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 partial evaluation, a process referred to as *collapsing of towers of interpreters*. Towers collapsed thus far have been homogeneous, focusing on reflective and meta-circular interpreters. We investigate how to collapse *heterogeneous* towers of interpreters, identifying the requirements due to varying data representations.

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 [10] 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. [13] 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.

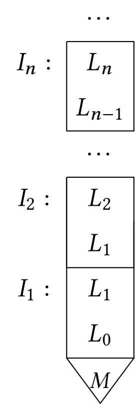


Figure 1. A tower of interpreters where each interpreter I_n is written in language L_{n-1} and interprets a language L_n, for some $n \geq 0$. In literature the tower often grows downwards, however, in our study we refer to I_0 as the base interpreter and grow the tower upwards for convenience. M is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

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 [11] and Danvy et al.'s language Blond [5] hinted at the 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

[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 [13], for instance, new interpreters in a tower are spawned through a built-in *reflect* operator

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

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) [9], 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 work.

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], 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. Discuss the effects that heterogeneity in towers imposes on TDPE
3. Evaluate the effect that staging at different levels within our tower has on residual programs
4. Demonstrate issues with and potential approaches to staging abstract machines, specifically a SECD machine, using TDPE

In this short paper, we focus on points 1 and 2. In section 2, we describe what it means to construct and *collapse* a tower of interpreters and the methodology Pink uses to do so [1]. In section 3 we discuss changes that have to be considered when applying these techniques to a heterogeneous setting.

2 Collapsing Reflective Towers

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*.

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 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 lifting introductory forms such as values; 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. The result is a collapsed tower in figure 3c where all intermediate interpreters, I_1 through

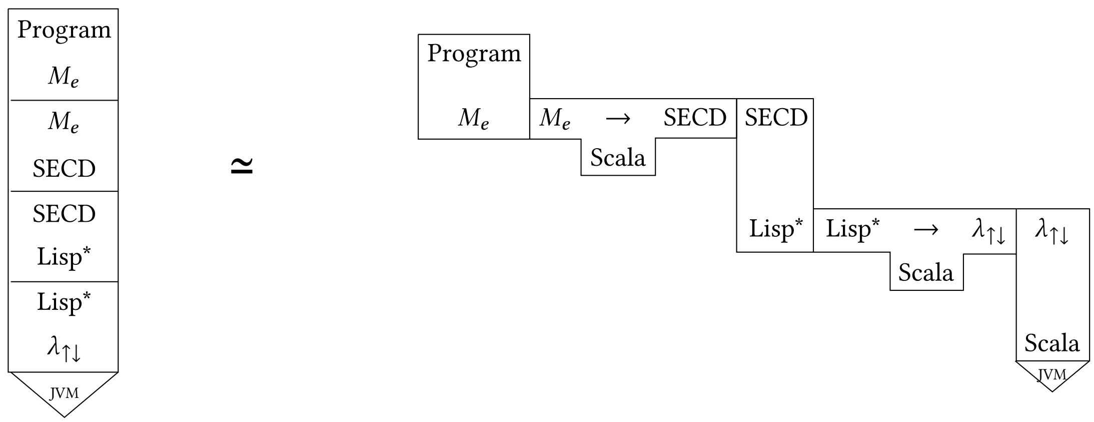
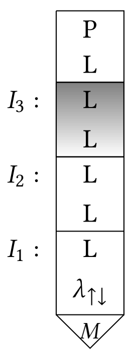
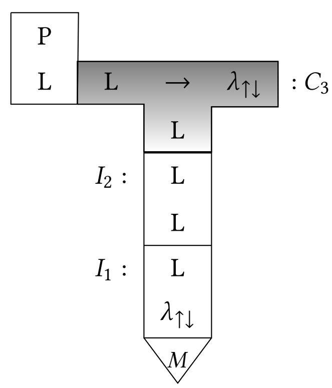


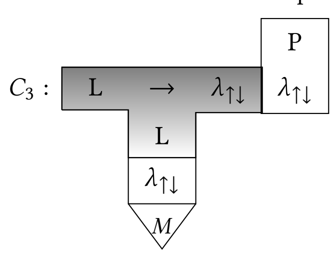
Figure 2. Tombstone diagrams that represent two versions of our experimental tower of interpreters. M_e is a toy Lisp language, $\lambda_{\uparrow\downarrow}$ refers to the multi-level language introduced as part of Pink [1] and Lisp* is $\lambda_{\uparrow\downarrow}$'s Lisp based front-end. *JVM* in 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 and its vital role in collapsing heterogeneous towers.



(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 $\lambda_{\uparrow\downarrow}$

I_3, have been removed from the tower (assuming the absence of side-effects at individual levels).

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 [11], 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, $I_1^L, I_2^L, \ldots, I_n^L$ 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 [10], Brown [13], 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 denotational 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 $\lambda_{\uparrow\downarrow}$ 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.

[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.

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 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. 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. 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.

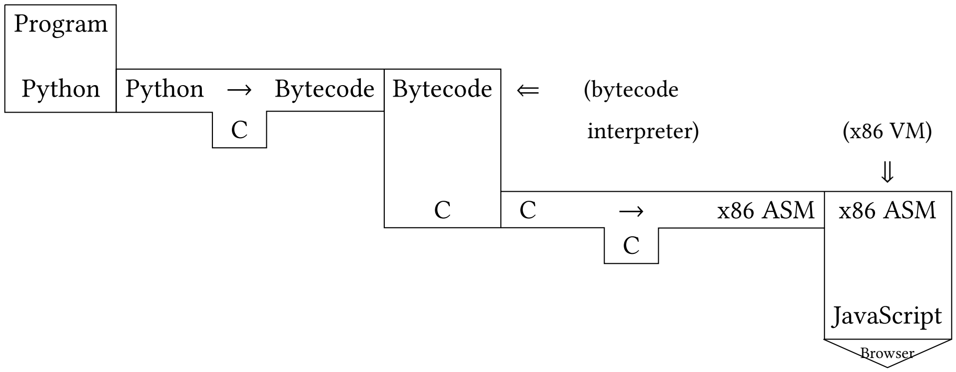


Figure 4. A hypothetical tower of interpreters that serves as the model for the tower we built (figure 2). The diagram depicts a x86 virtual machine (VM) written in JavaScript running a Python [12] interpreter that in turn executes some Python program. In this model, Python is first translated to bytecode 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.

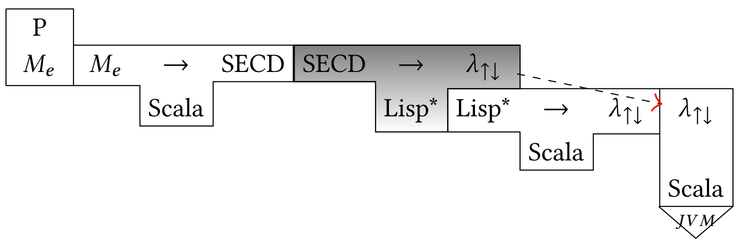


Figure 5. Our heterogeneous tower of interpreters (2) after staging at the SECD level (shaded tombstone). All computation to the left of the staged interpreter is carried into the residual program. The collapse essentially moved the code from above the SECD machine to the base.

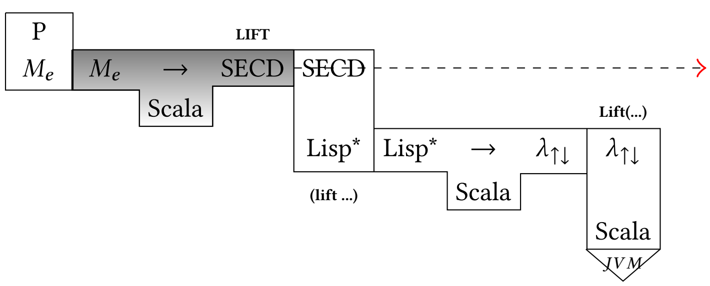


Figure 6. Our heterogeneous tower of interpreters after staging at the M_e level (shaded tombstone). At each level, starting from M_e, the *lift* operator is implemented differently but together they achive the effect of moving code from the M_e interpreter to the base.

4 Conclusions

We started by generalizing the concept of reflective towers to ones that consisted of non-meta-circular interpreters. To model a tower that could potentially contain layers of translation between languages, such as the one in figure 4, we also introduced the notion of *semantic gaps*, which dictate the extent to which two adjacent interpreters differ in operational semantics or representation of data. A combination of these two properties (non-meta-circularity and semantic gaps) form the new class of towers of interpreters that we call *heterogeneous*. The theoretical implications from heterogeneity on a tower's structure and procedure to collapse them was a focal point for our experiments. We envisioned two problems with collapsing heterogeneous towers: (1) we needed to devise a strategy to signal to the PE at the base of the tower which expressions to residualize without a meta–circular *lift* (even through levels of compilation) (2) semantic gaps required us to perform complex transformations on program constructs in one interpreter to adhere to their representation in adjacent interpreters. We constructed and then collapsed a 5-level heterogeneous tower with a SECD machine as one of its levels to provide evidence for these hypotheses and gain more insights into heterogeneity.

Staging an interpreter amounts to reifying literals, lambdas and product types it returns. An abstract machine is not guaranteed to distinguish these types by data structures or a type-system but can instead rely on dedicated instructions to differentiate data of these types. Hence, the points to reify at are dictated by the architecture of the underlying machine. In our experiments we created a conservative division tailored to the SECD stack-registers and reduced static expressions in Pink's reflect operator to achieve optimal residualization. In order to propagate the decision of whether to generate or evaluate an expression through levels in the tower, we implemented a **lift** operation for all languages in the tower. Observationally, *lifting* an expression moves it through the tower to the base and is the key to achieving the effect of collapsing. Whereas in homogeneous towers this role of *lift* was implicit and trivial due to meta-circularity, we found that heterogeneity explicates this process. We hope this furthered the understanding of some of the subtleties of collapsing towers of interpreters.

With our experiments we demonstrated the successful collapse of a heterogeneous tower. We also showed the ability of a TDPE-style *reify* operation to essentially move code across levels of a tower, which also worked across levels of compilation. However, realizing our methodology on a practical setting such as the Python-x86-JavaScript tower will require additional work. Our approach to propagating the TDPE binding-time information involved the implementation of a reification operator in each interpreter that is missing it. This then required the reverse-engineering and conversion of types in an interpreter to the representation that the interpreter below in the tower expects. In practice these changes would require intimate knowledge of and intrusive changes to an interpreter or compiler. Additionally, in our experimental tower we did not consider the residualization of side-effects which a useful collapse procedure would need for wider applicability. Our methodology could, however, help the optimization of smaller-scale systems in practice where towers consist of embedded DSL interpreters or regular expression matchers (even in the absence of meta-circularity and presence of translation layers).

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

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

[10] 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.

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

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

[13] 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.

[14] David HD Warren. 1983. An abstract Prolog instruction set. *Technical note 309* (1983).

[15] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. 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, 187–204.