Specializing a Meta-Interpreter

JIT Compilation of DynSem Specifications on the Graal VM

Vlad Vergu
TU Delft
The Netherlands
v.a.vergu@tudelft.nl

Eelco Visser
TU Delft
The Netherlands
visser@acm.org

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. DynSem specifications can be executed to interpret programs in the language under development. To enable fast turnaround during language development, we have developed a meta-interpreter for DynSem specifications, which requires minimal processing of the specification. In addition to fast development time, we also aim to achieve fast run times for interpreted programs.

In this paper we present the design of a meta-interpreter for DynSem and report on experiments with JIT compiling the application of the meta-interpreter on the Graal VM. By interpreting specifications directly, we have minimal compilation overhead. By specializing pattern matches, maintaining call-site dispatch chains and using native control-flow constructs we gain significant run-time performance. We evaluate the performance of the meta-interpreter when applied to the Tiger language specification running a set of common benchmark programs. Specialization enables the Graal VM to JIT compile the meta-interpreter giving speedups of up to factor 15 over running on the standard Oracle Java VM.

CCS CONCEPTS

• **Software and its engineering → Interpreters**; **Domain specific languages**; *Semantics*;

KEYWORDS

dynamic semantics, interpretation, JIT, run-time optimization

ACM Reference Format:
Vlad Vergu and Eelco Visser. 2018. Specializing a Meta-Interpreter: JIT Compilation of DynSem Specifications on the Graal VM. In *15th International Conference on Managed Languages & Runtimes (ManLang'18), September 12–14, 2018, Linz, Austria.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3237009.3237018

1 INTRODUCTION

The dynamic semantics of a programming language defines the run time execution behavior of programs in the language. Ideally, the design of a programming language starts with the specification of its dynamic semantics to provide a high-level readable and unambiguous definition. However, understanding the design of a programming language also requires experimentation by actually running programs. Therefore, this ideal route is rarely taken, but language designs are embodied in the implementation of interpreters or compilers instead.

We have previously designed DynSem [33], a high-level meta-DSL for dynamic semantics specifications of programming languages, with the aim of supporting readable *and* executable specification. It supports the definition of modular and concise semantics by means of reduction rules with implicit propagation of contextual information. DynSem's executable semantics entails that specifications can be used to interpret object language programs.

In our early prototypes, DynSem specifications were compiled to an interpreter. The process of generating a Java implementation of an interpreter and compiling that generated code caused long turnaround times during language prototyping. In order to support rapid prototyping with short turnaround times, we turned to interpreting specifications directly instead of compiling them. A DynSem interpreter is a *meta-interpreter* since the programs it interprets are themselves interpreters. Figure 1 depicts the high-level architecture of the DynSem meta-interpreter. First, a DynSem specification is desugared (explicated) to make implicit passing of semantic components explicit. The resulting specification in DynSem Core is then loaded into the meta-interpreter together with the AST of the interpreted object program. The interpreter consumes the program as input enacting the specification. This produces the desired result of a short turnaround time for experimenting with dynamic semantics specifications.

Meta-interpretation reduces the turnaround time at the expense of execution performance. At run time there are two interpreter layers operating (the meta-language interpreter and the object-language interpreter) which introduces substantial overhead. While we envision DynSem as a convenient way to prototype the dynamic semantics of programming languages, ultimately we also envision it as a convenient way to bridge the gap between the prototyping and production phases of a programming language's lifecycle. Thus, we not only want an interpreter fast, but we also want a fast interpreter, which raises the question: Can we achieve fast object-language interpreters by optimizing the meta-interpretation of dynamic semantics specifications?

Direct vanilla interpreters are in general slow to begin with, even when they are implemented in a host language that is JIT-ed. This is because the host JIT is unable to see patterns in the object language and to meaningfully optimize the interpreter. The task of optimizing an interpreter has traditionally been long and

ManLang'18, September 12–14, 2018, Linz, Austria

ACM ISBN 978-1-4503-6424-9/18/09…$15.00
https://doi.org/10.1145/3237009.3237018

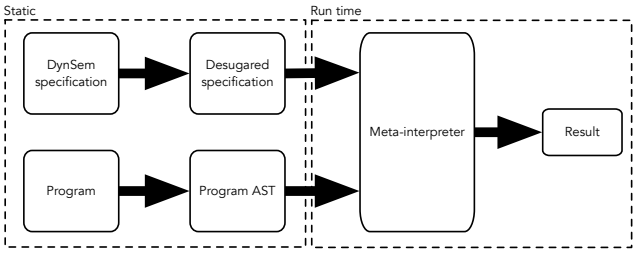

Figure 1: Meta-interpretation overview

difficult and required specialized knowledge. Recent advances in JIT technology, such as meta-tracing [7] and meta-compilation [36] strive to make it tractable for non-JIT-experts to define interpreters that are as fast as native code. JIT-ing a meta-interpreter requires dynamic compilation of the two interpreter layers concomitantly, complicating the problem.

In this paper, we present the design of a meta-interpreter for DynSem specifications and report on experiments with JIT-compiling the application of the meta-interpreter on the Graal Virtual Machine [36]. We begin with a pure meta-interpreter (Section 3) and evolve it to a hybrid interpretation approach (Section 4) which combines generated components with interpreted specifications. We tackle the problem of non-linear control-flow in the specification by designing a dispatch mechanism (Section 5) that allows the JIT to restructure the specification by inlining it so that the only remaining dispatch is that of the interpreted program. We then open the door to a hybrid between meta-circular and definitional interpreters without sacrificing the clarity of the specification. This allows us to distinguish looping control-flow at the meta-level from loops at the program level (Section 6). Concretely, the contributions of this paper are:

- A hybrid meta-interpretation technique which combines generational and interpreted components leading to obtain increased performance without sacrificing turnaround time during language prototyping.
- The design of a dispatch mechanism that combines dispatch caching and dispatch reorganization to allow control-flow to stabilize even in the face of local backtracking.
- The combination of definitional interpreters with meta-circular components for looping control-flow to maximize the amount of inlining that the JIT can provide.

We evaluate our meta-interpreters (Section 7) by applying them to a specification of the Tiger [3] programming language. When run on the Graal VM the meta-interpreter shows improved performance by factor 6 to 15, depending on the workload.

2 BACKGROUND

Our aim is to design a meta-interpreter for DynSem [33] specifications of dynamic semantics so that they can be just-in-time compiled on the Oracle Graal VM.

We briefly introduce DynSem and the specification of our case study language Tiger. We then give an overview of developing interpreters using Truffle and Graal.

2.1 Semantics Specifications with DynSem

The dynamic semantics of a programming language defines the run-time behavior of its constructs. DynSem is a meta-DSL for specifying the dynamic semantics of programming languages, included in the Spoofax Language Workbench [18]. DynSem is part of the effort to create programming environments from high-level specifications [34]. Specifications are given in terms of syntax-oriented rules over named arrows from program terms to values, in a big-step style [17]. Rules can access contextual evaluation information from read-only components (mentioned left of the ⊢ symbol) and from read-write components. Read-only components propagate downwards (environment semantics), read-write components thread through the rules (store semantics).

We use DynSem to specify the dynamic semantics of the Tiger [3] programming language. Tiger is a statically typed programming language with let bindings, functions, records and control-flow constructs. It was introduced by Andrew Appel in his book Modern Compiler Implementation [3]. We use Tiger as the running case study for the remainder of this paper. The specification consists of signature declarations and reduction rules.

Signatures. Figure 2a shows a fragment of Tiger signatures. It declares two sorts of terms: `Exp` for program expressions, and `V` to be used for value terms. A constructor term named `Plus` has two children, both expressions. Itself the `Plus` constructor has type *Plus*/2, where 2 is its arity, and is of sort *Exp*. Figure 2a also declares an arrow named "" (the empty string) that reduces expression terms (of sort *Exp*) to value terms (of sort *V*). DynSem specifications are statically checked with respect to declared signatures and construction or matching of morphologically incorrect terms are rejected.

Rules. Rules relate program terms to value terms. A rule has a conclusion and arbitrarily many premises; they should be read like standard derivation rules with the exception that premises are ordered. The rule of Figure 2b gives the semantics of arithmetic addition in Tiger. The conclusion consists of three parts: the input pattern match, the arrow, and the construction of the output term. The input pattern `Plus(e1, e2)` serves two purposes. Firstly, it ensures that the input term is of type *Plus*/2 and it binds meta-variables `e1` and `e2` to its two sub-expressions. Secondly, the input pattern together with the arrow, declare that the rule reduces terms of type *Plus*/2 and that it populates the arrow declaration `Exp` ⟶ `V`.

Each of the two premises evaluates a sub-expression over rules of the `Exp` ⟶ `V` arrow and enforces the requirement that the result of evaluation is a term of type *IntV*/1, binding the sub-term to a meta-variable. The enforcement is by means of applying the pattern matches `IntV(i)` and `IntV(j)` to the result of evaluating `e1` and `e2`, respectively. The rule constructs and emits (returns) a term of type *IntV*/1. The term `addI(i, j)` is an invocation of primitive operator `addI` which performs the arithmetic addition. The rule propagates the semantic components `E` and `H` to the evaluation of the sub-expressions. Semantic component `E` (representing a variable environment) is passed as a read-only semantic component, while `H` (representing a store) is threaded through the computation and returned from the rule.

The order of evaluation in a rule is as follows: first the input pattern match is applied, then premises are evaluated in the order of declaration, finally the output term is constructed and returned.

Rules can be overloaded. If Tiger, for example, overloaded the + operator to also allow string concatenation, then the specification could contain a second rule for the type *Plus*/2 overloading the arithmetic addition one.

A rule only has to explicitly mention those semantic components which it modifies, other components can be left implicit. Since the rule of Figure 2b does not modify either semantic component it may leave both implicit and it can be written as in Figure 2c. A static analysis infers what semantic components must be propagated and informs a source-to-source transformation that explicates the rule of Figure 2c into that of Figure 2b.

The arithmetic addition rule can be written more concisely by making the reductions of `e1` and `e2` implicit, as the rule of Figure 2d does. A static analysis determines that sub-terms of sort *Exp* can only conform to the type *IntV*/1 by reduction from *Exp* to *V* over arrow `Exp ⟶ V`. The analysis informs a source-to-source transformation that lifts the implicit reductions to explicit premises.

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 Tiger function call of Figure 2e is a good example of meta-function use in DynSem. Meta-function `readVar` implements semantics for environment and store access; the rule for function calls uses it to retrieve the closure to be applied. DynSem does not (yet) have higher-order rules, so the meta-function `evalArgs` implements a zip- and fold-like transformation that evaluates arguments and bind them in an environment. The meta-functions `lookup`, `read`, `write` and `allocate` of Figure 2e are reusable semantic units for environment and store operations.

2.2 Truffle Interpreters on the Graal VM

We briefly and informally discuss the structure of Truffle-based interpreters and how they can be optimized when they are run on a Graal VM. For a definitive guide we refer the reader to the Truffle and Graal literature [15, 35–37].

Truffle interpreters are AST interpreters implemented in Java. An AST interpreter organizes interpretation logic (semantics of the language) in tree-like data structures whose nodes correspond to syntactic constructs of the interpreted language. Execution in an AST interpreter flows downwards in the tree and results flow upwards. A language with a binary arithmetic addition, for example, will have an interpreter node called `Plus` with two executable children and `execute` method. The `execute` method invokes evaluation of the two children, performs the addition and returns the result. Each `execute` method is parameterized with a `frame` which is propagated downwards with the evaluation. The frame is used to store bindings for local variables of the program.

To avoid tree traversals when resolving function calls, the AST of the program is broken down into smaller trees that usually correspond to function definitions in the interpreted program. Each tree has a root node which has no parent. Each root node defines a call target. A call target is an evaluation entry point which does not receive a frame but instead creates a frame when it is invoked and passes it to its root node. A function call in a program consists of looking up and invoking the intended call target.

The Truffle and Graal method consists of rewriting the interpreter AST at run time so that its nodes are specialized to handling

```
signature
  sorts Exp V
  constructors
    Plus: Exp * Exp → Exp
    Call: Id * List(Exp) → Exp
    IntV: Int → V
  arrows
    Exp ⟶ V
```

(a)

```
E ⊢ e1 :: H1 ⟶ IntV(i) :: H2;
E ⊢ e2 :: H2 ⟶ IntV(j) :: H3
-----------------------------------------------
E ⊢ Plus(e1, e2) :: H1 ⟶ IntV(addI(i, j)) :: H3
```

(b)

```
e1 ⟶ IntV(i);
e2 ⟶ IntV(j)
-------------------------------
Plus(e1, e2) ⟶ IntV(addI(i, j))
```

(c)

```
Plus(IntV(i), IntV(j)) ⟶ IntV(addI(i, j))
```

(d)

```
readVar(f) ⟶ ClosureV(params, e, E);
evalArgs(params, args) ⟶ E';
E {E', E} ⊢ e ⟶ v
-----------------------------------
Call(f, args) ⟶ v

evalArgs([],[]) ⟶ {}

allocate(v) ⟶ a; evalArgs(args, es) ⟶ E
-------------------------------------------------------
evalArgs([FArg(x : Id, _)|args], [v|es]) ⟶ {x ↦ a,E}

readVar(x) ⟶ read(lookup(x))

E ⊢ lookup(x) ⟶ E[x]

read(a) :: H ⟶ H[a]

write(a, v) :: H ⟶ v :: H {a ↦ v,H}

fresh⇒ a; write(a, v) ⟶ _
--------------------------
allocate(v) ⟶ a
```

(e)

Figure 2: (a) Signature declaration for Tiger program and value terms (fragment). Reduction rules for arithmetic addition in Tiger in three flavors: (b) fully explicit (c) implicit semantic components (d) implicit semantic components and reductions. (e) Reduction rule for the function call in Tiger, and meta-functions for variable lookup and store access.

the types of runtime values that they observe, giving up generic functionality. For example, a binary addition/concatenation node which has only observed numeric operands and never strings can be replaced by a node dedicated to numeric addition. This dedicated node is guarded by a check that the operands are numbers. If the check fails the node is replaced by a polymorphic addition node. Code under specialization guards is treated as fast-path code. Code to be executed in the event of guard failure is slow-path code. Truffle provides a DSL [15] consisting of class and method annotations that reduce the programming burden of implementing specializations and polymorphic inline caches. Programmers annotate specialization methods with the `@Specialization` annotation and a compiler generates the boilerplate.

Truffle expects that interpreter trees stabilize at run time. The Graal JIT has special knowledge of Truffle interpreter nodes and it can inline their execution methods once the tree is stable. Inlining eliminates dispatch, Graal eliminates the local variable frame (replacing it with local variables) and fast-path code is compiled to machine code.

The interpreter must exhibit stable function dispatch to allow further optimizations. Caching call targets into their call-site interpreter nodes achieves this. Graal treats a cached call target as a constant, clones its corresponding root interpreter node (in unspecialized state) and inlines it into the call site. There are two benefits to doing this. Firstly, the overhead of the lookup and dynamic dispatch is eliminated. Secondly, the inlined call target code can now specialize to values specific to its former call site.

The success of applying Truffle and Graal to DynSem specifications depends on being able to specialize the specification interpreter first to the program structure and then to the runtime values of the running program.

3 A PURE TRUFFLE META-INTERPRETER

In this section, we describe the architecture of a *pure* meta-interpreter for DynSem that satisifies the requirement of a fast turnaround time during language development by interpreting (instead of compiling) semantics specifications. The architecture is illustrated by the diagram in Figure 1. The meta-interpreter loads an explicated DynSem specification, parses an object program to obtain an abstract syntax tree, and then evaluates the object program by reducing its abstract syntax tree according to the rules of the specification.

We implement the meta-interpreter in Java as a Truffle AST interpreter. In a meta-interpretation approach the AST nodes that are interpreted represent the specification AST, rather than the object program AST. The interpreter executes the specification consuming the program terms as input. This is in contrast to a direct interpretation approach where interpreter nodes are derived from the program AST and their execution directly evaluates the program.

3.1 Rule Registry

At startup the interpreter builds a rule registry, analogous to a function table in other programming languages. The registry maintains a mapping from specialized arrows to bundles of rules. A generic arrow name (x, S), together with the constructors that populate sort S, derive specialized arrows (x, ty) for each constructor of sort S with type ty[1]. The left-hand side pattern of a rule together with its arrow name indicate the specialized arrow that it populates.

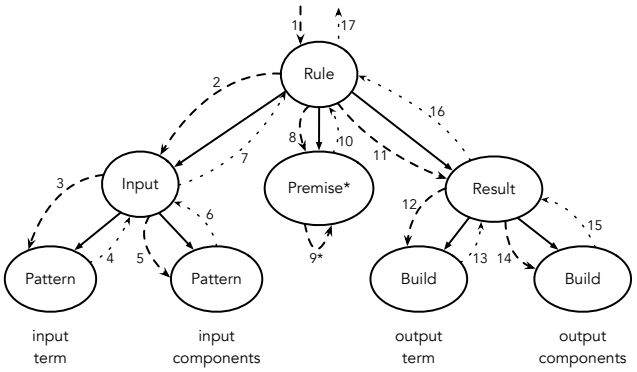

Figure 3: Structure of the reduction rule interpreter node. Dashed lines and their labels indicate control flow during interpretation.

Overloaded rules will populate the same specialized arrow. For example, the `Plus` rules of Figure 2 all populate the specialized arrow $(\text{"+"}, Plus/2)$. The rule registry bundles together rules that populate the same specialized arrow.

The type of terms and patterns is encoded as follows. The type of a constructor term $C(k^*)$ of sort S is $C/|k^*|$. The type of the type of $C(k^*)$ is S, the sort of $C/|k^*|$. A list term has type $List(S)$ where S is the sort of the elements contained. Finally, a tuple term has type $Tuple(S^*)$ where S^* is a list of the element sorts in the makeup of the tuple. A type together with an arrow name identify a set of rules.

3.2 Rule Execution

The meta-interpreter creates an interpreter AST for each rule of the specification. A rule node is an executable root node, i.e. it does not have a parent. Rule nodes conform to the structure of Figure 3. Each rule has an *inputs node*, any number of *premise nodes* and a *result node*.

A rule maintains a frame descriptor of the meta-variables declared in the rule. The frame descriptor informs the instantiation of a new rule frame for each invocation of the rule. The execution methods of all nodes in the interpreter are parameterized with the rule frame so that they can read and write meta-variables.

On invocation, a rule receives an array of the input program term and the semantic components to be propagated. Evaluation proceeds as indicated by the dashed arrows of Figure 3. The rule evaluates the *input node* which pattern matches the input term and binds meta-variables to sub-terms. The rule then evaluates each of the *premise nodes* in order, updating the bindings in the rule frame. The rule wraps up by evaluating the *result node* which extracts the computed result and the values of the read-write semantic components from the frame and returning.

[1] In the algebraic terminology of DynSem, the only types are 'sorts' (such as Exp). Here we convert to object-oriented terminology, also turning the constructors of a sort into types. Rule bundles define the implementation of the arrow (execution method) for a particular type.

```
public abstract class ConBuild extends TermBuild {
  private final String name;
  private final String sort;
  @Children private final TermBuild[] children;

  public ConBuild(SourceSection source, String name,
             String sort, TermBuild[] children) {
    super(source);
    this.sort = sort;
    this.name = name;
    this.children = children;
  }

  @Specialization
  public ApplTerm doBuild(VirtualFrame frame) {
    Object[] args = new Object[children.length];
    for (int i = 0; i < children.length; i++) {
      args[i] = children[i].executeGeneric(frame);
    }
    return new ApplTerm(sort, name, args);
  }
}
```

Figure 4: Interpreter node for building constructor terms.

3.3 Term Building and Pattern Matching

A rule is an orchestration of the basic DynSem operations of term construction and pattern matching. We implement both term construction and pattern matching as executable meta-interpreter nodes.

Term builders. A term building node is a either a literal term construction node or a meta-variable reading node. Term building nodes are created at run time from term building operations in the specification. The building of `IntV(42)`, for example, is interpreted by a constructor build node that instantiates a constructor term with the sub-term `42` and associates sort V with the newly built term. Figure 4 shows the node which implements constructor term building. When executed the node builds an array of sub-terms and instantiates a new constructor term. Construction of list and tuple terms is similar.

Pattern matchers. A pattern match node is a conditional decomposition of a term into its sub-terms which optionally binds meta-variables in the process or simply a meta-variable write node. The match pattern `Plus(e1, e2)`, for example, checks that a term is a constructor term of type *Plus*/2 and executes pattern matches `e1` and `e2` on its sub-terms. In this case both `e1` and `e2` are interpreter nodes which bind a meta-variable in the rule frame. The node implementing constructor matching is shown in Figure 5.

Patterns can be arbitrarily nested. The pattern `[a, IntV(42) | _]`, for example, checks that a term is a list with at least two elements, checks that the second element matches the pattern `IntV(42)` and binds the meta-variable a to the head of the list. Pattern match failure is an intrinsic control-flow mechanism in DynSem. Note that the node of Figure 5 raises an exception (`PremiseFailureException`) upon failure. This exception is caught only by the call-site of the rule that owns the failed pattern match. The exception is thrown to quickly abort computation of the rule and return control to the caller.

```
public abstract class ConMatch extends MatchPattern {
  private final String name;
  @Children private final MatchPattern[] children;

  public ConMatch(SourceSection source,
             String name, MatchPattern[] children) {
    super(source);
    this.name = name;
    this.children = children;
  }

  @Specialization @ExplodeLoop
  public void doCon(VirtualFrame frame, ApplTerm con) {
    if (!stringEq(name, con.name()) ||
                   children.length != con.size()) {
      throw PremiseFailureException.SINGLETON;
    }
    Object[] subterms = con.subterms();
    for(int i = 0; i < children.length; i++) {
      children[i].executeMatch(frame, subterms[i]);
    }
  }

  @TruffleBoundary
  private boolean stringEq(String a, String b) {
    return a.equals(b);
  }
}
```

Figure 5: Interpreter node for pattern matching constructor terms.

3.4 Premises

Premise nodes are either *match premise* nodes or *relation premise* nodes.

Match premises. Match premise nodes derive from DynSem premises of the form `b⇒ p`, where `b` is a term builder and `p` is a match pattern. Math premises are used to either deconstruct terms into sub-terms, or to bind meta-variables, or both. The match premise `e ⇒ Plus(e1, e2)`, for example, is a conditional match premises which invokes term build `e` and applies pattern `Plus(e1, e2)` to the built term. If the term conforms to the pattern (i.e. that the type of `e` is *Plus*/2) meta-variables `e1` and `e2` will be bound to the sub-terms of `e` in the rule frame. A failure to match will return control directly outside of the rule holding the premise.

Relation premises. A relation premise is analogous to a function call in other programming languages. It consists of building an input for the call, looking up a bundle of rules and of dispatching to it providing the input. Consider premise `E ⊢e1 :: H1 ⟶ IntV(i) :: H2` from the rule of Figure 2b. Reducing `e1` requires looking up the bundle of rules of specialized arrow ("", *ty*), where *ty* is the type of `e1`, and applying it on the input term consisting of `e1` and semantic components `E` and `H1`. The pattern `IntV(i)` is applied to the result of bundle evaluation.

A bundle consists of one or more rules. The call-site dispatch is responsible for selecting the correct rule from the bundle. The mechanism works as follows. Assume, for example, that `e1` has type *Plus*/2. Firstly, a rule registry lookup retrieves the bundle identified by specialized arrow ("", *Plus*/2). Secondly, rules in the bundle are invoked in succession until the first one that succeeds. When a rule

fails it raises an exception (`PremiseFailureException`), which is intercepted by the dispatch node. When a rule succeeds it returns a result. The result of dispatch is the result of the first successful rule from the bundle.

There is a fallback mechanism for constructor terms. If all rules identified by the specialized arrow ("", *Plus*/2) fail and, since *Plus*/2 is a constructor type, a new dispatch to sort-wide rules is made. In our example, this dispatch attempts to apply the bundle of rules identified by unspecialized arrow ("", *Exp*) to the same input. This fallback mechanism is analogous to `doesNotUnderstand` in Smalltalk [5], or `methodMissing` in Ruby. The distinguishing feature of DynSem is that there is no a priori way to determine whether a rule will succeed or not. This is a limited form of backtracking in the dispatch mechanism. Backtracking is limited because it is a meta-programming error if no rules are able to reduce a term.

4 HYBRID META-INTERPRETER

Pure meta-interpreters are slow because generic term libraries obfuscate program-specific structures from the VM thereby defeating the JIT. We first analyze why this happens and the opportunity loss this represents, and we propose a *hybrid meta-interpretation* solution that combines meta-interpretation with generated components.

4.1 Performance of Pure Meta-Interpretation

In a pure meta-interpreter the term data type used to represent the object-language directly interprets the specification to construct and decompose instantiations using a generic (untyped) term library. A term construction `IntV(1)` is interpreted as the instantiation of a generic constructor term with the name `IntV` and an array of size one for the sub-term. As a consequence, a pattern match for `IntV(1)` expects a generic constructor term and performs string and arity comparisons to decide the match.

Such an approach leads to a correct semantics but comes with execution overhead that is hard to specialize. First, string comparisons are slow. We can work around this by realizing that pattern matching does not require general string comparison but only string equality checking. A maximal sharing scheme (e.g. string interning) can improve enable reference equality checking, solving the immediate issue. However, deciding a pattern match still involves three checks and a few calls: a loose type guard that the term is a constructor, a string reference equality check and an arity equality check. These checks cannot be readily JIT-ed away so their cost remains. Furthermore, accessing sub-terms remains a generic operation because the layout of the term is generic. The bottom line is that the VM treats object language terms as opaque dynamic values flowing through the meta-interpreter. To improve on this situation we need to give the VM a fast and specific way to check pattern matches and some hints of term layouts.

4.2 Solution: Specializing Term Structure

The solution is to promote object language (program and value) terms to first class citizens of the meta-interpreter, i.e. to give each object language term constructor a specialized type and each term operation (matching and constructing) a specialized implementation. In this scenario a pattern match becomes a tight type check.

```
public abstract class Exp implements IApplTerm {
@Override
public final Class<? extends IApplTerm> getSortClass() {
  return Exp.class;
}
```

(a)

```
public final class Plus_2 extends Exp {
  public final static String  CONSTRUCTOR = "Plus";
  public final static int     ARITY       = 2;

  private final Exp _1;
  private final Exp _2;

  public Plus_2(Exp _1, Exp _2) {
    this._1 = _1;
    this._2 = _2;
  }

  public Exp get_1() { return _1; }

  public Exp get_2() { return _2; }
```

(b)

Figure 6: (a) signatures of Exp sort and Plus term, (b) generated class for sort Exp and (c) generated term class for Plus constructor

The tight type check informs sub-term accesses about the precise layout of the term. Knowing the precise layout of a term allows the JIT to eliminate the dynamic dispatch for sub-term accesses and inline them. This applies in general: the tighter a type guard the more specific the JIT-ed code will be.

We make term types of the object language explicit by statically generating a term data type from the signatures of the object language. The data type consists of classes for constructors, lists and tuples as well as term building, pattern matching and type casting nodes specific to each signature. The meta-interpreter enacts a specification with respect to this term data type. The result is a hybrid meta-interpreter which combines interpreted and compiled code. We explain the makeup of this data type and how it works.

4.3 Terms

We use term signatures to derive a set of term classes that is specific to an object-language. Instances of the term classes are created by the parser.

A sort declaration `S` derives an abstract term class `S`, as depicted in Figure 6a for the sort `Exp`. The class provides a method `getSortClass` which returns a reference to the class `S` itself. This method is used when dispatching to fallback rules in order to determine the arrow (x, Exp) of the fallback rules.

Constructor declarations `C: S1 * S2 * ... * Sn → S` derive term classes named `C_n` extending `S`. Classes declare fields `_i` of type `Si` and accessor methods for every child as illustrated in Figure 6. Fields are declared final to ensure that terms are immutable. Term immutability allows the meta-interpreter to safely cache terms and potentially allows the JIT to inline field accesses. The precise type declaration of fields provides type information to the VM and prevents accidental construction of malformed trees. Tuples derive

term classes by a similar mechanism, encoding every element of the tuple as a typed field. Lists are encoded as *cons-nil* lists, one of every type of list used and are also immutable. Standard Java classes are used to represent leaf types such as `String` and **`int`**.

4.4 Term Construction

```
public abstract class Exp_B extends TermBuild {
  public Exp_B(SourceSection source) {
    super(source);
  }

  @Override
  public abstract Exp executeGeneric(VirtualFrame frame);

  @Override
  public abstract Exp executeEvaluated(
          VirtualFrame frame,
          Object... terms);
}
```

(a)

```
@NodeChildren({
@NodeChild(value = "tb_1", type = Is_Exp.class),
@NodeChild(value = "tb_2", type = Is_Exp.class) })
public abstract class Plus_2_B extends Exp_B {
  public Plus_2_B(SourceSection source) {
    super(source);
  }

  @Specialization(limit = "1", guards = {
                    "t_1 == t_1_cached",
                    "t_2 == t_2_cached" })
  public Plus_2 doCached(Exp t_1, Exp t_2,
          @Cached("t_1") Exp t_1_cached,
          @Cached("t_2") Exp t_2_cached,
          @Cached("doUncached(t_1, t_2)")
                              Plus_2 cachedTerm) {
    return cachedTerm;
  }

  @Specialization(replaces = "doCached")
  public Plus_2 doUncached(Exp t_1, Exp t_2) {
    return new Plus_2(t_1, t_2);
  }
}
```

(b)

```
@NodeChild(value = "tb", type = TermBuild.class)
public abstract class Is_Exp extends TermBuild {
  public abstract Exp executeCast(VirtualFrame frame);
  public abstract Exp executeCastEvaluated(Object term);

  @Specialization
  public final Plus_2 doPlus_2(Plus_2 term) {
    return term;
  }
  ...
}
```

(c)

Figure 7: Generated classes for (a) term builds of the Exp sort, (b) term builds of the Plus term and (c) type cast node for sort Exp

Object-language terms are initially constructed by the parser but additional terms will need to be constructed at run time. This happens when value terms are built and when semantics of an object-language construct is given by desugaring to another construct of the language.

A sort declaration `S` derives an abstract interpreter node `S_B` as illustrated in Figure 7a. It provides extension points for concrete constructor building nodes. Constructor declarations `C:S1*S2*...*Sn→ S` derive interpreter nodes for term construction named `C_n_B` extending `S_B` as shown in Figure 7b. The generated nodes use the Truffle specialization DSL [15] to declare term building children for expected sub-terms.

Each term construction node declares two `@Specialization` annotated methods. The `doCached` method maintains an inline cache of a term being built. The cache is guarded by referential equality of the children terms with respect to those in the cache. Caching ensures that as long as the sub-terms built are constant the emitted term is constant. The runtime effect is that entire tree allocations can be elided by reusing cached terms. The cache mechanism is transparent with respect to rule frames and rule results. As long as the sub-terms are constant the allocation can be elided.

Caching is most useful for rules which give the semantics of a construct in terms of another construct. For example, the rule `Gt(e1, e2)` ⟶ `Lt(e2, e1)`, generates program terms at runtime. Since the program is stable, the sub-terms `e1` and `e2` will be constant and the entire construction of `Lt(e2, e1)` can be replaced by a guarded cache access at run time.

Caching is beneficial beyond avoiding allocations. As long as the input term to a pattern match is constant (or at least perceived to be constant) the type check guarding the pattern match can be eliminated and the term decomposition can be constant folded. The cache has a single cell. Once a cache miss occurs the node transitions to the `doUncached` specialization. A cache miss in a subtree will propagate a wave of cache misses to its parents without affecting caching in siblings.

Construction nodes for lists and tuples are similarly generated. The meta-interpreter provides hand-implemented nodes for accessing meta-variables and constructing terms of leaf types (e.g. `String` and **`int`**). Interning of strings and unboxing (actually avoiding boxing altogether) for numbers keeps reference equality checks in cache guards sane.

4.5 Type Casts

Leaf-type guards, i.e. checks for types at the leafs of the inheritance hierarchy using **`instanceof`**, result in faster machine code than looser type guards on intermediate types, especially when running on the Graal VM. Intuitively this is easy to explain. First, checking for leaf types is the cheapest kind of type check, since it just requires checking for type equality instead of traversing the type hierarchy to do subtype checking. Second, after code is inlined it will contain some nested type guards. The tighter the outer guards, the more of the nested guards will be eliminated, since they are implied by the outer guards. Third, code that invokes a method on an object whose precise type is known (through a leaf-type guard) is known to always invoke the same target method and so a lookup is no longer required.

```
public abstract class Plus_2_M extends MatchPattern {
  @Child private MatchPattern p_1;
  @Child private MatchPattern p_2;

  public Plus_2_M(SourceSection source,
          MatchPattern p_1, MatchPattern p_2) {
    super(source);
    this.p_1 = p_1;
    this.p_2 = p_2;
  }

  @Specialization
  public void doMatch(
    VirtualFrame frame, Plus_2 term) {
    p_1.executeMatch(frame, term.get_1());
    p_2.executeMatch(frame, term.get_2());
  }

  @Fallback
  public void doFailed(Object term) {
    throw PatternMatchFailure.SINGLETON;
  }
}
```

Figure 8: Generated match pattern for the Plus constructor term.

In a naive term data type, non-leaf type checks in the meta-interpreter occur for every child of a term construction node. Consider the signatures of Figure 2a and the specialization methods of Figure 2c. Prior to invoking the evaluation method, sub-terms have to be verified to be instances of `Exp`. This is a non-leaf type check.

We wrap every sub-term construction into a type cast node. Every sort declaration `S` derives a type cast node `Is_S`. Consider the signatures of Figure 2a and the derived type casting node of Figure 7c. The type cast node declares a specialization method for each constructor of sort `S`. A specialization method is applicable if the sub-term construction builds a term that passes the type guard of the method. The specialization `doPlus_2` is only applicable if the sub-term constructed is an instance of `Plus_2`. Type cast nodes have a number of advantages.

First, they perform leaf-type checks only. This is particularly useful when sub-terms are read from meta-variables and nothing is known about their type. Second, a type cast node speculates that the concrete type of the sub-term is constant at run time and specializes to it. Third, inlined code coming from type cast nodes will contain leaf-type guards. Subsequent type guards can be eliminated and method dispatches can be inlined. Both result in faster machine code.

4.6 Pattern Matching

A constructor pattern match node is responsible for deciding whether an input term is a constructor term of the expected kind and for dispatching matches on its sub-terms. Constructor declarations `C: S1 * S2 * ... * Sn→ S` derive pattern matching interpreter nodes named `C_n_M`. For example, the `Plus` signature of Figure 2a derives the pattern matching node `Plus_2_M` of Figure 8. Each matching node declares two methods to handle the success and failure cases. If the incoming term is of the expected type (`doMatch` case) then pattern matching is dispatched on the sub-terms. If the incoming term is not of the expected type (`doFailed` case) the pattern fails immediately by raising an exception. Pattern match nodes for lists and tuples are derived similarly.

We model pattern match failure by raising a specific exception, conversely we model success by uninterrupted evaluation. This reflects our expectation that pattern matches succeed more often than they fail. Alternatively, we could model outcomes with a **`boolean`** flag and chain pattern matching of sub-terms with a short-circuiting conjunction (`&&`) operator. We observed no significant performance difference between the two designs. We choose to model pattern match failure with a control flow exception because it makes for cleaner code. Notice that success and failure methods of the pattern matching node are annotated with `@Specialization` and `@Fallback`, respectively. This reflects our speculation that a pattern match tends to either always succeed or always fail. Pattern matches are either applied to program terms or to value terms. Program terms are not likely to change and neither is the outcome of a pattern match on them.

The outcome of the outermost pattern match on a value term is unlikely to flip-flop because the type of the value term that is being matched is determined by the program term that reduced to it. Consider the pattern match `IntV(i)` of the first premise of Figure 2c and assume that `e1` is a function call. Tiger is a statically typed language, so if the interpreted program type-checked the pattern match will always succeed at run time. In a dynamic language however, it is possible that `e1` will evaluate to varying types during execution, i.e. the outcome of the pattern match may vary. However, after specialization, call target splitting and inlining, it is very likely that the function call of `e1` exhibits a constant type and so the pattern match will stabilize. If its type is truly dynamic the pattern match node will flip-flop with some overhead. We have yet to experience this in the wild though.

5 EFFICIENT RULE DISPATCH

We propose two mechanisms by which to improve the efficiency of rule invocation in the meta-interpreter. The two mechanisms work together to stabilize rule dispatch and thus to allow Graal to eliminate it completely.

In the ideal case, sufficient rule splitting and inlining occurs such that the executing interpreter tree has exactly the same shape as the AST of the program, i.e. there is no more specification-induced dispatch.

The issue is circular and widespread: if the target of a relation premise is not stable (e.g. due to varying input terms) then it is not inline-able. If a rule is not inlined then it will be exposed to varying program terms and so the rules it calls will not be inline-able either.

5.1 Caching Rule Bundles

Statically, a relation premise of the form `e ⟶ v` identifies the set of all rule bundles populating the arrow ("", S) and its specializations ("", ty), where S is the sort and ty is the type of `e`. For example, the premises in Figure 2b identify all the bundles of arrow ("", Exp) and its specializations. But at run time, in the presence of a concrete type ty the premise identifies the specialized arrow ("", ty) and hence a specific bundle of rules. We speculate on the boundedness

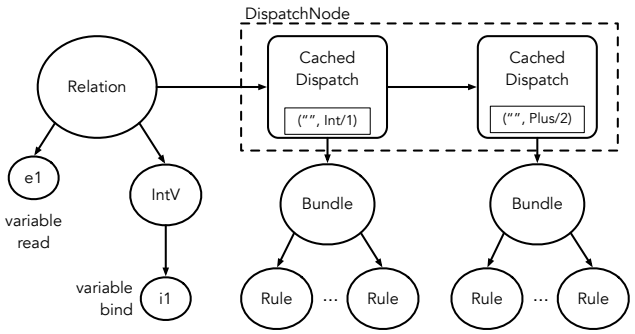

Figure 9: AST of the first relation premise of Figure 2c after it has been applied to two different `Exp` terms.

and stability of program terms, and therefore of the specialized arrows invoked by premises to enable caching of rule bundles at the call-site.

Figure 9 shows a schematic view of the AST of relation premise `e1` ⟶ `IntV(i1)` of Figure 2c. The relation has already invoked specialized arrows ("", $Plus/2$) and ("", $Int/1$) because it has observed two types of input terms: a nested addition and an integer literal.

A dispatch node maintains a polymorphic inline cache of rule bundles. The premise of Figure 9 has cached rule bundles for specialized arrows ("", $Plus/2$) and ("", $Int/1$). When the premise is executed again, if its input term has either type $Plus/2$ or $Int/1$ the cache will be hit and the rule registry lookup will be avoided. Otherwise a cache miss occurs and a new lookup is performed. The cache is bounded and the dispatch node will perform full lookups if the cache is full. There is no cache invalidation because DynSem rules are fixed during execution.

The larger the cache size the more variability the dispatch node can tolerate and the more it can specialize to program structure. On the one hand a small cache size means a lower memory footprint and shorter cache-lookup times. On the other hand an infinite cache guarantees that all call targets are linked into the relation premise and the hot ones can be inlined. When Graal clones the parent rule for inlining the entire cache is discarded so the overhead disappears.

5.2 Dispatch Trees

Overloaded DynSem rules are bundled together in the rule registry, as we discussed in Section 3. There is no caller-side mechanism to select from the bundle a rule that will succeed without actually executing it. At the same time executing a rule and observing that it fails has a non-zero computational cost. Additionally, the fallback mechanism for constructor reductions requires that all rules have failed prior to its enactment. The second of two mechanisms we propose addresses the issue of optimizing call-sites to efficiently handle bundled rules and fallbacks.

We speculate that a rule that has succeeded once is likely to succeed again and conversely a rule that has failed once is likely to fail again. We encode bundles and fallback bundles as dispatch trees. Dispatch trees draw from the dispatch chains of Marr et al. [22]. The core idea is to maintain at the dispatch node a sequence of rules that have been applied. The sequence of rules grows at the head such that successful rules are kept closest to the dispatch node.

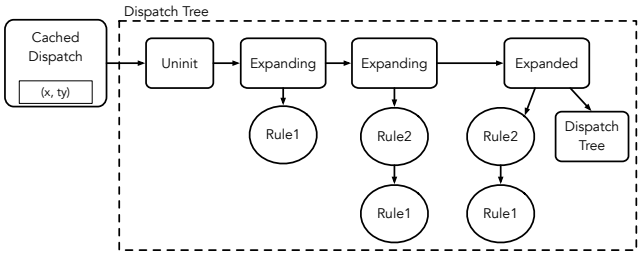

Figure 10: Structure and evolution of a dispatch tree.

Figure 10 shows how a dispatch tree evolves over successive executions. At first, a dispatch tree is uninitialized (`Uninit`). At its first execution it retrieves a rule bundle from the rule registry and replaces itself with an `Expanding` dispatch tree.

An `Expanding` node maintains a bundle and a chain of rule targets. When invoked an `Expanding` node will evaluate the chain of rules until the first one that succeeds. Initially the chain consists of a single rule. The dispatch tree will remain fixed for as long as the chain succeeds. Upon failure of the chain, the `Expanding` node removes a new rule from the bundle, inserts it at the beginning of the chain and evaluates just this first target. The process repeats until either a succeeding rule is inserted or the bundle is empty. In the latter case the `Expanding` node transitions to an `Expanded` node.

The `Expanded` node inherits the fully-expanded rule chain and creates a new dispatch tree for fallback dispatch. In subsequent executions the `Expanded` node will first evaluate the entire dispatch chain and only execute the fallback tree if the chain fails. The tree remains `Expanded` until the parent rule is cloned again or execution ends. Note that sort-wide fallback rules only exist for constructors; an `Expanded` node will not create a fallback dispatch for other terms. Therefore at most two dispatch trees are ever nested in a dispatch node.

DynSem specifications do not regularly have heavily overloaded rules. For such rules the dispatch trees are compiled efficiently to simple control-flow. In the case of heavily overloaded rules the peak performance that should be achievable is comparable but the compilation workload is significantly higher due to chain length. We discuss this and possible solutions in Section 7.

6 RECURSION ELIMINATION

DynSem does not have higher-order functions nor any builtin loop constructs. As a result specifications rely on recursive rules to specify list-traversals and loop semantics. The `evalArgs` meta-function of Figure 2e is a classic example of using recursion to traverse a list in DynSem.

We do not perform any source-to-source transformations to remove recursive calls. Recursion elimination is not always possible and, as we will explain, not always required. We distinguish two types of recursion in specifications: statically bounded and dynamically bounded. Recursion in rules such as `evalArgs` is statically bounded: the recursion depth is syntactically fixed by the program. Statically bound recursion is beneficial because it clearly exposes the structure of the interpreted program. We rely on the just-in-time compiler to unroll and inline the recursion call tree for statically bounded recursion. At run time, after the top-level invocation of

the `evalArgs` rule is inlined, the rest of the call tree will also be inlined.

6.1 Dynamically Bounded Recursion – Loops

Consider the rule of figure Figure 11a describing a Tiger while loop by desugaring it to an `if` term. Notice the arbitrarily deep call tree resulting from successive invocations of the `While/2` rule. The depth of the tree is determined by the number of loop iterations which is dynamic. From the perspective of a just-in-time compiler loops that depend on runtime values are unbounded. If Graal would begin inlining the calls that make up the loop call tree it would give up due to too deep inlining. Rules such as the one of Figure 11a are also non-trivial to optimize statically. At a minimum we must at least instruct Graal that a call-tree is part of a loop construct and those calls should not be inlined.

As a solution we provide a library of loop semantics that semanticists can reuse to describe looping constructs. The library comes in two variants: a first provides a pure (recursive) DynSem implementation to be used for reasoning about programs; a second relies on a native implementation of a loop provided by the meta-interpreter. The two have the same API and semantics and are interchangeable. The language designer can choose between them by changing an import.

The two variants share the signatures of Figure 11b. Figure 11e reduces while loop expression of Tiger to the meta-function provided by the library. The native library implementation of Figure 11d is simply a wrapping call to the natively-provided `_while` rule. DynSem has special knowledge of this native rule and replaces use-sites of `_while` by special calls to a repeating node. Invocations of the `_while` rule are evaluated iteratively. This ensures that the size of call tree is bounded and the just-in-time compiler can inline rules. The repeating node of the meta-interpreter uses a dedicated Truffle node for repeating constructs [36, 37].

The performance benefit of native loops is visible for loops with many iterations. The call tree of short loops is relatively small and Graal can usually compile it efficiently if we allow it to. On long loops however, inlining would not be possible without the native loop library. The primary benefit of the native library is therefore that inlining is at all possible.

```
w@While(e1, e2) ⟶ IfThen(e1, Seq([e2, w]))
```

(a)

```
signature
  sorts Cond Eval
  constructors
    whileLoop: Cond * Eval * V ⟶ Eval
  arrows
    Eval ⟶ V
    Cond  ⟶ Bool
```

(b)

```
c ⟶ cv;
case cv of
  false ⟶ v0 ⇒ v
  true  ⟶ e ⟶ _; w ⟶ v
-------------------------
w@whileLoop(c, e, v0) ⟶ v
```

(c)

```
signature
  arrows
    _while(Cond, Eval, V) ⟶ V {native}
rules
  whileLoop(c, e, v) ⟶ _while(c, e, v)
```

(d)

```
While(e1, e2) ⟶ whileLoop(e1, e2 , UnitV())
```

(e)

Figure 11: (a) Indirectly recursive rule for a while loop (b) shared signatures for library while loop (c) pure DynSem while loop (d) native while loop (e) loop library usage in Tiger

7 EVALUATION

We evaluate the speedup the techniques presented in this paper give to the Tiger interpreter when running on the Graal VM as compared to the standard Java Virtual Machine. We describe the experiment setup and discuss results.

Subjects. Four meta-interpreter variants are evaluated: a hybrid meta-interpreter complete with all techniques discussed (`Full`), a pure meta-interpreter without generated components (`-Hybrid`), a complete hybrid meta-interpreter without dispatch caching (`-DCache`), and a complete hybrid meta-interpreter without dispatch trees (`-DTree`).

We also evaluate three styles of semantics for Tiger: a specification using native loops (`Regular`), a specification using pure (recursive) loops (`PureLoop`), and a specification with very heavily overloaded rules (`Overload`). The heavily overloaded flavor is unrealistic in that all evaluation flows through a single specialized arrow that is populated by 44 rules.

Workloads. We migrated a subset of benchmark programs [20] that could be implemented in Tiger, resulting in 5 workloads: *list* creation and traversal, solver for the 8-*queens* problems, array *permutations*, *sieve* of Eratosthenes, *towers* of Hanoi. We created an additional workload consisting of a long-running *while-loop*. We modified the programs to repeat 30 times and record the duration of each repetition.

Method. Measurements are taken on an otherwise idle MacBook Pro 11,1, Intel Core i7 2.8Ghz, 16GB of RAM running macOs 10.13.4. JVM and Graal VM (GVM) workloads are run on Java Virtual Machine `1.8.0_144` and Graal VM version `1.0.0-rc1`, respectively. VM arguments are passed to reduce garbage collection work and increase stack size[2]. An additional argument is passed to runs on GVM to disable background compilation[3].

We discard the first 10 measurements to account for warm-up time and we use the median execution time of a workload on the JVM as the baseline for that workload. Other execution times for a workload are normalized against this baseline on the JVM. The

[2] `-Xss64m -Xms1g -Xmx1g`
[3] `TruffleBackgroundCompilation=false`

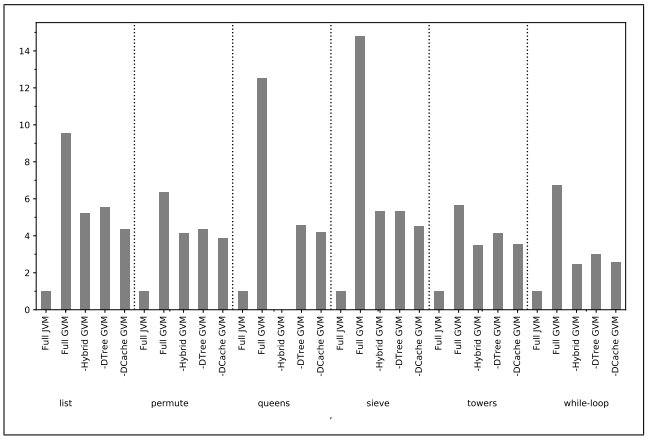

Figure 12: Speedup on the Graal VM compared to the JVM for different meta-interpreter flavors (higher is better).

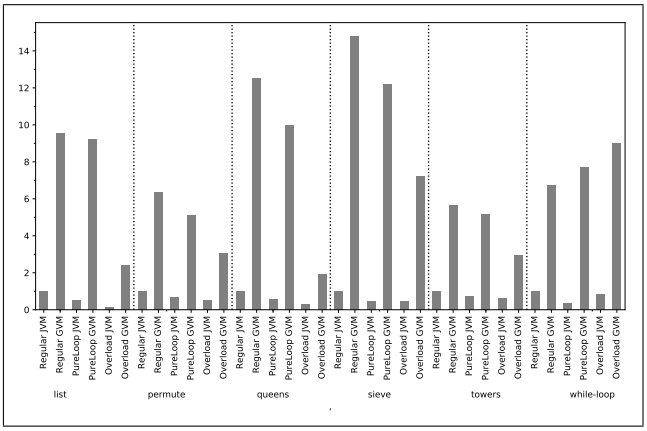

Figure 13: Speedup on the Graal VM compared to JVM for different specification flavors (higher is better).

aggregate speedup of a benchmark is the median of its normalized execution times.

Reproducibility. An artifact[4] is available for recreating the experiments. The archive contains the raw data collected, workloads, meta-interpreter flavors, Tiger flavors and instructions.

7.1 Results

Meta-interpreter flavors. Figure 12 shows the speedup of meta-interpreter flavors with respect to running on the JVM. The fully featured hybrid meta-interpreter is between 6 and 15 times faster on the GVM than on the JVM. We observe a benefit of hybrid interpretation (`Full`) over the pure interpretation (`-Hybrid`). The benefit is partially due to term-specific pattern matching, which results in more efficient code. This supports our decision to statically generate term data types.

Dispatch caching and dispatch trees, or the lack thereof (`-DCache` and `-DTree`), play a significant role in speedup of execution. Note that the effect of these optimizations is felt only when both are used. When dispatch targets are not cached (`-DCache`) the effect of dispatch trees is not significant. To take advantage of dispatch trees Graal must first evaluate that the target of a dispatch is stable. When dispatch caching is enabled but full bundles are cached instead of trees (`-DTree`) the performance is comparable. Dispatch involves attempting the targets in order until the first that succeeds. The problem is twofold: control-flow inside the dispatch node is obfuscated by the bundle iteration, and successful rules are not prioritized over failing ones. Using both dispatch caching and dispatch trees (`Full`) allows dispatch to stabilize and calls to be inlined.

Specification styles. Turnaround across the specification styles averages at 5.2*s* consisting of 2.4*s* for specification processing such as explication of semantic components, and 2.8*s* for term library generation. Additional time is spent to compile the generated Java sources. Speedups on the Graal VM with respect to the JVM are shown in Figure 13 for the three styles of Tiger specification. Using native loops (`Regular`) delivers better performance than recursive loops (`PureLoops`) on most workloads. The benefit is quite small though and on the *while-loop* workload the effect is adverse. We conjecture that there are two reasons for this. First, the native implementation of the loop node must do intricate bookkeeping for semantic components which adds some overhead. Second, it is achievable for Graal to inline the recursive loop iterations because the program is trivial and the interpreter AST does not explode. The issue requires further investigation.

The unrealistically overloaded specification (`Overload`) incurs significant overhead, both on the JVM and on GVM. On the JVM this is to be expected: potentially very many rules have to be invoked to find a successful call target. This translates to many wasted dispatches. The overhead on GVM is unexpected though. The cause is that the interpreter AST is polluted with unsuccessful rules. A dispatch tree grows with every attempted rule, whether successful or not, resulting in inlining being aborted due to excess tree size. We think a solution is to grow dispatch chains only with rules that have succeeded, and to keep failing rules in a circular buffer.

Comparison with native interpreters. We implemented a direct interpreter for Tiger in Java. It is a vanilla AST interpreter and comes in two flavors: one using mutable maps, and one using persistent maps for environments. On average, persistent environments increase run time by 4 times. The DynSem meta-interpreter also relies on persistent maps for environments. The `Regular` specification style on the `Full GVM` runtime is on average 4 times slower than the native interpreter with mutable environments, and 25% slower than the interpreter with persistent environments. Although Graal can remove much of the meta-interpretation overhead, there remains a significant performance gap between derived interpreters and natively implemented ones. We expect to be able to reduce the remaining 25% performance gap to a native interpreter with persistent environments by tuning cache sizes to allow further stabilization of interpreter trees. Reducing the gap to a native interpreter with mutable environments requires that we eliminate persistent environments from meta-interpreters. One possibility to eliminate persistent environments is to derive memory layouts from scope graphs [25, 32], the result of static name analysis of a program, following the scope-as-frames [29] approach. In the future we plan to investigate using frame-based memory layouts in

[4] https://bitbucket.org/slde/manlang18-benchmarks/src/manlang18

DynSem specification and whether and by how much performance improves if Graal can see through the memory of a program.

8 RELATED WORK

DynSem is part of the larger family of dynamic semantics definitions formalisms.

Typical DynSem specifications are in a big-step style [17]. Petterson [26] describes how to merge overloaded big-step rules and generate an interpreter. In our early experiments [33] with this method the time spent in compilation was significant and when run on an early Graal VM performance was poor. This served as motivation for the development of the meta-interpreter.

The big-step rules of a DynSem specification that uses meta-functions and implicit reductions resemble SOS [27] rules while retaining big-step semantics. Most related to I-MSOS [24], DynSem borrows its idea of semantic components and their implicit propagation. An I-MSOS specification derives an equivalent MSOS [23] specification by explication of semantic component propagation. MSOS rules compile to Prolog clauses. Small-step MSOS rules can be specialized by refocusing and striding [28] to reduce the amount of tree traversals required during evaluation. *Funcons* [8] intends to provide a definitive collection of reusable semantics. Semantics of *funcons* are specified in I-MSOS.

PLT Redex [11, 19] is a domain-specific language for Felleisen-Hieb reduction rules embedded into the Racket [12] programming language. Pycket [4] is a tracing JIT compiler for Racket implemented in RPython [2]. It shows improved performance on regular benchmarks. We were unable to determine if and how Redex semantics specifications benefit from Pycket. K [31] is a semantics formalism that has been applied to production-sized languages (C [10] and Java [6]). K specifications derive an interpreter in Maude [9].

Partial evaluation [16] is an automatic program specialization technique. Graal is a JIT compiler which partially evaluates a program at run time. Truffle provides a set of APIs and annotations to guide the partial evaluation. When applied to the DynSem meta-interpreter, Graal partially evaluates the specification interpreter with respect to a specification, akin to the first Futamura projection [13, 14] of the specification interpreter. Our experiments reveal that some meta-interpreter overhead persists after JIT compilation, therefore the resulting code is not Jones optimal [16]. Amin et al [1] describe a mechanism to eliminate multiple layers of interpretation by staging all interpreter layers together. Our approach differs significantly in that it has no expectation of a similarity between the object-language and DynSem and in that we rely on an existing online partial evaluator which is not under our control.

PyPy [30] is an alternative to the Graal meta-compilation approach. PyPy translates an interpreter to C code which is executed in an interpreter containing a tracing JIT compiler [7]. The C code is JIT-compiled using trace information. In contrast, Truffle interpreters are AST interpreters that undergo online partial evaluation to remove the interpretation overhead. Marr et al. [21] perform a performance-oriented comparison of the the two approaches.

9 CONCLUSION & DISCUSSION

DynSem is a domain-specific language for concise specification of the dynamic semantics of programming languages, aimed at rapid prototyping and evolution of language designs. In our early prototypes, DynSem specifications were compiled to an interpreter. The process of generating an interpreter caused long turnaround times during language prototyping. In order to shorten turnaround times, we turned to interpreting specifications directly instead of compiling them.

We presented the architecture of a meta-interpreter for DynSem specifications which requires minimal pre-processing of specifications. We built the meta-interpreter as a Truffle interpreter in Java. Striving for short runtimes we experimented with optimization techniques that would allow JIT compilation of the meta-interpreter on the Graal VM. We statically generated a term data type specific to an object language, resulting in a hybrid meta-interpreter. The data type allows Graal to optimize pattern matching and observe the stability of program terms.

A meta-interpreter incurs a significant performance penalty from rule dispatch. We presented an approach to improve dispatch by dispatch caching and by reorganizing overloaded rules into dispatch trees. This approach allows the meta-interpreter to be JIT-ed such that all dispatch induced by the specification is eliminated, and only program-induced dispatch remains.

When reasoning about recursive rules we distinguished statically-bound recursion that is induced syntactically by the object program from dynamically-bound recursion which is a consequence of looping constructs in the object language. We designed a reusable DynSem library for loops that allows specifications to swap between purely defined loops and meta-circularly defined ones without sacrificing declarativity. While meta-circular loops show an improvement in execution time, their most important contribution is that they prevent explosive AST growth due to inlining, which would cause the JIT to abort compilation.

We have evaluated the effect of our optimization on the meta-interpreter by means of an empirical evaluation on a specification of the Tiger programming language. The combination of generated term data type, dispatch caching and dispatch trees unlocks JIT compilation on the Graal VM. Graal is able to inline reduction rules and to partially evaluate pattern matching. The speedup obtained ranges from factor 6 to factor 15, depending on the workload.

While we have achieved an order of magnitude performance improvement by specializing a meta-interpreter, there is still a large gap with respect to the performance of hand-written interpreters. We believe there are further opportunities to specializing meta-interpreters that can narrow this gap. In the future we plan to investigate using frame-based memory layouts [29] derived from static name binding information [25, 32] to make the memory of running programs transparent to the JIT compiler.

ACKNOWLEDGMENTS

This research was supported by a gift from the Oracle Corporation and by the NWO VICI *Language Designer's Workbench* project (639.023.206). We would like to thank Stefan Marr for our constructive conversations, the Truffle group at the Institute for System Software of the Johannes Kepler University Linz and the wider Truffle and Graal community for their assistance.

REFERENCES

[1] Nada Amin and Tiark Rompf. 2018. Collapsing towers of interpreters. *PACMPL* 2, POPL (2018). https://doi.org/10.1145/3158140

[2] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. 2007. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 Symposium on Dynamic Languages, DLS 2007, October 22, 2007, Montreal, Quebec, Canada*, Pascal Costanza and Robert Hirschfeld (Eds.). ACM, 53–64. https://doi.org/10.1145/1297081.1297091

[3] Andrew W. Appel. 1998. *Modern Compiler Implementation in ML*. Cambridge University Press.

[4] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. 2015. Pycket: a tracing JIT for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 22–34. https://doi.org/10.1145/2784731.2784740

[5] Hubert Baumeister, Harald Ganzinger, Georg Heeg, and Michael Rüger. 1987. Smalltalk-80. *it - Information Technology* 29, 4 (1987), 241–251. https://doi.org/10.1524/itit.1987.29.4.241

[6] Denis Bogdanas and Grigore Rosu. 2015. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 445–456. https://doi.org/10.1145/2676726.2676982

[7] Carl Friedrich Bolz. 2014. *Meta-Tracing Just-in-Time Compilation for RPython*. Ph.D. Dissertation. Heinrich Heine University Düsseldorf. https://doi.org/1057957054

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

[9] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. 1999. The Maude System. In *Rewriting Techniques and Applications, 10th International Conference, RTA-99, Trento, Italy, July 2-4, 1999, Proceedings (Lecture Notes in Computer Science)*, Paliath Narendran and Michaël Rusinowitch (Eds.), Vol. 1631. Springer, 240–243. https://doi.org/link/service/series/0558/bibs/1631/16310240.htm

[10] Chucky Ellison and Grigore Rosu. 2012. An executable formal semantics of C with applications. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 533–544. https://doi.org/10.1145/2103656.2103719

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

[12] Matthias Felleisen, Robby Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA (LIPIcs)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 113–128. https://doi.org/10.4230/LIPIcs.SNAPL.2015.113

[13] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391. https://doi.org/content/l46w6q3720n57607/

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

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

[16] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science. ISBN number 0-13-020249-5 (pbk).

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

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

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

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

[21] Stefan Marr, Tobias Pape, and Wolfgang De Meuter. 2014. Are We There Yet?: Simple Language Implementation Techniques for the 21st Century. *IEEE Software* 31, 5 (2014), 60–67. https://doi.org/10.1109/MS.2014.98

[22] Stefan Marr, Chris Seaton, and Stéphane Ducasse. 2015. Zero-overhead metaprogramming: reflection and metaobject protocols fast and without compromises. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 545–554. https://doi.org/10.1145/2737924.2737963

[23] Peter D. Mosses. 2004. Modular structural operational semantics. *Journal of Logic and Algebraic Programming* 60-61 (2004), 195–228. https://doi.org/10.1016/j.jlap.2004.03.008

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

[25] Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *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 (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer, 205–231. https://doi.org/10.1007/978-3-662-46669-8_9

[26] Mikael Pettersson. 1996. A Compiler for Natural Semantics. In *Compiler Construction, 6th International Conference, CC 96, Linköping, Sweden, April 24-26, 1996, Proceedings (Lecture Notes in Computer Science)*, Tibor Gyimóthy (Ed.), Vol. 1060. Springer, 177–191.

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

[28] Casper Bach Poulsen and Peter D. Mosses. 2013. Generating Specialized Interpreters for Modular Structural Operational Semantics. In *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Madrid, Spain, September 18-19, 2013, Revised Selected Papers (Lecture Notes in Computer Science)*, Gopal Gupta and Ricardo Peña (Eds.), Vol. 8901. Springer, 220–236. https://doi.org/10.1007/978-3-319-14125-1_13

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

[30] Armin Rigo and Samuele Pedroni. 2006. PyPy s approach to virtual machine construction. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, Peri L. Tarr and William R. Cook (Eds.). ACM, 944–953. https://doi.org/10.1145/1176617.1176753

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

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

[33] Vlad A. Vergu, Pierre Néron, and Eelco Visser. 2015. DynSem: A DSL for Dynamic Semantics Specification. In *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland (LIPIcs)*, Maribel Fernández (Ed.), Vol. 36. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 365–378. https://doi.org/10.4230/LIPIcs.RTA.2015.365

[34] Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. 2014. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In *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*, Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz (Eds.). ACM, 95–111. https://doi.org/10.1145/2661136.2661149

[35] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language

runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen 0001 and Martin T. Vechev (Eds.). ACM, 662–676. https://doi.org/10.1145/3062341.3062381

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

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