TU Delft

Delft University of Technology

Document Version
Final published version

Licence
CC BY

Citation (APA)
Vergu, V., Tolmach, A., & Visser, E. (2019). Scopes and frames improve meta-interpreter specialization. In A. F. Donaldson (Ed.), *33rd European Conference on Object-Oriented Programming, ECOOP 2019* (Vol. 134, pp. 4:1-4:30). Article 4 Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing. https://doi.org/10.4230/LIPIcs.ECOOP.2019.4

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
In case the licence states “Dutch Copyright Act (Article 25fa)”, this publication was made available Green Open Access via the TU Delft Institutional Repository pursuant to Dutch Copyright Act (Article 25fa, the Taverne amendment). This provision does not affect copyright ownership.
Unless copyright is transferred by contract or statute, it remains with the copyright holder.

Sharing and reuse
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.

Scopes and Frames Improve Meta-Interpreter Specialization

Vlad Vergu
Delft University of Technology, Delft, The Netherlands
v.a.vergu@tudelft.nl

Andrew Tolmach
Portland State University, Portland, OR, USA
tolmach@pdx.edu

Eelco Visser
Delft University of Technology, Delft, The Netherlands
e.visser@tudelft.nl

Abstract

DynSem is a domain-specific language for concise specification of the dynamic semantics of programming languages, aimed at rapid experimentation and evolution of language designs. To maintain a short definition-to-execution cycle, DynSem specifications are meta-interpreted. Meta-interpretation introduces runtime overhead that is difficult to remove by using interpreter optimization frameworks such as the Truffle/Graal Java tools; previous work has shown order-of-magnitude improvements from applying Truffle/Graal to a meta-interpreter, but this is still far slower than what can be achieved with a language-specific interpreter. In this paper, we show how specifying the meta-interpreter using *scope graphs*, which encapsulate static name binding and resolution information, produces much better optimization results from Truffle/Graal. Furthermore, we identify that JIT compilation is hindered by large numbers of calls between small polymorphic rules and we introduce *rule cloning* to derive larger monomorphic rules at run time as a countermeasure. Our contributions improve the performance of DynSem-derived interpreters to within an order of magnitude of a handwritten language-specific interpreter.

2012 ACM Subject Classification Software and its engineering → Interpreters

Keywords and phrases Definitional interpreters, partial evaluation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.4

Funding This research was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206) and by a gift from the Oracle Corporation.

Acknowledgements We thank the anonymous reviewers for their feedback on previous versions of this paper, and we thank Laurence Tratt for his guidance on obtaining reliable runtime measurements and analyzing the resulting time series.

1 Introduction

A *language workbench* [9, 36] is a computing environment that aims to support the rapid development of programming languages with a quick turnaround time for language design experiments. Meeting that goal requires that (a) turning a language design idea into an executable prototype is easy; (b) the delay between making a change to the language and starting to execute programs in the revised prototype is short; and (c) the prototype runs programs reasonably quickly. Moreover, once the language design has stabilized, we will need a way to run programs at production speed, as defined for the particular language and application domain.


© Vlad Vergu, Andrew Tolmach, and Eelco Visser;
licensed under Creative Commons License CC-BY



Semantics specification languages such as Redex [10], K [30], and DynSem [34] provide abstractions for directly expressing the operational semantics rules of a language under design. For example, DynSem supports concise specification based on the implicitly modular operational semantics approach, which requires mentioning semantic components such as environments and stores only in rules that actually interact with those components [23, 22]. Such high-level specification languages reduce the effort of *defining* an object language. But how best to generate an executable prototype from such a definition?

Since we typically do not need the prototype to run especially fast, one natural approach is to generate an interpreter for the object language. For example, the original DynSem implementation [34] generates interpreters in Java. However, this approach requires a sequence of steps – generating code from the operational semantics definition, compiling that generated code, starting up a JVM, and running the generated interpreter on an object language AST – that altogether take on the order of a minute, even for very small language definitions. This delay inhibits workbench users from incorporating prototype generation and testing into their design iteration loop.

The standard solution to making a translated language more agile is to interpret it instead. An interpreter for an interpreter specification language is a *meta-interpreter*, resulting in two layers of interpretation: the meta-interpreter reads the AST of a specification and the AST of an object program, and interprets the rules from the specification, which in turn interpret the object language AST. While this reduces the code-to-run cycle, it increases the execution time of object programs by at least an order of magnitude, potentially limiting the scalability of tests or experiments. So, it seems that we either get slow interpreter generation or slow meta-interpreter execution. Can we get fast interpreter generation *and* fast interpreter execution?

There is reason to hope that we can: trace-based optimization frameworks such as RPython [4] and partial evaluation frameworks such as Truffle/Graal [38] have been successful in bringing the benefits of JIT compilation to (suitably instrumented) interpreters. We have been exploring whether such approaches will also work for meta-interpreters. In prior work [35] we demonstrated that specializing a meta-interpreter for DynSem using the Truffle/Graal framework can lead to an order of magnitude speed-up over a naive meta-interpeter. However, we were curious about whether we could do better still. Can we get close to the performance of a manual implementation of an object-language interpreter, or even of a production-quality object-language compiler?

In this paper, we report progress towards this goal. We show that the combination of the use of a uniform memory model and cloning semantics rules leads to a meta-interpreter for DynSem with a performance that is within a geometric mean factor of 4.7 of a hand-written object-language-specific interpreter for a small set of benchmarks on a simple object language. Both interpreters are implemented using the Truffle AST interpreter framework [40] and run with the Graal JIT compiler for the Java VM [38], which aggressively inlines stable method calls into efficient machine code. This work makes the following contributions:

- *Memory representation using "scopes and frames"*: The specifications of Vergu et al. [35] use environments for the representation of memory (environment and store) as is common in dynamic semantics specifications. However, this memory representation is language-specific and has high performance overhead. In this paper we use the "scopes and frames" approach [28], a uniform (language parametric) model for the representation of memory in dynamic semantics specifications based on scope graphs [25, 32]. By mapping frames onto Truffle's Object Storage Model, we can piggy-back on the optimizations for that representation.

- *Rule cloning*: The units of execution in a DynSem specification are reduction rules for language constructs. Since the same rule is used for all occurrences of a language construct in a program, the specializer considers them as *polymorphic*, with limited specialization as result. By cloning rules for each call site, rules become monomorphic, allowing Graal to inline them.
- *Evaluation*: We have evaluated the approach using the Tiger language [2]. We compare the performance of three variants of DynSem specifications for Tiger and a Tiger-specific interpreter implemented in Java, all running on the Graal VM. The variants compare memory representation (environments vs scopes-and-frames) and inlining vs not inlining. The results suggest that this is a viable approach, with performance of meta-interpretation using inlining and scopes-and-frames within an order of magnitude of the language-specific interpreter.

Outline. We proceed as follows. In the next section, we describe the DynSem specification language and review the Truffle/Graal framework. In Section 3 we discuss the design of the (hybrid) meta-interpreter. In Section 4 we review the "scopes-and-frames" approach, demonstrate its application in DynSem specifications, and discuss the mapping of frames to Truffle's Object Storage Model. In Section 5 we discuss the design of rule cloning in the meta-interpreter driven by a light-weight binding time analysis. In Section 6 we present the set-up of the evaluation experiment and discuss the results. In Section 7 we discuss related and future work.

2 Background

In this section we discuss the background on the DynSem specification language and the Truffle and Graal framework.

2.1 DynSem

DynSem [34] is a meta-DSL for specifying the dynamic semantics of programming languages. It is included in the Spoofax Language Workbench [17] and is a part of a larger effort to derive programming environments from high-level specifications [36]. In DynSem, programs are represented as terms and program execution is modeled as reduction of program terms to value terms. We illustrate the key concepts of DynSem with the example in Figure 1.

Signatures. The structure of terms is defined by means of an algebraic signature, which defines the sorts (types) of terms, term constructors, typed reduction arrows, and semantic components. Figure 1a illustrates these concepts for a subset of the term signatures of Tiger [2]. Tiger is a simple programming language originally invented for teaching about compilers; it is a statically typed language and has let bindings, functions, records, control-flow constructs and arrays. Figure 1a declares two sorts of terms: `Exp` for program expressions, and `Val` for value terms. A constructor declaration defines the arity and types of terms that a constructor can be applied to. For example, the `Plus` constructor is used to construct terms of the form `Plus(e1, e2)` where the subterms `e1` and `e2` are terms of sort `Exp`. Note that just like program expressions, value terms are represented by a sum type to represent different kinds of values, unified in the `Val` sort. The example defines integer and closure values.

An *arrow* defines the source and target sort of a reduction. For example, the `Exp` $\longrightarrow$ `Val` arrow states that `Exp` terms can be reduced to `Val` terms using the $\longrightarrow$ arrow. Semantic components are used to represent the run-time state of programs. In the example, semantic components for environments `E` (mapping identifiers to locations) and heaps (stores) `H` (mapping locations to values) are defined.

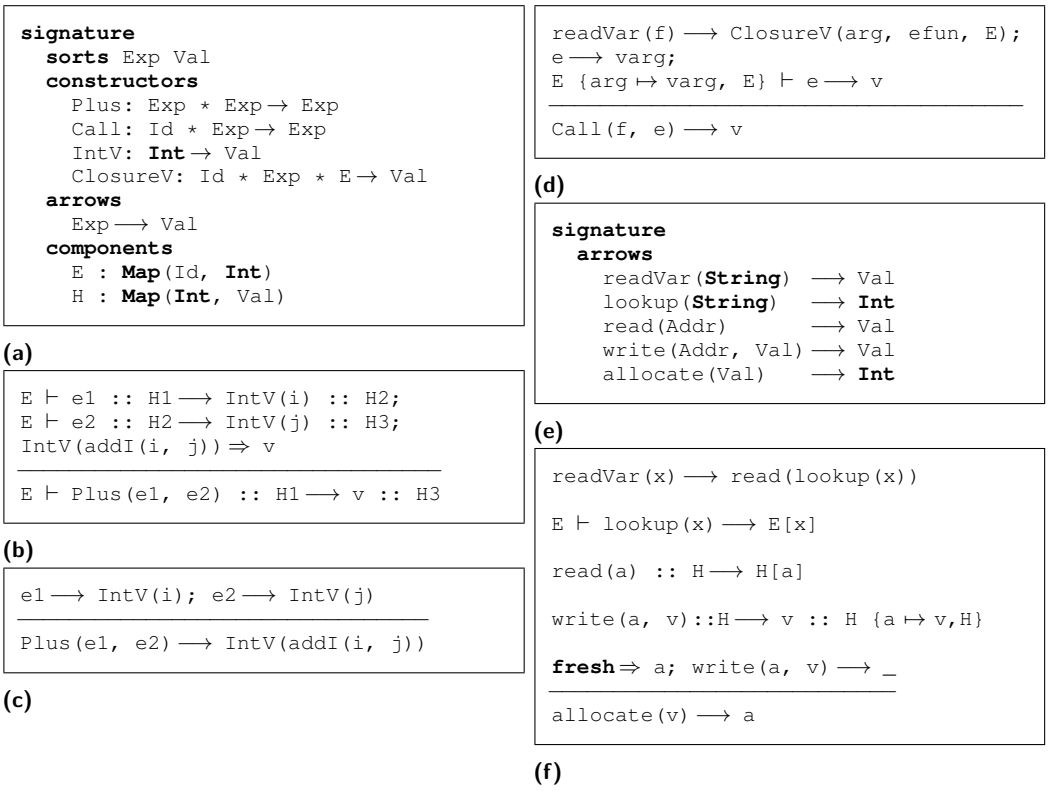


Figure 1 (a) Algebraic term signatures in DynSem. (b) Fully elaborated rule for arithmetic addition and (c) its concise equivalent with implicit propagation of semantic components. (d) Semantics of a unary function call. (e) Signatures of auxiliary meta-functions for environment and store operations and (f) their corresponding rules.

DynSem specifications are statically checked with respect to signatures. The checker ensures that term patterns in rules are consistent with constructor declarations and that arrow arguments are of the right sort.

Rules. Reduction rules define the dynamic semantics of programs by reduction of program terms to value terms. A rule has the form

```
prem1; prem2; ...
-----------------
lhs ⟶ rhs
```

where the conclusion is an arrow declared in the signature. It defines that a term matching `lhs` is reduced to the instantiation of term `rhs`, provided that the premises `prem1; prem2; ...` succeed. Premises are either recursive arrow applications or pattern matches. An arrow application premise `lhs ⟶ rhs` instantiates the pattern `lhs` with the substitutions for meta-variables from the left-hand side of the conclusion or from earlier premises, reduces it with the arrow, and matches the result against the `rhs` pattern. A pattern matching premise `lhs ⇒ rhs` instantiates the pattern `lhs`, which may possible involve application of meta-functions (see below), and matches it to the pattern `rhs`. Arrows are usually defined in a big-step style [16]. That is, a rule reduces a program term to a value term in one step, using recursive invocation of arrows in the premises. This is illustrated in Figure 1c, which defines the reduction of `Plus(e1, e2)` terms with the ⟶ arrow by completely reducing the argument terms to value terms. The right-hand side of the conclusion constructs the resulting value term by using the `addI` meta-function.

Semantic Components. The rule in Figure 1c does not account for the evaluation of an expression in the context of an environment binding variables in scope and a heap storing values with longer lifetimes. DynSem supports the propagation of such contextual information by means of so called *semantic components*, which are distinguished in read-only components and read-write components. A read-only component is mentioned to the left of the $\vdash$ symbol, and propagates downwards (environment semantics). A read-write component is mentioned after the `::` symbol and is threaded through the evaluation of the relation.

The rule in Figure 1b propagates semantic components `E` and `H` through the evaluation of the sub-expressions of `Plus`. Semantic component `E` (representing a variable environment) propagates as a read-only semantic component, while component `H` (representing a store) is threaded through the computation and returned from the rule.

A rule only has to explicitly mention those semantic components that it modifies; other components can be left implicit. The rule of Figure 1b modifies neither environment nor store and both may therefore be left implicit, as shown in Figure 1c. A static analysis infers which semantic components must be propagated and informs a source-to-source transformation that makes all components explicit.

Meta-Functions. DynSem allows standalone units of semantics to be separately defined as meta-functions. This supports reuse across rules and promotes concise rules. The semantics of a unary function call given in Figure 1d illustrate the use of meta-functions in DynSem.

Meta-functions `readVar`, `lookup`, `read`, etc. with their signatures and semantics of Figure 1e and Figure 1f, respectively, provide a library of memory operations. The operations are used, for example, to `lookup` the heap address of a variable in the environment by its name, and to `read` the value associated with this address from the heap. The `readVar` combines these two operations in a single meta-function which is used, for example, in the `Call` rule of Figure 1d to retrieve the function closure.

2.2 Truffle and Graal

We use Truffle [40] and Graal [38] as runtime frameworks for the execution of DynSem specifications. For a definitive guide we refer the reader to the Truffle and Graal literature [40, 39, 14, 38]. Throughout this section it is useful to keep in mind that a runtime derived from a DynSem specification is an interpreter of DynSem specifications that consumes an object-language specification and a program to execute, as depicted in the architecture overview Figure 6. We provide an overview of this in Section 3.

Truffle Interpreters. Truffle [40] is a Java framework for implementing high-performance interpreters, in particular interpreters for dynamic languages. Truffle interpreters are AST interpreters. In an AST interpreter the syntactic structure of the program determines the organization of the interpreter. Each AST node implements the semantics of the language construct it represents. In a typical Truffle interpreter the parser instantiates the AST of the interpreter given a particular program. Execution in the interpreter flows downwards in the tree and results flow upwards. Truffle provides the logistics for implementing interpreter nodes and maintaining the AST.

Figure 2 shows the skeletons of the two base classes that provide the basis for implementing language-specific nodes. A `Node` is the basic building block of a Truffle interpreter. The language developer extends the `Node` class to give semantics to language constructs. The `Node` class provides facilities for constructing and modifying trees of nodes and for traversing the tree, downwards and upwards. For example, a node for binary addition has two children

```
abstract class Node ... {
  Node parent;

  Node getParent() {
    return parent;
  }

  RootNode getRootNode() {
    Node rootNode = this;
    while (rootNode.getParent() != null) {
      rootNode = rootNode.getParent();
    }
    return (RootNode) rootNode;
  }

  Node replace(Node newNode){ ... }

  Node adopt(Node child) { ... }
}

abstract class RootNode ... {
  abstract Object execute(VirtualFrame f);
}
```

Figure 2 Skeletons of Truffle Node and RootNode classes and logistics for traversing the AST upwards.

nodes, one for each of its subexpressions, and provides an execution method that performs the addition and returns the result. If the implemented language has variables, the execute method is parameterized with an environment-like data structure, called a `Frame`, that contains the variables in scope at that location of the program.

An interpreter node without a parent is a `RootNode`. Each tree of interpreter nodes has a root, which is an entry point for execution and typically corresponds to a function in the object program. Multiple interpreter trees exist at run time, typically one for each function of a program. Each root node is parameterized by a frame descriptor defining the structure of the `Frame` that is propagated downwards during evaluation. For example, if a root node corresponds to a function, its frame descriptor defines the variables bound in the body of the function. The Truffle runtime uses the frame descriptor to instantiate a frame to be used when calling the function.

Specializing Truffle Interpreters. Truffle interpreters are particularly suited to dynamic languages because the AST structure of the interpreter allows each node to self-optimize based on runtime information. The core idea is that the interpreter AST evolves at run time to a more efficient implementation based on runtime values. For example, the plus operator of a dynamic language may embed semantics for both arithmetic addition and string concatenation, and at runtime specialize itself to one of these two semantics based on the (dynamic) values of its operands. A node may replace itself by a more specific variant by using the `replace` method, which updates the node's parent to point to the new variant. Alternatively, a node may decide to replace one of its children by a more efficient one, or adopt a new child altogether, by using the `adopt` method. Truffle provides a set of class and method annotations, collectively known as the Truffle DSL [14], that reduce the implementation effort (and boilerplate) of developing node specializations. The annotations drive a (compile-time) code generator which emits highly-efficient implementations of behavior specialization and inline caching.

The Graal JIT Compiler. Graal [38] is a high-performance JIT compiler for the Java VM with powerful partial evaluation and component inlining phases. Graal aggressively inlines stable method calls in order to generate efficient machine code. Runtime decisions about what calls are inlined are based on the outcome of a cost-benefit analysis. Truffle and Graal are designed to work together to obtain JIT-compiled Truffle interpreters with little effort. Graal treats each Truffle AST root node as a single compilation unit, i.e. Graal compiles root nodes individually. Once a Truffle interpreter tree stabilizes (i.e. node rewriting has stopped) Graal inlines all method calls of the nodes which are under a common root and emits machine code for that tree. A `Frame` that is never stored in a class field can remain virtualized – `VirtualFrame`. Since all the execution methods are inlined, the virtual frame can be eliminated, resulting in highly efficient machine code. If, after compilation, a node has to be re-specialized, for example due to a specialization that is no longer valid, the VM transfers execution of the entire executing tree back to interpreted code, disregards the machine code, and the tree is recompiled to machine code once its structure has stabilized again. The size of a tree therefore greatly affects the cost-benefit analysis of JIT compilation for that subtree. As we discuss in Sections 5 and 6, small trees compile cheaply but with little benefit, whereas JIT-compiling large trees delivers better peak performance but at an increased risk of costly recompilation.

3 Meta-Interpreters

The DynSem runtime of Vergu et al. [35] is a meta-interpreter, i.e. it interprets dynamic semantics specifications of a language. Figure 3 gives a macroscopic view of the components at play in meta-interpretation. A DynSem specification undergoes lightweight source-to-source transformations (syntactic desugaring, semantic component explication, factorization, etc.) to make it amenable to interpretation. The meta-interpreter enacts the desugared DynSem specification with respect to a program's AST in order to evaluate the program. Each rule of the specification is loaded in the meta-interpreter as a callable function. The body of a function is made up of meta-interpreter nodes that implement the semantics of the DynSem instructions used within the rule. This results in two layers of interpretation: the meta-interpreter interprets the rules of the specification which in turn interpret the object language AST.

While meta-interpretation reduces the code-to-run cycle, it increases the execution time of object programs, potentially limiting the scalability of tests or experiments. So, it seems that we either get slow interpreter generation or slow interpreter execution. Motivated by the goal of having fast interpreter generation *and* fast interpreter execution, the DynSem meta-interpreter is implemented as a Truffle [40] AST interpreter and executes on an Oracle Graal VM [38]. Much of the original meta-interpretation research [35] is focused on determining an interpreter morphology and providing runtime information to the Graal JIT such that it can remove the meta-interpreter layer.

Hybrid Meta-interpretation. Because meta-interpretation is slowed down by interpretation of generic term operations (pattern matching and construction), and because term operations for an object language are specific to that language, the DynSem meta-interpreter replaces generic term operations with statically generated language-specific term operations, which are derived from the DynSem specification of the language. Vergu et al. named the combination of specification meta-interpretation and generated term operations *hybrid*

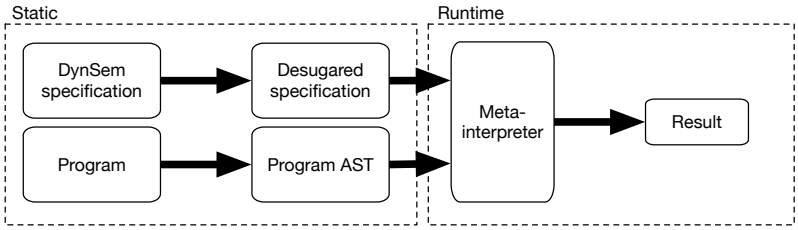


Figure 3 Overview of meta-interpretation.

meta-interpretation [35]. The original hybrid meta-interpreter starts up with generic term operations that immediately specialize themselves to the language-specific operation at their first execution, which is essentially a form of local JIT compilation.

Meta-interpreter Modifications. We apply the improvements presented in this paper to the DynSem hybrid meta-interpreter with two small modifications. First, we replace the rule dispatch mechanism by a simple rule call mechanism with an inline cache. The simplified rule call mechanism looks up the callee rule in the registry of rules and invokes it. The inline cache allows the call mechanism to remember callee rules so that the lookup is avoided in future calls. We chose to make this simplifying refactoring to allow a redesign of the rule call specialization mechanism, as we will show in Section 5. Second, we refactored the meta-interpreter to directly use the generated term operations instead of lazily replacing generic ones at run time. At best this leads to one less iteration required until warmup, but it simplifies interpreter initialization. The change does not have an effect after warmup and thus has no impact on the evaluation of the contributions of this paper.

Limitations of Name Resolution with Maps. In the original DynSem work [34], typical language specifications model name binding, resolution and program memory using abstractions for environments (mapping names to addresses) and stores (mapping addresses to values). Thus, for example, every reference to an object program variable involves a string-based lookup of the variable name in an environment data structure. Environments and stores are themselves implemented using ordinary DynSem reduction rules on top of a built-in type of persistent (i.e. functional) maps. The approach has previously been identified as a DynSem performance bottleneck [35]. The performance penalty is due in the first instance to the inherent cost of (hash-)map operations. But a more fundamental issue is that the JIT compiler cannot see the algorithms of the underlying maps, which means it cannot comprehend the operation of environments, and hence cannot comprehend name resolution in object programs. Observing and optimizing name resolution is, however, an essential ingredient in JIT compilation. Moreover, to write an environment-based DynSem specification, a language developer must define name binding and resolution in the dynamic semantics. Typically, they do this by writing higher-level DynSem meta-functions, such as variable lookup, that abstract from the low-level details of environment manipulation and encapsulate the object language's name resolution policy (Section 2.1). Unfortunately, such meta-functions are typically language-specific, making them difficult to reuse.

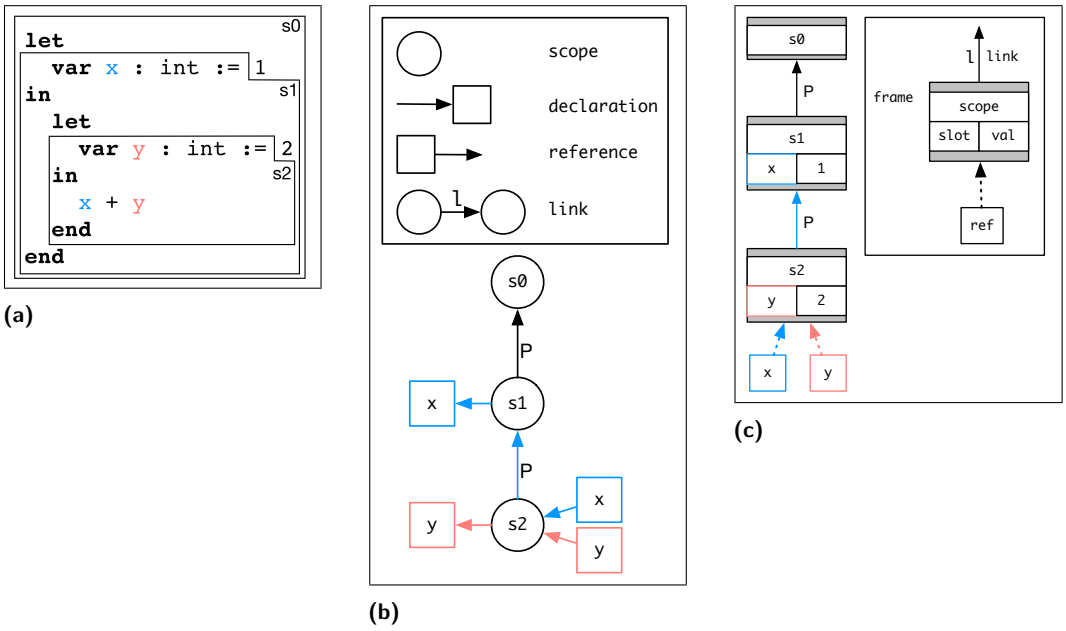


Figure 4 (a) Program with nested let bindings. The labelled box surrounding a code fragment indicates the scope the fragment resides in. Declarations and references for the same name are shown in the same color. (b) The scope graph describing the name binding structure of the program. Colors highlight name resolution paths from references to declarations. (c) Heap of frames at the end of program evaluation.

4 Scopes and Frames

To address the performance issues of the use of maps for the representation of name binding, we adopt the *scopes-and-frames* approach of Poulsen et al. [28]. In this section, we provide an overview of the previous work on *name resolution with scope graphs* and *frames* to represent scopes at run time. Then we discuss the extension of DynSem with support for scopes-and-frames and its implementation in terms of Truffle's Object Storage Model.

4.1 Name Resolution with Scope Graphs

Our approach is based on the theoretical framework of a *resolved scope graph* [25], which is a distillation of a program's name-binding structure that supports name resolution in a mostly language-independent way. Consider the small program of Figure 4a and its corresponding resolved scope graph in Figure 4b. Scopes are code regions that behave uniformly with respect to name binding and resolution. They are marked in code with labelled boxes and are shown in the scope graph as named circles. Scopes contain declarations, shown as named boxes with an incoming arrow, and references, shown as named boxes with an outgoing arrow. Visibility inclusion between scopes is shown as a labelled directed arrow between scopes. For example, the fact that declarations of the outer **`let`** are visible in the inner **`let`** is indicated by the arrow from scope `s2` to `s1`. Arrow labels characterize visibility inclusion relationships. In this case the P label indicates a lexical parent inclusion relationship. Resolving a name involves determining a path in the graph from the scope containing the name reference to the scope containing its declaration. The reference `y` resolves to the local declaration by the red path in the scope graph, while reference `x` resolves to the declaration in the parent scope by the blue path. The name resolution of a program is the set of paths which uniquely relate each reference to a declaration.

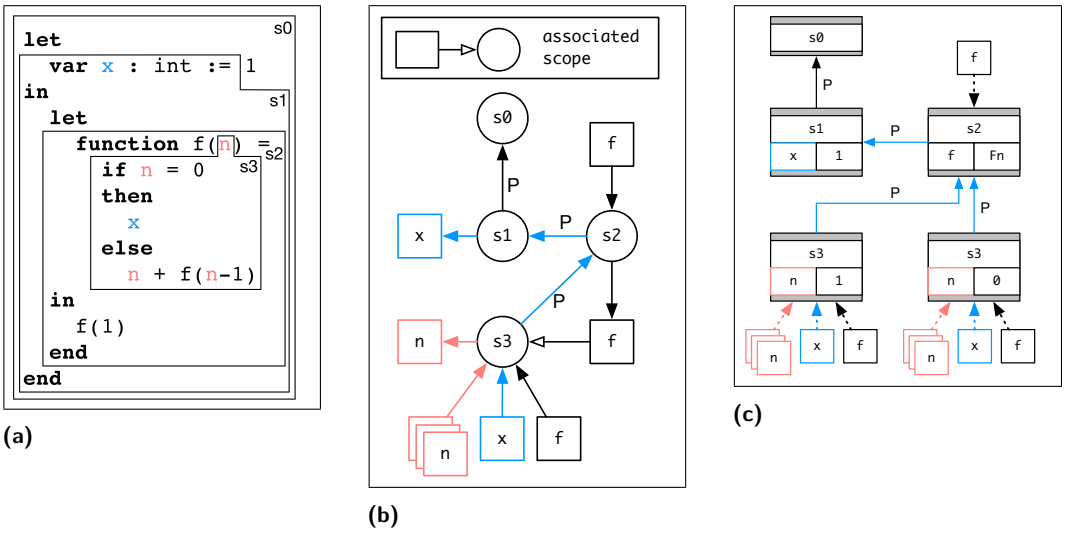


Figure 5 (a) Program with nested let bindings and a recursive function. (b) The scope graph describing the name binding structure of the program. (c) Heap of frames at the end of the evaluation of the program.

The example in Figure 5 shows how function scopes are modeled using scope graphs. These examples demonstrate examples of lexical scope, in which declarations in inner scopes shadow declarations in outer scopes. The Tiger language, which is used for the experiments in this paper, also has records and recursive type definitions. However, scope graphs are not limited to these patterns, but rather support the formalization of a wide range of name binding patterns, including variations of let bindings (sequential, parallel, recursive), modules with (recursive and transitive) imports, classes with inheritance, packages [25, 24], type-dependent name resolution [32], and structural and generic types [33]. The framework allows modeling a variety of visibility policies by configuring path specificity and path well-formedness predicates [32].

Frames. Poulsen et al. [28] provide the theoretical foundation for using a resolved scope graph to describe the layout of frames in a heap and the semantics of the base memory operations: allocation, lookup, access, and update. Declarations and references of a scope provide a recipe for constructing a memory frame at run time. A heap of frames, for example that of Figure 4c, results from program evaluation. A new frame is created when evaluation enters a new scope. The structure of the frame is determined by the declarations and references in its describing scope, which become slots of the frame. Newly created frames are linked to existing frames in accordance to their scope links. In the frame heap, references are related to slots by the name resolution path from the scope graph. Resolving a reference to a slot is performed by traversing frame links in accordance with the path. A new frame is created each time evaluation enters a scope. We illustrate this in the program of Figure 5, where the function body is evaluated in a fresh frame for each function call. Note that for a recursive function like this, multiple frames for a single scope can exist simultaneously.

Architecture. In the rest of this section we describe how we incorporate scopes-and-frames into DynSem. Figure 6 gives an architectural overview of the approach. The static semantics of the object language is described in the constraint-based NaBL2 [32] language. Notably, it

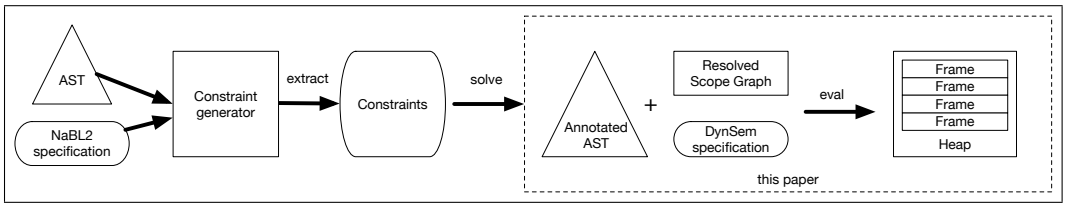


Figure 6 Architecture of the approach: static analysis on a program's AST via constraints produces an AST with explicit name and type information, which is the input for interpretation in accordance with a dynamic semantics specification.

uses scope graphs to represent the binding structure of the programs. The result of type checking with an NaBL2 specification is an annotated AST and a resolved scope graph. The DynSem specification for the object language uses frames based on scopes in the scope graph to represent memory and paths in the scope graph to resolve names to declarations in the frame heap.

4.2 Static Semantics with NaBL2

The scope graph for a program is constructed during type checking. The type checker derived from an NaBL2 specification generates constraints for an object program, which are solved by a language-independent constraint solver. We give a brief introduction to static semantics specifications with NaBL2 [32] using the rules in the left column of Figure 8 for the subset of the Tiger language used in the examples in Figure 4 and Figure 5. The signature of the abstract syntax of this subset is defined in Figure 7. (For the sake of conciseness of the presentation we have simplified the constructs in the subset to unary instead of n-ary let bindings and function definitions and calls. Furthermore, we use type equality instead of subtyping. For the experiments we have used the full Tiger language.)

An NaBL2 rule of the form `Srt[[C(e1, e2, ...) ^ (s) : t ]]:= C.` specifies that the (abstract syntax of) language construct `C(e1, e2, ...)` in the context of scope `s` has type `t` provided that the constraint `C` is satisfied. The constraint in the body of a rule is typically a conjunction of multiple simpler constraints. Constraints include recursive invocations `Srt[[C(e1, e2, ...) ^ (s) : t ]]` of constraint rules on subterms, unification constraints on constraint variables, and scope graph constraints, which support the introduction of a new scope (**new** `s`), the definition of a scope edge ($s \xrightarrow{P} s'$), the definition of a declaration in a scope ($o \leftarrow s$), the definition of a reference in a scope ($o \rightarrow s$), the association of a type with an occurrence (`o : t`), and the resolution of a reference to a declaration ($o \mapsto d$). Here `o` denotes an *occurrence* `NS{x}` consisting of a namespace `NS` and a concrete occurrence of a name `x` in a program. The NaBL2 constraint `@l.scopeOf := s'` attaches the newly created scope `s'` as a property on the program term to make it available to the runtime.

For example, the rule for `Let` introduces a new scope `s_let`, links it to the parent scope, and passes it on as the binding scope for the declaration and as the scope of its body expression. The rule for `VarDec` introduces the variable `x` as a bound variable in the binding scope `s'` and associates the type of the initializer expression with it. The rule for `Var` declares `x` as a reference in the scope of the variable, resolves the name to a declaration `d`, and retrieves the associated type `ty`. The rule for `FunDec` creates a new scope `s_fun` for the body of the function and declares the formal parameter `x` as a declaration in that scope.

```
signature
  sorts Id
  sorts Dec constructors
    VarDec : Id * Type * Exp → Dec
    FunDec : Id * Id * Type * Type * Exp → Dec
  sorts Exp constructors
    Let    : Dec * Exp → Exp
    Var    : Id → Exp
    Call   : Id * Exp → Exp
    Plus   : Exp * Exp → Exp
    Minus  : Exp * Exp → Exp
```

Figure 7 Signature for an adapted subset of Tiger.

```
Exp[[ l@Let(dec, e) ^ (s) : ty ]] :=
  new s_let, s_let -P-> s,
  @l.scopeOf := s_let,
  Dec[[ dec ^ (s_let, s) ]],
  Exp[[ e ^ (s_let) : ty ]].
```

```
newframe(scopeOfTerm(l)) ⇒ F';
link(F', L(P(), F) ⇒ _;
Fs (F', F) ⊢ dec ⟶ _;
F' ⊢ e ⟶ v
————————————————————————
F ⊢ l@Let(dec, e) ⟶ v
```

```
Dec[[ VarDec(x, t, e) ^ (s', s) ]]:=
  Tp[[ t ^ (s) : ty ]],
  Exp[[ e ^ (s) : ty ]],
  Var{x} ← s', Var{x} : ty.
```

```
F ⊢ e ⟶ v2;
set(F', x, v2) ⇒ _
————————————————————————————————
Fs (F', F) ⊢ VarDec(x, _, e) ⟶ U()
```

```
Exp[[ Var(x) ^ (s) : ty ]] :=
  Var{x} → s, Var{x} ↦ d, d : ty.
```

```
F ⊢ Var(x) ⟶ get(lookup(F, x))
```

```
Dec[[ d@FunDec(f, x, t1, t2, e) ^ (s', s) ]]:=
  new s_fun, s_fun -P-> s',
  @d.scopeOf := s_fun,
  Tp[[ t1 ^ (s) : ty1 ]],
  Tp[[ t2 ^ (s) : ty2 ]],
  Var{x} ← s_fun, Var{x} : ty1,
  Exp[[ e ^ (s_fun) : ty2 ]],
  Var{f} ← s', Var{f} : FUN(ty1, ty2).
```

```
FunV(F, scopeOfTerm(d), arg, e) ⇒ clos;
set(F, f, clos) ⇒ _
————————————————————————————————
Fs (F', F) ⊢ d@FunDec(f, arg, _, e) ⟶ U()
```

```
Exp[[ Call(f, e) ^ (s) : ty2 ]]:=
  Var{f} → s, Var{f} ↦ d, d : FUN(ty1, ty2),
  Exp[[ e ^ (s) : ty1 ]].
```

```
get(lookup(F, f)) ⇒
  FunV(Fp, s_fun, x, e_fun);
link(newframe(s_fun), L(P(), Fp)) ⇒ Fcall;
F ⊢ e ⟶ varg;
set(Fcall, x, varg) ⇒ _;
Fcall ⊢ e_fun ⟶ v
————————————————————————————————
F ⊢ Call(f, e) ⟶ v
```

```
Exp[[ Plus(e1, e2) ^ (s) : INT() ]]:=
  Exp[[ e1 ^ (s) : INT() ]],
  Exp[[ e2 ^ (s): INT() ]].
```

```
e1 ⟶ IntV(i1); e2 ⟶ IntV(i2)
————————————————————————————
Plus(e1, e2) ⟶ IntV(plusI(i1, i2))
```

```
Exp[[ Minus(e1, e2) ^ (s) : INT() ]]:=
  Exp[[ e1 ^ (s) : INT() ]],
  Exp[[ e2 ^ (s): INT() ]].
```

```
e1 ⟶ IntV(i1); e2 ⟶ IntV(i2)
————————————————————————————
Minus(e1, e2) ⟶ IntV(subI(i1, i2))
```

Figure 8 Left: static semantics in NaBL2 for an adapted subset of Tiger. Right: corresponding dynamic semantics in DynSem using scopes and frames.

Note that the rule for `VarDec` analyzes the initializer expression using scope `s`, which is the outer scope of the corresponding `Let`. This entails that the variable declaration cannot be recursive (refer to itself). On the other hand, the rule for `FunDec` makes the scope `s'` in which the function is added as declaration, a parent scope `s_fun`, the scope of the body of the function. This entails that functions *can* be recursive.

```
sorts Val Frame Addr Occurrence

components
  F : Frame

sorts Link constructors
  L: Label * Frame → Link

arrows
  newframe(Scope) ⟶ Frame
  link(Frame, Link) ⟶ Frame
  lookup(Frame, Occurrence) ⟶ Addr
  get(Addr) ⟶ Val
  get(Frame, Occurrence) ⟶ Val
  set(Addr, Val) ⟶ Val
  set(Frame, Occurrence, Val) ⟶ Val
```

Figure 9 DynSem API for frame operations.

4.3 DynSem with Scopes-and-Frames

Frame-based DynSem specifications rely on primitive frame operations provided as a language-independent library. Figure 9 declares the most important frame operations but elides their implementation. We discuss their semantics here; a reference dynamic semantics is given by Poulsen et al. [28].

The collection of linked frames is called the *heap*. The `newframe` operation instantiates a new frame in the heap given a `Scope`, which is a reference to a scope in the scope graph. This creates the required frame and frame slots for declarations and references but does not link the new frame. The `link` operation adds a link to a given frame. All links are labelled as in the scope graph. An `Occurrence` is a unique identification of the use of a name at a specific location in the program. Static name analysis transforms the program AST to replace each name occurrence, be it a declaration or a reference, with a unique identifier. Due to its uniqueness each occurrence is in precisely one scope. Given a reference occurrence and a frame, the `lookup` operation traverses the heap from the given frame to the frame holding a slot for the declaration occurrence by using the statically computed name resolution path. A lookup result is an `Address` specifically identifying a slot in a frame. Operations `get` and `set` read and update slots, respectively. Both operations come in a basic form operating on an address, and in a form directly operating on a frame and a slot.

Frame operations provide the building blocks for defining frame-based dynamic semantics specifications. The right column of Figure 8 shows the dynamic semantics in DynSem for the subset of Tiger discussed above. Each DynSem rule is listed next to the NaBL2 rule for the same construct. The binding in the DynSem rules follows the static semantics. Where the NaBL2 rule uses a scope, the DynSem rule uses a corresponding frame. Where the NaBL2 predicate is indexed by a scope (or scopes), the DynSem arrow is indexed by a corresponding frame (or frames). Thus, the language constructs are evaluated with the `Fs (Frame, Frame) ⊢Dec ⟶ Unit` and `F ⊢Exp ⟶ Val` arrows.

Where the NaBL2 rule creates a new scope, the DynSem rule creates a corresponding frame. There is some choice in the decision *when* to create a frame for a scope. For example, in the case of a `Let`, the frame is created as soon as the construct is evaluated. (Note that the scope is obtained from the NaBL2 `scopeOf` AST property, which is read using `scopeOfTerm` operator.) However, the evaluation rule for a function declaration does *not* create an instantiation of the scope of the function. Rather, a closure (`FunV`) is created that

records the scope and the *parent* frame (`F`) of the function declaration. Only evaluation of the corresponding function *call* creates the function call frame and links it to the parent frame from the closure.

Where the NaBL2 rule declares a name, a DynSem rule assigns a value to the corresponding slot. For example, the `VarDec` rule assigns the value of the initializer expression to the slot for the variable in the binding frame. In the case of a function, the assignment of the value of the actual parameter is only done once the frame is created by the function call.

Where the NaBL2 rule resolves a name, the DynSem rule uses `lookup` to find the corresponding slot, using the path obtained from resolving the name in the scope graph. For example, the `Var` rule looks up the address of the slot for the variable and gets the value stored there. Similarly, the `Call` rule looks up the address of the function name and gets the closure stored there.

The systematic correspondence between static and dynamic name binding exhibited by the rules in Figure 8 extends to all name binding patterns covered by scope graphs. The Tiger language used for the evaluation of this paper has n-ary sequential let bindings, mutually recursive function declarations, type declarations, (recursive) record types, and arrays. The scope of a record describes the layout of its fields. A record instance is a frame derived from record's scope and holds field values. Record instantiation involves retrieving the scope of the record and creating a new frame from it.

4.4 Native Library for Scopes-and-Frames

A resolved scope graph is the result of static name and type analysis; once created, the graph and all the scopes it describes remain constant at run time. Thus, all frames created for a given scope will have the same structure, and the edges between frames follow the pattern fixed by scope graph edges. For example, a particular local variable reference in a program will always have the same name resolution path and will always identify the same slot in its declaration frame. This means that at run time we can partially evaluate a variable lookup to a number of frame link traversals and an offset in a declaration frame, similar to the way an optimizing compiler would optimize lookups statically.

The implementation strategy presented in this section is designed to allow the JIT compiler of the hosting VM (an Oracle Graal VM) to observe that frame structure is constant and to perform optimizations based on this observation. Our approach is to provide a Java implementation of the scopes and frames API of Figure 9, to be used in DynSem specifications. The library implements language-independent optimizations on frame operations which any language with a frame-based DynSem specification can benefit from, out of the box.

Object Storage Model. Our implementation choice is to model scopes and frames using the Truffle Object Storage Model (OSM) [37] and to implement scope and frame operations on this model. The OSM was designed as a generic framework for modeling memory in languages with dynamic name binding and typing. In particular the OSM provides a framework for modeling objects in memory that undergo shape changes, similar to objects in prototype-based languages such as Javascript. Truffle and Graal have special knowledge of the classes that make up the OSM and can perform optimizations on memory allocation and operations. Applying the OSM to a scope graph, which is by definition fixed at run time, is akin to applying it to its ideal corner case: all shapes of all objects are constant. It is however possible that the OSM introduces a certain amount of overhead that persists even in this ideal situation. As an alternative implementation strategy, one could map a scope to a Truffle

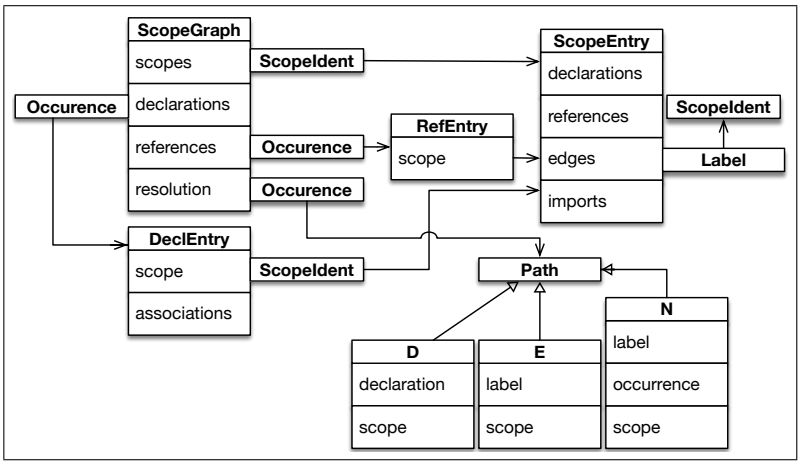


Figure 10 Components of a scope graph.

FrameDescriptor and a heap frame to a VirtualFrame. However, this mapping is intricate and would require all linked frames to be materialized in order to support frame linking. It is our understanding that materialized frames are slower than frames on the OSM.

We give a brief overview of the mapping of scopes and frames to the OSM. The OSM has three basic building blocks: objects, shapes and properties. A shape is a manifest of the properties of a family of objects and how they are laid out, akin to a prototype for an object or a class for an instance object. Shapes act as both descriptors for objects and factories for objects. A shape can be used to check whether a given object conforms to it, to retrieve properties of the object and to create new objects of that shape. A property uniquely identifies a slot and provides additional metadata to the JIT, such as whether the slot is mutable, nullable, and the type of values that it will store. The metadata informs the shape as to how the storage area for an object is to be constructed. Additionally, a property of a shape is the most efficient way to read or write the slot it identifies in an object of that shape. A property can therefore be seen as both a slot descriptor and a slot offset into an object.

Scope Graphs on OSM. Figure 10 shows the components in the makeup of a scope graph. We model them using the Truffle OSM. Declarations of layout interfaces inform the Truffle DSL to generate their implementations. A scope graph consists of scopes, declarations and references. A name resolution complements the scope graph with resolution paths from references to declarations. Paths start at the reference scope and end at the declaration scope. We use occurrences to uniquely identify declarations and references, and scope identifiers to uniquely identify scopes. Scope identifiers and occurrences are the keys to associative arrays maintained by the scope graph and are used to access detailed data. Note that we store scope graph data in a flattened representation; it is more efficient to look up scopes, declarations and references in flat associative maps than to search in graph-like structures. In the implementation, the associative arrays are instances of `DynamicObject` from the Truffle OSM. This allows Graal to optimize allocations and lookups, and gives us a set of tools for efficient access. `Occurrence` and `ScopeIdent` are optimized to have efficient hash code computation and fast equality checking.

At run time there exists precisely one scope graph. The meta-interpreter keeps a reference to the scope graph in a global interpreter context which is accessible to any interpreter node. This allows scope graph information to be accessed from anywhere in the meta-interpreter.

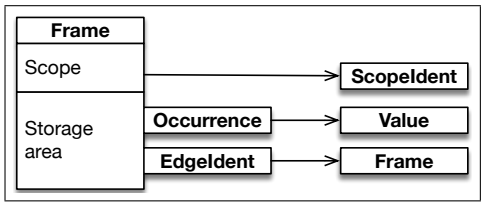


Figure 11 Structure of natively implemented frames.

Frames on OSM. We map frames and their respective operations onto the three core concepts of the OSM. Figure 11 describes the makeup of a frame. We implement a frame as an OSM object. A frame is made up of a scope uniquely identified by a `ScopeIdent` and an area for data storage. Each scope defines a unique frame shape. Each declaration is identified by its `Occurrence` and derives a frame slot property. Each edge of a scope is identified by an `EdgeIdent` – a pair of the edge label and the destination scope, and becomes a shape property and a slot in a frame. A shape dictates the structure of the storage area of a frame. Note that, by construction, all frames of a scope have the same shape. By checking whether any two frames have the same shape we effectively check whether they are frames of the same scope and vice versa.

Given a reference `Occurrence` and a starting frame, we look up the intended slot by traversing frame links as dictated by the name resolution path from the resolved scope graph. The result of the lookup is the address of the slot. The address is a pair of the frame and declaration `Occurrence` of the slot. The `Occurrence` identifies a slot property in the shape of the frame. This slot property can be used to efficiently access the slot in all frames of that shape. By definition, the relationship between a code fragment at a particular location and its surrounding scope is static. This means that code at that particular location will always execute in the context of frames derived from the same scope. This allows slot properties to be cached after their first lookup and later applied to access the slot efficiently, speeding up memory operations considerably. Such caching is particularly efficient because it can be left unguarded, since there is a static guarantee that the cached property will always be valid for that particular code location.

An advantage of mapping scopes and frames onto the Truffle OSM is that it allows the JIT compiler to observe memory operations. Since the JIT compiler can see through the memory of a running interpreter, we expect that the improvement will not be limited to just faster memory operations, but that the JIT will also optimize the running program by optimizing memory allocations. An additional advantage of using native frames is that garbage collection of frames is automatic and requires no effort from the language developer.

The native scopes and frames library makes the frame heap implicit and mutable, and does not allow it to be captured or reset. On the other hand, the vanilla DynSem library for scopes and frames uses explicit persistent data structures to model the heap. Although the heap is normally hidden from view (as an implicitly threaded semantic component), a language designer could intentionally define a semantics that observes it, captures or resets it. However, we have not encountered a language for which this would be a desirable implementation strategy. For example, even if a language needed transactional memory, capturing and resetting the entire heap would not be a good implementation approach; something finer-grained is needed. A more realistic approach would be to wrap the scopes and frames library to provide transaction support. This would work for both the vanilla DynSem and native scope and frames libraries.

5 Rule Inlining

The DynSem meta-interpreter [35] relies on Graal to optimize code within a rule and calls across rules. A rule call in the meta-interpreter corresponds to a function call in a regular interpreter. The JIT compiler will try to inline stable callees in order to reduce the number of dispatches and to generate larger compilation units. We observe that the vast majority of DynSem rules do not perform stable calls. The underlying cause is that most rules are intermediate rules, i.e. they adapt the input program term and call other rules to further reduce sub-terms. Consider, for example, the program of Figure 12a and the rule call tree of Figure 12b corresponding to its evaluation. With the exception of `FunDef`, `Var` and **`Int`**, all rules are intermediate. With the exception of meta-functions which are identified statically by their name, a callee rule is identified at runtime by the sub-term to be reduced, which in turn depends on the caller's input term. In other words a callee rule is looked up by what the JIT compiler sees as a runtime parameter to the caller. If it cannot determine that a caller's input term is constant, the JIT cannot decide to inline callees.

Not inlining of an intermediate callee rule leaves that rule exposed to calls from various callers on various program terms. We call a rule *polymorphic*, if throughout its invocations it reduces different terms. Conversely, a rule that always reduces the same term is *monomorphic*. For example, the `Call`, **`Int`** and `Var` rules of Figure 12b are polymorphic. (In this simple example, relatively many rules are monomorphic. In practice most rules in a specification are polymorphic, because the corresponding language constructs are used more than once in the program under evaluation.) Callees of polymorphic rules are not inlined, and not inlining increases the number of polymorphic rules. In larger programs, the net result is many small polymorphic rules which perform dynamic calls.

We distinguish two kinds of rule dispatch in a DynSem interpreter: *dynamic dispatch*, which depends on runtime values of the object program, and *structural dispatch*, which depends on the object program AST. In the call tree of Figure 12b all star-labeled arrows represent structural dispatch. It is desirable, and plausible, that all structural dispatch be eliminated by the JIT compiler; however, the issues outlined above prevent this. In this section we address this problem by presenting improvements to the DynSem interpreter that enable it to take explicit inlining decisions. In the ideal case the only remaining calls are those corresponding to dynamic dispatches, as illustrated in Figure 12d. The improvements consist of the following components:

- A rule-level source-to-source transformation on DynSem specifications that explicitly annotates structural rule dispatch.
- A load-time fusion of overloaded rules.
- A run-time rule-level signaling mechanism which allows any interpreter node to query whether its surrounding rule is monomorphic.
- A modified rule dispatch mechanism that can explicitly inline callee rules.

Binding-time Analysis. We introduce a lightweight source-to-source transformation of DynSem specifications that analyzes rules and identifies structural dispatches by marking meta-variables whose binding depends solely on the object program structure. Consider the arithmetic addition rule of Figure 13a where meta-variables `e1` and `e2` are annotated with `const`. The meaning of the `const` annotation on a meta-variable is twofold: (1) the meta-variable is known to stem from the rule's input without dependence on evaluation context or rule calls, and (2) the meta-variable will be bound to a term that will be constant if the surrounding rule is monomorphic. The `const` annotations of the meta-variables that are

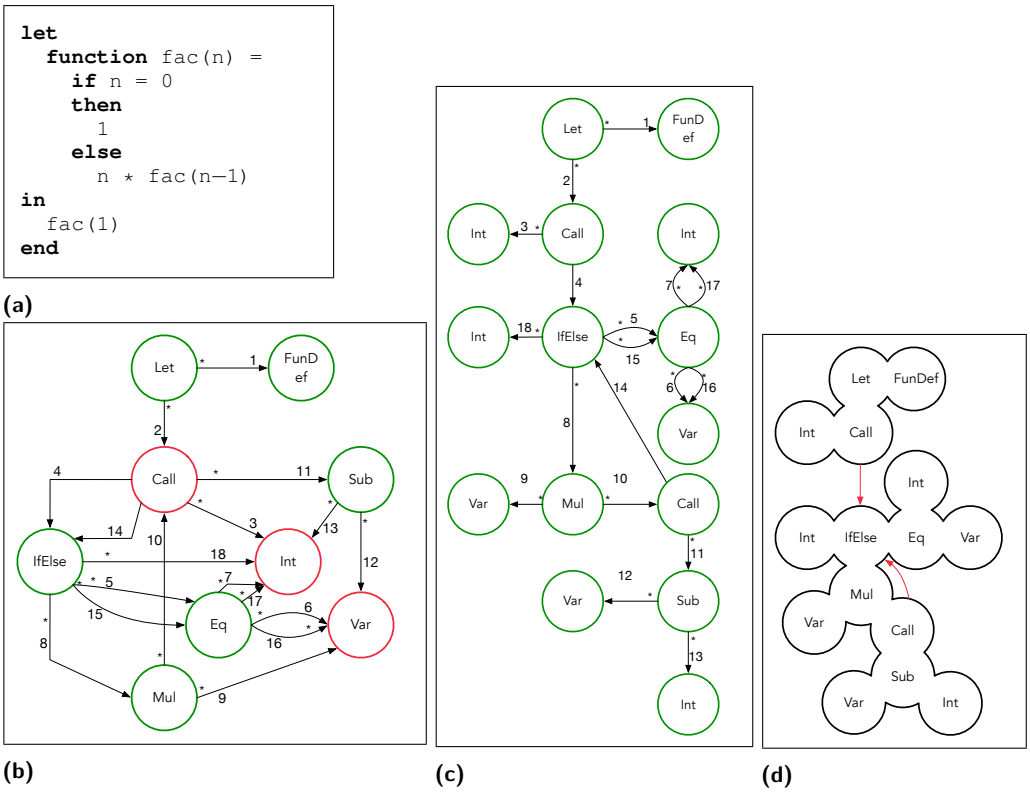


Figure 12 (a) Tiger program, (b) Rule call tree of program evaluation, (c) Rule call tree with cloned rules, (d) Rule call tree with rule inlining. Arrows marked with $*$ indicate calls on constant terms. Rules with green circles are monomorphic, those with red circles are polymorphic. Arrow numbers in figures (b) and (c) indicate execution order.

the inputs to the first two relation premises effectively mark the two rule calls as performing structural rule dispatch. It is the propagation of the `const` annotation to rule call premises that allows structural dispatch in Figure 12b to be identified and arrows labeled.

Consider the rule for a unary function call of Figure 13b. The meta-variable `e` bound to the parameter expression is `const` annotated. This identifies the evaluation of the parameter expression as requiring structural dispatch. At run time the evaluation of the parameter expression can be inlined if the surrounding rule is monomorphic. The function body `efun` retrieved from the closure is not `const` and its evaluation requires dynamic dispatch.

Fusion of Overloaded Rules. We call multiple DynSem rules that match the same pattern *overloaded* rules. Consider the six `eqV` rules of Figure 14a as an example of overloaded rules. The meta-interpreter loads overloaded rules as bundles. At rule call-time the rules in a bundle are executed one by one until the first applicable one is found and the call site caches the applicable rule at the call site. Subsequent executions of the call site first attempt the cached rules. In the event of a cache miss the remaining bundled rules are tried and the cache is grown with the newly applicable rule.

We observe that the success of a rule from the bundle is more likely to be determined by the state of the object program rather than by its structure. Consider for example a bundle of the two rules for an `if-then-else` statement. Indeed selecting one of the `if-then-else` rules depends on the result of evaluating its guard condition. By this reasoning we cannot estimate

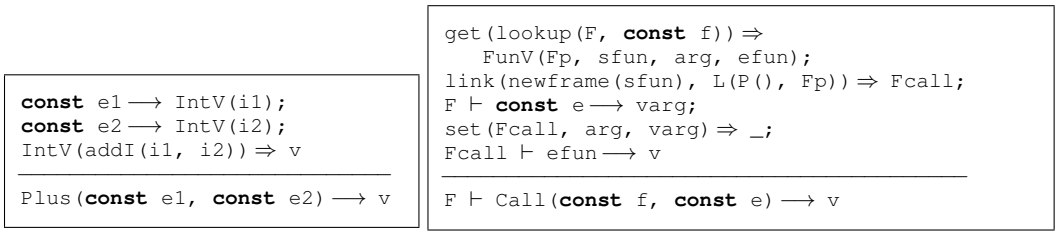


Figure 13 DynSem rules for (a) arithmetic addition and (b) unary function call with annotated meta-variables after binding-time analysis.

the risk of a cache miss locally; and the price to pay for a cache miss is the decompilation of the caller rule. The risk of a cache miss increases further if the call is a dynamic dispatch or the caller is polymorphic.

We propose that a better strategy is to not force the caller to select a successful rule, and instead to let the callee choose the applicable rule. We do this by introducing a rule node that combines rules of a bundle into a single executable node, as shown in Figure 14b. At rule load-time, the meta-variable environments of the fused rules are concatenated and a `FusedRule` node is created for each rule bundle. The execution method of a `FusedRule` iterates through the rules, returning the result of the first applicable rule. Since the number of rules in a fused bundle is fixed at run time, the JIT compiler can completely unroll the iteration, and additional profiling can be performed on the actual number of iterations required. In addition to mitigating the risk of decompilation due to a callee cache miss, fusing rules drastically simplifies call-site logic. In the remainder of this section we refer to a rule obtained by fusion generically as a rule.

Signaling Monomorphic Rules. A structural dispatch call site (a call site which reduces a term assigned to a `const`-annotated meta-variable) must be able to query whether the surrounding caller is monomorphic or polymorphic and use this information to decide which call site optimizations can be performed. In the terms of Figure 12b, this means that a star-labelled outgoing arrow should be able to observe whether its source rule is green or red, i.e. monomorphic or polymorphic. To achieve this we install a flag at the root of each rule, as shown in the left panel of Figure 15. The flag is visible to all nodes within a rule, thus also to the nodes that implement variable reading semantics and call sites. A rule starts off as monomorphic and remains so as long as it is always invoked on the same program term. A rule becomes polymorphic, and its flag is invalidated, if and when it is invoked on a different program term. This is the case for the `Call` rule of Figure 15 which is invoked both from the body of the `let` construct, and from within the function body. We implement flag invalidation at the rule level, as shown in the left panel of Figure 15.

In the figure we describe the flag as a boolean, but in reality we implement the signal using a Truffle Assumption. Graal ensures that checking whether Assumptions are valid from JIT-ed code is very cheap, so using an assumption as a cache guard, or as a specialization guard is very efficient. While guard checking with assumptions is very cheap, the cost of decompilation and recompilation is still high.

Inlining Rules. In the call tree of Figure 12b, although dispatches to `Call`, `Int` and `Var` are all structural, the rules themselves are polymorphic because their different callees pass different input terms. However, we know that since a program is fixed, even a polymorphic

```
eqV(IntV(i), IntV(j)) ⟶ eqI(i, j)

eqV(StringV(x), StringV(y)) ⟶ eqS(x, y)

eqV(NilV(), NilV()) ⟶ 1

eqV(NilV(), RecordV(_)) ⟶ 0

eqV(RecordV(_), NilV()) ⟶ 0

eqV(RecordV(F1), RecordV(F2)) ⟶ eqFrames(F1, F2)
```

(a)

```
class FusedRule extends Rule {
  final Rule[] rules;

  FusedRule(Rule[] rules) {
    this.rules = rules;
  }
  Result execute(VirtualFrame frame) {
    for (int i = 0; i < rules.length; i++) {
      try {
        return rules[i].execute(frame);
      } catch (RuleFailure e) {}
    }
    throw new ReductionFailure("No more rules to try");
  }
}
```

(b)

Figure 14 (a) Overloaded equality rules. (b) Sketch implementation of the fused rule node.

rule has a finite set of behaviors. This set of behaviors is bound in the set of program terms that match the rule's pattern. We can create a specialized copy of the rule for each program term in this set, thereby reducing a polymorphic rule to a set of monomorphic rules. The specialized copies can be inlined to replace structural dispatches within other monomorphic rules. Applying rule cloning to the call tree of Figure 12b results in the call tree of Figure 12c; all rules in the tree are monomorphic. The dynamic dispatches that remain are those that reduce computed terms, i.e. the two closure applications (arrows 4 and 14).

We modify the meta-interpreter to inline (at run time) callees into their call site if two conditions are met: (1) the caller is monomorphic; and (2) the dispatch is structural. The right panel of Figure 15 sketches the inlining mechanism. At call time, if the conditions hold, the `uninitclone()` method copies the callee in an uninitialized state (i.e., in its state prior to any invocation), and the copy is adopted into the caller, becoming a child node. For subsequent calls, the inlined callee is executed directly as long as the rule stays monomorphic. The inlined callee is discarded and replaced by dynamic dispatch if the rule becomes polymorphic. Dynamic dispatch will attempt to cache callees locally to avoid repeated lookups; Figure 15 omits caching details for conciseness. Note that a callee is inlined without its root node, which allows calls to `getRootNode()` from within the callee to resolve to the root node of the caller. This has the advantage of sharing a single `monomorphic` flag for all inlined rules within a tree.

If we apply the cloning and inlining mechanism to the call tree of Figure 12b, the JIT will compile a monomorphic caller together with its inlined callees in a single compilation unit, thereby eliminating dispatches between rules altogether. This results in the call tree of Figure 12d where the red arrows correspond to the only two dynamic dispatches that remain. Inlining of structural dispatches creates rules which do more work locally and perform

```
class RuleRoot extends RootNode {
  boolean monomorphic = true;
  Rule rule;

  Result execute(VirtualFrame f) {
    return rule.execute(f);
  }
}

class Rule extends Node {
  Pattern patt;
  Premise[] premises;
  TermBuild output;
  Term tInit;

  Result execute(VirtualFrame f) {
    Term t = getInputTerm(f);
    patt.match(t);
    if (tInit == null) {
      tInit = t;
    } else if (getRootNode().monomorphic
                && tInit != t) {
      getRootNode().monomorphic = false;
    }
    for (Premise p : premises) {
      p.execute(f);
    }
    return output.build(f);
  }
}
```

```
class Premise extends Node { ... }

class RelationPremise extends Premise{
  TermBuild input;
  Pattern output;
  Rule callee;

  void execute(VirtualFrame f) {
    Term t = input.build(f);
    Result res;
    if (getRootNode().monomorphic
        && input.isconst()) {
      if (callee == null) {
        callee = adopt(
          ruleRegistry().lookup(t)
            .rule.uninitclone()
        );
      }
      res = callee.execute(...);
    } else {
      callee = null;
      res = ruleRegistry().lookup(t)
              .execute(...);
    }
    output.match(res);
  }
}
```

Figure 15 Schematic implementation of rule calls with rule cloning.

fewer dynamic calls. In addition to reducing dynamic calls, this enables more intra-rule optimizations. Disadvantages of this method are longer compilation times due to larger compilation units and overhead during warmup due to rule cloning. Additionally, while larger compilation units enable better partial evaluation, this partial evaluation possibly takes longer, requiring more warmup rounds.

6 Evaluation

We evaluate our performance improvement techniques using DynSem specifications for Tiger, a simple programming language originally invented for teaching about compilers [2]. Tiger is a statically typed language with let bindings, functions, records and control-flow constructs. Our evaluation compares execution times across different flavors of Tiger implementations.

6.1 Experiment Set-up

Subjects. We evaluate four different implementations of Tiger: three meta-interpreted DynSem specifications and one hand-written Tiger interpreter. These are:

- `Meta-Env`: an environment-based DynSem specification interpreted on the runtime described in Section 3. This was the state-of-the-art DynSem runtime prior to the contributions of this paper.
- `Meta-SF`: a DynSem specification using Scopes & Frames as described in Section 4.3, interpreted on the runtime with native Scopes & Frames bindings of Section 4.4.
- `Meta-SF-Inline`: specification and runtime identical to `Meta-SF` with runtime rule inlining enabled.

- `Hand`: a Truffle-based AST interpreter using Scopes & Frames and implementing common Truffle optimization techniques (e.g. loop unrolling, polymorphic inline caches, branch profiles).

Workloads. We adapted the set of Tiger benchmark programs of Vergu et al. [35], which are translations of the Java programs of Marr et al. [19]. During earlier experimentation we discovered that benchmark runtime was too short on the faster meta-interpreters for a reliable time measurement. We addressed this by making the problems solved harder, resulting in the following six programs:

- `queens`: a solver for the 16-queens problem. The implementation uses let bindings, arrays, recursive function calls, for loops and nested conditional constructs.
- `list`: builds and traverses cons-nil lists. The program makes use of records, recursive function calls, while loops and conditionals.
- `towers`: a solver for the Towers of Hanoi game, primarily exercising records and recursive function calls.
- `sieve`: Sieve of Eratosthenes algorithm finding prime numbers smaller than 14,000. The program primarily exercises variable declarations, variable access in nested lexical scopes, and nested loops.
- `permute`: generates permutations of an array of size 8.
- `bubblesort`: performs bubble sort on a cons-nil list of 500 integers, initially in reverse order. The lists are built using records.

Methodology. We modified the four Tiger runtimes to repeat the evaluation of a program 200 times in the same process and to record the duration of each repetition. The time recorded is strictly program evaluation time, i.e. it excludes VM startup, program parsing, static analysis and interpreter instantiation. Each sequence of 200 in-process repetitions is repeated 30 times, as separate processes. We run the experiment on a Hewlett Packard ProLiant MicroServer Gen 8 with an Intel Xeon CPU E3-1265L V2 running at 2.5Ghz. The CPU has four cores; we disable one of the cores to ensure that heat dissipation is sufficient, and we disable hyper-threading to improve predictability. The machine has 16 GB of DDR3 memory, divided in two sockets, operating at a maximum frequency of 1.6Ghz, with ECC mode enabled. The operating system is a fresh minimal installation of Ubuntu Server `18.04.2` running a Linux kernel version `4.15.0-48`. All non-essential system daemons and networking are disabled before running the experiment, and we connect to the machine through out-of-band management facilities. All benchmark programs are run on the Oracle Graal Enterprise Edition VM version `1.0.0-rc9`.

We are interested in the steady state performance of each benchmark and VM combination. We use warmup_stats, part of the Krun [3] benchmarking system, to process and analyze the recorded timeseries. It performs statistical analyses to determine whether each combination of benchmark and VM shows stable performance and to compute this steady state performance.

6.2 Results

Table 1 shows the steady state runtimes, in seconds, for each configuration of benchmark and runtime. A missing measurement indicates that the configuration did not exhibit steady performance according to warmup_stats. We first consider the performance difference between traditional environment-based (`Meta-Env`) and scopes-and-frames (`Meta-SF`) specifications. For the remainder of this section, when we describe average speedup, we are referring to the geometric mean.

Table 1 Median steady state execution times, expressed in seconds, for combinations of benchmarks and VMs. The 99% confidence interval is shown in small font. Execution times for combinations which do not exhibit stable performance are excluded.

	Meta-Env	Meta-SF	Meta-SF-Inline	Hand
queens	1.7019 ±0.72583	0.0682 ±0.18626	0.0208 ±0.09366	0.0047 ±0.00085
list	0.2396 ±0.01789	0.0965 ±0.03700	0.0773 ±0.06191	
towers	9.5841 ±0.49535	0.6647 ±0.05259	0.0508 ±0.00460	0.0107 ±0.00030
sieve		0.0041 ±0.01925	0.0025 ±0.00196	0.0003 ±0.00053
permute	12.7514 ±1.91232	0.3216 ±0.02547	0.1108 ±0.00241	0.0260 ±0.00050
bubblesort	2.3551 ±0.34690	0.1164 ±0.01155	0.0147 ±0.00502	0.0060 ±0.02275

Table 2 Median number of repetitions required to reach steady state performance, and in small font the interquartile range. In parentheses (in normal font): the average duration, in seconds, of a warmup iteration.

	Meta-Env	Meta-SF	Meta-SF-Inline	Hand
queens	1	10.5 (1.0, 75.4) (1.78s)	51 (1.0, 77.1) (0.49s)	20 (18.5, 40.0) (0.25s)
list	98.5 (56.7, 121.5) (0.51s)	38.5 (25.0, 125.3) (0.49s)	81 (1.0, 106.5) (0.18s)	
towers	1	18 (18.0, 25.0) (2.49s)	89.5 (75.3, 119.5) (0.18s)	50.5 (42.4, 58.0) (0.09s)
sieve		106 (73.4, 146.6) (0.12s)	126 (5.5, 143.1) (0.04s)	9 (8.0, 17.6) (0.18s)
permute	1	68.5 (65.0, 84.5) (0.60s)	44 (40.0, 52.0) (0.28s)	30 (30.0, 43.5) (0.09s)
bubblesort	1	49 (31.4, 89.1) (1.42s)	67.5 (57.0, 85.5) (0.13s)	1

The `Meta-SF` interpreter improves on `Meta-Env` performance by an average 15x, with the highest gains for `permute` (39x) and smallest gains for `list` (2.5x). The runtimes on the two VMs are strongly correlated (correlation coefficient of 0.75), suggesting that adopting scopes and frames improves all benchmarks fairly uniformly. However, we also find a moderate correlation (correlation coefficient of 0.64) between the runtimes of `Meta-Env` and speedup gains exhibited by `Meta-SF`, suggesting that the longer the benchmark runtime on `Meta-Env`, the higher the speedup offered by `Meta-SF`. This may be due either to `Meta-SF` optimizing precisely the bottlenecks in `Meta-Env`, or simply to more complex programs benefiting more.

The `Meta-SF-Inline` VM improves on the performance of `Meta-SF` in 50% of the cases, while in the other 50% of the cases they are statistically indistinguishable. `Meta-SF-Inline` is always faster than `Meta-Env` by at least an order of magnitude and typically by two orders of magnitude, with the exception of `queens` for which it is at least 8.5x faster. There is strong correlation (0.79) between the runtime of benchmarks on `Meta-SF` and the speedup on `Meta-SF-Inline`. Coupled with only a moderate correlation (0.42) of runtimes on the two VMs, this suggests that, for the programs benchmarked, inlining addresses precisely the bottlenecks in `Meta-SF`. We do note the overlap in confidence intervals of runtime on `Meta-SF` and on `Meta-SF-Inline` for benchmarks `queens`, `sieve` and `list` which makes them statistically indistinguishable.

The handwritten interpreter `Hand` is on average 4.7x faster than `Meta-SF-Inline`, but not more than 30x faster. Some of these benchmarks have very short runtimes, but focusing on the two benchmarks with longest runtimes on `Hand`, `permute` and `towers`, produces a very similar overhead figure of 4.5x.

The number of iterations that are required until reaching steady state is an indication both of how JIT-able a benchmark/VM combination is and of how much particular optimizations compromise warmup time for maximum performance. Table 2 shows the median number of warmup iterations required until steady state is reached and the median duration of an iteration during warmup. With the exception of `list`, benchmarks on the environment-based VM do not seem to warm up well: they reach steady state performance in one iteration and never improve after that. It is noteworthy that `list`, the only benchmark that warms up on `Meta-Env`, is also the one least improved on by `Meta-SF`. In contrast to `Meta-Env`, the JIT compiler is able to optimize programs on the `Meta-SF` VM, but requires an average of 37 iterations to do so. We find a similar pattern for `Meta-SF-Inline`, typically requiring more warmup iterations than `Meta-SF` but resulting in faster code. We observe that even when the median warmup round on `Meta-SF-Inline` is slower than the steady-state performance on `Meta-SF`, it is within an order of magnitude slower, and that the average median warmup time on `Meta-SF-Inline` is shorter than on `Meta-SF`. From Table 1 we note that runtime confidence intervals are wider for the `Meta-SF` and `Meta-SF-Inline` VMs than they are for `Hand`; in particular for benchmark `queens` on `Meta-SF`, and for benchmarks `queens` and `list` on `Meta-SF-Inline`. The wide confidence intervals appear correlated with benchmark-VM combinations that have one or more non-warmup process executions (Table 2, combinations for which the 25th quantile is 1.0). This suggests some non-determinism over which we have little current understanding.

We find that replacing environments and stores by scopes and frames has a strictly beneficial effect on the execution time, and that meta-interpreters derived from scopes-and-frames specifications have better warm up characteristics. Adopting scopes and frames "out of the box" allows the JIT compiler to optimize the executing code. The JIT can see through memory operations and examine the memory layout of the program which enables partial evaluation of memory operations. Since our experiment does not measure the garbage collection activity, it is unclear to what degree the reported performance numbers are affected, positively or negatively, by garbage collection activity. We proposed in Section 5 that the fine granularity of code that the JIT is optimizing in the meta-interpreter case is a bottleneck in the optimizations that it can perform, and we introduced cloning and inlining of monomorphic rule calls at run time to attempt to improve on this situation. The expectation was that increasing the size of the rules, and thereby minimizing the number of calls across rules, would make the program easier to optimize. This expectation is borne out: in 50% of cases `Meta-SF-Inline` faster than `Meta-SF`, and in the other cases it is not slower. Inlining of rules increases the size of compilation units, aligns the structure of the rule call tree with the syntactic structure of the executing program, and the JIT can produce faster code.

Overall the combination of scopes and frames with inlining delivers a meta-interpreter that is always faster than using environments and stores. The speedup is at least one and typically two orders of magnitude. Moreover, the best meta-interpreter is within 10x slower (approximately 4.7x) than our optimized handwritten interpreter.

7 Discussion; Related and Future Work

The work presented in this paper is a performance improvement on the state of the art DynSem meta-interpreter. The improvement is achieved by (1) using scope and frames to model memory in dynamic semantics and (2) applying inline expansion of DynSem rules at run time.

Our work demonstrates a significant reduction in the execution time of meta-interpreted specifications of dynamic semantics using two techniques. The first exploits the systematic correspondence between static and run-time name binding exhibited by scopes and frames [28]. The second inlines reduction rules at run time to obtain coarser-grained rules that reflect the structure of the interpreted program. Combining these two techniques results in meta-interpreters that are at least one order of magnitude and generally two orders of magnitude faster than the state of the art DynSem meta-interpreter; and within a factor 5 from an optimized handwritten interpreter.

We remark that optimizations made to frame operations are in fact optimizations made to the executing program, not to the meta-interpreter. A resolved scope graph and paths in the scope graph representing the results of name resolution are program specific. Using the scope graph to inform optimizations of frame operations results in optimizations that are program specific. The JIT of the hosting VM, which hosts the meta-interpreter, is thus traversing the meta-interpreter layer to operate on the top-level interpreter. In the end, the program-specific optimizations performed by the JIT unlock further meta-interpreter optimizations than those limited to syntax-driven optimizations. Another indication that this is happening is, aside from the increased performance, the number of iterations required for code to warm up.

Related Work. DynSem [34], as a dynamic semantics framework, is part of the family of structural operational semantics (SOS) frameworks. This family contains big-step SOS (or natural semantics [16]); small-step SOS as originally introduced by Plotkin [27]; and reduction semantics with evaluation contexts (e.g. [11]), of which PLT Redex [10] is an instantiation. MSOS [22] and its extension I-MSOS [23] improve on the modularity and conciseness of traditional SOS by allowing *semantic components* such as environments and stores to be propagated implicitly through rules that do not modify those components. DynSem borrows the notion of implicit semantic propagation from I-MSOS and implements a systematic transformation of specifications with implicit components into equivalent specifications with explicit components. Typical DynSem specifications are in big-step style with implicit propagation of semantic components.

Dynamic semantics specifications take one of two approaches to specifying name binding: (a) eagerly substituting values for names or (b) propagating semantic components such as environments or stores that associate values with names. Specifications in Redex [18] and Ott [31] typically use substitution, while specifications in K [30] and funcons [7] typically use semantic components. Prior to the developments presented in this paper, DynSem specifications modeled name binding using semantic components that map identifiers to addresses and addresses to values and embedded name resolution semantics in terms of operations on these components. The DynSem extensions of Section 4 use scope graph [25] information to automatically derive a memory layout in terms of frames [28] and provide a set of primitives for operating on memory. The approach replaces environments, stores and other custom semantic components with a generic representation of memory stored in an implicitly propagated store. The only components passed in rules are frame references into the store.

Given a dynamic semantics for an object language there are three conceptual approaches to obtaining a execution engine for that language: (1) compile the semantics to an interpreter, (2) compile the semantics to a compiler or (3) interpret the semantics. DynSem, Redex [18] fall into the final category, i.e. a runtime is obtained by (meta-)interpreting a semantics. An older runtime for K [30] generated an interpreter for object language, but more recently K

specifications can be directly interpreted. Significant amounts of research have gone into generating compilers from semantics [21, 26, 8] with varying degrees of applicability and usually with slow compilation or slow execution or both. For example, the SIS compiler generator of Mosses [21] compiled denotational semantics to a code generator, demonstrating that it was possible to compile code generators from declarative specifications. However, both the generated compiler and its emitted code were quite slow.

Translating a dynamic semantics specification to an efficient (and optimizing) compiler requires some form of offline partial evaluation [15]. The three approaches to make semantics specification executable are conceptually related to partial evaluation [15] and the Futamura projections [12, 13]. The first Futamura projection of a meta-interpreter and a semantics specification yields an interpreter, and the first Futamura projection of that interpreter and a program yields an executable. The second Futamura projection of a meta-interpreter and a semantics yields a compiler derived from the semantics. Amin et al. [1] describe the construction of a one-pass compiler that collapses all interpreter layers in a hierarchy-of-layers, thus eliminating the overhead of stacked interpretation.

Our approach to make DynSem specifications executable is through meta-interpretation with minimal pre-compilation. This raises the challenge of eliminating the overhead of meta-interpretation. The problem is more complicated than just optimizing an interpreter at runtime (as is done in just-in-time (JIT) compilation), because both the hosting and the hosted interpreters must be optimized simultaneously. The hosting meta-interpreter cannot effectively be partially evaluated without the hosted object interpreter, whose optimization in turn requires the program input.

There are two mainstream directions for implementing efficient interpreters, both relying on JIT compilation: meta-tracing and online partial evaluation. Meta-tracing, as provided by RPython [4] and applied to PyPy [5, 6] traces the execution of an interpreter to obtain a JIT compiler specific to that interpreter. The obtained JIT monitors the execution of the interpreter and compiles frequently executed code (of the interpreter) into highly efficient machine code. Only recently has online partial evaluation been shown as a practical meta-compilation technique of AST interpreters. Würthinger et al. [40] have developed Truffle, a framework for implementing interpreters. Truffle interpreters are AST interpreters, i.e. the control-flow of the interpreter follows the syntactic structure of the executing program. The Graal partial evaluator [39, 38] determines compilation units by resolving control-flow jumps across parts of the AST. For a practical comparison and evaluation of both meta-tracing and online partial evaluation of interpreters, we refer the reader to the research of Marr et al. [20].

To the best of our knowledge, neither meta-tracing nor online partial evaluation have been applied to two stacked layers of interpretation. Conceptually, meta-interpretation of a program with respect to a semantics specification involves a syntax-directed sequence of rule applications. A fixed program informs a fixed arrangement of rule applications, i.e. the rules of a specification are arranged such that they follow the AST of the program. This observation has motivated the choice of Truffle as an implementation target for the DynSem meta-interpreter. Conceptually, the Graal JIT has sufficient information to construct a tree of rules that strictly mimics the program AST. Construction of such a tree requires inlining of structural dispatch to rules, as discussed in Section 5. The inlining introduced in Section 5 is designed to aid the JIT in identifying control-flow jumps in the hosting meta-interpreter that are known to be stable but that the JIT cannot observe as such due to the intermediate interpreter layer.

Future Work. In the future we plan to investigate using Graal to perform optimizations with respect to program values. To some limited extent this is happening already: checks on value terms from within DynSem rules are observable by the JIT, and frame slot allocation takes into consideration the type of the declaration. There also still are opportunities for optimization with respect to rule inlining. Currently not all static bindings in rules are recognized as monomorphic. For example, while for a particular object language a function call is known to always resolve to a specific closure, the DynSem static analysis cannot currently determine this. While we can allow the language developer to explicitly annotate `const` meta-variables, we believe a better solution would be to uncover more static bindings automatically. We expect that combining a program, its scope graph, and a DynSem specification provides sufficient information to determine this. The scopes-and-frames approach may also apply to dynamic languages. We plan to investigate if by building frame structures dynamically and caching results of run-time name resolution we can obtain similar performance gains. Yet another research avenue is to explore whether using DynSem to define intrinsically-typed interpreters [29] for object languages provides further benefits for specialization.

References

1 Nada Amin and Tiark Rompf. Collapsing towers of interpreters. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018. doi:10.1145/3158140.

2 Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

3 Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), 2017. doi:10.1145/3133876.

4 Carl Friedrich Bolz. *Meta-Tracing Just-in-Time Compilation for RPython*. PhD thesis, Heinrich Heine University Düsseldorf, 2014. URL: http://d-nb.info/1057957054.

5 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In Ian Rogers, editor, *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS 2009, Genova, Italy, July 6, 2009*, pages 18–25. ACM, 2009. doi:10.1145/1565824.1565827.

6 Carl Friedrich Bolz and Laurence Tratt. The impact of meta-tracing on VM design and implementation. *Science of Computer Programming*, 98:408–421, 2015. doi:10.1016/j.scico.2013.02.001.

7 Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. Reusable Components of Semantic Specifications. *Transactions on Aspect-Oriented Software Development*, 12:132–179, 2015. doi:10.1007/978-3-662-46734-3_4.

8 Olivier Danvy and René Vestergaard. Semantics-Based Compiling: A Case Study in Type-Directed Partial Evaluation. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs, 8th International Symposium, PLILP 96, Aachen, Germany, September 24-27, 1996, Proceedings*, volume 1140 of *Lecture Notes in Computer Science*, pages 182–197. Springer, 1996.

9 Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015. doi:10.1016/j.cl.2015.08.007.

10 Matthias Felleisen, Robby Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

11 Matthias Felleisen and Robert Hieb. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 103(2):235–271, 1992.

12 Yoshihiko Futamura. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. URL: `http://www.springerlink.com/content/l46w6q3720n57607/`.

13 Yoshihiko Futamura. Partial Evaluation of Computation Process, Revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.

14 Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A domain-specific language for building self-optimizing AST interpreters. In Ulrik Pagh Schultz and Matthew Flatt, editors, *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014*, pages 123–132. ACM, 2014. `doi:10.1145/2658761.2658776`.

15 Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993.

16 Gilles Kahn. Natural Semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

17 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM. `doi:10.1145/1869459.1869497`.

18 Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robby Findler. Run your research: on the effectiveness of lightweight mechanization. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 285–296. ACM, 2012. `doi:10.1145/2103656.2103691`.

19 Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-language compiler benchmarking: are we fast yet? In Roberto Ierusalimschy, editor, *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*, pages 120–131. ACM, 2016. `doi:10.1145/2989225.2989232`.

20 Stefan Marr and Stéphane Ducasse. Tracing vs. partial evaluation: comparing meta-compilation approaches for self-optimizing interpreters. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 821–839. ACM, 2015. `doi:10.1145/2814270.2814275`.

21 Peter D. Mosses. Compiler Generation Using Denotational Semantics. In Antoni W. Mazurkiewicz, editor, *Mathematical Foundations of Computer Science 1976, 5th Symposium, Gdansk, Poland, September 6-10, 1976, Proceedings*, volume 45 of *Lecture Notes in Computer Science*, pages 436–441. Springer, 1976.

22 Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004. `doi:10.1016/j.jlap.2004.03.008`.

23 Peter D. Mosses and Mark J. New. Implicit Propagation in Structural Operational Semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49–66, 2009. `doi:10.1016/j.entcs.2009.07.073`.

24 Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A Theory of Name Resolution with extended Coverage and Proofs. Technical Report TUD-SERG-2015-001,

Software Engineering Research Group. Delft University of Technology, January 2015. Extended version of ESOP 2015 paper "A Theory of Name Resolution".

25 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A Theory of Name Resolution. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. doi:10.1007/978-3-662-46669-8_9.

26 Lawrence C. Paulson. A Semantics-Directed Compiler Generator. In *POPL*, pages 224–233, 1982.

27 Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

28 Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.20.

29 Casper Bach Poulsen, Arjen Rouvoet, Andrew P. Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018. doi:10.1145/3158104.

30 Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. doi:10.1016/j.jlap.2010.03.012.

31 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010. doi:10.1017/S0956796809990293.

32 Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In Martin Erwig and Tiark Rompf, editors, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 49–60. ACM, 2016. doi:10.1145/2847538.2847543.

33 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 2018. doi:10.1145/3276484.

34 Vlad A. Vergu, Pierre Néron, and Eelco Visser. DynSem: A DSL for Dynamic Semantics Specification. In Maribel Fernández, editor, *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*, volume 36 of *LIPIcs*, pages 365–378. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.RTA.2015.365.

35 Vlad A. Vergu and Eelco Visser. Specializing a meta-interpreter: JIT compilation of Dynsem specifications on the Graal VM. In Eli Tilevich and Hanspeter Mössenböck, editors, *Proceedings of the 15th International Conference on Managed Languages & Runtimes, ManLang 2018, Linz, Austria, September 12-14, 2018*. ACM, 2018. doi:10.1145/3237009.3237018.

36 Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, pages 95–111. ACM, 2014. doi:10.1145/2661136.2661149.

37 Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An object storage model for the truffle language implementation framework. In Joanna Kolodziej and Bruce R. Childers, editors, *2014 International Conference*

on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014, pages 133–144. ACM, 2014. doi:10.1145/2647508.2647517.

38 Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In Albert Cohen 0001 and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 662–676. ACM, 2017. doi:10.1145/3062341.3062381.

39 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 187–204. ACM, 2013. doi:10.1145/2509578.2509581.

40 Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In Alessandro Warth, editor, *Proceedings of the 8th Symposium on Dynamic Languages, DLS '12, Tucson, AZ, USA, October 22, 2012*, pages 73–82. ACM, 2012. doi:10.1145/2384577.2384587.