] (maybe-lift)
- $\lambda\uparrow\downarrow$features:
  - *run residual code*

  - binding-time/stage polymorphism [14]
  - preserves execution order of future-stage expressions
  - does not require type system or static analysis
    * TDPE [15] (great explanation also at [16]): **polymorphic Lift** operator turns static values into dynamic (future-stage) expressions

4.1 Towers of Interpreters Project Overview

4.1.1 Scala

- base.scala: implements definitional interpreter for $\lambda\uparrow\downarrow$

5 Results

- 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

5.1 Benchmarks

We extended the benchmarks provided as part of the original framework [11] with timings for staging the stack machine with respect to a user factorial program and timings for evaluating said program. The compilation output yielded and is essentially a loop unrolling of the (non-recursive SECD) factorial program without traces of the SECD emulator left:

```
Let(
Gt(Lit(3),Lit(1)),
Let(
Equ(Var(3),Lit(0)),
Let(
If(Var(4),
Lit(1),
Let(
Lift(Sym(.)),
Let(
Cons(Lit(3),Sym(.)),
Let(
Fst(Var(6)),
Let(
Minus(Lit(1),Var(7)),
Let(
Times(Var(8),Lit(-1)),
Let(
Fst(Var(6)),
Let(
```

```
Times(Var(9),Var(10)),
Let(
Snd(Var(6)),
Let(
Minus(Lit(1),Var(7)),
Let(
Times(Var(13),Lit(-1)),
Let(
Minus(Lit(1),Var(14)),
Let(
Times(Var(15),Lit(-1)),
Let(
Times(Var(16),Var(11)),
Let(
Minus(Lit(1),Var(14)),
Let(
Times(Var(18),Lit(-1)),
Var(17)))))))))))))))))),
Var(5))))
```

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

5.1.1 Similarities to Mix

In the Mix partial evaluator [6] 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 ...

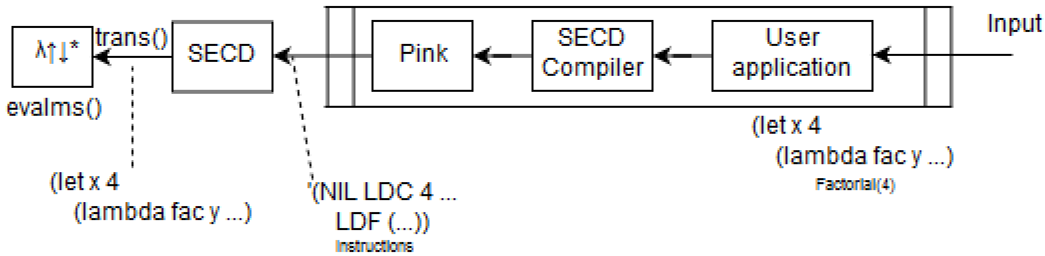

Figure 2: “Effectively functional” $\lambda\uparrow\downarrow^*$with SECD tower above it

6 Problems

A useful analogy is the one presented in [11]: a Python interpreter running on a JavaScript emulator of a x86 CPU. 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 side-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 [17, 11] (for a *reflective* language Black) and GraalVM [18]

To tackle the first of these problems we construct a similar yet condensed form of the setting as shown in 2.

7 Contributions

- Combine the disparate theories of reflective towers and mix-like partial evaluators
- Generalized framework for collapsable towers. A generilzation of [11] for “heterogeneous” towers
- Extension of the core $\lambda\uparrow\downarrow$to support side effects, combining previous insights into multi-level λ [19] and work on side-effects in partial evaluators [17].
- Denotational account of original base language/Pink and the new $\lambda\uparrow\downarrow$(including a store, which was not part of [4])
- Development a CESK-style abstract machine/ abstract interpreter for said extended $\lambda\uparrow\downarrow$
- Form a basis for further work on towers by providing a stage polymorphic base evaluator capable of modelling functional or imperative languages
- Mimick a practical tower through a SECD machine on top of the base evaluator and show compilation without staging commands throughout the tower

- Theoretical proposal of how one might achieve collapsing in practice

We deviate from traditional research in reflective towers in that we do not develop a separate language that demonstrates reflective tower capabilities and part from the constraints of metacircularity and reflection. Instead of generating levels in the tower dynamically through reflection and reification operators we construct a pre-determined tower resembling towers of interpreters in practice. We demonstrate initially how meta-circularity and reflection eases the collapsing process and then wire the tower in a way that breaks key implicit assumptions of said technique. Finally we propose a generalization of the original framework that deals with the constraints such a semantic gaps and lack of reflection and reification. We evaluate the framework on a set of abstract machines that are convenient to implement in Lisp-like fashion but are capable of modelling a broad set of functional and non-functional language properties.

8 Normalization By Evaluation (NBE)

Useful Slides NBE paper More slides Supercompilation by Evaluation Supercompilation and Normalization by Evaluation

9 Type-Directed Partial Evluation (TDPE)

[16]

9.1 Staging

There are two types of partial evaluation methodologies [20]:

- Offline partial evaluation
- Online partial evaluation [21]

Namin et al. [11], propose two languages Pink and Purple. Pink uses a form of online partial evaluation 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.

Our language extends Pink with side effects and a stack machine that makes use of pointer like semantics for Lisp-like cons pairs. Thus we build on top of the NBE-style lift operator for staging. However, calling into the base-level lift requires knowledge about its use to be passed from the layers above. We can employ several strategies of doing this:

- A basic approach exposes the base layer staging operation to the level above. This is how the original Pink implementation works.
- At every layer *deduce* whether we need to call the underlying interpreter staging operator
  - This requires every level to include an implementation of such staging operations
- A mixture of passing staging operations to the layer below or implementing ones own operators
- Find a method of passing staging decisions through each layer in a generic way without intrusive changes to the evaluators of the layers

- Decide about calling staging operations at a particular point in the tower and apply previous points

We are interested in the last two point. In heterogeneous and practical towers a programmer does not have the liberty to introduce intrusive changes along each layer. 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.

10 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 [18]).

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 [20].

An curious use-case for staging towers of interpreters started with the challenge of compiling the reflective language proposed by Asai et al. [17, 9]. The authors are able to compile a Lisp-like reflective language, built through the infinite tower of interpreters model [22, 3, 4], with respect to the initial semantics of the tower. Amin et al. [11] then extended this work to allow compilation of such towers under modified semantics i.e. dynamically changing behaviour of individual interpreters. An interesting consequence demonstrated in their paper is the ability to derive translators in the process of collapsing.

10.0.1 Example of Deriving Translators

A trivial but useful example is logging. Given the tower in 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)

Nonetheless, it is an interesting consequence of compiling towers that we can collect side-effects in individual levels (in the original framework at no extra cost), and have an interpreter above exhibit them. Analogous to a sieve, we take a coarse-grained collection of functionality, loosen it and extract only the individual grains we want.

Imagine the sieve being coated with a thin film. All grains passing through will now have the film applied to them. Here sieves are the levels below and grains are the levels above. (REFINE ANALOGY)

Our study examines this property by testing the limits of how we can get the side effects to stick to interpreters in a useful way. One could imagine optimizations, parallelization or instrumentation as possible use cases. Under certain side-effects, we may, however, reach limits in terms of security (TROJANS IN HYPERVISORS) or ability to reason about a system. We are interested in the extent of these limits. (concurrency as a side-effect: instead of launching missiles we launch threads)

11 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

12 On Side-effects and Dynamic Semantics of Programming Languages

References:

-

12.1 Adding Levels

Added stack machine DIAGRAMS

12.2 Result of collapsing

12.3 Cost of Emulation (or interpreteation)

[23] Turing Tax Turing Tax specific slides

13 Methodologies

13.1 Abstract Machines

13.2 Collapsing Towers

In order to achieve the collapsing effect in Pink [11], the authors make use of two keys points: (1) side-effects are deferred to $\lambda\uparrow\downarrow$ (2) staging commands are available throughout the tower

To achieve the same effect under constraints imposed by heterogeneity, we address both requirements first by showing heterogeneous towers are a special case of a generalized model for reflective towers and then providing a framework that uses the concept of a meta-controller from previous work on towers of interpreters to encapsulate staging information and pass it between stages.

References

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

[2] D. P. Friedman and M. Wand, "Reification: Reflection without metaphysics," in *Proceedings of the 1984 ACM Symposium on LISP and functional programming.* ACM, 1984, pp. 348–355.

[3] M. Wand and D. P. Friedman, "The mystery of the tower revealed: A nonreflective description of the reflective tower," *Lisp and Symbolic Computation*, vol. 1, no. 1, pp. 11–38, 1988.

[4] 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.

[5] J. C. Sturdy, "A lisp through the looking glass." 1993.

[6] 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.

[7] 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.

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

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

[10] 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.

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

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

[13] R. Glück, "Is there a fourth futamura projection?" in *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation.* ACM, 2009, pp. 51–60.

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

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

[16] 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.

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

[18] 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.

[19] F. Nielson and H. R. Nielson, "Multi-level lambda-calculi: an algebraic description," in *Partial evaluation.* Springer, 1996, pp. 338–354.

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

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

[22] B. C. Smith, "Reflection and semantics in a procedural language (ph. d. thesis)," *Technical Report*, 1982.

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