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 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 for IML itself. 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) [24] 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) [16, 17] 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, 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) [18] 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 [30], the specification language used by the Funcon Framework [9], and DynSem [31], 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–based specifications to a common intermediate language (IML), and to apply optimisations to IML specifications. Optimised IML specifications are then used to generate interpreters for the programming language specified by the original I-MSOS–based specification. Our intermediate language consists of two levels:

- The IML Rule Format, a high-level representation of I-MSOS rules. It serves as the source of an IML compiler which translates instances of the IML Rule Format into the low-level IML language.
- IML, a lower level language that models I-MSOS rules as sequences of statements. 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.
- For expressing and reasoning about optimisations, IML is a more suitable object language than the existing

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

Figure 1: Grammar for a small While language.

declarative specification languages, as rules are represented by statements with an explicit flow of control.

- 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.

The focus of this paper is on the two levels of our intermediate language. In particular, we discuss the distinguishing aspects of the IML Rule Format in comparison to I-MSOS, and the operational semantics of IML programs, presented in Sections 3 and 4 respectively. A scheme translating instances of rule format into IML code is given in Section 4.2. The potential of IML with respect to code optimisations is discussed in Section 5.

A detailed exploration of optimisations on IML programs, and an analysis of the efficiency gains they bring in practice, will be the topic of a subsequent paper.

2. BACKGROUND

This section provides an overview of SOS and I-MSOS, and introduces 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, 24].

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 (both referred to as 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 (Rule 8).

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

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

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

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

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

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

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

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

Figure 2: SOS small-step rules for commands.

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

$$V \Downarrow V \quad (7)$$

$$\frac{\sigma \vdash E_1 \Downarrow I_1 \qquad \sigma \vdash E_2 \Downarrow I_2}{\sigma \vdash \textbf{plus}\ E_1\ E_2 \Downarrow I_3}\,(I_3 = I_1 + I_2) \quad (8)$$

$$\frac{\sigma \vdash E_1 \Downarrow I_1 \qquad \sigma \vdash E_2 \Downarrow I_2}{\sigma \vdash \textbf{leq}\ E_1\ E_2 \Downarrow \textbf{true}}\,(I_1 \leqslant I_2) \quad (9)$$

$$\frac{\sigma \vdash E_1 \Downarrow I_1 \qquad \sigma \vdash E_2 \Downarrow I_2}{\sigma \vdash \textbf{leq}\ E_1\ E_2 \Downarrow \textbf{false}}\,(I_1 \not\leqslant I_2) \quad (10)$$

$$\frac{}{\sigma \vdash R \Downarrow V}\,(V = \sigma(R)) \quad (11)$$

Figure 3: SOS big-step rules for expressions.

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

$$\langle \textbf{done}, \sigma\rangle \overset{[]}{\Rightarrow} \langle \textbf{done}, \sigma\rangle \quad (12)$$

$$\frac{\langle C, \sigma\rangle \xrightarrow{\alpha} \langle C', \sigma'\rangle \qquad \langle C', \sigma'\rangle \overset{\beta}{\Rightarrow} \langle \textbf{done}, \sigma''\rangle}{\langle C, \sigma\rangle \xrightarrow{\alpha +\!\!+ \beta} \langle \textbf{done}, \sigma''\rangle} \quad (13)$$

Figure 4: SOS repeated small-step transitions.

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.

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 [24] 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 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 the resulting value of the conclusion.

$$\frac{C_1 \rightarrow C_1'}{C_1 \;;\; C_2 \rightarrow C_1' \;;\; C_2} \tag{14}$$

$$\mathbf{done}; C_2 \rightarrow C_2 \tag{15}$$

$$\frac{\mathsf{env}(\sigma) \vdash E \Downarrow V}{\langle R := E, \mathsf{store}(\sigma)\rangle \rightarrow \langle \mathbf{done}, \mathsf{store}(\sigma[R \mapsto V])\rangle} \tag{16}$$

$$\frac{\mathsf{env}(\sigma) \vdash E \Downarrow V}{\langle \mathbf{print}\ E, \mathsf{store}(\sigma)\rangle \xrightarrow{\mathsf{out}([V])} \langle \mathbf{done}, \mathsf{store}(\sigma)\rangle} \tag{17}$$

$$\frac{\mathsf{env}(\sigma) \vdash E \Downarrow \mathbf{false}}{\langle \mathbf{while}\ E\ \mathbf{do}\ C\ \mathbf{od}, \mathsf{store}(\sigma)\rangle \rightarrow \langle \mathbf{done}, \mathsf{store}(\sigma)\rangle} \tag{18}$$

$$\frac{\mathsf{env}(\sigma) \vdash E \Downarrow \mathbf{true}}{\begin{array}{l}\langle \mathbf{while}\ E\ \mathbf{do}\ C\ \mathbf{od}, \mathsf{store}(\sigma)\rangle \rightarrow \\ \quad \langle C \;;\; \mathbf{while}\ E\ \mathbf{do}\ C\ \mathbf{od}, \mathsf{store}(\sigma)\rangle\end{array}} \tag{19}$$

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

$$V \Downarrow V \tag{20}$$

$$\frac{E_1 \Downarrow I_1 \qquad E_2 \Downarrow I_2}{\mathbf{plus}\ E_1\ E_2 \Downarrow I_3}\,(I_3 = I_1 + I_2) \tag{21}$$

$$\frac{E_1 \Downarrow I_1 \qquad E_2 \Downarrow I_2}{\mathbf{leq}\ E_1\ E_2 \Downarrow \mathbf{true}}\,(I_1 \leqslant I_2) \tag{22}$$

$$\frac{E_1 \Downarrow I_1 \qquad E_2 \Downarrow I_2}{\mathbf{leq}\ E_1\ E_2 \Downarrow \mathbf{false}}\,(I_1 \not\leqslant I_2) \tag{23}$$

$$\frac{}{\mathsf{env}(\sigma) \vdash R \Downarrow V}\,(V = \sigma(R)) \tag{24}$$

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

$$\mathbf{done} \Rightarrow \mathbf{done} \tag{25}$$

$$\frac{C \rightarrow C' \qquad C' \Rightarrow \mathbf{done}}{C \Rightarrow \mathbf{done}} \tag{26}$$

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 [34]. 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-

relation($\rightarrow$)
relation($\Downarrow$, VAL-REFLEXIVE)
entity(env, RO, `map-empty`)
entity(store, RW, `map-empty`)
entity(out, WO)

Figure 8: IML declarations.

only entities to model throwing exceptions or other forms of abnormal control-flow (see e.g. [26]). An SOS specification would have to change the *syntax* of the transition relation, and hence all existing rules, to permit 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 defines the IML Rule Format, which decides on a concrete treatment of particular abstract aspects of I-MSOS rules, e.g. meta-operations and side-conditions, permitting executable rules. We further define inference rule-based IML specifications. We accompany the precise mathematical definitions with Haskell (data) types. The Haskell types cannot be expected to be as precise. For example, values are defined as a synonym for terms, ignoring the constraints placed on values by the mathematical definition.

The IML Rule Format serves as an intermediate form in a mechanical translation of I-MSOS rules into interpreters and is intended as a target for compiling inference rules developed in the CBS and DynSem specification languages. We expect many SOS specifications may be written as IML specifications as well, for example those found in [3, 15], and can thus be made executable.

3.1 Terms and Expressions

When defining a formal semantics for a programming language, we distinguish between the *object language*, which is the programming language being defined by means of a specification, and the *meta-language*, which is the language being used to express the semantics of the object language. This section describes a minimal meta-language, on top of the IML Rule Format. We will continue to use the WHILE language as an example object language. We refer to the variables of a meta-language as *meta-variables* and reserve 'variables' for variables of the object language.

A *rule-based IML specification* Φ contains *meta-variables* $X(\Phi)$; *constructors* $F(\Phi)$, with an associated arity $n \geqslant 0$; a nominated set of *value constructors* $F^V(\Phi) \subset F(\Phi)$; a set of *operator names* $O(\Phi)$; a set of *relation symbols* $R(\Phi)$, with an associated set of predicates; a set of *entity identifiers* $E(\Phi)$, with an associated direction RO (read-only), WO (write-only), or RW (read-write), and an associated operator name (providing a default value); and a set of inference rules. The constructors $F(\Phi) \setminus F^V(\Phi)$ are refered to as *computation constructors*. Sets $X(\Phi)$, $F(\Phi)$, $O(\Phi)$, and $R(\Phi)$ are pairwise disjoint. The specification's inference rules must

$$\frac{C_1 \rightarrow C_1'}{seq(C_1, C_2) \rightarrow seq(C_1', C_2)} \tag{27}$$

$$seq(\mathbf{done}, C_2) \rightarrow C_2 \tag{28}$$

$$\frac{\mathsf{env}(\sigma_1) \vdash E \Downarrow V \quad \texttt{map-insert}(\sigma_1, R, V) \rhd \sigma_2}{\langle assign(R, E), \mathsf{store}(\sigma_1)\rangle \rightarrow \langle \mathbf{done}, \mathsf{store}(\sigma_2)\rangle} \tag{29}$$

$$\frac{\mathsf{env}(\sigma) \vdash E \Downarrow V}{\langle print(E), \mathsf{store}(\sigma)\rangle \xrightarrow{\mathsf{out}([V])} \langle \mathbf{done}, \mathsf{store}(\sigma)\rangle} \tag{30}$$

$$\frac{\mathsf{env}(\sigma) \vdash E \Downarrow \mathbf{false}}{\langle while(E, C), \mathsf{store}(\sigma)\rangle \rightarrow \langle \mathbf{done}, \mathsf{store}(\sigma)\rangle} \tag{31}$$

$$\frac{\mathsf{env}(\sigma) \vdash E \Downarrow \mathbf{true}}{\langle while(E, C), \mathsf{store}(\sigma)\rangle \rightarrow \langle seq(C, while(E, C)), \mathsf{store}(\sigma)\rangle} \tag{32}$$

Figure 9: IML small-step rules for commands.

$$\frac{E_1 \Downarrow I_1 \quad E_2 \Downarrow I_2 \quad \texttt{plus}(I_1, I_2) \rhd I_3}{plus(E_1, E_2) \Downarrow I_3} \tag{33}$$

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

$$\frac{E_1 \Downarrow I_1 \quad E_2 \Downarrow I_2 \quad \texttt{is-leq}(I_1, I_2) \rhd \mathbf{false}}{leq(E_1, E_2) \Downarrow \mathbf{false}} \tag{35}$$

$$\frac{\texttt{map-lookup}(R, \sigma) \rhd V}{\mathsf{env}(\sigma) \vdash ref(R) \Downarrow V} \tag{36}$$

Figure 10: IML big-step rules for expressions.

be instances of the IML Rule Format defined in the remainder of this section.

Figures 8, 9 and 10 present an IML specification for the WHILE language. The IML rules are similar to the I-MSOS rules in Figures 5 and 6, but program fragments are represented by terms rather than syntactically. Operations on values have been given names shown in `typewriter` font. Value operations appear on the left-hand side of the $\rhd$ operator used for side-conditions and pattern-matching. Constructors are written in *italics* and value constructors in **boldface**. Meta-variables are optionally subscripted Greek letters or Roman capitals. Rules (20), (25), and (26) appear to be missing. As we shall see later, these rules are implicitly active following from the relation declarations of Figure 8.

Definition 3.1. A *term* is either a meta-variable $x \in X(\Phi)$, or the application of a constructor $f \in F(\Phi)$ to n terms, denoted by $f(x_1, \ldots, x_n)$, with $n \geqslant 0$ is the arity of f. The set of Φ-terms is denoted $T(\Phi)$. A term is said to be *open* if it contains meta-variables, and *closed* otherwise.

Definition 3.2. Values are terms whose outermost constructor (if any) is a *value constructor* $f \in F^V(\Phi)$. The set

of Φ-values is denoted $V(\Phi)$, a subset of $T(\Phi)$.

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 (variable and integer, respectively) and X, Y are meta-variables. An example of a closed term is $\mathit{assign}(\text{“}x\text{”}, 10)$.

Terms are represented by the following Haskell types:

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

type F = String          type R = String
type X = String          type E = String
type V = T               type O = String
```

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

SOS and I-MSOS specifications typically make use of objects and operations from established mathematical frameworks, particularly for providing values and operations on values. For example Rules (21), (22), (16), and (36), perform integer addition, integer comparison, map insertion and map lookup, respectively. The semantics of value operations are assumed and not always given as part of a specification. The set $O(\Phi)$ contains the *names* of the value operations used by IML specification Φ. An interpreter for the object language defined by Φ has to provide implementations for the operations in $O(\Phi)$. In the IML rules for WHILE, we use the names `plus`, `is-leq`, and `map-lookup` among others.

Definition 3.3. An *operator expression* (or *expression*) is either a value, or the application of an operator name $o \in O(\Phi)$ to a sequence of expressions, denoted by $o(e_1, \ldots, e_n)$.

```
type Exprs = [Expr]
data Expr  = Val V
           | 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. IML 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; non-value terms can only be matched to meta-variables.

Definition 3.4. A *pattern* is a term in which all constructors are value constructors and in which no meta-variable occurs more than once. The set of Φ-patterns is denoted $P(\Phi)$, a subset of $T(\Phi)$.

```
data P = PVar X
       | PCons F [P]
```

3.2 Rule Representation

I-MSOS rules may have *side-conditions*, such as '$I_1 \leqslant I_2$' in Rule (22). In IML rules side-conditions appear as $e \rhd p$, with e an expression and p a pattern. A side-condition is satisfied if e evaluates (Section 4.1.5) to a value matching p.

Definition 3.5. A *side-condition* is a pair (e, p), denoted as $e \rhd p$, with e an expression and p a pattern.

```
data SideCon = SideCon Expr P
```

The inference rules of an IML specification define the relations identified by the relation symbols in $R(\Phi)$.

Based on the following inference rules, we obtain a definition for the 'iterative version' $\rightsquigarrow^*$ of any relation $\rightsquigarrow \in R(\Phi)$.

$$\frac{\texttt{is-value}(V_0) \rhd \textbf{true}}{V_0 \rightsquigarrow^* V_0} \tag{37}$$

$$\frac{\texttt{is-value}(T_0) \rhd \textbf{false} \quad T_0 \rightsquigarrow T_1 \quad T_1 \rightsquigarrow^* V_2}{T_0 \rightsquigarrow^* V_2} \tag{38}$$

These IML rules are implicitly present in any IML specification. Other rules can be made available by associating predicates with relation symbols. For example, the '$\Downarrow$' relation is declared to be reflexive on values see (Figure 8), thus enabling the IML variant of I-MSOS Rule (20):

$$\frac{\texttt{is-value}(V) \rhd \textbf{true}}{V \Downarrow V} \tag{39}$$

Recall that an entity identifier is associated with a direction and a default value provided as an operation name.

```
data Decls     = RODecl E O
               | RWDecl E O
               | WODecl E   -- default is [] (see 4.1)
               | RelDecl R [Predicate]
data Predicate = ValReflexive | ...
```

A *reference* to a read-only or write-only entity in a transition is either an *access* or an *update*, which is a pair (eid, p) or a pair (eid, t), respectively (for $\mathit{eid} \in E(\Phi), p \in P(\Phi), t \in T(\Phi)$). Read-only entities are accessed in the conclusion of a rule, where they are matched against a pattern, and updated in the premise of a rule, where they are set to a term. Conversely, write-only entities are accessed in a premise and updated in the conclusion.

A reference to a read-write entity is always both an access and an update, and is either a triple (eid, p, t) in a conclusion (the access is on the left-hand side of the transition and the update is on the right-hand side) or a triple (eid, t, p) in a premise (the update is on the left-hand side and the access is on the right-hand side). For consistency with read-only and write-only entities we refer to the former as read-write accesses and to the latter as read-write updates. The premise of Rule (31) is prefixed by read-only *update* (env, σ), while the conclusion contains a read-write *access* $(\mathsf{store}, \sigma, \sigma)$.

```
type Ac^RO  = (E, P)        type Up^WO = (E, T)
type Up^RO  = (E, T)        type Ac^RW = (E, P, T)
type Ac^WO  = (E, P)        type Up^RW = (E, T, P)
```

We define the *conclusion* of an IML rule as a sextuple $(p, r, t, \mathit{ro}, \mathit{rw}, \mathit{wo})$; with $p, t \in T(\Phi)$; $r \in R(\Phi)$; *ro* a sequence of read-only accesses; *rw* a sequence of read-write accesses; and *wo* a sequence of write-only updates. The same entity identifier may not occur twice within a single sequence of entity references. Term p must be of the form $f(p_1, \ldots, p_n)$, with f a computation constructor and $p_1, \ldots, p_n$ patterns, following the *value-added tyft* format given in [8].

```
data Concl = Concl F [P] R T [Ac^RO] [Ac^RW] [Up^WO]
```

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

The same entity identifier may not occur twice within a single sequence of entity references.

```
data Prem = Prem T Rel P [Up^RO] [Up^RW] [Ac^WO]
data Rel  = Rel R Rep
data Rep  = NoRep | Rep   -- either ⇝ or ⇝*
```

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

$$\frac{\{(t_i, \rightsquigarrow_i, p_i, ro_i, rw_i, wo_i) \ : \ i \in I\} \quad \{e_j \rhd p_j \ : \ j \in J\}}{(f(w_1, \ldots, w_n), \rightsquigarrow_0, t, ro_0, rw_0, wo_0)} \tag{40}$$

The tuple below the horizontal bar is a conclusion, and the $|I|$ tuples above the bar premises. Whenever an entity is referenced in a conclusion or premise, it must be referenced in the conclusion and all premises.

```
data Rule = Rule Concl [Either Prem SideCon]
type Spec = [Either Decl Rule]
```

Consider again the IML specification for WHILE. We adopt the same syntactic style as SOS and I-MSOS when displaying IML rules: we write read-only entity references before the turnstile symbol, write-only references above the arrow, and read-write references paired with program terms using angle brackets.

The following section shows an IML specification is executable via a translation into the lower level IML language.

4. INTERMEDIATE LANGUAGE

The previous section defines IML specifications based on inference rules, giving both Haskell and mathematical definitions. In this section we define transaction-based IML specifications (transaction are discussed later), mostly using Haskell. The goal is to present the IML language and develop an operational semantics by means of a reference interpreter. Section 4.2 shows how to generate transactions from IML rules and thus how rule-based IML specifications are translated into transaction-based IML specifications.

A transaction-based IML specification (simply specification henceforth) is not executable in itself—it does not specify where input comes from nor what to do with input. We assume an execution environment intends to send one or more interpretation requests to an interpreter generated from a specification. For this purpose we define *queries*. A query is a pair (r, t), with t a closed term, r either $\rightsquigarrow$ or $\rightsquigarrow^*$, and $\rightsquigarrow$ a relation symbol. An IML *program* is defined as an IML specification with a number of queries.

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

The abstract syntax of transactions is given by the Haskell types in Figure 11. We use IML (short for I-MSOS intermediate language), or the IML language, to refer to the set of valid IML programs. Not all constraints on valid IML programs are captured by the Haskell types. The missing constraints are discussed throughout this section.

The types X, F, T, P, *Expr*, R, and *Rep* are defined in Section 3. To explain IML transactions, we (loosely) borrow terminology related to database transactions. We say a transaction is a procedure that reads from, and writes to, some external state, by performing a series of statements.

```
data TransDecl = Trans R F [Stmts]
type Label     = Int
type Stmts     = [Stmt]
data Stmt
  = PMArgs  [P]                -- match input arguments
  | PM      Expr P             -- match given expression
  | Single  R T X  Label       -- T ⇝ X transition
  | Many    R T X  Label       -- T ⇝* X transition
  | ROGet   E X                -- read-only access
  | ROSet   E Expr Label       -- read-only update
  | RWGet   E X    Label       -- read-write access
  | RWSet   E Expr Label       -- read-write update
  | WOGet   E X    Label       -- write-only acces
  | WOSet   E Expr             -- write-only update
  | Branches [Stmts]           -- internal branching
  | Commit  T                  -- finalise transition to T
```

Figure 11: IML abstract syntax.

Each statement may fail, causing the transaction to *abort* and 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*. IML rules are implemented by transactions if we consider a meta-environment (binding meta-variables to terms) and the values of semantic entities as the external state. 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. A commit delivers both a term and any modifications made to semantic entities to the context in which the transaction was applied.

An IML transaction is defined for a relation symbol $\rightsquigarrow$ and a (computation) constructor f. The transaction's body is an arbitrary number of branches, each a sequence of statements with a commit or branching statement at the very end (and not earlier). Figure 12 shows a transaction generated for IML rules (31) and (32). The statements of transactions are shown with 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 IML Semantics

We define the semantics of IML programs as a set of semantic functions forming a reference interpreter. The focus is on presentational clarity and not efficient execution. We emphasise the operational semantics so that we can give a detailed explanation of the propagation of semantic entities. 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

Declarations for read-only and read-write entities determine their default values. Entity declarations evaluate to a *Ents*, mapping entity identifiers to values and directions.

```
type Ents = Map E (V, Dir)
data Dir  = RO | RW | WO
```

The value of a write-only entity is a list of values and the empty list by default. Relation declarations evaluate to a mapping between relation symbols and predicates.

```
type RelPreds = RSymb → Predicate
```

→ **TRANSACTION FOR: while**

pm-args(E, C);	*pm-args*(E, C);
rw-get(store, σ, 0);	*rw-get*(store, σ, 0);
ro-set(env, σ, 1);	*ro-set*(env, σ, 1);
single("⇓", E, X_0, 1);	*single*("⇓", E, X_0, 1);
pm(X_0, **false**);	*pm*(X_0, **true**);
rw-set(store, σ, 0);	*rw-set*(store, σ, 0);
COMMIT: done	**COMMIT: seq**$(C,$ **while**$(E, C))$

Figure 12: An IML transaction for Rules 31 and 32.

Transaction declarations evaluate to a *TransMap*, mapping relation symbols and constructor pairs to the list of branches mentioned in the declaration. An IML program contains at most one declaration for each possible pair. The empty list is returned if there is no declaration for a certain pair.

```
type TransMap = (R, F) → [Stmts]
```

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. Term t must be of the form $f(t_1, \ldots, t_n)$, where f is the computation constructor for which the transaction is defined. We refer to $t_1, \ldots, t_n$ as the *input arguments*. A *PMArgs* statements attempts to match the input arguments to a given list of patterns.

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

```
data Res_STMT = Done | ⊥ | Commit (T, Ents)
```

Done indicates the successful execution of a statement. Outcome *Commit* indicates the transaction has been successfully completed and provides the output term of the executed transition. Component *Ents* provides values for some read-write and write-only entities: referred to as the *additional output* of the transition. Outcome ⊥ indicates the transaction has been aborted.

The context of transaction is formed by a *TransMap*, some *additional input* in the form of entity values, the mapping between relation and predicates computed from relation declarations, and input arguments.

```
type Ctxt_T = (TransMap, Ents, RelPreds, [T])
```

Single and *Many* statements execute the premises of an IML rule: *Single* performs a $\rightsquigarrow$ transition and *Many* a $\rightsquigarrow^*$ transition, for a given relation symbol $\rightsquigarrow$, on the given input term. The given meta-variable is bound to the output term. Their respective semantics is described in detail in Section 4.1.3. The unique label associated with a premise is used by getter and setter statements to access (or update) entity values specific to a premise [2]. A Δ contains entity values specific to a certain label.

```
type Δ          = Map Label Ents
lookup_Δ        :: (E, Label) → Δ      → Maybe V
delete_Δ        :: (E, Label) → Δ      → Δ
project_Δ       :: Label      → Δ      → Ents
override_Δ      :: Label      → Ents → Δ → Δ
```

[1] ⊥ is *not* Haskell's undefined, sometimes also typeset as ⊥

[2] IML transactions have simple control flow because of labels, but more complicated semantics. Code-blocks could have been used as an alternative.

Support function $lookup_\Delta$ finds the value optionally stored for a given entity under the given label. Functions $delete_\Delta$ deletes the value of a given entity under the given label. The *Ents* storing all entities for a certain label is obtained by calling $project_\Delta$, while $override_\Delta$ replaces the *Ents* stored for the given label.

The state of a transaction consists of a meta-environment, binding meta-variables to (closed) terms, a Δ, and a list of labels containing the labels of the executed premises.

```
type State_T = (MetaEnv, Δ, [Label])
init_st      = ({}, {}, [])   -- default/empty state
```

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 map of type Δ, and an empty list of labels.

The semantics of statements is captured by the type [3]:

```
type Sem_STMT = Ctxt_T → State_T → (Res_STMT, State_T)
```

The semantics of abortion is defined as follows:

```
sem_abort :: Sem_STMT
sem_abort ctx st = (⊥, st)
```

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

```
sem_stmts :: [Sem_STMT] → Sem_STMT
sem_stmts []  _ _          = error "branch w/o commit"
sem_stmts (s : ss) ctx st = case s ctx st of
  (Done, st') → sem_stmts ss ctx st'
  r           → r
```

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 failing branch.

```
sem_branches :: [[Sem_STMT]] → Sem_STMT
sem_branches []       = sem_abort
sem_branches (b : bs) = sem_branch
  where sem_branch ctx st = case sem_stmts b ctx st of
          (Done, _) → error "branch without commit"
          (⊥, _)    → sem_branches bs ctx st
          r         → r
```

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, i.e. in any context there is at most one rule applicable. Interpreters may be generalised however, for example using a list of successes [33].

4.1.3 Implicit entity propagation

Mosses and New suggest multiple ways of interpreting I-MSOS rules with two or more premises [18]. We determine that premises can be executed in the order they appear in the rule, or in any order that satisfies their dependencies. Having chosen an order, unmentioned entities can be propagated. Alternatively, unobserved effects in the presence

[3] 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

of multiple premises can be considered illegal, i.e. a rule omitting a read-write or write-only entity may not execute a transition involving that entity.

For practical reasons we choose to consider the premises in the order they appear. We discuss the propagation of unmentioned entities, although the IML semantics are easily extended to support the alternative.

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 $\rightarrow_i$ relation symbols (or their iterative variation), with $0 \leqslant i \leqslant n$. The empty list is written on the conclusion's relation symbol, when $n = 0$ in Rule (43).

$$\frac{\mathsf{ro\text{-}ent}(\rho) \vdash t_1 \rightarrow_1 p_1 \quad \ldots \quad \mathsf{ro\text{-}ent}(\rho) \vdash t_n \rightarrow_n p_n}{\mathsf{ro\text{-}ent}(\rho) \vdash t_0 \rightarrow_0 p_0} \tag{41}$$

$$\frac{\begin{array}{c}\langle t_1, \mathsf{rw\text{-}ent}(\sigma_0)\rangle \rightarrow_1 \langle p_1, \mathsf{rw\text{-}ent}(\sigma_1)\rangle \\ \ldots \\ \langle t_n, \mathsf{rw\text{-}ent}(\sigma_{n-1})\rangle \rightarrow_n \langle p_n, \mathsf{rw\text{-}ent}(\sigma_n)\rangle\end{array}}{\langle t_0, \mathsf{rw\text{-}ent}(\sigma_0)\rangle \rightarrow_0 \langle p_0, \mathsf{rw\text{-}ent}(\sigma_n)\rangle} \tag{42}$$

$$\frac{t_1 \xrightarrow{\mathsf{wo\text{-}ent}(\alpha_1)}_1 p_1 \quad \ldots \quad t_n \xrightarrow{\mathsf{wo\text{-}ent}(\alpha_n)}_n p_n}{t_0 \xrightarrow{\mathsf{wo\text{-}ent}(\alpha_1 +\!\!+ \ldots +\!\!+ \alpha_n)}_0 p_0} \tag{43}$$

Implicit entity propagation is considered part of the semantics of IML transactions (we do not rely on a procedure explicating entity references). As a result, IML transactions are as modular as I-MSOS rules and have the same meaning in isolation as 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 values for all of the read-only and read-write entities
- When occurring in the result of a transition (commit), an *Ents* contains only those values of read-write and write-only entities that have been mentioned in the transition
- Before a premise with label l has been executed, an entry $l \mapsto es$ in a 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*.

```
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 $x \mapsto v$.

4.1.4 Executing transitions

We use a helper function $prop\_in :: [Ents] \rightarrow Ents$ for computing the additional input to a transition. The function merges the elements 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 $prop\_out :: [Ents] \rightarrow Ents$ 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 $+\!\!+$. Functions *prop_in* and *prop_out* unite *Ents* such that additional input and output is computed in accordance with Rules 41-43.

The semantics of performing a (possibly repeated) transition is implemented by functions *outer* and *inner*. Function *outer* performs substitution (function $subs :: MetaEnv \rightarrow T \rightarrow T$) 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 additional input stored under label l by the additional output (es_1) , 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 project_Δ δ) (σ ++ [l]))
```

A transition is executed by executing the branches found by applying *TransMap tm* to relation symbol r and computation constructor f. Constructor f and input arguments *args* are obtained by pattern-matching on input term t_0.

```
inner :: Rep → R → T → Ctxt_T → Res_STMT
inner _ _ (TVar _) _ = error "open term"
inner rp r t0@(TCons isVal f args) (tm, es0, rinfo, _) =
  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
    _ → if isVal ∧ reflVal then Commit (t0, { })
                           else ⊥
  where reflVal = case rp of
                    NoRep → ValReflexive ∈ rinfo r
                    Rep   → True
        rec t1 es1 = case inner rp r t1 (tm, es1', []) of
          Commit (t2, es2) → Commit (t2, es2')
          _                → Commit (t1, es1)
          where es2' = prop_out [es1, es2]
        where es1' = prop_in [es0, es1]
```

A branch committing (t_1, es_1) indicates a successful transition from t_0 to t_1, forming the overall result if $rp \equiv NoRep$. If there is no such branch, we may apply Rule (39) if relation r is declared as reflexive on values, and commit $(t_0, [])$. The case of repeated transitions is covered by greedily applying inference Rules (37) and (38). If there is no committing branch, and t_0 is a value, we apply Rule (37), and commit $(t_0, \{\})$. If a transition is possible, we apply Rule (38) and make a recursive call to *inner*. The implicit propagation of entities between premises and conclusion of Rule (38) is handled by *prop_in* and *prop_out*.

Single and *Many* are then 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. $Ents\ es$ contains the union of all unobserved side-effects of any premises, together with the additional output provided by write-only setters and read-write 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 project_Δ δ) (σ ++ [0]))
```

4.1.5 Evaluating expressions

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. The result of evaluating an (operator-)expression may be *Just* a value 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 V
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 (flip ($)) γ) es
        opApp "is-leq" args = ...
        opApp "plus"   args = ...
        ...
```

4.1.6 Pattern matching

The semantics of *PMArgs* statements is to match the input arguments against the given sequence of patterns. We rely on an unspecified function $matches :: [T] \to [P] \to Maybe\ MetaEnv$ to perform pattern matching. We write $\gamma_1\ [\gamma_2]$ to refer to the right-biased union of γ_1 and γ_2.

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

Side-conditions and premises require matching a specific term to a pattern, which is executed by *PM* statements. The first argument is an expression to be evaluated. The statement aborts if evaluating the expression does not yield a value matching the pattern. Any new bindings are added to the current meta-environment in case of a match.

```
sem_pm :: Sem_EXPR → P → Sem_STMT
sem_pm e p ctx st@(γ1, δ, σ) = 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, σ))
```

4.1.7 Executing queries

A query is executed by invoking *sem_single* or *sem_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 to be executed with the initial state, and a context resulting from evaluating the specification's declarations. The result of executing a query is reported back to some execution environment, which 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 seeds for producing fresh labels and meta-variables.

```
type Fresh a = State (Label, Int) a
fresh_var    :: Fresh X
fresh_lab    :: Fresh Label
```

The following infix operators, together with ++, concatenate sequences of statements (possibly) generated by *Fresh* computations.

```
infixr 5 ⟨++⟩ , ⟨++ , ++⟩
⟨++⟩ :: Fresh Stmts → Fresh Stmts → Fresh Stmts
⟨++  :: Fresh Stmts → Stmts       → Fresh Stmts
++⟩  :: Stmts       → Fresh Stmts → Fresh Stmts
```

An IML rule generates an IML transaction declaration. The translation of getters and setters are straightforward and are omitted. Read-write references in the conclusion generate getters and setters with label 0.

```
gBody :: Rule → Fresh TransDecl
gBody (Rule (Concl f ps r rhs ro rw wo) conds) =
  TransDecl r f ∘ ([]) ⟨$⟩
        [PM_Args ps]          -- match arguments
    ++⟩ gROgets ro            -- ro-get statements
    ++⟩ gRWgets 0 rw          -- rw-get statements
    ⟨++ gConditions conds     -- 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).

Above the bar of an IML rule we find conditions—premises and side-conditions—translated by *gPrem* and *gSideCon* respectively.

```
gConditions :: [Either Prem SideCon] → Fresh Stmts
gConditions = (concat ⟨$⟩) ∘ mapM gCondition
  where gCondition (Right sc) = gSideCon sc
        gCondition (Left pr)  = gPrem pr
```

Side-conditions generate one or two *PM* statements depending on whether the side-condition's pattern p is a meta-variable. Separate statements are generated for evaluating

the side-condition's expression and matching the outcome to p.

```
gSideCon :: SideCoc → Fresh Stmts
gSideCon (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]
```

A premise requires a unique label l and a *Single* or *Many* statement surrounded by getters and setters with label l.

```
gPrem :: Premise → Fresh Stmts
gPrem (Prem t rel p ro rw wo) = do
  l ← fresh_lab
  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 definition of *gTransition* is slightly clever as redundant statements of the form $pm(y, x)$, where $y = x$, are avoided.

```
gTransition :: Label → T → Rel → P → Fresh 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
```

5. PROGRAM TRANSFORMATIONS

Sections 3 and 4 describe two representations of I-MSOS inference rules, the high-level IML rules and the low-level IML transactions, and compilation from the former to the latter. The motivation for compiling to IML transactions is the application of program transformations that are not natural at the inference rule level.

As a low-level language, IML makes evaluation order and control flow explicit. The statements of IML transactions have been chosen to separate the actions that need to be performed. For example, a premise may require setting a new environment, performing a transition, and matching the result term to a pattern. Each of these actions is performed by a separate statement in an IML transaction. On the other hand, IML is sufficiently general and makes no assumptions about the style in which I-MSOS rules are written (e.g. big-step or small-step) and about back-ends implementing it.

Well-known optimisations are applicable to IML transactions, such as common subexpression elimination [19], common prefix elimination (left-factoring), and reordering based on commutativity. The primary optimisation we discuss is *left-factoring*, which can reduce the amount of work *un*done by backtracking. Left-factoring has been applied by other authors to generate efficient code from inference-based language specifications [21, 31, 22], as well as in other domains such as syntax analysis [11, 2].

5.1 Left-factoring

Left-factoring is the merging of common prefixes between branches of IML transactions. Thus pushing branching points 'downstream', making backtracking less destructive. For example, the IML transaction of Figure 12 consists of two branches beginning with four identical statements. Figure 13 shows the transaction after left-factoring has been applied. The statements generated for the patterns of Rules (31) and (32) have merged. This simple improvement could have been achieved by applying the pattern-matching specific optimisations discussed in [22]. However, the benefits of left-factoring extend beyond pattern matching. The premises of the original inference rules of our example have been partially merged: environment σ is set and a transition on term E is performed as part of the same branch. However, the result of the transition is matched with **true** or **false** in separate branches. This is possible because in the IML transaction the premise is represented by three separate statements.

→ **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$(X_0, \mathbf{false})$;	*pm*$(X_0, \mathbf{true})$;
rw-set$(\mathsf{store}, \sigma, 0)$;	*rw-set*$(\mathsf{store}, \sigma, 0)$;
COMMIT: done	**COMMIT: seq**$(C, \mathbf{while}(E, C))$

Figure 13: Left-factored version of Figure 12.

Rules (31) and (32) are presented such that the same meta-variables bind the arguments of *while* (E and C) and the current store (σ), making the rules easier to compare for the reader *and* compiler. Left-factoring can be extended by incorporating unification, a technique applied for similar purposes in the domains of polymorphic type inference [20]. Rather than relying on strict syntactic equality, we use a notion of equality insensitive to the names of meta-variables in binding positions. Two statements are considered unifiable if their respective meta-variables in binding positions can be renamed such that the resulting statements are syntactically equal.

5.2 Statement reordering

IML transactions are intended as a sequential representation of I-MSOS rules. The statements are to be executed, and the branches to be tried, in the order that they are given (top to bottom and left to right respectively, for example in Figures 12 and 13).

As discussed in Section 4.1.2, the outcome of a transaction is either failure ($\bot$) or a committed term (together with changes to some entities). Reordering a transaction's statements does not change the outcome of the transaction, owing to the statements' declarative origin [4]. The order of a transaction's statements may influence its execution time in a context in which it aborts—less work is wasted if an aborting statement is executed sooner rather than later. Prioritising statements that are more likely to fail may thus prove beneficial. Reordering statements may also increase the effectiveness of left-factoring. As an example, consider again Figure 13. We can observe that the statements *rw-set*$(\mathsf{store}, \sigma, 0)$ and $pm(X_0, true)$ of the right branch may be swapped. Because the same is possible in the left branch, *rw-set*$(\mathsf{store}, \sigma, 0)$ can be moved further upwards into the common prefix.

The order in which branches are considered does not in-

[4]An order is valid if the dependencies between statements are respected and if there is a commit or branching statement (only) at the end of every branch

fluence the outcome of a transaction if the branches originates from a deterministic I-MSOS specification (see Section 4.1.2). Under this assumption, branches can be reordered to prioritise branches that are *less* likely to fail.

Experiments have shown that reordering branches and statements can have a major effect.

6. RELATED WORK

The IML language is motivated and inspired by several recent specification languages based on I-MSOS: CSF [9], its successor CBS [30], and DynSem [31]. Where CBS and DynSem utilise I-MSOS, the Ott meta-language [27] and the $\mathbb{K}$ Framework [25] embody their own formalisms for describing semantics. These approaches 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. The rewrite rules in a $\mathbb{K}$ specification define a concurrent transition relation describing the evolution of state configurations. CSF and CBS also provide a fixed set of transition relations which the user extends with new inference rules, whereas in DynSem and Ott users introduces their own transition relations. IML supports custom relations defined by fully general I-MSOS rules, possibly written in big-step style, small-step style, or combinations of the two.

Higher Order Attribute Grammars (HOAGs) extend context free grammars with attributes for providing context information [32], forming a powerful method for describing language semantics and tree-based computations in general. The UUAG formalism implements HOAGs [28], and enables modular specification by implicitly propagating unmentioned attributes.

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 enhance the efficiency of small-step based interpreters (see below). Left-factoring was first applied to semantic specifications based on inference rules in [21] and is also applied to DynSem specifications [31].

Interpreters generated directly from small-step SOS specifications suffer from a linear overhead [10]. *Refocusing* is an alternative evaluation strategy tackling this problem, applicable to SOS and MSOS rules of a certain form [4, 6]. The strategy is 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. 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.

CSF/CBS, DynSem, Ott, and $\mathbb{K}$ rely on compilation to obtain reference interpreters from a high-level specification. Their respective formalisms require an understanding of logical inference or forms of term rewriting. The alternative is to write specifications as programs in a general-purpose language. However, developing an embedded, modular and type-safe specification in a principled fashion requires a substantial understanding of advanced programming techniques, often based on categorical concepts. To summarise, modular syntax composition can be achieved by defining injections (projections) of syntactic categories into (out of) a mutual coproduct [29]. Monads and monad transformers enable modular composition of semantic functions by implicitly propagating contextual information [14]. The effect handler approach, based on the notion of a free monad [23], is very promising. Fully modular composition of both syntax and semantics has been demonstrated for a large set of programming constructs [35].

7. FUTURE WORK

We have motivated and presented the IML language, designed as a target for translating the I-MSOS components of semantic specification languages. In future work we intend to show how IML can be used in the implementation of such specification languages.

The IML Rule Format forms a minimal meta-language and executable formal semantics can thus be developed directly as IML specifications. Together with this paper we release an IML compiler accompanied by a substantial set of value operations on a closed universe of values. The compiler translates rule-based IML specifications to reference interpreters and LaTeXversions of the IML rules, typeset in the style of this paper, giving language designers and semanticists a lightweight tool for developing executable formal specifications. A medium sized case-study should demonstrate the practicality of developing specifications directly in IML.

The reference interpreter for IML programs provided in this paper forms a basis for developing IML back-ends, and enables reasoning about the soundness of program transformations. The development of efficient IML back-ends, and a full exploration of the possible program transformation techniques improving the efficiency of IML programs, is the topic of future work. In particular, we are interested in exploring both static and dynamic techniques for finding more efficient orderings over IML statements and branches.

The PLanCompS project developed a component-based approach to programming language semantics [9]. Fundamental language constructs, called *Funcons*, are specified by I-MSOS small-step rules. We are interested in using IML to generate faster interpreters from Funcon specifications. A library of highly reusable Funcons is used in several case studies and future case-studies would immediately benefit from any efficiency improvements.

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] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *22nd Symposium on Principles of Programming Languages*, pages 333–343. ACM, 1995.

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

[16] 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.

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

[18] 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.

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

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

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

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

[23] G. Plotkin and M. Pretnar. *Handlers of Algebraic Effects*, pages 80–94. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[24] 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.

[25] G. Roşu and T. F. Şerbănuţă. K overview and SIMPLE case study. *Electronic Notes in Theoretical Computer Science*, 304:3–56, 2014.

[26] 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.

[27] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.

[28] S. D. Swierstra, P. R. A. Alcocer, and J. Saraiva. Designing and implementing combinator languages. In *Advanced Functional Programming*, pages 150–206. Springer Berlin Heidelberg, 1999.

[29] W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008.

[30] 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.

[31] 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.

[32] H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 131–145, New York, NY, USA, 1989. ACM.

[33] 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.

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

[35] N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 1–12, New York, NY, USA, 2014. ACM.