An Intermediate Language for Efficient Interpretation of Implicitly Modular Structural Operational Semantics

L. Thomas van Binsbergen
Royal Holloway,
University of London
ltvanbinsbergen@acm.org

Neil Sculthorpe
Nottingham Trent University
neil.sculthorpe@ntu.ac.uk

Adrian Johnstone
Royal Holloway,
University of London
a.johnstone@rhul.ac.uk

Elizabeth Scott
Royal Holloway,
University of London
e.scott@rhul.ac.uk

ABSTRACT

Structural Operational Semantics (SOS) is a well-established framework for specifying the semantics of programming languages. *Implicitly Modular* SOS (I-MSOS) is a variant of SOS that has recently been used as the basis for several formal specification languages, including CBS and DynSem. These specification languages are intended to be executable: it should be possible to generate a reference interpreter for a programming language from the inference rules that specify its semantics.

The topic of this paper is the efficient implementation of I-MSOS rules. Our approach is to compile I-MSOS rules from different specification languages to a common intermediate language (IML). IML is a lower level imperative language, designed to facilitate refactoring and optimisation. Sets of IML rules can then be compiled to produce a reference interpreter for the programming language specified by the original I-MSOS rules.

In this paper we motivate and present IML, together with a reference interpreter. We also define a compilation scheme from declarative I-MSOS rules to IML, and discuss the key optimisations IML supports.

1. INTRODUCTION

Structural Operational Semantics (SOS) [22] is a framework for specifying computational behaviour, where the behaviour of a program is specified by transition relations between program terms, defined inductively by inference rules and axioms. The transition relations used in SOS are typically augmented with auxiliary semantic entities, such as environments, stores or signals, which are used to model computational side-effects.

IFL '16 August 31 – September 2, 2016, Leuven, Belgium

ACM ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

Modular Structural Operational Semantics (MSOS) [15, 16] is a variant of SOS that allows the semantics of a programming construct to be specified independently of any semantic entities with which it does not directly interact. For example, the semantics of function application can be specified by MSOS rules without mentioning stores or output signals, even if the transition rules for other constructs in the same language specification do modify a store or emit output.

Implicitly Modular SOS (I-MSOS) [17] is a variant of MSOS that has a notational style similar to conventional SOS; it can be viewed as syntactic sugar for MSOS. Several formal specification languages are based on I-MSOS, including CBS [24], the specification language used by the Funcon Framework [9], and DynSem [25], the language for dynamic semantics used in the Spoofax Language Workbench [13]. CBS and DynSem specifications are intended to be executable, which enables programming-language designers to run test programs and validate their specifications as part of the design process.

The topic of this paper is the efficient implementation of sets of I-MSOS rules. In particular, our approach is to compile I-MSOS rules from existing specification languages to a common intermediate language (IML), and to apply optimisations to that intermediate language. The optimised IML specification is then used to generate an interpreter for the programming language specified by the original I-MSOS rules.

Our intermediate language consists of two levels:

- The IML Rule Format, which is a high-level abstract syntax for representing I-MSOS rules. Its purpose is to provide a target for compiling I-MSOS rules from existing formal specification languages.
- IML, which is a lower level imperative language that models I-MSOS rules as sequences of actions. Its purpose is to facilitate refactoring and optimisation.

Our approach has a number of benefits:

- The optimisations are independent of any source specification language and target host language, giving them strong potential for reuse.

$$
\begin{array}{lll}
C : \mathit{command} & ::= & C \ ; \ C \\
& | & R := E \\
& | & \textbf{print}\ E \\
& | & \textbf{while}\ E\ \textbf{do}\ C\ \textbf{od} \\
& | & D \\
D : \mathit{done} & ::= & \textbf{done} \\
E : \mathit{expr} & ::= & \textbf{plus}\ E\ E \mid \textbf{leq}\ E\ E \mid V \\
V : \mathit{value} & ::= & \textbf{false} \mid \textbf{true} \mid I \mid R \\
I : \mathit{integer} & ::= & \ldots \\
R : \mathit{reference} & ::= & \ldots
\end{array}
$$

Figure 1: Grammar for a small While language.

- For expressing and reasoning about optimisations, IML is a more suitable object language than the existing declarative specification languages, because evaluation order and control flow is explicit.
- I-MSOS rules from any I-MSOS–based formal specification language can be implemented (or even defined) simply by giving a translation to the IML Rule Format.

In summary, the contributions of this paper are as follows:

- We motivate and design IML, an intermediate language for modelling the execution of I-MSOS rules.
- We introduce the IML Rule Format, an abstract syntax for representing I-MSOS rules that forms an interface between specification languages and IML.
- We define a compiler translating instances of the IML Rule Format into IML.
- We present a semantics for IML in the form of a reference interpreter written in Haskell, which forms a basis for reasoning about the correctness of optimisations.
- We discuss optimisations over IML, in particular a left-factoring optimisation to reduce the negative impact of backtracking.

2. BACKGROUND

This section will provide an overview of SOS and I-MSOS, and introduce the running example we will use throughout the paper, which we have adapted from [3]. We assume the reader has some familiarity with SOS, for example [3, 22].

2.1 Structural Operational Semantics

Consider the WHILE language presented in Figure 1, which is a simple imperative language of commands and expressions. Note that special syntactic sort *done* would not typically be part of the source-language grammar, but is added to the semantic specification as a means of denoting a terminated computation.

Figures 2 and 3 contain SOS axioms and inference rules that define transition relations expressing the execution of commands and the evaluation of expressions, respectively. The conclusion of each rule contains an instance of the transition relation being defined, whereas the premise(s) of each rule may contain instances of the same or other transition relations. SOS rules may also have side conditions that restrict their applicability, or perform some meta-level operations (such as integer addition in Rule 7).

$$\boxed{\langle C, \sigma\rangle \xrightarrow{\alpha} \langle C, \sigma\rangle}$$

$$\frac{\langle C_1, \sigma\rangle \xrightarrow{\alpha} \langle C_1', \sigma'\rangle}{\langle C_1 \ ; \ C_2, \sigma\rangle \xrightarrow{\alpha} \langle C_1' \ ; \ C_2, \sigma'\rangle} \quad (1)$$

$$\langle \textbf{done} \ ; \ C_2, \sigma\rangle \xrightarrow{[]} \langle C_2, \sigma\rangle \quad (2)$$

$$\frac{\sigma \vdash E \Downarrow V}{\langle R := E, \sigma\rangle \xrightarrow{[]} \langle \textbf{done}, \sigma[R \mapsto V]\rangle} \quad (3)$$

$$\frac{\sigma \vdash E \Downarrow V}{\langle \textbf{print}\ E, \sigma\rangle \xrightarrow{[V]} \langle \textbf{done}, \sigma\rangle} \quad (4)$$

$$\frac{\sigma \vdash E \Downarrow \textbf{false}}{\langle \textbf{while}\ E\ \textbf{do}\ C\ \textbf{od}, \sigma\rangle \xrightarrow{[]} \langle \textbf{done}, \sigma\rangle} \quad (5)$$

$$\frac{\sigma \vdash E \Downarrow \textbf{true}}{\langle \textbf{while}\ E\ \textbf{do}\ C\ \textbf{od}, \sigma\rangle \xrightarrow{[]} \langle C \ ; \ \textbf{while}\ E\ \textbf{do}\ C\ \textbf{od}, \sigma\rangle} \quad (6)$$

Figure 2: SOS small-step rules for commands.

$$\boxed{\sigma \vdash E \Downarrow V}$$

$$\frac{\sigma \vdash E_1 \Downarrow I_1 \qquad \sigma \vdash E_2 \Downarrow I_2}{\sigma \vdash \textbf{plus}\ E_1\ E_2 \Downarrow I_3}\,(I_3 = I_1 + I_2) \quad (7)$$

$$\frac{\sigma \vdash E_1 \Downarrow I_1 \qquad \sigma \vdash E_2 \Downarrow I_2}{\sigma \vdash \textbf{leq}\ E_1\ E_2 \Downarrow \textbf{true}}\,(I_1 \leqslant I_2) \quad (8)$$

$$\frac{\sigma \vdash E_1 \Downarrow I_1 \qquad \sigma \vdash E_2 \Downarrow I_2}{\sigma \vdash \textbf{leq}\ E_1\ E_2 \Downarrow \textbf{false}}\,(I_1 \not\leqslant I_2) \quad (9)$$

$$\frac{}{\sigma \vdash R \Downarrow V}\,(V = \sigma(R)) \quad (10)$$

Figure 3: SOS big-step rules for expressions.

$$\boxed{\langle C, \sigma\rangle \overset{\alpha}{\Rightarrow} \langle D, \sigma\rangle}$$

$$\langle \textbf{done}, \sigma\rangle \overset{[]}{\Rightarrow} \langle \textbf{done}, \sigma\rangle \quad (11)$$

$$\frac{\langle C, \sigma\rangle \xrightarrow{\alpha} \langle C', \sigma'\rangle \qquad \langle C', \sigma'\rangle \overset{\beta}{\Rightarrow} \langle \textbf{done}, \sigma''\rangle}{\langle C, \sigma\rangle \overset{\alpha +\!\!+ \beta}{\Longrightarrow} \langle \textbf{done}, \sigma''\rangle} \quad (12)$$

Figure 4: SOS repeated small-step transitions.

As well as relating program terms, SOS transition relations may also involve *auxiliary semantic entities* that facilitate expressing the semantics of computational side effects. In this case, the rules make use of a *store* that contains the contents of mutable references, and an output signal containing a list of printed values. We have used Greek letters as meta-variables ranging over semantic entities (σ for stores and α, β for output signals), and capitalised Roman letters as meta-variables ranging over WHILE-language terms (as introduced in Figure 1).

The box at the top of each figure contains a template for the relation being defined. The template for the '$\rightarrow$' relation expresses that it relates a command/store pair to another command/store pair, as well as to an output signal. The first pair represents the program term and store contents before the transition, while the second pair represents the program term and store contents after the transition. The output signal represents the list of values printed during the transition (if any).

The template for the '$\Downarrow$' relation expresses that it relates an expression, store and value. This represents the expression being evaluated, the store contents during that evaluation, and the resulting value. Notice the different treatment of the store in the two relations: when executing commands the store can be updated, and hence there is an initial and resulting store in each rule; whereas when evaluating expressions the contents of the store can be read but not modified, so there need only be one store in the relation.

The rules defining the execution of commands are expressed in the small-step [22] style, whereas the rules defining evaluation of expressions are expressed in the big-step [12] style. To model a complete run of a WHILE program, we introduce a third relation (Figure 4) representing repeated iteration of the small-step relation.

2.2 Implicitly Modular SOS

A transition relation in SOS involves a *fixed* set of auxiliary semantic entities, distinguished by where they appear syntactically in the transition (such as before the turnstile, or above the arrow). The key distinction of I-MSOS is that *arbitrary* semantic entities may be included or omitted when writing a transition relation in a rule. Any omitted semantic entities are *implicitly propagated* between the premise(s) and conclusion. This allows semantic entities that do not interact with the programming construct being specified to be omitted from the rule, leading to clearer and more concise specifications. Furthermore, this makes I-MSOS rules *modular*: rules that mention some semantic entities can be combined with rules that mention different semantic entities, and the unmentioned entities are implicitly propagated.

I-MSOS classifies semantic entities into three kinds:

- *Read-only* entities, which represent inputs to a transition (e.g. an environment). If a read-only entity is omitted from a rule, then it is implicitly propagated from the conclusion to the premises (if any).
- *Write-only* entities, which represent lists of output from a transition (e.g. printed values). If a write-only entity is omitted from a rule, then the outputs of the premises are concatenated together to form the output of the conclusion.
- *Read-write* entities, which represent states that are mutated by a transition (e.g. a store). If a read-write entity is omitted from a rule, then it is "threaded" through the premises of the rule from left to right, with the initial value of the conclusion being the initial value of the first premise, and the resulting value of the final premise being the resulting value of the conclusion.

$$\frac{C_1 \rightarrow C_1'}{C_1 \mathbin{;} C_2 \rightarrow C_1' \mathbin{;} C_2} \tag{13}$$

$$\mathbf{done}; C_2 \rightarrow C_2 \tag{14}$$

$$\frac{\mathsf{env}(\sigma) \vdash E \Downarrow V}{\langle R := E, \mathsf{store}(\sigma)\rangle \rightarrow \langle \mathbf{done}, \mathsf{store}(\sigma[R \mapsto V])\rangle} \tag{15}$$

$$\frac{\mathsf{env}(\sigma) \vdash E \Downarrow V}{\langle \mathbf{print}\ E, \mathsf{store}(\sigma)\rangle \xrightarrow{\mathsf{out}([V])} \langle \mathbf{done}, \mathsf{store}(\sigma)\rangle} \tag{16}$$

$$\frac{\mathsf{env}(\sigma) \vdash E \Downarrow \mathbf{false}}{\langle \mathbf{while}\ E\ \mathbf{do}\ C\ \mathbf{od}, \mathsf{store}(\sigma)\rangle \rightarrow \langle \mathbf{done}, \mathsf{store}(\sigma)\rangle} \tag{17}$$

$$\frac{\mathsf{env}(\sigma) \vdash E \Downarrow \mathbf{true}}{\begin{array}{l}\langle \mathbf{while}\ E\ \mathbf{do}\ C\ \mathbf{od}, \mathsf{store}(\sigma)\rangle \rightarrow \\ \quad \langle C \mathbin{;} \mathbf{while}\ E\ \mathbf{do}\ C\ \mathbf{od}, \mathsf{store}(\sigma)\rangle\end{array}} \tag{18}$$

Figure 5: I-MSOS small-step rules for commands.

$$\frac{E_1 \Downarrow I_1 \qquad E_2 \Downarrow I_2}{\mathbf{plus}\ E_1\ E_2 \Downarrow I_3}\,(I_3 = I_1 + I_2) \tag{19}$$

$$\frac{E_1 \Downarrow I_1 \qquad E_2 \Downarrow I_2}{\mathbf{leq}\ E_1\ E_2 \Downarrow \mathbf{true}}\,(I_1 \leqslant I_2) \tag{20}$$

$$\frac{E_1 \Downarrow I_1 \qquad E_2 \Downarrow I_2}{\mathbf{leq}\ E_1\ E_2 \Downarrow \mathbf{false}}\,(I_1 \not\leqslant I_2) \tag{21}$$

$$\frac{}{\mathsf{env}(\sigma) \vdash R \Downarrow V}\,(V = \sigma(R)) \tag{22}$$

Figure 6: I-MSOS big-step rules for expressions.

$$\mathbf{done} \Rightarrow \mathbf{done} \tag{23}$$

$$\frac{C \rightarrow C' \qquad C' \Rightarrow \mathbf{done}}{C \Rightarrow \mathbf{done}} \tag{24}$$

Figure 7: I-MSOS repeated small-step transitions.

Interpreting I-MSOS rules operationally, these propagation schemes correspond to those of a reader, writer and state monad [27], respectively.

As examples, figures 5–7 present I-MSOS equivalents of the SOS rules in figures 2–4, respectively.

For consistency with SOS, an I-MSOS transition is written with read-only entities before the turnstile symbol, write-only entities above the arrow, and read-write entities paired

with the program term. However, unlike SOS, an I-MSOS rule can contain an arbitrary number of comma-separated entities in each of these positions. For example, a specification of a programming language may use additional write-only entities to model throwing exceptions or other forms of abnormal control-flow (see e.g. [23]). An SOS specification would have to change the *syntax* of the transition relation, and thence all existing axioms and inference rules, to permit such additional entities.

To allow multiple semantic entities in the same position to be distinguished, any semantic entities in an I-MSOS transition relation are tagged with a name (`store` and `out` in Figure 5; `env` in Figure 6). To enable further conciseness, I-MSOS allows the turnstile symbol to be omitted if there are no explicit read-only entities, and the pairing brackets to be omitted if there are no explicit read-write entities.

3. RULE FORMAT

This section presents the *IML Rule Format*. Then, in Section 4, we will present the IML intermediate language and show how to compile I-MSOS rules from the IML Rule Format into IML. To facilitate defining such a compiler, we accompany our definitions in this section with Haskell data types representing the abstract syntax of the IML Rule Format.

The purpose of this rule format is to serve as the starting point of a mechanical translation of I-MSOS rules into (efficient) interpreters. The rule format is sufficiently general to embed inference rules developed in the existing CBS and DynSem specification languages. Moreover, the format also generalises SOS specifications, for example those found in [3, 14], since any SOS specification can be translated into an I-MSOS specification by generating a name for each semantic entity.

3.1 Term Representation

When defining a formal semantics for a programming language, we distinguish between the *object language*, which is the programming language being defined, and the *meta-language*, which is the language being used to express the semantics of the object language. Here, the IML Rule Format is our meta-language, and we will continue to use the WHILE language as an example object language.

To distinguish between variables of the meta-language and variables of the object language, we refer to the former as *meta-variables*. We refer to program fragments of the object language as *terms*. Assuming a set of meta-variables X, and a set of term-constructor names F, we define terms as follows:

Definition 3.1. A *term* is either a meta-variable $x \in X$, or the application of a constructor $f \in F$ to n terms, denoted by $f(x_1, \ldots, x_n)$, where n is the arity of f. The set of all terms is denoted T. A term $t \in T$ is said to be *open* if it contains meta-variables, and *closed* if it does not.

An important subset of terms are *values*:

Definition 3.2. Values are terms whose outermost constructor is a *value constructor* $f \in F^V$, where $F^V \subset F$ is the set of all value constructors [8]. The set $F^C = F \setminus F^V$ contains all *computation constructors*. The set of all values is denoted V, which is a subset of T.

Example (open) terms for representing WHILE programs are $\mathit{while}(\mathit{leq}(\text{“}x\text{”}, X), \mathit{assign}(Y, 10))$, in which “$x$” and 10 are values (object-language variable and integer, respectively) and X, Y are meta-variables. An example of a closed term is $\mathit{assign}(\text{“}x\text{”}, 10)$.

The abstract syntax of terms can be represented using a Haskell data type:

```
type X = String
type F = String
data T = TVar X
       | TCons Bool F [T]
```

The Boolean flag indicates whether the constructor is a value constructor or not.

Formal specification languages typically make use of objects and operations from established mathematical frameworks, particularly in regard to providing values and operations on those values. For example, in the specification of the WHILE language, rules 19, 20, 15, and 22, perform integer addition, integer comparison, map insertion and map lookup, respectively.

Definition 3.3. A *value operation* is a variadic total or partial function of type $V \times \ldots \times V \to V$.

```
type O   = String
type VOP = [T] → T
```

The semantics of such value operations are assumed and not given as part of a specification. An interpreter for the object language has to provide definitions for any value operations used in the object language's specification. We use operator names `plus`, `less-or-eq`, `map-insert`, and `map-lookup` for the value operations used in the WHILE specification.

Definition 3.4. An *operator-expression* is either a value, or the application of a value operation o to a sequence of expressions of arbitrary length n, denoted by $o(e_1, \ldots, e_n)$.

```
type Exprs = [Expr]
data Expr  = Val T
           | Op O [Expr]
```

Pattern matching is a common concept in declarative programming languages. A term is matched against a pattern in order to deconstruct it, binding variables to the term's components. The IML Rule Format provides pattern matching at the meta-level, allowing object-language terms to be bound to meta-variables in a *meta-environment*. Following [8], we only allow values to be deconstructed; computation terms can only be matched to meta-variables.

Definition 3.5. A *pattern* is a value in which no meta-variable occurs more than once, formed only by the application of value constructors. The set of all patterns is denoted P. A term t *matches* a pattern p, resulting in the meta-environment γ, if and only if:

- The meta-variables occurring in p are bound in γ.
- Term t is obtained by replacing the meta-variables in p with the terms to which they are bound in γ.

Substitution is the process in which some meta-variables occurring in a term are replaced by the terms bound to those meta-variables in a meta-environment. If all meta-variables

in term t are bound in meta-environment γ then performing substitution on t with γ yields a closed term.

$$\begin{aligned}&\textbf{data } P = PVar\ X\\ &\quad\mid PCons\ F\ [P]\end{aligned}$$

3.2 Rule Representation

I-MSOS rules inductively define relations. We assume an abstract set of relation symbols R.

$$\textbf{type } R = String$$

The rules for WHILE in Figures 5 and 6 contain the relation symbols '$\rightarrow$' and '$\Downarrow$'.

The behaviour described by Rules (23) and (24) follow a general pattern of applying a transition repeatedly. This pattern is specified by the I-MSOS Rules (25) and (26), defining repetition for arbitrary relations $\rightarrow \in R$. We denote the iterative variant of a relation $\rightarrow$ by $\rightarrow^*$.

$$V_0 \rightarrow^* V_0 \tag{25}$$

$$\frac{T_0 \rightarrow T_1 \quad T_1 \rightarrow^* V_2}{T \rightarrow^* V_2} \tag{26}$$

Transitions may include auxiliary semantic entities, which provide contextual information and model computational side effects.

Definition 3.6. An *entity identifier* $eid \in E$ is associated with a *direction*: read-only, write-only or read-write, denoted by $\downarrow$, $\uparrow$, or $\updownarrow$ respectively.

The WHILE I-MSOS specification contains the read-write entity store, read-only entity env, and write-only entity out.

$$\begin{aligned}&\textbf{type } E = String\\ &\textbf{data } Dir = RO \mid RW \mid WO\end{aligned}$$

An entity reference is either a pair (eid, p), with $eid \in E$ and $p \in P$, or a pair (eid, t), with $t \in T$, depending on whether the value of entity is *accessed* or *updated*. Read-write entities are always referenced on both sides of a relation symbol, and thus come in triples (eid, t, p) (premise) or (eid, p, t) (conclusion).

$$\begin{aligned}&\textbf{type } Ac^{\downarrow} = (E, P)\\ &\textbf{type } Up^{\downarrow} = (E, T)\\ &\textbf{type } Ac^{\uparrow} = (E, P)\\ &\textbf{type } Up^{\uparrow} = (E, T)\\ &\textbf{type } Ac^{\updownarrow} = (E, P, T)\\ &\textbf{type } Up^{\updownarrow} = (E, T, P)\end{aligned}$$

The *conclusion* of a rule is a sextuple (p, r, t, ro, rw, wo); with $t \in T$; $\rightarrow \in R$ and r either $\rightarrow$ or $\rightarrow^*$; ro a sequence of read-only accesses; rw a sequence of read-write accesses; and wo a sequence of write-only updates. Pattern p must be a compound pattern $f(p_1, \ldots, p_n)$, with f a computation constructor, following the *value-added tyft* format given in [8].

$$\begin{aligned}&\textbf{data } Concl = Concl\ F\ [P]\ Rel\ T\ [Ac^{\downarrow}]\ [Ac^{\updownarrow}]\ [Up^{\uparrow}]\\ &\textbf{data } Rel = Rel\ R\ Rep\\ &\textbf{data } Rep = NoRep \mid Rep \quad \text{-- either } \rightarrow \text{ or } \rightarrow^*\end{aligned}$$

The *premise* of a rule is a sextuple (t, r, p, ro, rw, wo); with $t \in T$; $p \in P$; $\rightarrow \in R$ and r either $\rightarrow$ or $\rightarrow^*$; ro a sequence of read-only updates; rw a sequence of read-write updates; and wo a sequence of write-only accesses.

$$\begin{aligned}&\textbf{type } Prems = [Prem]\\ &\textbf{data } Prem = Prem\ T\ Rel\ P\ [Up^{\downarrow}]\ [Up^{\updownarrow}]\ [Ac^{\uparrow}]\end{aligned}$$

The premise of Rule 15 is prefixed by read-only update '$\mathsf{env}(\sigma)$', while the conclusion of Rule 22 is prefixed by read-only access '$\mathsf{env}(\sigma)$'. In both cases the sequences of read-write updates/accesses and write-only updates/accesses is empty. The conclusion of Rule 15 contains a read-write access $(\mathsf{store}, \sigma, \sigma)$.

I-MSOS rules may also contain *side-conditions*, such as '$I_1 \leqslant I_2$' in Rule 20.

Definition 3.7. A *side-condition* is a pair (e, p), denoted as $e \rhd p$, with e an operator-expression and p a pattern. A side-condition $e \rhd p$ holds if and only if e evaluates to a value matching the pattern p.

$$\begin{aligned}&\textbf{type } SideCons = [SideCon]\\ &\textbf{data } SideCon = SideCon\ Expr\ P\end{aligned}$$

3.2.1 IML rule format

We define the IML Rule Format using index sets I and J, with $0 \notin I$.

$$\frac{\{t_i \rightsquigarrow_i p_i \ :\ i \in I\} \quad \{e_j \rhd p_j \ :\ j \in J\}}{f(w_1, \ldots, w_n) \rightsquigarrow_0 t} \tag{27}$$

Arrow $\rightsquigarrow_0$ abbreviates a conclusion and each arrow $\rightsquigarrow_i$, with $i \in I$, abbreviates a premise. Each conclusion and premise contains an arbitrary relation symbol $\rightarrow \in R$ or its iterative version $\rightarrow^*$. For clarity we have omitted any entity references from the conclusion and premises. Whenever an entity is referenced in any conclusion or premise, it must be referenced in the conclusion and all premises. Possibly decorated identifiers t ranges over terms; e over operator-expressions; w and p over patterns; and f over computation constructors.

$$\textbf{data } Rule = Rule\ Concl\ Prems\ SideCons$$

Definition 3.8. An IML specification is a pair (D, M), where D is a set of rules such that each rule is an instance of the IML rule format, and M is a function $E \rightarrow Expr$ defining an initial value for each read-write and read-only entity. The initial value for entity $eid \in E$ is obtained by evaluating the operator-expression $M(eid)$. Every entity identifier occurs consistently as either a read-only, write-only or read-write entity in the rules D.

$$\begin{aligned}&\textbf{type } Spec_{\text{HI}} = [Either\ EntDecl\ Rule]\\ &\textbf{data } EntDecl = RODecl\ E\ Expr\\ &\qquad\mid RWDecl\ E\ Expr\end{aligned}$$

Figure 8 shows rules 13, 14, 17, 18, 19, 20, and 21 as instances of the rule format. To increase conciseness we omit the turnstile if a premise/conclusion has no read-only updates/accesses, and the pairing brackets if there are no read-write updates/accesses. Note that although *done* is not a value of the object language WHILE, it is a value constructor at the meta-level.

4. INTERMEDIATE LANGUAGE

We introduce the I-MSOS intermediate language (IML) and describe its semantics by means of a reference interpreter (Section 4.1). We then give a translation from the IML Rule Format to IML (Section 4.2).

$$\frac{C_1 \to C_1'}{seq(C_1, C_2) \to seq(C_1', C_2)} \quad (28)$$

$$seq(done, C_2) \to C_2 \quad (29)$$

$$\frac{\mathsf{env}(\sigma) \vdash E \Downarrow false}{\langle while(E, C), \mathsf{store}(\sigma)\rangle \to \langle done, \mathsf{store}(\sigma)\rangle} \quad (30)$$

$$\frac{\mathsf{env}(\sigma) \vdash E \Downarrow true}{\langle while(E, C), \mathsf{store}(\sigma)\rangle \to \langle seq(C, while(E, C)), \mathsf{store}(\sigma)\rangle} \quad (31)$$

$$\frac{E_1 \Downarrow I_1 \quad E_2 \Downarrow I_2 \quad \texttt{plus}(I_1, I_2) \rhd I_3}{plus(E_1, E_2) \Downarrow I_3} \quad (32)$$

$$\frac{E_1 \Downarrow I_1 \quad E_2 \Downarrow I_2 \quad \texttt{is-leq}(I_1, I_2) \rhd true}{leq(E_1, E_2) \Downarrow true} \quad (33)$$

$$\frac{E_1 \Downarrow I_1 \quad E_2 \Downarrow I_2 \quad \texttt{is-leq}(I_1, I_2) \rhd false}{leq(E_1, E_2) \Downarrow false} \quad (34)$$

Figure 8: Selection of IML rules.

```
type Spec_LO   = [Either EntDecl TransDecl]
data TransDecl = Trans R F [Stmts]
type Stmts     = [Stmt]
data Stmt      = Branches [Stmts]
               | PMArgs   [P]
               | PM       Expr P
               | Commit   T
               | Single   R T X  Label
               | Many     R T X  Label
               | ROGet    E X
               | ROSet    E Expr Label
               | RWGet    E X    Label
               | RWSet    E Expr Label
               | WOGet    E X    Label
               | WOSet    E Expr
type Label     = Int
```

Figure 9: IML abstract syntax.

4.1 IML Grammar and Semantics

The abstract syntax of IML is presented in Figure 9. The types X, F, T, P, *Expr*, R, *Rep*, and *EntDecl* are shared with the IML Rule Format defined in Section 3. IML is a low-level imperative language, with familiar constructs for declarations, expressions and statements.

We define the semantics of IML as a set of *semantic functions* that form a reference interpreter for IML specifications. The purpose of this interpreter is presentational clarity, rather than efficient execution. Efficient compilation of IML to generate specialised interpreters is the subject of future work. The presentation of the interpreter in this paper covers the main aspects of IML, but is not exhaustive. The complete interpreter is available online [1].

4.1.1 Declarations and transactions

→ TRANSACTION FOR: while

pm-args(E, C);	*pm-args*(E, C);
rw-get$(\mathsf{store}, \sigma, 0)$;	*rw-get*$(\mathsf{store}, \sigma, 0)$;
ro-set$(\mathsf{env}, \sigma, 1)$;	*ro-set*$(\mathsf{env}, \sigma, 1)$;
single$(\text{“}\Downarrow\text{”}, E, X_0, 1)$;	*single*$(\text{“}\Downarrow\text{”}, E, X_0, 1)$;
pm$(X_0, false)$;	*pm*$(X_0, true)$;
rw-set$(\mathsf{store}, \sigma, 0)$;	*rw-set*$(\mathsf{store}, \sigma, 0)$;
COMMIT: done	**COMMIT: seq**$(C,$ **while**$(E, C))$

Figure 10: An IML transaction for Rules 30 and 31.

Declarations for read-only and read-write entities determine their default values. From the entity declarations we obtain a value of type *Record*, mapping entity identifiers to values and directions.

```
type Record = Map E (T, Dir)
```

Write-only entities are not declared; their default value is the empty list. To explain the third type of declaration, *TransDecl*, we we borrow terminology related to (database) *transactions*.

Let a transaction be a procedure that reads from, and writes to, some external state, by performing a series of statements. Each statement may fail, causing the transaction to be *aborted*. As part of an abort, any changes made to the external state are undone. When all statements of a transaction have been successfully executed, the changes to the external state are *committed*.

An I-MSOS rule is loosely analogous to a transaction if we consider a meta-environment (binding meta-variables to terms) and the value of semantic entities as the external state of a transaction. The different components of a rule - conclusion, premises and side-conditions - correspond to one or more statements of the transaction. Abortion of a transaction indicates that the rule is not applicable.

An IML transaction is defined for a relation symbol $\to$ and a (computation) constructor f. The transaction's body is an arbitrary number of *branches*, each a sequence of statements generated for an IML rule (see Section 4.2). From the transaction declarations of a specification we obtain a map, mapping pairs of relation symbols and constructors to branches.

```
type TransMap = Map (RSymb, F) [Stmts]
```

Figure 10 shows a transaction generated for IML rules (30) and (31). In the concrete syntax representation of transactions we use lower-case hyphenated constructor names. Some constructors are left implicit, e.g. a meta-variable may be both a term, a value, a pattern, and an expression, depending on where it occurs.

4.1.2 Executing transactions by backtracking

The purpose of a transaction is to perform a (single) transition $t \to t'$, where we refer to t and t' as the (closed) *input term* and *output term* respectively. The transaction succeeds if it was generated from a specification that contains a rule applicable to t. Term t must therefore be of the form $f(t_1, \ldots, t_n)$, where we refer to $t_1, \ldots, t_n$ as the *input arguments*.

To determine whether a transition is possible, a transaction executes statements, with one of three outcomes [1]:

[1] $\bot$ is *not* Haskell's undefined, sometimes also typeset as $\bot$

$$\textbf{data}\ Res_{\text{STMT}} = Done \mid \bot \mid Commit\ (T, Ents)$$

Each statement is generated from a particular rule (or multiple rules after left-factoring). Outcome $\bot$ indicates that the rule from which the statement originates is not applicable in the current state and context of the transaction (defined later). Outcome *Commit* shows the rule is applicable and provides the output term of the resulting transition. Component *Ents* provides the values of read-write and write-only entities: the *additional output* of the transition. Outcome *Done* indicates the successful execution of a statement and is neutral with respect to rule applicability.

The context of transaction is formed by a *TransMap*, entity values as *additional input*, and input arguments.

$$\textbf{type}\ Ctxt_T = (TransMap, Ents, [T])$$

Single and *Many* statements are for executing the premises of an inference rule. Their respective semantics is described in detail in Section 4.1.3. The unique *label* associated with a premise enables the *getter* and *setter* statements to obtain (or provide) entity values specific to a premise. By using labels we have avoided the introduction of code blocks, and as a result IML transactions have simple control-flow. The type Δ stores entity values specific to a certain label.

$$\textbf{type}\ \Delta = Map\ Label\ Ents$$

The state of a transaction consists of a meta-environment, binding meta-variables to (closed) terms, a value of type Δ, and a list of labels.

$$\begin{aligned}&\textbf{type}\ State_T = (MetaEnv, \Delta, [Label])\\ &init\_st \qquad\quad = (\{\,\}, \{\,\}, [\,])\end{aligned}$$

The list of labels is for remembering the order in which premises have been executed, required for implicit entity propagation (also discussed in Section 4.1.3). The initial state *init_st* consists of an empty meta-environment, an empty Δ, and an empty list of labels.

The semantics of statements is captured by the type [2]:

$$\textbf{type}\ Sem_{\text{STMT}} = Ctxt_T \to State_T \to (Res_{\text{STMT}}, State_T)$$

The semantics of abortion is defined as follows:

$$\begin{aligned}&sem\_abort :: Sem_{\text{STMT}}\\ &sem\_abort\ ctx\ st = (\bot, st)\end{aligned}$$

The semantics of a sequence of statements is to execute the sequence in order, until a statement commits or aborts.

$$\begin{aligned}&sem\_stmts :: [Sem_{\text{STMT}}] \to Sem_{\text{STMT}}\\ &sem\_stmts\ [\,]\ \_\ \_ \qquad\quad = error\ \texttt{"branch without commit"}\\ &sem\_stmts\ (s : ss)\ ctx\ st = \textbf{case}\ s\ ctx\ st\ \textbf{of}\\ &\quad (Done, st') \to sem\_stmts\ ss\ ctx\ st'\\ &\quad r \qquad\qquad\ \to r\end{aligned}$$

When a rule is not applicable to an input term t, another rule may still be applicable. Rules are selected by backtracking between branches. The flow of control is to jump back to the last branching location, executing another branch with the same state as the branch that failed.

$$\begin{aligned}&sem\_branches :: [[Sem_{\text{STMT}}]] \to Sem_{\text{STMT}}\\ &sem\_branches\ [\,] \qquad\quad = sem\_abort\\ &sem\_branches\ (b : bs) = sem\_branch\\ &\quad \textbf{where}\ sem\_branch\ ctx\ st = \textbf{case}\ sem\_stmts\ b\ ctx\ st\ \textbf{of}\\ &\qquad\qquad (Done, \_) \to error\ \texttt{"branch without commit"}\\ &\qquad\qquad (\bot, \_) \qquad\ \to sem\_branches\ bs\ ctx\ st\\ &\qquad\qquad r \qquad\qquad\ \to r\end{aligned}$$

The result of the first successfully executed branch is the overall result. Interpreters generated from IML specifications are *not* complete in the sense that not every possible transition is actually performed. We assume that specifications are *deterministic*. Interpreters may be generalised however, for example using a list of successes [26].

4.1.3 Implicit entity propagation

In [17], Mosses and New suggest multiple ways for I-MSOS rules to implicitly propagate the values of unmentioned semantic entities. From their suggestions we extract the following possible strategies:

(1) Execute premises in the order in which they appear

(2) Alternatively, consider all possible orders

(a) Given an order in which premises are executed, thread unmentioned read-write entities according to the 'state convention' [14], and concatenate unmentioned write-only entities in that order

(b) Alternatively, do not allow 'unobserved side-effects' when a rule has more than one premise. Every entity value in the additional output of a premise must be obtained by a getter statement. The order in which premises are executed is irrelevant in this case

Strategy (2) may be realised using the backtracking facilities discussed previously, generating branches for each valid[3] permutation of a rule's premises. However, we expect it is not likely for a branch to fail because of the order in which premises are executed. The cost of backtracking are high if a branch fails for any other reason, as possibly many permutations are to be considered. For this pragmatic reason we choose (1). Further, IML implements both strategies (a) and (b), although here we only discuss (a). Which of (a) and (b) is chosen is determined on a rule-by-rule basis when translating I-MSOS into the IML Rule Format.

The following inference rules show how values of unmentioned read-only, read-write, and write-only entities are propagated for rules with $n \geqslant 0$ premises. All t_i are arbitrary terms, p_i patterns, and $\to_i$ relation symbols (or their iterative variation), with $0 \leqslant i \leqslant n$.

$$\frac{\mathsf{ro\text{-}ent}(\rho) \vdash t_1 \to_1 p_1 \quad \ldots \quad \mathsf{ro\text{-}ent}(\rho) \vdash t_n \to_n p_n}{\mathsf{ro\text{-}ent}(\rho) \vdash t_0 \to_0 p_0} \tag{35}$$

$$\frac{\begin{array}{c}\langle t_1, \mathsf{rw\text{-}ent}(\sigma_0)\rangle \to_1 \langle p_1, \mathsf{rw\text{-}ent}(\sigma_1)\rangle\\ \ldots\\ \langle t_n, \mathsf{rw\text{-}ent}(\sigma_{n-1})\rangle \to_n \langle p_n, \mathsf{rw\text{-}ent}(\sigma_n)\rangle\end{array}}{\langle t_0, \mathsf{rw\text{-}ent}(\sigma_0)\rangle \to_0 \langle p_0, \mathsf{rw\text{-}ent}(\sigma_n)\rangle} \tag{36}$$

$$\frac{t_1 \xrightarrow{\mathsf{wo\text{-}ent}(\alpha_1)}_1 p_1 \quad \ldots \quad t_n \xrightarrow{\mathsf{wo\text{-}ent}(\alpha_n)}_n p_n}{t_0 \xrightarrow{\mathsf{wo\text{-}ent}(\alpha_1 +\!\!+ \ldots +\!\!+ \alpha_n)}_0 p_0} \tag{37}$$

The empty list is written on the conclusion's relation symbol, when $n = 0$ in Rule (37).

[2] The type is in close correspondence with a combination of a *Reader* and *State* monad. We have decided against implicitly propagating context and state using a monad, for reasons of clarity

[3] Premises may have dependencies

Implicit entity propagation is considered part of the semantics of IML transactions, rather than relying on a procedure to explicate entity references. As a result, IML transactions are as modular as I-MSOS rules and have the same meaning in isolation and in combination with other rules.

We establish invariants on values of type *Ents* and Δ, helping to ensure entity values are propagated according to the above rules:

- In the *context* of a transition, an *Ents* contains the values of all read-only and read-write entities that are declared in the specification
- When occurring in the result of a transition, an *Ents* contains only those values of read-write entities that have been modified by the transition, together with the values of zero or more write-only entities
- Before a premise with label l has been executed, an entry $l \mapsto es$, in a value of type Δ, indicates that the entity values in es form additional input to the premise with label l, and contains only values of read-only and read-write entities.
- After a premise with label l has been executed, an entry $l \mapsto es$ indicates that es are the *unobserved side-effects* of executing the premise with label l. A getter with label l observes a side-effect, binding the entity value and removing it from Δ. The values in es are for read-write and write-only entities only.

As an example of 'observing a side-effect', we give the semantics of *wo-get*. A possible entry $[eid \mapsto a] \in es$, with $[l \mapsto es] \in \delta$ and for some a, is removed from es by applying $delete_\Delta$.

```
sem_woget :: E → X → Label → Sem_STMT
sem_woget eid x l _ (γ, δ, σ) = case lookup_Δ (eid, l) δ of
  Nothing → (γ [x ↦ []], δ, σ)   -- defaults to empty list
  Just v  → (γ [x ↦ v], delete_Δ (eid, l) δ, σ)
```

We use the non-Haskell shorthand $\gamma\ [x \mapsto v]$ to extend meta-environment γ with the new binding $x \mapsto v$. Setters use $insert_\Delta :: (E, Label) \to (T, Dir) \to \Delta \to \Delta$ to provide additional input for a future premise (we assume the meanings of $insert_\Delta$ and $lookup_\Delta$ are clear from their names).

4.1.4 Executing transitions

We use a helper function $prop\_in :: [Ents] \to Ents$ for computing the additional input to a transition. The function merges the components of its first argument by applying a binary map-union operator that is right-biased with respect to both read-only and read-write entities. Similarly, helper function $prop\_out :: [Ents] \to Ents$ merges the additional output of one or more premises by applying a map-union operator that is right-biased with respect to read-write entities and concatenates any write-only entities using list-append operator $+\!\!+$.

The semantics of performing a (possibly repeated) transition is implemented by functions *outer* and *inner*. Function *outer* performs substitution (function $subs :: MetaEnv \to T \to T$) on a term to create the input term ct of the transition, before calling *inner* to execute the transition. If the result is an output term ct_1 and an $Ents\ es_1$ then the state is updated as follows: extend the meta-environment with a new binding $x \mapsto ct_1$, replace the old es in δ with es_1 under label l, add label l to the list of executed premises.

```
outer :: Rep → R → T → X → Label → Sem_STMT
outer rp r t x l (tm, es, _) (γ, δ, σ) =
  case inner rp r ct (tm, es-, []) of
    Commit (ct1, es1) →
        (Done, (γ [x ↦ ct1], override_Δ l es1 δ, σ ++ [l]))
    _ → sem_abort ctx (γ, δ, σ)
  where
    ct    = subs γ t
    es-   = prop_in (es : map (flip obtain_Δ δ) (σ ++ [l]))
```

A transition is executed by executing the branches stored in *TransMap tm* under relation symbol r and computation constructor f. The context of the surrounding transaction contains input arguments *args*.

```
inner :: Rep → R → T → Ctxt_T → Res_STMT
inner _ _ (TVar _) _ = error "open term"
inner rp r t0@(TCons isVal f args) (tm, es0, _) =
  case sem_branches (tm ! (r, f)) (tm, es0, args) init_st of
    Commit (t1, es1)
      → case rp of
          NoRep → Commit (t1, es1)
          Rep   → rec t1 es1
    _ → case rp of
          NoRep → ⊥
          Rep   → if isVal then Commit (t0, { })
                           else ⊥
  where rec t1 es1 = case inner rp r t1 (tm, es1-, []) of
          Commit (t2, es2) → Commit (t2, es2-)
            where es2- = prop_out [es1, es2]
          where es1- = prop_in [es0, es1]
```

If there is a successful branch returning (t_1, es_1), this is the result of performing a single transition. If there is no such branch, the single transition failed and the statement aborts.

Repeated transitions are implemented by greedily applying inference Rules (25) and (26). If there is no transition possible, and t_0 is a value, we apply Rule (25), and return $(t_0, \{\})$. If a single transition is possible, we apply Rule (26) by making a recursive call to *inner*. The implicit propagation of entities between premises and conclusion of Rule (26) is handled by applying *prop_in* and *prop_out*. The semantics of *Single* and *Many* are defined in terms of *outer*.

```
sem_single, sem_many :: R → T → X → Label → Sem_STMT
sem_single = outer NoRep
sem_many   = outer Rep
```

A commit action returns $Commit\ (ct, es)$ (together with unmodified state *st*), where *ct* is the closed term acquired by performing substitution on the given term t with the current meta-environment γ. The *Ents es* contains the merger of all unobserved side-effects of any premises, together with the additional output provided by setters with label 0.

```
sem_commit :: T → Sem_STMT
sem_commit t ctx st@(γ, δ, σ) = (Commit (ct, es), st)
  where ct = subs γ t
        es = prop_out (map (flip obtain_Δ δ) (σ ++ [0]))
```

4.1.5 Evaluating expressions

The result of evaluating an (operator-)expression may be *Just* a (closed) value term or *Nothing*. *Nothing* is returned if, for example, a partial value operation is applied to values outside its domain. A meta-environment is required to perform substitution on values.

```
type Sem_EXPR = MetaEnv → Maybe T   -- closed value
```

An expression is either a (possibly open) value or the application of a value operation to expressions. In the former case, substitution is applied to the given value.

```
sem_val :: T {-value -} → Sem_EXPR
sem_val t γ = subs γ t
```

IML requires the definitions of value operations to be available in some library accessible by implementations of IML.

```
sem_op :: O → [Sem_EXPR] → Sem_EXPR
sem_op op es γ
  | all isJust margs = opApp op (map fromJust margs)
  | otherwise        = Nothing
  where margs = map ($γ) es
        opApp "is-leq" args = ...
        opApp "is-int" args = ...
        ...
```

4.1.6 Pattern matching

The left-hand side of a rule's conclusion is implemented by the *pm-args* statement. Its semantics is to match the input arguments against a sequence of patterns, and extend the current meta-environment with any bindings resulting from this matching. We rely on an unspecified function *matches* :: $[T] \to [P] \to$ *Maybe MetaEnv* to perform pattern matching. We write $\gamma_1\ [\gamma_2]$ to mean the right-biased union of two meta-environments.

```
sem_pm_args :: [P] → Sem_STMT
sem_pm_args pats ctx@(_,_,args) st@(γ1, δ, prod) =
  case matches args pats of
    Nothing → sem_abort ctx st
    Just γ2 → (Done, (γ1 [γ2], δ, prod))
```

Side-conditions and premises require matching a specific term to a pattern, which is executed by *pm* statements. The first argument of *pm* is an expression that requires evaluation. A *pm* statement aborts if evaluating the expression does not yield a value matching the pattern. Any new bindings are added to the current meta-environment otherwise.

```
sem_pm :: Sem_EXPR → P → Sem_STMT
sem_pm e p ctx st@(γ1, δ, prod) = case e γ1 of
  Nothing → sem_abort ctx st
  Just v  → case matches [v] [p] of
    Nothing → sem_abort ctx st
    Just γ2 → (Done, (γ1 [γ2], δ, prod))
```

4.1.7 Executing queries

A semantic specification is not executable in itself; it does not specify what to do with any input nor where that input is coming from. We need a way to to tie up a specification in an execution environment. We assume an execution environment intends to send one or more interpretation requests to interpreters generated from specifications. For this purpose we add *queries* to IML. An IML *program* is then an IML specification and an arbitrary number of queries.

```
data Program = Program Spec_LO Queries
type Queries = [Query]
data Query   = Query R T Rep
```

Each query is a pair $(\Rightarrow, t)$, where $\Rightarrow$ is either $\to$ or $\to^*$, $\to \in R$, and t a closed term. A query is executed by invoking *single* or *many*, and committing the resulting output term.

```
sem_query :: R → T → Rep → Sem_STMT
sem_query r t rep =
  sem_stmts [ sem_prem r t "x0"
            , sem_commit (TVar "x0")]
  where sem_prem case rep of
    NoRep → sem_single
    Rep   → sem_many
```

A query is executed with the initial state, and a context that is produced from the specification's declarations containing the default values of read-only and read-write entities. The result of executing a query is reported back to the execution environment, which then decides how to present this information to the user.

4.2 Translating Rules to IML

This section shows how IML rules generate IML transactions. We introduce a state monad *VarGen*, propagating a seed for producing fresh meta-variables.

```
type F_VAR a = State Int a
fresh_var :: F_VAR X
```

The following infix operators, together with ⧺, concatenate sequences of statements (possibly) generated by F_{VAR} computations.

```
infixr 5 ⟨⧺⟩ , ⟨⧺ , ⧺⟩
⟨⧺⟩ :: F_VAR Stmts → F_VAR Stmts → F_VAR Stmts
⟨⧺  :: F_VAR Stmts → Stmts       → F_VAR Stmts
⧺⟩  :: Stmts       → F_VAR Stmts → F_VAR Stmts
```

An IML rule generates an IML transaction declaration. We omit the functions generating getters and setters, as their definitions are straightforward.

```
gBody :: Rule → F_VAR TransDecl
gBody (Rule (Concl f ps r rhs ro rw wo) prs scs) =
  TransDecl r f ∘ ([]) ⟨$⟩
        [PM_Args ps]    -- match arguments
  ⧺⟩  gROgets ro        -- ro-get statements
  ⧺⟩  gRWgets 0 rw      -- rw-get statements
  ⟨⧺⟩ gPrems prs        -- statements for premises
  ⟨⧺  gSideCons scs     -- pm statements
  ⧺   gRWsets_ 0 rw     -- rw-set statements
  ⧺   gWOsets wo        -- wo-set statements
  ⧺   [Commit rhs]      -- commit
```

The resulting *TransDecl* has just one branch. However, as stated in Section 4.1.1, a transaction's body must contain the branches for all rules with the same relation symbol and (computation) constructor. Transaction declarations with the same relation symbol and constructor are simply fused by concatenating their branches (function not given).

Premises require generating unique labels, which we achieve by applying *zipWith* to $[1\,..]$ and the list of premises.

```
gPrems :: Prems → F_VAR Stmts
gPrems =
  (concat ⟨$⟩) ∘ sequence ∘ zipWith gPremise [1..]
gPrem :: Label → Premise → F_VAR Stmts
gPrem l (Prem t rel p ro rw wo) =
        gROsets l ro      -- ro-set statements
  ⧺⟩  gRWsets l rw        -- rw-set statements
  ⧺⟩  gTransition l t rel p  -- single/many
  ⟨⧺⟩ gRWgets_ l rw       -- rw-get statements
  ⟨⧺⟩ gWOgets l wo        -- wo-get statements
```

The statements executing a premise's transition are generated by *gTransition*. The definition is slightly clever as it avoids generating redundant statements of the form $pm(y, x)$, where $y = x$.

```
gTransition :: Label → T → Rel → P → F_VAR Stmts
gTransition l t (Rel r rep) p = case p of
  PVar x → return [prem r t x l]
  _      → do x ← fresh_var
              return [prem r t x l
                     , PM (Val (TVar x)) p]
```

```
    where prem = case rep of NoRep → Single
                             Rep   → Many
```

Side-conditions generate one or two *PM* statements, depending on whether the side-condition's pattern p is a meta-variable or a compound pattern. Having separate statements for evaluating the side-condition's expression and matching the outcome to p is beneficial when left-factoring (discussed in Section 5.1).

```
gSideConds :: SideCons → F_VAR Stmts
gSideConds = (concat ⟨$⟩) ∘ mapM gSideCond
  where gSideCond (SideOP e p) = case p of
    PVar x → return [PM e (PVar x)]
    _      → do x ← fresh_var
                return [PM e (PVar x)
                       ,PM (Val (TVar x)) p]
```

5. PROGRAM MANIPULATION

In this section we discuss transformations on IML transactions that have the potential to greatly improve the runtime of their execution. The transformations are back-end independent such that any back-end of IML directly benefits. Standard techniques can be applied such as common subexpression elimination [18] and reordering based on commutativity. In the final version of this paper we intend to include experimental results reporting on the efficiency improvements.

The primary optimising is *left-factoring*, to minimising the amount of work undone by backtracking. Left-factoring has previously been applied by other authors to generate efficient code from inference-based language specifications [20, 25, 21], as well as in other domains such as syntax analysis [11, 2]. Performing left-factoring at the level of IML is beneficial because IML transactions contain more detail compared to high-level declarative inference rules. We show that the precision of left-factoring is enhanced when performed on IML transactions that have been subjected to statement reordering, which to our knowledge is a novel approach.

5.1 Left-factoring

The purpose of left-factoring is to find and merge common prefixes between the branches of IML transactions. As a result, branching points are pushed 'downstream', making backtracking less destructive. Left-factoring requires a notion of equality up to substitution of meta-variables. For example, statements *pm-args*(x_1) and *pm-args*(x_2) are candidates for merging, even though x_1 and x_2 are different meta-variables. We refer to the process of deciding whether two statements are equal up to substitution as *unification*. Unification is a concept well-known in the domain of polymorphic type inference [19], where unification decides whether two types, possibly containing type variables, may be considered equal. We define a binary unification operator $\uplus$ as a function over statements, returning $\bot$ if there is no unification, and *substitution functions* θ_1 and θ_2 otherwise. The substitution functions act as proof that the two operands are unifiable: if $a_1 \uplus a_2 = (\theta_1, \theta_2)$ then the statements obtained by applying θ_1 to a_1 and θ_2 to a_2 are syntactically equal. When a substitution function is applied to an statement a, zero or more meta-variables occurring in a are replaced by fresh meta-variables.

→ **TRANSACTION FOR: while**

pm-args(E, C);	
rw-get$(\mathsf{store}, \sigma, 0)$;	
ro-set$(\mathsf{env}, \sigma, 1)$;	
single$(\text{“}\Downarrow\text{”}, E, X_0, 1)$;	
pm(X_0, false);	*pm*(X_0, true);
rw-set$(\mathsf{store}, \sigma, 0)$;	*rw-set*$(\mathsf{store}, \sigma, 0)$;
COMMIT: done	**COMMIT: seq**$(C,$ **while**$(E, C))$

Figure 11: Left-factoring applied to the transaction of Figure 10.

A simple left-factoring procedure attempts to find a common prefix between two sequences of statements by, starting with $i = 1$, applying the unification operator to the i'th statement of both sequences. If there is no unification, the end of the common prefix has been reached and a branching point is introduced. If there is a unification, say $a_i^1 \uplus a_i^2 = (\theta_1, \theta_2)$, the resulting substitution functions θ_1 and θ_2 are applied to the first and second sequences of statements respectively. The common prefix is extended with the unified statement $a_i^{\uplus} = \theta_1(a_i^1) = \theta_2(a_i^2)$ and left-factoring continues. Following this approach, the branches of the transactions in Figure 10 are almost completely merged, resulting the transaction of Figure 11.

As a result of left-factoring, the work associated with the patterns of Rules (30) and (31) is now shared. This improvement could have been achieved by applying the pattern-matching specific optimisations discussed in [21]. However, as shown by the example, the benefits of reordering and left-factoring extend beyond pattern matching.

5.2 Statement reordering

Inspection of the right branch in Figure 11 tells us that there is no need for *rw-set*$(\mathsf{store}, \sigma, 0)$ to follow *pm*(X_0, true). The statements can be swapped. However, we cannot, for example, swap the second and third statement of the common prefix, as *ro-set*$(\mathsf{env}, \sigma, 1)$ depends on the binding for σ introduced by *rw-get*$(\mathsf{store}, \sigma, 0)$. To reason about the ways in which statements can be reordered we use *dependency graphs*. Figure 12 shows a dependency graph for the right branch of Figure 10. There is an edge from a to b if statement a needs to be executed before b. A meta-variable bound by a statement appears in blue, other meta-variables appear in red. Commit statements are not shown as they depend on all other statements. The edges in the graph are computed

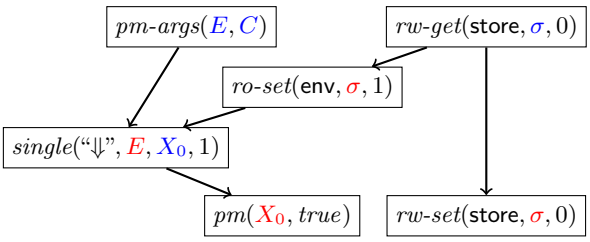

Figure 12: Dependency graph computed for right-hand side of Figure 10

by these rules:

- There is an edge from the statement binding meta-variable X to every statement that requires a binding

for X

- There is an edge from every setter statement with label i to the *single/many* statement with label i.
- There is an edge from the *single/many* statement with label i to every getter statement with label i.

We refer to the procedure that determines an execution order between statements based on such a dependency graph as *scheduling*. Consider the following basic scheduling algorithm. Let the set U contains all scheduled statements, and let the set R contain all statements that have their dependencies met. Initially U is empty and R contains the graph's *entry-points*: the nodes that have no incoming arrows. Repeat until R is empty:

1. Based on heuristic H, remove node $H(R)$ from R;
2. Add $H(R)$ to U;
3. Add every child of $H(R)$ to R, if and only if all its ancestors are in U.

The order in which nodes are added to U determines the schedule. The algorithm terminates as only finitely many nodes are added to R. All nodes are scheduled if the graph is acyclic, as a node is either an entry-point, or has an ancestor.

The precision of left-factoring is improved when the same scheduling heuristic is applied consistently to each branch of a transaction. When, in our example, *rw-set*($\mathsf{store}, \sigma, 0$) and $pm(X_0, true)$ are swapped by scheduling, left-factoring may then move *rw-set*($\mathsf{store}, \sigma, 0$) into the common prefix.

A scheduling heuristic may prevent merging statements. We may prioritise statements with a high 'likelihood-to-fail over predicted runtime' ratio. This heuristic is sensible, as it may decrease the runtime of transactions that abort, while not increasing the runtime of successful transactions. However, this heuristic will not swap *rw-set*($\mathsf{store}, \sigma, 0$) and $pm(X_0, true)$, as former always succeeds. We therefore expect that it is beneficial to have two scheduling phases, one taking place before and one after left-factoring.

In future work we aim to develop an overall strategy towards improving the efficiency of IML programs. The strategy will be inspired by experimental data obtained from exploring scheduling tactics.

6. RELATED WORK

Several recent specification languages have been based on I-MSOS, including CSF [9], its successor CBS [24], and DynSem [25]. These languages vary in their intended usage and the facilities they provide. For example, DynSem is designed primarily for expressing dynamic semantics in the big-step style, whereas CSF/CBS are designed primarily for the small-step style. CSF and CBS both provide a fixed set of transition relations which the user extends with new inference rules, whereas in DynSem the user introduces her own transition relations. IML considers I-MSOS in its full generality, and supports both big-step style, small-step style, and combinations of the two. The IML Rule Format was specifically designed to be able to subsume rules written in all of these specification languages.

Formal specification languages often provide mechanisms for implicitly generating rules, or components of rules, that would otherwise require tedious and repetitive manual definitions. For example, CSF and CBS support the implicit generation of *congruence rules* [9], and DynSem generates *implicit coercions* [25] between sorts within rules. A translation into the IML Rule Format should convert from these implicit mechanisms into explicit rules. Note that a translation should *not* make entity propagation explicit—implicit propagation of semantic entities is fundamental to the semantics of IML transactions. IML transactions are as modular as the I-MSOS rules from which they originate and IML components may be compiled individually: changes to an entity declaration or inference rule require only their code to be regenerated.

As a declarative programming language, PROLOG is a natural choice for developing interpreters based on SOS or I-MSOS specifications [6, 9]. In [6], the authors generate PROLOG clauses from MSOS rules. The clauses are left-factored, and refocusing is applied to greatly enhance the efficiency of small-step based interpreters (see below). Left-factoring was first applied to semantic specifications based on inference rules in [20] and has also been applied to DynSem specifications [25].

Interpreters generated directly from small-step SOS specifications suffer from a linear overhead, as finding the next reducible subterm may require full term traversal at each step. *Refocusing*, introduced in the context of reduction semantics [10], is an alternative evaluation strategy that tackles this problem. Refocusing is applicable also to SOS and MSOS rules of a certain form [4, 6]. The strategy is to keep reducing the same subterm until no further reductions are possible, by applying the same transition iteratively, similar to executing a pretty-big-step specification [7, 5]. IML may take advantage of refocusing by automatically transforming small-step IML rules into pretty-big-step IML rules. This transformation is straightforward—iterative transitions are native to the IML rule format—for specifications consisting of small-step rules. Although refocusing has previously been applied to small-step I-MSOS rules [24], adding a pretty-big-step transformation to IML would be the first attempt to combine refocused small-steps rules and big-step rules in executable specifications.

7. CONCLUSION

We have motivated and presented the design of IML, an intermediate language for generating efficient interpreters from I-MSOS specifications of programming languages. The presented semantics of IML forms a basis of IML implementations, and enables reasoning about the soundness of program manipulations. In future work we aim to discuss IML back-ends, and generating efficient interpreters from IML specifications.

IML is designed to be general enough to form a target language for the (I-M)SOS components of different semantic frameworks. We have already defined a translation from CBS rules to IML rules (not presented in this paper), and we aim to define a similar translation from DynSem to IML. Our hope is that other semantic frameworks may benefit from IML in the same fashion.

IML specifications concretise the workload associated with executing I-MSOS specifications, and may be subjected to scheduling algorithms and other forms of program manipulation. Some such efficiency improving transformations have been suggested in this paper. In future work we intend to develop a sequence of algorithms for incrementally transforming IML specifications to improve the runtime perfor-

mance of generated interpreters. The algorithms can then be instrumented and fined-tuned, based on experiments with different scheduling tactics.

8. REFERENCES

[1] Online Resources for "*An Intermediate Language for Efficient Interpretation of Implicitly Modular Structural Operational Semantics*". www.plancomps.org/ifl2016, 2016.

[2] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice Hall, 1972.

[3] E. Astesiano. Inductive and operational semantics. In *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 51–136. Springer, 1991.

[4] C. Bach Poulsen. *Extensible Transition System Semantics*. PhD thesis, Swansea University, 2016.

[5] C. Bach Poulsen and P. D. Mosses. Deriving pretty-big-step semantics from small-step semantics. In *23rd European Symposium on Programming*, volume 8410 of *Lecture Notes in Computer Science*, pages 270–289. Springer, 2014.

[6] C. Bach Poulsen and P. D. Mosses. Generating specialized interpreters for modular structural operational semantics. In *23rd International Symposium on Logic-Based Program Synthesis and Transformation*, pages 220–236. Springer, 2014.

[7] A. Charguéraud. Pretty-big-step semantics. In *22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2013.

[8] M. Churchill and P. D. Mosses. Modular bisimulation theory for computations and values. In *16th International Conference on Foundations of Software Science and Computation Structures*, volume 7794 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2013.

[9] M. Churchill, P. D. Mosses, N. Sculthorpe, and P. Torrini. Reusable components of semantic specifications. In *Transactions on Aspect-Oriented Software Development XII*, volume 8989 of *Lecture Notes in Computer Science*, pages 132–179. Springer, 2015.

[10] O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. BRICS Research Series RS-04-26, Department of Computer Science, Aarhus University, 2004.

[11] D. Grune and C. J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood, 1990.

[12] G. Kahn. Natural semantics. In *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[13] L. C. L. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *International Conference on Object Oriented Programming Systems Languages and Applications*, pages 444–463. ACM, 2010.

[14] R. Milner, M. Tofte, and D. MacQueen. *The Definition of Standard ML*. MIT Press, 1997.

[15] P. D. Mosses. Pragmatics of modular SOS. In *International Conference on Algebraic Methodology and Software Technology*, volume 2422 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2002.

[16] P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:195–228, 2004.

[17] P. D. Mosses and M. J. New. Implicit propagation in structural operational semantics. In *Fifth Workshop on Structural Operational Semantics*, volume 229(4) of *Electronic Notes in Theoretical Computer Science*, pages 49–66. Elsevier, 2009.

[18] S. S. Muchnick. *Advanced Compiler Design Implementation*. Academic Press, 1997.

[19] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[20] M. Pettersson. *Compiling Natural Semantics*, volume 1549 of *Lecture Notes in Computer Science*. Springer, 1999.

[21] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[22] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004. Reprint of Technical Report FN-19, DAIMI, Aarhus University, 1981.

[23] N. Sculthorpe, P. Torrini, and P. D. Mosses. A modular structural operational semantics for delimited continuations. In *2015 Workshop on Continuations*, volume 212 of *Electronic Proceedings in Theoretical Computer Science*, pages 63–80. Open Publishing Association, 2016.

[24] L. T. van Binsbergen, N. Sculthorpe, and P. D. Mosses. Tool support for component-based semantics. In *Companion Proceedings of the 15th International Conference on Modularity*, pages 8–11. ACM, 2016.

[25] V. Vergu, P. Neron, and E. Visser. DynSem: A DSL for dynamic semantics specification. In *26th International Conference on Rewriting Techniques and Applications*, volume 36 of *Leibniz International Proceedings in Informatics*, pages 365–378. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015.

[26] P. Wadler. How to replace failure by a list of successes. In *Conference on Functional Programming Languages and Computer Architecture*, pages 113–128. Springer, 1985.

[27] P. Wadler. The essence of functional programming. In *19th Symposium on Principles of Programming Languages*, pages 1–14. ACM, 1992.