DynSem: A DSL for Dynamic Semantics Specification

Vlad Vergu, Pierre Neron, and Eelco Visser

Delft University of Technology
Delft, The Netherlands
`{v.a.vergu|p.j.m.neron|visser}@tudelft.nl`

Abstract

The formal semantics of a programming language and its implementation are typically separately defined, with the risk of divergence such that properties of the formal semantics are not properties of the implementation. In this paper, we present DynSem, a domain-specific language for the specification of the dynamic semantics of programming languages that aims at supporting both formal reasoning and efficient interpretation. DynSem supports the specification of the *operational semantics* of a language by means of *statically typed conditional term reduction rules*. DynSem supports *concise* specification of reduction rules by providing *implicit build and match coercions* based on reduction arrows and implicit term constructors. DynSem supports *modular* specification by adopting implicit propagation of semantic components from I-MSOS, which allows omitting propagation of components such as environments and stores from rules that do not affect those. DynSem supports the declaration of *native operators* for delegation of aspects of the semantics to an external definition or implementation. DynSem supports the definition of auxiliary *meta-functions*, which can be expressed using regular reduction rules and are subject to semantic component propagation. DynSem specifications are *executable* through automatic generation of a Java-based AST interpreter.

1998 ACM Subject Classification F.3.2. Semantics of Programming Languages (D.3.1.)

Keywords and phrases programming languages, dynamic semantics, reduction semantics, semantics engineering, IDE, interpreters, modularity

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.365

1 Introduction

The specification of the dynamic semantics is the core of a programming language design as it describes the runtime behavior of programs. In practice, the implementation of a compiler or an interpreter for the language often stands as the *only* definition of the semantics of a language. Such implementations, in a traditional programming language, often lack the clarity and the conciseness that a specification in a formal semantics framework provides. Therefore, they are a poor source of *documentation* about the semantics. On the other hand, formal definitions are not executable to the point that they can be used as implementations to run programs. Even when both a formal specification and an implementation co-exist, they typically diverge. As a result, important properties of a language as established based on its formal semantics may not hold for its implementation. Our goal is to unify the semantics engineering and language engineering of programming language designs [22] by providing a notation for the specification of the dynamic semantics that can serve both as a readable formalization as well as the source of an execution engine.

In this paper, we present DynSem, a DSL for the concise, modular, statically typed, and executable specification of the dynamic semantics of programming languages. DynSem supports the specification of the *operational semantics* of a language by means of *conditional term reduction rules.*

- DynSem rules are *statically typed* with respect to the declaration of the signature of constructors of abstract syntax nodes and values, and the declaration of typed (and optionally named) reduction arrows.
- DynSem supports *concise* specification of reduction rules by providing *implicit build and match coercions* based on reduction arrows and implicit term constructors.
- DynSem supports *modular* specification by adopting the implicit propagation of semantic components from the I-MSOS (Implicitly Modular Structural Operational Semantics) formalism [16], which allows omitting propagation of components such as environments and stores from rules that do not affect those. The implicit semantic components are explicated through a source-to-source transformation, to produce a complete and unambigous specification. This formalizes informal conventions that are often adopted in language specifications such as *The Definition of Standard ML* [14]. The declaration of variable schemes for semantic components allows their concise identification in rules, following typical naming conventions in typeset renderings of semantic rules.
- DynSem supports the declaration of *native operators* for delegation of aspects of the semantics to an external definition or implementation. For example, the details of the semantics of numeric operations may be abstracted over by delegating to a library of native numeric operators.
- DynSem supports the definition of auxiliary *meta-functions*, which can be expressed using regular reduction rules and are subject to semantic component propagation. Thus, one can provide abstractions over run-time operations that one *does* want to define explicitly in the semantics, without having to inline their definition in each rule that uses them.
- DynSem specifications are *executable* through automatic generation of a Java-based AST interpreter. DynSem specifications can be written either in big-step style or in small-step style, but our interpreter generator is geared towards big-step style rules. While small-step style rules are compiled as well, the interpreter generator does not (yet) provide any optimizations (such as refocusing [18]) to reduce the excessive traversals inherent in small-step specifications. The interpreter generated from big-step style specifications has reasonable performance, allowing one to run and experiment with the design of a language on concrete programs.

The DynSem language is integrated into the Spoofax language workbench, a tool for the definition of (domain-specific) programming languages [11]. From a language definition, Spoofax generates a complete programming environment including parser, type checker, and language-aware editor (IDE). The DynSem integration of Spoofax extends these programming environments with generated interpreters. DynSem itself is built using Spoofax and comes with a full-fledged IDE.

The rest of the paper is organized as follows. In the next section we introduce the definition of term reduction rules and the definition of signatures that declare sorts, constructors, and arrows. In Section 3 we discuss the features that DynSem provides for supporting concise definitions of rules. In Section 4 we describe the generation of interpreters from DynSem specifications. Finally, in Section 5 and Section 6 we discuss related and future work.

2 DynSem Reduction Rules

The specification of a language semantics in DynSem is expressed using term reduction rules such as the one presented in Figure 1. This rule is the fully explicit version of the definition

```
rules
  Env E ⊢ Plus(e1, e2) :: Sto s1 ⟶ NumV(PlusI(n1, n2)) :: Sto s3
  where
    Env E ⊢ e1 :: Sto s1 ⟶ NumV(n1) :: Sto s2,
    Env E ⊢ e2 :: Sto s2 ⟶ NumV(n2) :: Sto s3
```

Figure 1 Fully explicit addition reduction rule.

of the reduction of an addition constructor `Plus` in a language whose semantics uses both an environment and a store. In general, a reduction rule in DynSem has the form

```
    Rs ⊢ t1 :: Ws —a→ t2 :: Ws′
    where ps
```

and defines the reduction `t1` $\xrightarrow{a}$ `t2` from a term matching term pattern `t1` to a term obtained by instantiating term pattern `t2`, provided that the premises `ps` succeed. Reduction arrows are identified by their name and the type of their input (left) argument term. Arrows can be, and typically are, unnamed (i.e. use the default name) as is the case in Figure 1.

The *premises* of a reduction rule are of one of the following forms:

- A *reduction premise* is the (recursive) invocation `Rs ⊢ t :: Ws` $\xrightarrow{b}$ `t′ :: Ws′` of a reduction on the term `t`, matching the result against the term pattern `t′`. The premises in Figure 1 are of this form.
- An *equality or inequality premise* tests the equality or inequality of two terms. For example, in Figure 6, the `Ifz` constructor has two corresponding reduction rules that are disambiguated depending on the integer the first sub-term (the condition) reduces to.

`Ws′` are used to describe the context in which a particular reduction takes place. For example, the environment `Env E` and the store `Sto S` are semantics components in the rule in Figure 1.

In the conclusion of a rule, the left-hand side terms (`Rs`, `t1` and `Ws`) are patterns binding the meta-variables they contain, while the terms on the right-hand side (`t2` and `Ws′`) are instantiated to construct the result of reduction. This is reversed in reduction premises, where the left-hand side terms are term instantiations, while the right-hand side terms are binding patterns. A variable can only be bound once per rule.

DynSem rules are *statically typed* with respect to the declaration of the signature of term constructors, the declaration of typed (and optionally named) reduction arrows, and the declaration of semantic components.

Term Signature. DynSem rules define reductions on *terms* that represent the *abstract syntax trees* of a programming language. The abstract syntax tree constructors of a language are declared as an algebraic signature, as illustrated in Figure 2. Semantic rules typically use additional intermediate representations and final values, which are also represented using terms and should also be declared in the algebraic signature. For example, in a big-step semantics the values produced by reductions are typically not a subset of the abstract syntax, but are defined as a separate set of terms.

The base components of an algebraic signature are *sorts* and *constructors*. The sorts of the object language such as the sorts `Expr` and `Bind` in Figure 2 and the sort of the intermediate or output values such as the sort of values `V` in Figure 3 are declared in the

```
signature
  sorts Expr Bind
  constructors
    Num    : String → Expr
    Plus   : Expr * Expr → Expr
    Var    : String → Expr
    Bind   : String * Expr → Bind
    Let    : List<Bind> * Expr → Expr
    Fun    : String * Expr → Expr
    App    : Expr * Expr → Expr
    Box    : Expr → Expr
    Unbox  : Expr → Expr
    SetBox : Expr * Expr → Expr
    Seq    : Expr * Expr → Expr
    Ifz    : Expr * Expr * Expr → Expr
```

Figure 2 Signatures of the object language.

```
signature
  sorts V
  semantic—components
    Env = Map<String, V>
    Sto = Map<Int, V>
  constructors
    NumV  : Int → V  {implicit}
    ClosV : Expr * Env → V
    BoxV  : Int → V
  arrows
    Expr  ⟶ V
    List<Bind> -b⟶ Env
  native operators
    plusI   : Int * Int → Int
    str2int : String → Int
```

Figure 3 Auxiliary components.

```
signature
  constructors
    allocate : V  ⟶ Int
    write    : Int * V  ⟶ V
    read     : Int  ⟶ V
```

Figure 4 Meta-functions.

```
signature
  variables
    E : Env
    S : Sto
```

Figure 5 Variable Schemes.

corresponding `sort` section in the **`signature`**. Constructor declarations consist of a name, the sorts of the argument terms, and the result sort.

In addition to the declared sorts, the parametric sorts, **`List`** and **`Map`** can be used in signatures. Given a sort `X`, **`List`**`<X>` is the sort of lists of terms of sort `X`. The corresponding terms are `[]`, which denotes the empty list and `[hd|tl]`, which denotes the list with head element `hd` and tail `tl` (following Prolog list notation). Given two sorts `X` and `Y`, **`Map`**`<X,Y>` denotes the sort of finite maps from terms of sort `X` to terms of sort `Y`. The corresponding terms are `{}`, which denotes the empty mapping, `{x ↦ v, S}`, which extends the mapping `S` by associating `v` to `x`, and `S[l]`, which denotes the term associated with `l` in mapping `S`. These sorts can be used directly in constructor signatures such as the `Let` constructor in Figure 2 that takes a list of binders as first parameter.

Semantic Components. The specification of operational semantics using big-step rules defines reduction relations that involve not only abstract syntax terms but also other entities, known as *semantic components*, representing the state of a program under reduction, such as an environment or a store. DynSem distinguishes two types of semantic components based on how they are passed through the reduction rules: *read-only* and *read-write* semantic components. Terms of any sort may be used as semantic component after being declared as such. The **`semantic—components`** section in Figure 3 declares `Env` as a semantic component of sort **`Map`**`<String,V>`, mapping identifiers (strings) to values, and `Sto` as a semantic component of sort **`Map`**`<Int,V>`, mapping locations (integers) to values.

Semantic components appearing left of $\vdash$ (in `Rs`) are *read-only*. These components propagate downwards, i.e. are passed into premises, but do not propagate horizontally, i.e. changes introduced by one premise are not visible in the next premise. A typical example of a read-only component is an environment, mapping variables to values. Downward propagation corresponds to the lexical scope of variables. Components that appear right of :: are *read-write* semantic components. Since the value of such components is threaded through the computation, they appear on both the right and the left side of a reduction. A typical example of a read-write component is a store mapping locations to values. Threading a store is expressed as `e :: Sto st` $\longrightarrow$ `v :: Sto st'` when reduction of expression `e` into value `v` changes the corresponding store form `st` to `st'`. Store threading corresponds to the dynamic extent of store mappings.

Arrows. A reduction arrow defines a relation between terms, possibly involving semantics components. However, only the sorts of the source and target terms of an arrow have to be declared in its signature. The set of semantic components an arrow uses is inferred, as described in Section 3. The signature in Figure 3 declares two arrows, the main evaluation arrow $\longrightarrow$ from expressions `Expr` to values `V`, and the binder reduction $\xrightarrow{b}$ from lists of binders to environments. Arrow declarations can be overloaded on the input sort. That is, one can declare several arrows with the same name, as long as they have different input types. For example, in Figure 3 we could also use $\longrightarrow$ instead of $\xrightarrow{b}$ to denote the reduction from lists of binders to environments.

Native Operators and Meta Functions. The core rules of a semantics define reductions on the abstract syntax terms of the language under consideration. However, some aspects of the semantics we would like to delegate, i.e. not define directly. DynSem provides two mechanism for such delegation.

A *native operator* is a function that is defined externally to DynSem. The specification just records its signature. A typical use of native operators is the API for numeric operations. For example, Figure 3 declares the native operators `plusI` for adding integers and `str2int` for parsing strings as integer literals. From the perspective of the type system, native operators are term constructors. From the perspective of reduction, a native operator comes with an implementation that replaces its invocation with a term of the appropriate sort.

Sometimes we do want to define the semantics of an operation in DynSem, but we want to abstract over a frequently occurring pattern. A *meta-function*, declared as a constructor `C: ts` $\longrightarrow$ `t` (with a long arrow), introduces an auxiliary term constructor in combination with an arrow evaluating it to its result type. For example, Figure 4 declares the signature of meta-functions `allocate`, `write`, and `read`, which abstract over operations on the store, simplifying the definition of rules using the store.

3 Conciseness and Modularity

The previous section introduced the general form of DynSem reduction rules and the signatures that declare the structure of terms and arrows. However, as one can observe in Figure 1, full-fledged reduction rules can be rather verbose, requiring quite a bit of boilerplate code for the reduction of subterms and the propagation of semantic components. Such verbosity is bad since it hides the essence of a rule; which parts of the rule in Figure 1 are truly related to the addition? Worse, the explicit propagation of semantic components is harmful to modularity. When introducing a new feature to a language that requires a new semantic component, all existing rules need to be adjusted to propagate the new semantic component. It is common practice to reduce the verbosity of rules by means of informal conventions [14]. DynSem provides several features formalizing such conventions so that rules are unambiguous, yet have can be defined concisely and modularly. These features can be separated in two

```
Num(n) ⟶ str2int(n)

Ifz(0, e1, _) ⟶ e1

Ifz(n: Int, _, e2) ⟶ e2
where
  n ≢ 0

E ⊢ Var(x) ⟶ E[x]

[] : List(Bind) -b⟶ {}

[Bind(x, v: V) | lb] -b⟶ {x ↦ v, E}
where
  lb -b⟶ E

E ⊢ Let(E_b, e) ⟶ v'
where
  Env {E_b, E} ⊢ e ⟶ v'

Seq(v1 : V, e2) ⟶ e2
```

```
Plus(NumV(i1), NumV(i2)) ⟶ plusI(i1, i2)

E ⊢ f@Fun(_, _) ⟶ ClosV(f, E)

App(ClosV(Fun(x, e), E), v: V) ⟶ v'
where
  Env {x ↦ v, E} ⊢ e ⟶ v'

Box(e) ⟶ BoxV(allocate(e))

Unbox(BoxV(loc)) ⟶ read(loc)

SetBox(BoxV(loc), e) ⟶ write(loc, e)

allocate(v) ⟶ loc
where
  fresh ⇒ loc, write(loc, v) ⟶ _

write(loc, v) :: S ⟶ v :: Sto {loc ↦ v, S}

read(loc) :: S ⟶ S[loc] :: S
```

Figure 6 DynSem reduction rules for a functional language with references (boxes). The meta-functions `allocate`, `write`, and `read` define reusable abstractions over store operations that are used in the definition of the rules for boxes.

categories: (1) *coercions* that allow implicit transformation from one sort into another, and (2) *implicit propagation of semantics components* that allows the omission of those semantic components that are not affected by a rule.

3.1 Implicit Coercions

Implicit Constructors. Including the terms of one sort in another sort requires the introduction of an explicit constructor. For example, in order to use integers (sort `Int`) as values (sort `V`), a constructor `NumV: Int → V` is required, which should be applied to wrap each use of an integer as a value. In order to avoid such unnecessary constructor applications we allow unary constructors to be declared as `implicit`. An implicit constructor `C` with signature `S1 →S2` defines a coercion between sort `S1` and sort `S2`. That is, one can use a term of sort `S1` in a context where a term of sort `S2` is expected. The constructor `C` will then be automatically inserted to match the context. Therefore, since the `NumV` constructor is declared as `implicit` in the signature presented in Figure 3, we can omit it when the expected sort is known, such as in the first rule in Figure 6 for Num(n) where the right-hand side is an `Int` while a `V` is expected according to the ⟶ arrow declaration. Coercion expansion will introduce this implicit constructor, transforming the rule into

```
Num(n) ⟶ NumV(str2int(n))
```

Arrows as Coercions. In big-step operational semantics, most of the reductions rules have a similar scheme. That is, for a given constructor, reduce its subterms to values using a (recursive) invocation of the arrow and then combine these values to produce the final value of the term. If the reductions of the sub-terms behave uniformly with respect to semantic components, we can infer the sub-reduction by treating an arrow as declaration of an implicit coercion. Implicit arrow coercions are made explicit by the introduction of reduction premises.

If a term `t` of sort `A` appears in a position where sort `B` is expected, and if there is an arrow from `A` to `B`, an additional premise reducing `t` to a variable of sort `B` will be introduced. Conversely, if a pattern `t` of sort `B` is used in a position where a pattern of sort `A` is expected, and there is an arrow from `A` to `B`, the pattern is replaced with a fresh meta-variable, which is subsequently reduced in a new reduction premise to pattern `t`. For example making coercions explicit in the `Ifz` rules of Figure 6 leads to the following rules:

```
rules
  Ifz(e0, e1, _) ⟶ v1
  where
    e0 ⟶ NumV(0),
    e1 ⟶ v1
```

```
  Ifz(e0, _, e2) ⟶ v1
  where
    e0 ⟶ NumV(n),
    n ≢ 0,
    e2 ⟶ v1
```

In order to force coercion applications, one can also annotate variables in a pattern with their required type (e.g. the `n : Int` pattern in the second `Ifz` rule) enforcing a coercion from this type to the expected one to be introduced.

Variable Schemes. Big-step rules for operational semantics often require the use of several semantics components. In such a case, the use of meta-variables to represent the different semantics components may lead to an ambiguity about which semantics component is represented by a particular variable. Therefore, the name of a semantic component is required to appear before the term that represents its value, as in Figure 1. DynSem supports writing semantic components using just a variable, through a declaration of variable schemes. Given a semantics components `SC`, a variable scheme declaration such as `S : SC` reserves the name `S` and several extensions (e.g. `S`n where n is an integer or `S_x` where `x` is any identifier extension) to be used as meta-variables for the semantic component `SC`. These reserved variables do not require the explicit semantic component name annotation as shown in the variable rule in Figure 6 where `E` is inferred to be the semantic component `Env` given the variable declaration in Figure 5.

3.2 Implicit Semantic Components

While the rule for addition in Figure 1 explicitly passes the environment and store to its premises, the rules in Figure 6 do not mention semantic components or only selectively. DynSem uses implicit propagation of semantic components, as introduced in I-MSOS [16], to support the ommission of semantic components in rules that are not affected them. For example, the rule in Figure 1 can be rewritten to the `Plus` rule in Figure 6. Explication of coercions and semantic components automatically transforms this rule to the one in Figure 1. In addition to being more concise, this rule is also more modular, since it can be used in combination with language constructs that require different semantic components.

Semantic Component Dependency. In order to ensure consistency of the reduction relations, all reduction rules for a given arrow have to be extended with the same set of read-only and read-write semantic components. Moreover a particular semantic component can not appear both as a read-only and read-write component. Therefore, before explicating the semantic components in all rules, we first infer which semantic components are implicitly used by each arrow in the specification.

The semantic components used by an arrow do not only depend on the rules defining the arrow but also on the semantic components used by other arrows on which it depends. Let $\xrightarrow{r}$ denote a particular reduction arrow, the set of semantic components that have to appear in the expansion of the reduction rules defining $\xrightarrow{r}$ has to satisfy the following properties:

```
Env E_0 ⊢ [Bind(x, e_0) | lb] :: Sto S_0 -b-> {x ↦ v, E} :: Sto S_2
 where
   Env E_0 ⊢ e_0 :: Sto S_0 ⟶ v :: Sto S_1,
   Env E_0 ⊢ lb :: Sto S_1 -b-> E :: Sto S_2

E ⊢ Let(b_0, e) :: Sto S_0 ⟶ v' :: Sto S_2
 where
   Env E_0 ⊢ b_0 :: Sto S_0 -b-> E_b :: Sto S_1,
   Env {E_b, E} ⊢ e :: Sto S_1 ⟶ v' :: Sto S_2
```

Figure 7 Explicated rules for Bind and Let.

- All the semantic components appearing explicitly in one of the occurrences of arrow $\xrightarrow{r}$, either in the premise or conclusion of a rule, should appear in its expansion.
- A semantic component appearing in the expansion of an arrow $\xrightarrow{r}$, which is used in the premise of a rule defining arrow $\xrightarrow{r}$, should appear in the expansion of $\xrightarrow{r}$.

Explication of Semantic Components. Given these dependencies, we define an *explication* transformation on reduction rules, which make the use of semantic components explicit. Assume that for every reduction relation $\xrightarrow{r}$ we know its corresponding set of read-only $R_{\mathtt{r}}$ and read-write $W_{\mathtt{r}}$ semantic components according to the dependencies introduced above. Then, for each reduction rule, we apply the following transformations:

- Extend the set of read-only and read-write semantic components of the left-hand side of the conclusion with free variables for each of the components in $R_{\mathtt{r}}$ and $W_{\mathtt{r}}$
- Extend the set of read-only semantic components of each reduction premise using the read-only semantic components of the conclusion (the second dependency rule ensures that there is one) as required by its dependencies.
- Thread the read-write semantic components through the premises, as follows:
  - Initialize the context of read-write semantic components Sc with the read-write semantic components of the left-hand side of the conclusion
  - For each reduction premise:
    - Extend the semantic components on the left-hand side as required by the dependencies of the arrow of the premise
    - Use fresh variables to extend the semantic components on the right-hand side
    - Update the context Sc by replacing the semantic components used in the left-hand side with the corresponding fresh variables used in the right-hand side.
  - Extend the semantic components on the right-hand side of the conclusion using the context returned by premises processing.

Coercion explication followed by the semantic components explication turns the `Plus` rule from Figure 6 in the rule of Figure 1. It also introduces the `Env` and `Sto` semantic components in the binder reduction rules $\xrightarrow{b}$ (through the implicit use of the $\longrightarrow$ arrow in `v : V`) leading to the explicated rules for binder lists and `Let` in Figure 7. Note that, unlike with usual inductive rules, due to the threading of semantics components, the order of the premises matters when using implicit semantics components.

4 Interpreter generation

Our objective is to compile DynSem specifications to efficient interpreters. Their performance should be acceptable and they must lend themselves to later optimizations. To achieve this we translate a DynSem specification into an interpreter in Java. ASTs of object language programs are mirrored to instantiations of the generated classes which yields executable ASTs. Programs in the object language are then directly executable after parsing. Executable ASTs are simpler to reason about than bytecode interpreters, since they preserve the program structure and since they lend themselves to local program optimizations. The runtime behavior of the generated executable ASTs resembles hand-written Java code which allows the JVM's JIT to recognize common patterns and optimize the running program.

In a nutshell, the generator maps signatures of the embedded language to classes in Java, arrow declarations to method stubs and general premisses to method bodies. Classes are generated into a class hierarchy dictated by the relation between sorts and between constructors and sorts. Reduction rule alternatives become callable in ancestor classes.

Signatures. A sort definition translates to an abstract Java class. Sorts related by a subtype relationship translate to Java classes in a subtype relation. A sort definition also derives a specialized Java implementation for a list of that specific sort.

$$\textbf{sort } S \implies \begin{cases} \textbf{abstract class } A_S \textbf{ extends } \mathit{AbstractNode} \,\{\,\ldots\,\} \\ \textbf{class } \mathit{List}_S \textbf{ implements List}\langle A_S\rangle \,\{\,\ldots\,\} \end{cases}$$

All generated list classes implement the $\textbf{List}\langle T\rangle$ interface and generated sort classes extend the framework-provided $\mathit{AbstractNode}$ class.

A constructor C of arity n and sort S derives a Java class C_n which extends the class A_S. Classes derived from non-nullary constructors form roots for program subtrees. The generation scheme is

$$C : S_1 * \ldots * S_n \to S_t \quad\implies\quad \textbf{class } C_n \textbf{ extends } A_{S_t}\{A_{S_1}\ \_1; \ldots; A_{S_n}\ \_n;\}$$

An explicated arrow declaration for $\xrightarrow{rel}$ produces a method declaration in the class corresponding to the arrow's source sort. The return type R_{A_T} of the method is a record of the target type and the read-write semantic component types:

$$R_1 \cdot \ldots \cdot R_{n_r} \vdash (S :: S_1 \cdot \ldots \cdot S_{n_s}) \xrightarrow{rel} (T :: S_1 \cdot \ldots \cdot S_{n_s})$$
$$\implies \begin{cases} \textbf{public } R_{A_T}\ rel(A_{R_1}, \ldots, A_{R_{n_r}}, A_{S_1}, \ldots, A_{S_{n_r}})\{\ \mathit{def}(S, rel)\ \} \text{ in } \mathit{target}(S) \\ \textbf{class } R_{A_T}\ \{A_T\ \_0; A_{S_1}\ \_1; \ldots; A_{S_{n_r}}\ \_n;\} \end{cases}$$

The class containing the method is given by the conclusion's source pattern:

$$\mathit{target}(t) = \begin{cases} \textbf{class } C_n & \text{if } t = C(x_1, \ldots, x_n) \\ \textbf{class } \mathit{List}_S & \text{if } t : \textbf{List}\langle S\rangle \\ \textbf{abstract class } A_S & \text{if } t : S, \text{ otherwise} \end{cases}$$

The generated method calls its ancestor or raises an exception if the containing class does not have an ancestor:

$$\mathit{def}(s, rel) = \begin{cases} \textbf{super}.rel(\ldots) & \text{if } s = C(x_0, \ldots, x_n) \\ \textbf{super}.rel(\ldots) & \text{if } s : T, T \neq \textbf{List}\langle T'\rangle \text{ and } \exists T'', T \leq T'' \\ \text{raise exception} & \text{otherwise} \end{cases}$$

term construction	$C_n(t_1, ..., t_n)$	$\Longrightarrow$ **new** $C_n(e_1, ..., e_n)$
list construction	$[t_1 \vert t_2]$ (: $\textbf{List}\langle T\rangle$)	$\Longrightarrow$ **new** $List_T(e_1, e_2)$
empty map	$\{\}$	$\Longrightarrow$ **new** $PersistentTreeMap()$
map extension	$\{t_1 \mapsto t_2, t_3\}$	$\Longrightarrow e_3.plus(e_1, e_2)$
map access	$t_1[t_2]$	$\Longrightarrow e_1.get(e_2)$
assignment	$t \Rightarrow v$ (: T)	$\Longrightarrow A_T\ v = e$
pattern match	$t \Rightarrow C(v_1, ..., v_n)$	$\Longrightarrow$ **if** $(e$ **instanceof** $C_n)$ { ... }
equality check	$t_1 = t_2$	$\Longrightarrow$ **if** $(e_1.equals(e_2))$ { ... }
relation	$t^*_{rs} \vdash t_s :: t_{sc} \xrightarrow{rel} v :: v_{tc}$ (: T)	$\Longrightarrow R_{A_T}\ \_v = e_s.rel(e^*_{rs}, e^*_{sc})$

Figure 8 Correspondence of DynSem premises in Java.

Reduction rules. A pre-processing step merges all rules defining reductions for the same constructor (e.g. the two rules for `Ifz` in Figure 6), common premises are factorized and an *or* combinator ($\vee$) is introduced to combine the different alternatives. The merging algorithm is similar to left-factoring as described by Pettersson [19]. A difference is that premise equivalence in DynSem is decided modulo alpha-equivalence. Non-trivial pattern matching on the right-hand side of reduction premises is factored out using a pattern matching premise `t⇒p`. When `p` is not a variable, we apply the following transformation on the premise:

```
Rs ⊢ t :: Ws ⟶ p :: Ws'   ⟹   Rs ⊢ t :: Ws ⟶ x :: Ws', x ⇒ p
```

with `x` a fresh variable. We use similar rules for non-trivial patterns in `Ws'`. A merged reduction rule has its conclusion in normal form and a single general premise. It derives a method into the class described above. The signature of the generated method respects the signature derived from the arrow declaration. The general premise derives the method body. Figure 8 gives the correspondence of DynSem premises to Java statements and expressions. The individual premises are combined either as a sequence or as alternatives. Premises that can fail, such as pattern matches, are always guarded. If at evaluation a guard fails, its alternative premise is evaluated. A successful general premise returns from the method. For each reduction rule $rs \vdash s :: sc \xrightarrow{rel} t :: tc$ **where** gp the weaving of Java statements is given by the function $gen(gp \cdot\ sup(s, rel), t)$:

$$\begin{aligned}
gen((p_1 \cdot p_2) \vee p_3, t) &:= \quad \textbf{if } gen_g(p_1) \textbf{ then } \quad gen_s(p_1)\ \textbf{;}\ gen(p_2, t)\\
&\qquad\qquad\qquad\quad\ \textbf{else} \quad gen(p_3, t)\\
gen(p_1 \cdot p_2, t) &:= \quad gen(p_1, t)\ \textbf{;}\ gen(p_2, t)\\
gen(\epsilon, t) &:= \quad \textbf{return}\ t\textbf{;}\\
gen(sup(s, rel), t) &:= \quad def(s, rel)
\end{aligned}$$

where $gen_g(p)$ and $gen_s(p)$ generate Java code according to Figure 8 for the guard and the successful case of premise p, respectively.

5 Related work

Definition of Standard ML. The implicit propagation of semantic components in DynSem is an implementation of the notation used in the *The Definition of Standard ML* [14] to define the dynamic semantics for a core of Standard ML. This core of Standard ML is defined, just like definitions in DynSem, in the style of natural semantics [10]. Although inspired by

I-MSOS [16], explication in DynSem is closer to the definition used by Milner et al. which defines an implicit order of premises and is applied to big-step style semantics. Following I-MSOS, rules in DynSem may also omit the unused read-only semantic components from rules, which are left implicit by Milner et al. *The Definition of Standard ML* uses a single reduction arrow symbol.

I-MSOS, MSOS & interpreter generation. Modular SOS (MSOS) [15] introduces relation arrows that have record values as arrow labels. This record value carries auxiliary entities to be propagated together in the label of the arrow. Labels can be composed but have to be explicitly mentioned on all uses of the relation arrow, regardless of whether the rule accesses the contents of the label or not.

I-MSOS [16], which is the inspiration for DynSem, improves on the modularity and conciseness of MSOS. It introduces the distinction between auxiliary entities (semantic components) that implicitly propagate either only downwards or are threaded through the premises of rules. Each I-MSOS specification has a translation to an MSOS specification by aggregating the semantic components on the arrow label. I-MSOS also supports multiple relation arrows, but in contrast to DynSem, using multiple arrows in the same rule propagates all of the semantic components to all of the arrows used together. As a special case of this difference DynSem also prevents propagation of auxiliary entities for ground terms.

I-MSOS and MSOS can derive specialized interpreters [1] in Prolog. In the naive generation strategy each reduction rule translates to a Prolog clause which calls a stepping predicate which searches for a next reduction in the program. The number of step inferences harms performance the interpreter. A refocusing strategy, following Danvy et al. [18], rewrites an MSOS specification to a specification in which each rule attempts to transitively completely evaluate intermediate values. This heuristic, speculating on locality of evaluation, significantly reduces the number of inferences and improves execution performance. Special rules are introduced to propagate abruptly terminated computations. Specialization of interpreters from MSOS involves left-factoring the rules as described by Pettersson [19]. Left-factoring is similar to the merging of rules in DynSem. While left-factoring eliminates only identical premises, merging in DynSem also eliminates mutually exclusive premises and premises which can be unified modulo alpha-renaming. The motivation for left-factoring is to reduce backtracking, merging additionally has the goal of grouping premises per constructor.

K Semantic Framework [20] is a mature language and toolchain for specification of dynamic semantics of programming languages. K has been applied to production-size languages such as C [7] and Java [4]. Semantics in K are given as rewrite rules. The homologue of semantic components in K are configurations. These consist of (nested) cells used to store the interpreted program and additional data structures. Configurations are automatically inherited into rules and rewriting takes place directly inside the cells of the configuration. Rules that do not mention the configuration automatically perform the rewrite inside the K (program) cell and resemble rewrite rules. Rules that explicitly mention the configuration resemble context-sensitive reduction rules. Cells in the configuration that are not accessed are left unchanged and implicitly propagated. Cells are only propagated from input to output since K rules do not have premises.

K requires that the syntax definition of the object language be embedded in the K specification. Annotations on the syntax definition can define the evaluation order or strictness requirements. For example a *strict*(1) annotation on the conjunction expression states that the left subexpression has to be evaluated first and completely. Once this syntax is defined in K one can use a mix of object language concrete syntax and K syntax to match and build terms. Mixing of concrete object language syntax and DynSem can be obtained by assimilation following [21]. K specifications derive textual representations of the rules and graphical layouts for the configurations and cells to aid in documenting the object language. K specifications are compiled to rewriting logic in Maude [6]. Our previous micro-benchmarks [22] revealed that on big-step style specifications of the same language in DynSem and K, interpreters derived from DynSem are much faster than those derived from K.

PLT Redex [8] is an executable functional domain-specific language for semantic models. In Redex the semantics of a language is defined using context-sensitive reduction relations and meta-functions. Redex comes with a rich standard library of functions that can be used by semantic models. Its toolchain has facilities targeted at semantics prototyping such as randomized testing in the style of QuickCheck [5] and step-wise visualization of reductions [12]. Semantics can use either explicit substitution or environments and states. If environments and stores are used they have to be explicitly mentioned in every reduction relation. Redex supports ellipsis pattern placement for list matching and redex matching in program trees using the *in-hole* pattern, features that are not supported in DynSem. Readability of reduction relations in Redex is reduced by Racket's syntax [9] and by the need to explicitly mention semantic components. Programs are run by interpreting their reduction semantics in Redex. Racket's JIT compiler improves the runtime of interpreted programs but performance is still hampered by the redex lookup overhead and by non-deterministic rules. Performance of the interpreter is not an explicit goal of PLT Redex according to the published papers.

Implicit parameters in Scala exhibit similarities to semantic components in DynSem. Function parameters that are declared as implicit in the function definition may be omitted from the function application. Propagation of the implicit parameters is resolved by the compiler. The compiler prioritizes the inherited implicit parameters over the locally defined instances of the same type. Implicit parameters can be used to emulate the implicit read-only semantic components of DynSem but special care has to be taken to the mutability of the objects used. Persistent semantic components cannot be emulated with implicit parameters. While implicit parameters can be omitted from function applications they cannot be omitted from function definitions.

Monad transformers [13] allow definition of aggregated data structures with implicit packing and unpacking of components. The monad abstraction allows interpreters to be modularly composed. Examples are the state and IO monads. Monad transformers deliver implicit propagation similar to that of read-write semantic components in DynSem. The infectious propagation of the IO monad in Haskell resembles the upwards propagation of the semantic components in DynSem. In Haskell, like in the Scala case, the type signature of the function has to explicitly declare the monad as a function parameter. Both Haskell and Scala programs are less suitable for verification than more formal specification languages.

XSemantics [3] (the successor of Xtypes [2]) is a DSL for the specification of type systems for languages written in Xtext [23]. XSemantics is also applicable to implementation of interpreters. Dynamic semantics are specified in a syntax similar to deduction rules but with the rule conclusion preceding the rule premises. Multiple relation symbols can be declared and relation symbols can be overloaded. Only a single, mutable environment, can be used in a rule and there is no implicit propagation of the environment into the premises.

6 Future work

DynSem provides a good starting point for the further exploration of the integration of semantics engineering and language engineering. We plan to further develop the DynSem language and its accompanying toolset, and to investigate opportunities for abstraction in specifications. In particular, we plan to investigate the relation between static and dynamic binding of names in programs building on our foundational work on name resolution [17].

A goal for the generated interpreters is to obtain high performance execution engines. Techniques such as meta-tracing and dynamic compilation are proving successful for the optimization of custom built interpreters. We want to investigate whether such techniques can be generically applicable at the level of interpreter generation. Two other research avenues are to grow DynSem to cover a wide range of semantic styles and to automatically check that DynSem rules are type preserving.

Acknowledgments. We thank Peter Mosses and Martin Churchill for hosting the productive visit of Eelco Visser to Swansea that led to the DynSem language. We thank the participants of the Summer School on Language Frameworks in Sinaia, Romania in July 2012 — in particular Grigore Rosu, Robby Findler, and Peter Mosses — for inspiring us to address dynamic semantics in language workbenches. 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.

References

1 Casper Bach Poulsen and Peter D. Mosses. Generating specialized interpreters for modular structural operational semantics. In *Proceedings of the 23rd international symposium on Logic Based Program Synthesis and Transformation*, LOPSTR, 2013.

2 Lorenzo Bettini. A dsl for writing type systems for xtext languages. In Christian W. Probst and Christian Wimmer, editors, *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ 2011, Kongens Lyngby, Denmark, August 24-26, 2011*, pages 31–40. ACM, 2011.

3 Lorenzo Bettini. Implementing java-like languages in xtext with xsemantics. In Sung Y. Shin and José Carlos Maldonado, editors, *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pages 1559–1564. ACM, 2013.

4 Denis Bogdanas and Grigore Rosu. K-java: A complete semantics of java. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 445–456. ACM, 2015.

5 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

6 Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

7 Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. 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 533–544. ACM, 2012.

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

9 Matthew Flatt. Plt. reference: Racket. Technical report, Technical Report PLT-TR-2010-1, PLT Inc., 2010.

10 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.

11 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.

12 Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce 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.

13 Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In *POPL*, pages 333–343, 1995.

14 Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990.

15 Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004.

16 Peter D. Mosses and Mark J. New. Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49–66, 2009.

17 Pierre Neron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *Programming Languages and Systems*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer Berlin Heidelberg, 2015.

18 Danvy Olivier, Nielsen, and Lasse R. *Refocusing in reduction semantics*. 2004.

19 Mikael Pettersson. *Compiling Natural Semantics*, volume 1549 of *Lecture Notes in Computer Science*. Springer, 1999.

20 Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

21 Eelco Visser. Meta-programming with concrete object syntax. In Don S. Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315. Springer, 2002.

22 Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Neron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël D. P. 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 SLASH '14, Portland, OR, USA, October 20-24, 2014*, pages 95–111. ACM, 2014.

23 Xtext documentation. `http://www.eclipse.org/Xtext/documentation/2.6.0/Xtext`, 2014.