Collapsing Heterogeneous Towers of Interpreters

Anonymous Author(s)*

Abstract

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

To collapse a tower means to *stage* an interpreter in the tower (i.e., convert the interpreter into a compiler by splitting its execution into several stages) and statically reduce all the evaluation performed by preceding interpreters. Where the procedure to collapse homogeneous towers is trivial because computation performed in one interpreter can be represented in terms of its interpreter and information of which operations to partially evaluate can be propagated using the same built-in operators, this is not the case in a heterogeneous setting. There, one would need to convert representations of program constructs at each interpreter boundary and find a way to pass information needed by the partial evaluator through the tower. Our contributions include: (1) we construct and collapse an experimental heterogeneous tower using Pink, a language that was previously used to collapse reflective towers through a modified variant of partial evaluation called *type-directed partial evaluation (TDPE)* (2) we stage a SECD abstract machine using TDPE which required modification of its operational semantics to ensure termination in the presence of recursive calls (3) we investigate the hypothesis that staging at different levels in the tower affects its optimality after collapse.

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS Concepts • **Software and its engineering → Interpreters**; **Source code generation**; Semantics; Translator writing systems and compiler generators.

Keywords interpreters,partial evaluation, meta-programming

ACM Reference Format:
Anonymous Author(s). 2019. Collapsing Heterogeneous Towers of Interpreters. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Towers of interpreters are a program architecture which consists of sequences of interpreters where each interpreter is interpreted by an adjacent interpreter (depicted as a tombstone diagram in figure 1). Each additional *level* (i.e., interpreter) in the tower adds a constant factor of interpretative overhead to the run-time of the system. One of the earliest mentions of such architectures in literature is a language extension to Lisp called 3-LISP [9] introduced by Smith. Smith describes the notion of a reflective system, a system that is able to reason about itself, as a tower of meta-circular interpreters, also referred to as a *reflective tower* [1]. Using this architecture 3-LISP enables an interpreter within the tower to access and modify internal state of its neighbouring interpreters. An interpreter is *meta-circular* when the language the interpreter is written in and the language it is interpreting are the same. Meta-circularity and the common data representation between interpreters are core properties of reflective towers studied in previous work. We refer to towers with such properties as *homogeneous*. Subsequent studies due to Wand et al. [12] and Danvy et al. [5] show systematic approaches for constructing reflective towers. The authors provide denotational semantic accounts of reflection and develop languages based on the reflective tower model called *Brown* and *Blond* respectively.

In the original reflective tower models only minimal attention was given to the imposed cost of performing new interpretation at each level of a tower. Then works by Sturdy [10] and Danvy et al.'s language Blond [5] hinted at the

[1] Reflective towers in theory are considered to be potentially infinite. Given enough computing resources one can create towers consisting of an unbounded number of interpreters. In Wand et al.'s reflective tower model [12], for instance, new interpreters in a tower are spawned through a built-in *reflect* operator

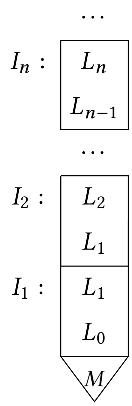


Figure 1. A tower of interpreters where each interpreter I_n is written in language L_{n-1} and interprets a language L_n, for some $n \geq 0$. In literature the tower often grows downwards, however, in our study we refer to I_0 as the base interpreter and grow the tower upwards for convenience. M is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

possibility of removing some of this overhead by partially evaluating (i.e., specializing) interpreters with respect to the interpreters below in the tower. Asai et al.'s language *Black* [3] is a reflective language implemented through a reflective tower. The authors use a hand-crafted partial evaluator, and in a later study use MetaOCaml [2], to efficiently implement the language. Asai and then, using the language Pink [1], Amin et al. demonstrate the ability to compile a reflective language while the semantics of individual interpreters in the underlying tower can be modified. Essentially this is achieved by specializing and executing functions of an interpreter at run-time to remove the cost of multiple interpretation; this effectively ***collapses*** a tower.

Parallel to all the above theoretical research into reflective towers, practical programmers have been working with towers of interpreters to some extent dating back to the idea of language parsers. Writing a parser in an interpreted language already implies two levels of interpretation: one running the parser and another the parser itself. Other examples include interpreters for embedded domain-specific languages (DSLs) or string matchers embedded in a language both of which form towers of two levels. Advances in virtualization technology has driven increasing interest in software emulation. Viewing emulation as a form of interpretation we can consider interpreters running on virtual hardware, such as the bytecode interpreter in the Java Virtual Machine (JVM) [8], as towers of interpreters as well.

However, these two branches of research do not overlap and work on towers of interpreters rarely studied their counterparts in production systems. It is natural to ask the question of what it would take to apply previous techniques in partial evaluation to a practical setting. This is the question Amin et al. pose in their conclusion after describing Pink [1] and is the starting point for this thesis.

We aim to bring previous work of removing interpretative overhead in towers using partial evaluation into practice. Our study achieves this by constructing a proof-of-concept tower of interpreters that more-closely resembles those in real-world systems. Figure 2 depicts two versions of our experimental tower. Traditionally reflective towers are thought of as completely vertical like the one on the left. However, details such as how a tower grows, shrinks and collapses while executing user programs worked rather mysteriously. We decided to implement our tower using occasional layers of compilation (as shown on the right). The two versions of our tower are extensionally equal since they yield the same output for a given program to evaluate. Part of our study is devoted to evaluating the effect of the intensional structure of towers on the act of collapsing them.

We then collapse the experimental tower under different configurations and evaluate the resulting optimized programs. We demonstrate that given a language capable of expressing types of variables that are available at run-time versus compile-time (i.e., a *multi-level language*) and a type-directed PE (TDPE), a lightweight partial evaluator due to Danvy [4] described in section **??**, we can partially evaluate individual interpreters in a heterogeneous tower and effectively generate code specialized for a user program (hopefully eliminating interpretative overhead in the process). Our work's contributions are:

1. Develop an experimental heterogeneous tower of interpreters and a strategy for collapsing it
2. Evaluate the effect that staging at different levels within our tower has on residual programs
3. Discuss the effects that heterogeneity in towers imposes on TDPE
4. Demonstrate issues with and potential approaches to staging abstract machines, specifically a SECD machine, using TDPE

In section 2 we explain background information that covers the fundamental topics we base our experiments and discussions on. We then define *heterogeneity* in towers of interpreters in section 3. In section **??** we describe the recipe that the language Pink [1] used to construct and collapse meta-circular towers and then show how this recipe changes as a result of heterogeneity. We present the implementation and evaluation of our experimental tower in section **??**. We systematically describe the process by which we create a heterogeneous tower of interpreters and incrementally collapse it in sections **??** through **??**. We conclude with an evaluation of our findings followed by a discussion of potential future work in section **??**.

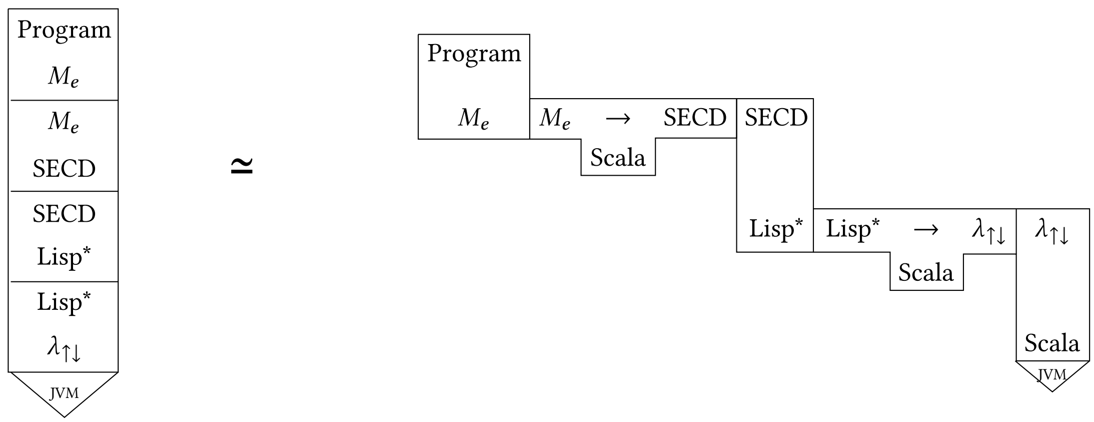


Figure 2. Tombstone diagrams that represent two versions of our experimental tower of interpreters. M_e is our toy language described in section **??**, $\lambda_{\uparrow\downarrow}$ refers to the multi-level language introduced as part of Pink [1] and Lisp* is $\lambda_{\uparrow\downarrow}$'s Lisp based front-end. *JVM* in our diagram also encompasses any underlying machinery necessary to run it. While the left depicts the intuitive view of a tower, we actually implement it using the architecture on the right. Not only is the tower on the right simpler to construct but it also highlights the power of the *lift* operator described in section **??** and its vital role in collapsing heterogeneous towers.

2 Background

2.1 SECD

2.2 Partial Evaluation

2.3 Collapsing Reflective Towers

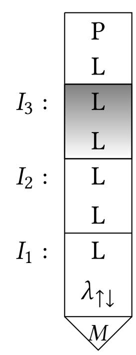
In this section we describe what it means to construct and *collapse* a tower of interpreters and the methodology Pink uses to do so [1]. Then we discuss changes that have to be considered when applying these techniques to a heterogeneous setting.

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

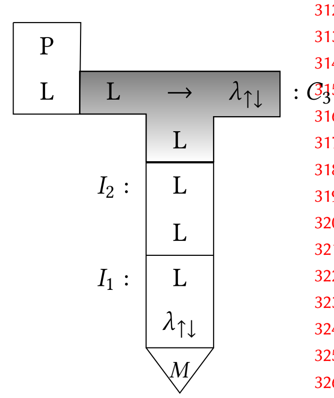
1. A multi-level language
2. A stage-polymorphic evaluator for the multi-level language
3. TDPE-style reification operator

Amin et al.'s multi-level language, $\lambda_{\uparrow\downarrow}$, differentiates between static and dynamic expressions using types; this allows $\lambda_{\uparrow\downarrow}$ to express binding-time information. It also defines a *lift* (i.e., TDPE's reify) operator such that the PE can coerce static to dynamic values. A single evaluator for $\lambda_{\uparrow\downarrow}$ operates on both dynamic and static expressions which enables so called *binding-time agnostic* staging [1]. This means to *stage an interpreter* we can annotate it with the polymorphic lift instantiated such that it invokes $\lambda_{\uparrow\downarrow}$'s *lift* (such as *compiler* in figure **??**).

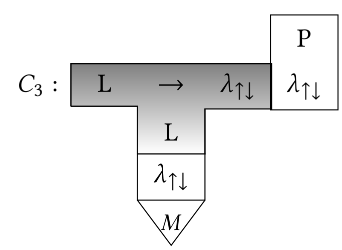
Figures 3a to 3c depict the process of collapsing a tower through tombstone diagrams. We start with a meta-circular tower of interpreters all written in the same language, L, and



(a) Tower of meta-circular interpreters, I_k, in language, L, running a program, P.



(b) Tower whose top interpreter is staged (i.e., converted into a compiler, C)



(c) Final representation of the tower in 3a after collapsing it. All intermediate interpretation (levels I_1 to I_3) has been eliminated (by evaluating it during PE time) and P has been specialized with respect to the top-most staged interpreter, C_3. The residual program P consists of $\lambda_{\uparrow\downarrow}$ terms in ANF-normal form.

Figure 3. Tombstone diagrams representing the process of collapsing a tower using $\lambda_{\uparrow\downarrow}$

Pink's $\lambda_{\uparrow\downarrow}$ at the base. The key benefit of meta-circularity is that the *lift* operator defined in $\lambda_{\uparrow\downarrow}$ is accessible to each interpreter. We can now stage some interpreter in the tower, in this example the user-most one I_3, by annotating it with *maybe-lift* expressions as described in section **??**; this interpreter is now equivalent to a compiler from L to $\lambda_{\uparrow\downarrow}$ (C_3 in figure 3b). When we execute the tower (i.e., invoke $\lambda_{\uparrow\downarrow}$'s partial evaluator) C_3 residualizes while all other levels in the tower evaluate (essentially *propagating* binding-time information of program P from the top to the base of the tower). At I_1 a call to *lift* now invokes $\lambda_{\uparrow\downarrow}$'s TDPE-style reify. Effectively after residualization the generated program will only include the values staged at the top-most interpreter while the rest of the tower was reduced at specialization time since the polymorphic lifts were instantiated to be the identity function (i.e., *interpreter* in figure **??**). The result is a collapsed tower in figure 3c where all intermediate interpreters, I_1 through I_3, have been removed from the tower (assuming the absence of side-effects at individual levels).

Although previous work did not provide an insight into the exact effect of staging at different levels in the tower, an intuitive reason we stage at the top-most level is that we want to eliminate as much interpretative overhead as possible which is achieved by collapsing the maximal set of interpreters in the tower. Sections **??** to **??** provide evidence to support this claim on our experimental tower.

3 Heterogeneous Towers

A central part of our study revolves around the notion of heterogeneous towers. Prior work on towers of interpreters that inspired some these concepts includes Sturdy's work on the Platypus language framework that provided a mixed-language interpreter built from a reflective tower [10], 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 towers of meta-circular interpreters into one-pass compilers. We continue from where the latter left off, namely the question of how one might achieve the effect of compiling multiple interpreters in heterogeneous settings. We view heterogeneous towers as a generalization of reflective towers and define *heterogeneous* as follows:

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

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

1. For any two adjacent interpreters I_k and I_{k-1} where $k \in \mathbb{N}_{\geq 1} : L_k \not\equiv L_{k-1}$ can hold
2. For any two adjacent interpreters used in the tower, I_k and I_{k-1}, the operational semantics and the representation of data can be different between the two even if the languages coincide; this gives us a way of addressing the difference in intensional structure between towers

3.1 Absence of: Meta-circularity

The first generalization described by observation 3.1 is that of mixed languages between levels of a tower. A practical challenge this poses for partial evaluators is the inability to reuse language facilities across interpreters. This also implies that one cannot in general define reflection and reification procedures as in 3-LISP [9], Brown [12], Blond [5], Black [3] or Pink [1].

3.2 Absence of: Reflection

Reflection in an interpreter enables the introspection and modification of its state during execution. It is a tool reflective languages can use to embed, for example, debuggers or run-time instrumentation into programs. Reflection in reflective towers implies the ability to modify an interpreter's interpreter which can be beneficial in the implementation of said tools. However, it also allows potentially destructive operations on a running interpreter's semantics which can become difficult to reason about or debug. Towers that we are interested in rarely provide reflective capabilities in their interpreters. Thus, we do not support or experiment with reflection in our study.

3.3 Semantic Gap and Mixed Language Systems

Danvy et al. mentioned the possibility of non-reflective non-meta-circular towers early on in his denontational description of the reflective tower model [5]. The authors explored the idea of having different denotations for data at every level of the tower. However, since it was not the focus of their study, the potential consequences were not further investigated but serve as an inspiration for the second point of observation 3.1. We call the difference in operational semantics or data representation between two interpreters a *semantic gap*.

Another motivation of ours stems from the realization that systems consisting of several layers of interpretation can feasibly be constructed. A hypothetical tower of interpreters that served as a model for the one we built throughout our work was described in Amin et al.'s paper on collapsing towers [1] and is depicted as a tombstone diagram in figure 4[2]. As a comparison our tower is shown in figure 2. We replace the x86 emulator with a SECD abstract machine interpreter and Python with our own functional toy language, M_e. The label

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

$\lambda_{\uparrow\downarrow}$ represents the multi-level core language from Pink [1] and Lisp* is the Lisp-like front-end to $\lambda_{\uparrow\downarrow}$. Although here the tower grows upwards and to the left, this need not be. The compilers, or *translators*, from M_e to SECD and from Lisp* to $\lambda_{\uparrow\downarrow}$ have been implemented in Scala purely for simplicity. To realize a completely vertical tower (i.e., consisting of interpreters only), the Lisp*-$\lambda_{\uparrow\downarrow}$ translator could be omitted so that the $\lambda_{\uparrow\downarrow}$ interpreter evaluates s-expressions directly. Similarly, the M_e-SECD compiler could be implemented in SECD instructions itself. However, we argue that the presence of compilation layers in our experimental tower more closely resembles practice and adds some insightful challenges to our experiments.

3.4 Effect of Heterogeneity

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

While a definitional interpreter can be staged using TDPE by simply lifting values it returns, the process of staging an abstract machine, as we show in section **??**, requires careful design of its division rules and consideration of which locations to lift to avoid non-termination at PE-time. Figure 5 depicts the effect of staging the SECD interpreter in our tower. After staging the SECD machine (highlighted in grey) it now operates as a compiler from SECD instructions to $\lambda_{\uparrow\downarrow}$ terms. Collapsing the tower in this configuration essentially means moving code from the SECD level to the base interpreter across the Lisp* level. Consistent with the composition rules of tombstones, the levels above SECD are residualized and present in the output as well.

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

A more subtle collapse of our tower occurs when we stage the M_e level (see figure 6). In this case staging has a slightly different effect which is not obvious from the tombstones. Since the M_e level is actually a translator already, staging the translator will simply yield another translator. Instead, staging M_e means we generate SECD instructions that *lift* (i.e., signal the PE to residualize) expressions of M_e using a new **LIFT** instruction (added in section **??**). As we can see from the annotated tombstone diagram, we use the *lift* operator as a mechanism to move code through each level to the base where it is residualized.

4 Construction and Collapse of a Heterogeneous Tower

5 Conclusions

5.1 Conclusion

5.2 Future Work

References

[1] Nada Amin and Tiark Rompf. 2017. Collapsing towers of interpreters. *Proceedings of the ACM on Programming Languages* 2 (2017), 52.

[2] Kenichi Asai. 2015. Compiling a reflective language using MetaOCaml. *ACM SIGPLAN Notices* 50, 3 (2015), 113–122.

[3] Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa. 1996. Duplication and partial evaluation. *Lisp and Symbolic Computation* 9, 2-3 (1996), 203–241.

[4] Olivier Danvy. 1999. Type-directed partial evaluation. In *Partial Evaluation: Practice and Theory*. Springer, 367–411.

[5] Olivier Danvy and Karoline Malmkjaer. 1988. Intensions and extensions in a reflective tower. In *Proceedings of the 1988 ACM conference on LISP and functional programming*. ACM, 327–341.

[6] Neil D Jones, Peter Sestoft, and Harald Søndergaard. 1989. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic computation* 2 (1989), 9–50.

[7] Brian W Kernighan, Dennis M Ritchie, Clovis L Tondo, and Scott E Gimpel. 1988. *The C programming language*. Vol. 2. prentice-Hall Englewood Cliffs, NJ.

[8] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java virtual machine specification*. Pearson Education.

[9] Brian Cantwell Smith. 1984. Reflection and semantics in Lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 23–35.

[10] John CG Sturdy. 1993. *A LISP through the looking glass*. Ph.D. Dissertation. University of Bath.

[11] Guido Van Rossum and Fred L Drake. 2011. *The Python language reference manual*. Network Theory Ltd.

[12] Mitchell Wand and Daniel P Friedman. 1988. The mystery of the tower revealed: A nonreflective description of the reflective tower. *Lisp and Symbolic Computation* 1 (1988), 11–38.

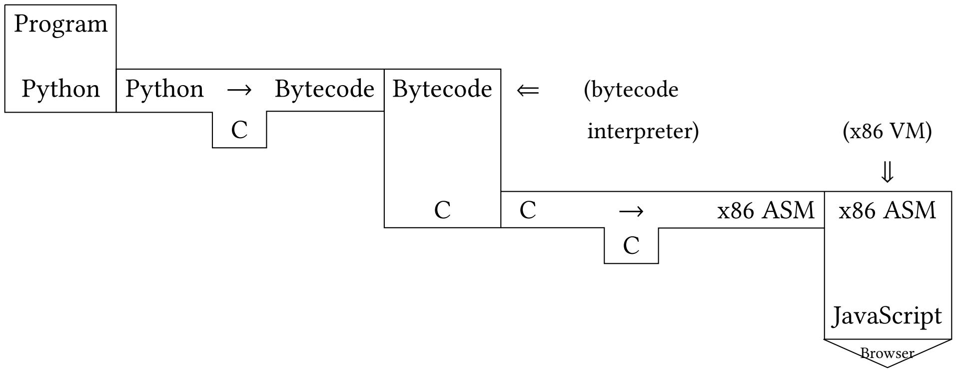


Figure 4. A hypothetical tower of interpreters that serves as the model for the tower we built (figure 2). The diagram depicts a x86 virtual machine (VM) written in JavaScript running a Python [11] interpreter that in turn executes some Python program. In this model, Python is first translated to bytecode which is then interpreted by some bytecode interpreter (written in the C language [7]). *Browser* encompasses the JavaScript interpreter within a browser and any underlying technologies required to host the browser.

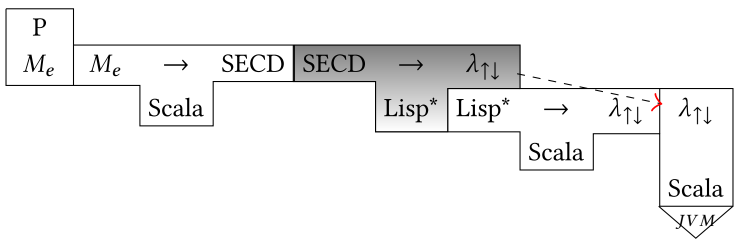


Figure 5. Our heterogeneous tower of interpreters (2) after staging at the SECD level (shaded tombstone). All computation to the left of the staged interpreter is carried into the residual program. The collapse essentially moved the code from above the SECD machine to the base.

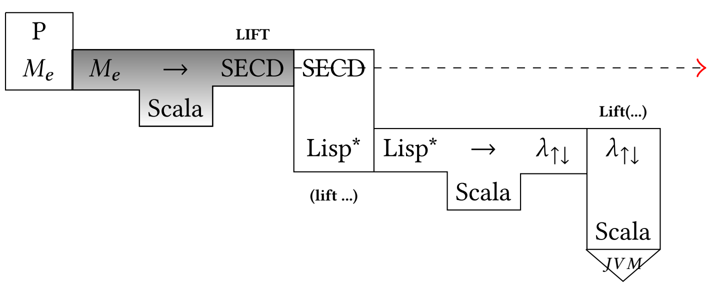


Figure 6. Our heterogeneous tower of interpreters after staging at the M_e level (shaded tombstone). At each level, starting from M_e, the *lift* operator is implemented differently but together they achive the effect of moving code from the M_e interpreter to the base.