Semantic Essence of AsmL: Extended Abstract*

Yuri Gurevich, Benjamin Rossman and Wolfram Schulte

Microsoft Research
One Microsoft Way, Redmond, WA 98052, USA

Abstract. The Abstract State Machine Language, AsmL, is a novel executable specification language based on the theory of Abstract State Machines. AsmL is object-oriented, provides high-level mathematical data-structures, and is built around the notion of synchronous updates and finite choice. AsmL is fully integrated into the .NET framework and Microsoft development tools. In this paper, we explain the design rationale of AsmL and sketch semantics for a kernel of the language. The details will appear in the full version of the paper.

1 Introduction

For years, formal method advocates have criticized specification and documentation practices of the software industry. They point out that neither more rigorous English nor semi-formal notation like UML protect us from unintended ambiguity or missing important information. The more practical among them require specifications to be linked to an executable code. Without such a linkage one cannot debug the specification or impose it. Non-linked specifications tend quickly to become obsolete.

We agree with the critique. We need specifications that are precise, readable and executable. The group of Foundations of Software Engineering at Microsoft Research [4] was not satisfied with existing solutions of the specification problem (we will address related work in the full paper) and has worked out a new solution based on the theory of abstract state machines [5, 6, 3, 7]. We think of specifications as executable models that exhibit the desired behavior on the appropriate level of abstraction. Abstract State Machine Language, AsmL, is a language for writing such models [1].

The FSE group has designed AsmL, implemented it and integrated it with the Microsoft runtime and tool environment. Furthermore, the group has built various tools on top of AsmL.

1.1 Language Features

The language features of AsmL were chosen to give the user a familiar programming paradigm. For instance, AsmL supports classes and interfaces in the same way as C# or Java do. In fact all .NET structuring mechanisms are supported: enumerations, delegates, methods, events, properties and exceptions. Nevertheless, AsmL is primarily a specification language. Users familiar with the specification language literature will find familiar data structures and features, like sets, sequences, maps, pattern matching, bounded quantification, and set comprehension.

But the crucial features of AsmL, intrinsic to ASMs, are massive synchronous parallelism and finite choice. These features give rise to a cleaner programming style than is possible with standard imperative programming languages. Synchronous parallelism means that AsmL has transactional semantics. This provides for a clean separation between the generation of new values and the committal of those values into the persistent state. For instance, when an exception is thrown, the

* Springer Lecture Notes in Computer Science 3188 (2004), 240-259

state is automatically rolled back rather than being left in an unknown and possibly inconsistent state. Finite choice allows the specification of a range of behaviors permissible for an (eventual) implementation.

1.2 AsmL-S, a Core of AsmL

AsmL is rich. It incorporates features needed for .NET integration and features needed to support the tools built on top of AsmL. AsmL-S represents the stable core of AsmL; the S alludes to "simple". In this semantical study we allow ourselves to compactify the syntax and ignore some features that do not add semantical complexity. In particular, maps, sequences and sets are first-class citizens of the full AsmL. In AsmL-S only maps are first-class citizens. Sets of type t can be represented as maps from t to a unit type.

Acknowledgments. Without the support from the FSE group this work would not be possible. Particular thanks go to Wolfgang Grieskamp and Nikolai Tillmann for developing the runtime mechanism of AsmL.

2 AsmL-S through Examples

One can see AsmL as a fusion of the ASM paradigm and the .NET type system, influenced to an extent by other specification languages like VDM [2] or Z [13]. This makes it a powerful modeling tool. On the other hand, we also aimed for simplicity. That is why AsmL is designed in such a way that its core, AsmL-S, is small. AsmL-S is expression and object oriented. It supports synchronous parallelism, finite choice, sequential composition and exception handling. The rest of this section presents examples of AsmL-S programs and expressions. For the abstract syntax of AsmL-S, see Fig. 1 in Section 3.

Remark 1. The definitions in this section are provisional, having been simplified for the purpose of explaining examples. The notions introduced here (*stores*, *effects*, *evaluation*, etc.) are, of course, defined formally in the full paper.

2.1 Expressions

In AsmL-S, expressions are the only syntactic means for writing executable specifications. Binding and function application are call-by-value. (The necessity of .NET integration is a good reason all by itself not to use lazy evaluation.)

Literal is the set of literals, like 1, *true*, *null* or *void*. We write the value denoted by a literal as the literal itself. Literals are typed; for instance, 1 is of type *Int* and *true* is of type *Bool*. AsmL-S has various operations on *Literal*, like addition over integers or conjunction, i.e. *and*, over *Bool*.

Exception is an infinite set of exceptions, that is disjoint from *Literal*. For now, think of exceptions as values representing different kinds of errors. We will discuss exceptions further in subsection 2.8.

If e is a closed expression, i.e. an expression without free variables, and v is a literal or an exception, then $e \xrightarrow{\text{v}} v$ means that e evaluates to v. The "v" above the arrow alludes to "value".

Examples 1–5 show how to evaluate simple AsmL-S expressions.

Evaluation of Simple Expressions

$$1 + 2 \xrightarrow{\text{v}} 3 \tag{1}$$

$$1/0 \xrightarrow{\text{v}} \mathit{argX} \tag{2}$$

$$\textbf{let } x = 1 \textbf{ do } x + x \xrightarrow{\text{v}} 2 \tag{3}$$

$$\textbf{let } x = 1/0 \textbf{ do } 2 \xrightarrow{\text{v}} \mathit{argX} \tag{4}$$

$$\textbf{if } \mathit{true} \textbf{ then } 0 \textbf{ else } 3 \xrightarrow{\text{v}} 0 \tag{5}$$

For instance, Example 4 shows that let-expressions expose call-by-value semantics: if the evaluation of the binding fails (in this case, resulting in the argument exception), then the complete let-expression fails, irrespective of whether the body is used the binding.

2.2 Object Orientation

AsmL-S encapsulates state and behavior in classes. As in C# or Java, classes form a hierarchy according to single inheritance. We use only the single dispatch of methods. Objects are dynamically allocated. Each object has a unique identity. Objects can be created, compared and passed around.

ObjectId is an infinite set of potential object identifiers, that is disjoint from *Literal* and *Exception*. *Normal values* are either object identifiers in *ObjectId* or literals. *Values* are either normal values or exceptions.

$$\mathit{Nvalue} = \mathit{ObjectId} \cup \mathit{Literal}$$
$$\mathit{Value} = \mathit{Nvalue} \cup \mathit{Exception}$$

A *type map* is a partial function from *ObjectId* to *Type*. It sends allocated objects to their runtime types. A *location* is an object identifier together with a field name drawn from a set *FieldId*. A *content map* is a partial function from *Location* to *Nvalue*. It records the initial bindings for all locations.

$$\mathit{TypeMap} = \mathit{ObjectId} \to \mathit{Type}$$
$$\mathit{Location} = \mathit{ObjectId} \times \mathit{FieldId}$$
$$\mathit{ContentMap} = \mathit{Location} \to \mathit{Nvalue}$$

If e is a closed expression, then $e \xrightarrow{\theta,\omega,\text{v}} \theta, \omega, v$ means that the evaluation of e produces the type map θ, the content map ω and the value v. Examples 6–14 demonstrate the object oriented features of AsmL-S.

$$\textbf{class } A\ \{\} : \textbf{new } A() \xrightarrow{\theta,\omega,\text{v}} \{o \mapsto A\}, \emptyset, o \tag{6}$$

The execution of a nullary constructor returns a fresh object identifier o and extends the type map. The fresh object identifier o is mapped to the dynamic type of the object.

$$\textbf{class } A\ \{i \textbf{ as } \mathit{Int}\},\ \textbf{class } B \textbf{ extends } A\ \{b \textbf{ as } \mathit{Bool}\} : \tag{7}$$
$$\textbf{new } B(1, \mathit{true}) \xrightarrow{\theta,\omega,\text{v}} \{o \mapsto B\},\ \{(o, i) \mapsto 1, (o, b) \mapsto \mathit{true}\},\ o$$

The default constructor in AsmL-S takes one parameter for each field in the order of their declaration. The constructor extends the type map, extends the field map using the corresponding arguments, and returns a fresh object identifier.

$$\textbf{class } A\ \{i \textbf{ as } Int\} : \textbf{new } A(1).i \xrightarrow{\text{v}} 1 \tag{8}$$

Instance fields can immediately be accessed.

$$\begin{aligned}&\textbf{class } A\ \big\{Fact(i \textbf{ as } Int) \textbf{ as } Int \textbf{ do } \big(\textbf{if } i = 0 \textbf{ then } 1 \textbf{ else } i * me.Fact(n-1)\big)\big\} :\\ &\quad \textbf{new } A().Fact(3) \xrightarrow{\theta,\omega,\text{v}} \{o \mapsto A\}, \emptyset, 6\end{aligned} \tag{9}$$

Method calls have call-by-value semantics. Methods can be recursive. Within methods the receiver object is denoted by *me*.

$$\begin{aligned}&\textbf{class } A\ \{One() \textbf{ as } Int \textbf{ do } 1,\\ &\qquad Two() \textbf{ as } Int \textbf{ do } me.One() + me.One()\},\\ &\textbf{class } B \textbf{ extends } A\ \{One() \textbf{ as } Int \textbf{ do } -1\} : \textbf{new } B().Two() \xrightarrow{\text{v}} -2\end{aligned} \tag{10}$$

As in C# or Java method, dispatch is dynamic. Accordingly, in this example, it is the redefined method that is used for evaluation.

$$\begin{aligned}&\textbf{class } A\ \{i \textbf{ as } Int\} :\\ &\qquad \textbf{let } x = \big(\textbf{if } 3 < 4 \textbf{ then } null \textbf{ else new } A(1)\big) \textbf{ do } x.i \xrightarrow{\text{v}} nullX\end{aligned} \tag{11}$$

If the receiver of a field or method selection is *null*, evaluation fails and throws a null pointer exception.

$$\textbf{class } A\ \{\},\ \textbf{class } B \textbf{ extends } A\ \{\} : \textbf{new } B() \textbf{ is } A \xrightarrow{\text{v}} true \tag{12}$$

The operator **is** tests the dynamic type of the expression.

$$\textbf{class } A\ \{\},\ \textbf{class } B \textbf{ extends } A\ \{\} : \textbf{new } B() \textbf{ as } A \xrightarrow{\theta,\omega,\text{v}} \{o \mapsto B\}, \emptyset, o \tag{13}$$

Casting checks that an instance is a subtype of the given type, and if so then yields the instance without changing the dynamic type of the instance.

$$\textbf{class } A\ \{\},\ \textbf{class } B \textbf{ extends } A\ \{\} : \textbf{new } A() \textbf{ as } B \xrightarrow{\text{v}} castX \tag{14}$$

If casting fails, evaluation throws a cast exception.

2.3 Maps

Maps are finite partial functions. A *map display* is essentially the graph of the partial function. For example, a map display $m = \{1 \mapsto 2,\ 3 \mapsto 4\}$ represents the partial function that maps 1 to 2 and 3 to 4. The map m consists of two *maplets* $1 \mapsto 2$ and $3 \mapsto 4$ mapping *keys* (or *indices*) $1, 3$ to values $2, 4$ respectively.

Remark 2. In AsmL, maps can be also described by means of comprehension expressions. For example, $\{x \mapsto 2 * x \mid x \in \{1, 2, 3\}\}$ denotes $\{1 \mapsto 2,\ 2 \mapsto 4,\ 3 \mapsto 6\}$. In AsmL-S map comprehension should be programmed.

The maps of AsmL-S are similar to associative arrays of AWK or Perl. Maps have identities and each key gives rise to a location. Arbitrary normal values can serve as keys. We extend the notion of a location accordingly.

$$Location = ObjectId \times (FieldId \cup Nvalue)$$

Maps may be modified (see Section 2.4). Maps are often used in forall and choose expressions (see Sections 2.5 and 2.7). Examples 15–19 exhibit the use of maps in AsmL-S.

$$\begin{aligned}&\textbf{new } \mathit{Int} \to \mathit{Bool}\ \{1 \mapsto \mathit{true}, 5 \mapsto \mathit{false}\} \\ &\xrightarrow{\theta,\omega,\mathrm{v}}\ \{o \mapsto (\mathit{Int} \to \mathit{Bool})\}, \{(o,1) \mapsto \mathit{true}, (o,5) \mapsto \mathit{false}\}, o\end{aligned} \tag{15}$$

A map constructor takes the map type and the initial map as arguments.

$$\textbf{new } \mathit{Int} \to \mathit{Bool}\ \{1 \mapsto \mathit{true}, 1 \mapsto \mathit{false}\}\ \xrightarrow{\mathrm{v}}\ \mathit{argconsistencyX} \tag{16}$$

If a map constructor is inconsistent (i.e. includes at least two maplets with identical keys but different values), then the evaluation throws an inconsistency exception.

$$\big(\textbf{new } \mathit{Int} \to \mathit{Bool}\ \{1 \mapsto \mathit{true}\}\big)\,[1]\ \xrightarrow{\mathrm{v}}\ \mathit{true} \tag{17}$$

The value of a key can be extracted by means of an index expression.

$$\big(\textbf{if } \mathit{true} \textbf{ then } \mathit{null} \textbf{ else new } \mathit{Int} \to \mathit{Int}\ \{1 \mapsto 7\}\big)\,[1] \quad \xrightarrow{\mathrm{v}}\ \mathit{nullX} \tag{18}$$

$$\big(\textbf{new } \mathit{Int} \to \mathit{Int}\ \{1 \mapsto 7\}\big)\,[2] \quad \xrightarrow{\mathrm{v}}\ \mathit{mapkeyX} \tag{19}$$

However, if the receiver of the index expression is *null* or if the index is not in the domain of the map, then the evaluation throws a null exception or a map-key exception, respectively.

2.4 Assignments

One of AsmL's unique features is its handling of state. In sequential languages, like C# or Java, assignments trigger immediate state changes. In ASMs, and therefore also in AsmL, an assignment creates an *update*. An update is a pair: the first component describes the location to update, the second the value to which it should be updated. An update set is a set of updates. A triple that consists of a type map, a content map and an update set will be called a *store*.

$$\begin{aligned}\mathit{Update} &= \mathit{Location} \times (\mathit{Value} \cup \{\mathit{DEL}\}) \\ \mathit{UpdateSet} &= \mathit{SetOf}(\mathit{Update}) \\ \mathit{Store} &= \mathit{TypeMap} \times \mathit{ContentMap} \times \mathit{UpdateSet}\end{aligned}$$

Note that we extended $Value$ with a special symbol *DEL* which is used only with locations given by map keys and which marks keys to be removed from the map.

If e is a closed expression, then $e \xrightarrow{\mathrm{s,v}} s, v$ means that evaluation of e produces the store s and the value v. Examples 20–23 show the three ways to create updates. Note that in AsmL-S, but not in AsmL, all fields and keys can be updated. AsmL distinguishes between constants and variables and allows updates only to the latter.

$$\begin{aligned}&\textbf{class } A\ \{i \textbf{ as } \mathit{Int}\} : \\ &\quad \textbf{new } A(1).i := 2\ \xrightarrow{\mathrm{s,v}}\ \big(\{o \mapsto A\}, \{(o,i) \mapsto 1\}, \{((o,i),2)\}\big),\ \mathit{void}\end{aligned} \tag{20}$$

A field assignment is expressed as usual. However, it does not change the state. Instead, it returns the proposed update.

$$\begin{array}{l}\big(\textbf{new}\ \mathit{Int} \to \mathit{Bool}\ \{1 \mapsto \mathit{true}\}\big)\,[2] := \mathit{false} \\ \xrightarrow{\text{s,v}}\ \big(\{o \mapsto \mathit{Int} \to \mathit{Bool}\}, \{(o,1) \mapsto \mathit{true}\}, \{((o,2), \mathit{false})\}\big),\ \mathit{void}\end{array} \tag{21}$$

A map-value assignment behaves similarly. Note that the update set is created irrespective of whether the location exists or not.

$$\begin{array}{l}\textbf{remove}\ \big(\textbf{new}\ \mathit{Int} \to \mathit{Bool}\ \{1 \mapsto \mathit{true}\}\big)\,[1] \\ \xrightarrow{\text{s,v}}\ \big(\{o \mapsto \mathit{Int} \to \mathit{Bool}\}, \{((o,1) \mapsto \mathit{true}\}, \{(o,1), \mathit{DEL})\}\big),\ \mathit{void}\end{array} \tag{22}$$

The remove instruction deletes an entry from the map by generating an update that contains the placeholder *DEL* in the location to delete.

$$\begin{array}{l}\textbf{class}\ A\ \{F(\mathit{map}\ \textbf{as}\ \mathit{Int} \to A,\ \mathit{val}\ \textbf{as}\ A)\ \ \textbf{as}\ \mathit{Void}\ \textbf{do}\ \mathit{map}[0] := \mathit{val}\}, \\ \textbf{class}\ B\ \textbf{extends}\ A\ \{\}\ : \\ \quad \textbf{let}\ a = \textbf{new}\ A()\ \textbf{do}\ a.F(\textbf{new}\ \mathit{Int} \to B\ \{\},\ a)\ \xrightarrow{\text{v}}\ \mathit{mapvalueX}\end{array} \tag{23}$$

Since $\mathit{Int} \to B$ is a subtype of $\mathit{Int} \to A$, it is reasonable that this piece of code type-checks successfully at compile time. However, the assignment fails at runtime and throws a map-assignment exception. Thus, map assignments must be type-checked at runtime. (The same reason forces runtime type-checks of array assignments in C# or Java.)

2.5 Parallel Composition

Hand in hand with the deferred update of the state goes the notion of synchronous parallelism. It allows the simultaneous generation of finitely many updates. Examples 24–27 show two ways to construct synchronous parallel updates in AsmL-S.

$$\begin{array}{l}\textbf{let}\ x = \textbf{new}\ \mathit{Int} \to \mathit{Int}\ \{\}\ \textbf{do}\ \big(x[2] := 4\ \parallel\ x[3] := 9\big) \\ \xrightarrow{\text{s,v}}\ \big(\{o \mapsto \mathit{Int} \to \mathit{Int}\}, \emptyset, \{((o,2),4), ((o,3),9)\}\big),\ \mathit{void}\end{array} \tag{24}$$

Parallel expressions may create multiple updates. Update sets can be inconsistent. A consistency check is performed when a sequential composition of expressions is evaluated and at the end of the program.

$$\begin{array}{l}\textbf{let}\ x = \textbf{new}\ \mathit{Int} \to \mathit{Int}\ \{\}\ \textbf{do} \\ \textbf{let}\ y = \textbf{new}\ \mathit{Int} \to \mathit{Void}\ \{2 \mapsto \mathit{void}, 3 \mapsto \mathit{void}\}\ \textbf{do} \\ \quad \textbf{forall}\ i\ \textbf{in}\ y\ \textbf{do}\ x[i] := 2 * i \\ \quad \xrightarrow{\text{s,v}}\ \big(\{o_1 \mapsto \mathit{Int} \to \mathit{Int},\ o_2 \mapsto \mathit{Int} \to \mathit{Void}\}, \\ \qquad \{(o_2,2) \mapsto \mathit{void}, (o_2,3) \mapsto \mathit{void}\},\ \{((o_1,2),4), ((o_1,3),9)\}\big),\ \mathit{void}\end{array} \tag{25}$$

Parallel assignments can also be performed using forall expressions. In a forall expression **forall** x **in** e_1 **do** e_2, the subexpression e_1 must evaluate to a map. The subexpression e_2 is then executed with all possible bindings of the introduced variable to the elements in the domain of the map.

$$\begin{array}{l}\textbf{let}\ x = \textbf{new}\ \mathit{Int} \to \mathit{Int}\ \{\}\ \textbf{do}\ \big(\textbf{forall}\ i\ \textbf{in}\ x\ \textbf{do}\ x[i] := 1/i\big) \\ \xrightarrow{\text{s,v}}\ (\emptyset, \emptyset, \emptyset),\ \mathit{void}\end{array} \tag{26}$$

If the range of a forall expression is empty, it simply returns the literal *void*.

$$\begin{array}{l}\textbf{let } x = \textbf{new } \mathit{Int} \to \mathit{Int}\ \{2 \mapsto 4\} \textbf{ do let } y = x[2] \textbf{ do } \big((x[2] := 8) \parallel y\big) \\ \quad \xrightarrow{\text{s,v}}\ \big(\{o \mapsto \mathit{Int} \to \mathit{Int}\}, \{(o,2) \mapsto 4\}, \{((o,2),8)\}\big),\ 4\end{array} \tag{27}$$

Parallel expressions can return values. In full AsmL, the return value is distinguished syntactically by writing **return**. In AsmL-S, the value of the second expression is returned, whereas forall-expressions return *void*.

2.6 Sequential Composition

AsmL-S also supports sequential composition. Not only does AsmL-S *commit updates on the state*, as in conventional imperative languages, but it also *accumulates updates*, so that the result of a sequential composition can be used in the context of a parallel update as well. Examples 28–31 demonstrate this important feature of AsmL-S.

$$\begin{array}{l}\textbf{let } x = \textbf{new } \mathit{Int} \to \mathit{Int}\ \{2 \mapsto 4\} \textbf{ do } \big((x[2] := 8)\ ;\ (x[2] := x[2] * x[2])\big) \\ \quad \xrightarrow{\text{s,v}}\ \big(\{o \mapsto \mathit{Int} \to \mathit{Int}\},\ \{(o,2) \mapsto 4)\},\ \{((o,2),64)\}\big),\ \mathit{void}\end{array} \tag{28}$$

The evaluation of a sequential composition of $e_1\,;e_2$ at a state S proceeds as follows. First e_1 is evaluated at S. If no exception is thrown and the resulting update set is consistent, then the update set is fired (or executed) at S. This creates an auxiliary state S'. Then e_2 is evaluated at S', after which S' is forgotten. The current state is still S. The accumulated update set consists of the updates generated by e_2 at S' and the updates of e_1 that have not been overridden by updates of e_2.

$$\begin{array}{l}\textbf{let } x = \textbf{new } \mathit{Int} \to \mathit{Int}\ \{2 \mapsto 4\} \textbf{ do} \\ \quad \big(x[2] := 8 \parallel x[2] := 6\big)\ ;\ x[2] := x[2] * x[2]\ \xrightarrow{\text{v}}\ \mathit{updateX}\end{array} \tag{29}$$

If the update set of the first expression is inconsistent, then execution fails and throws an inconsistency exception.

$$\begin{array}{l}\textbf{let } x = \textbf{new } \mathit{Int} \to \mathit{Int}\ \{1 \mapsto 2\} \textbf{ do} \\ \quad \big(x[2] := 4 \parallel x[3] := 6\big)\ ;\ x[3] := x[3] + 1 \\ \quad \xrightarrow{\text{s,v}}\ \big(\{o \mapsto \mathit{Int} \to \mathit{Int}\},\ \{(o,1) \mapsto 2)\},\ \{((o,2),4),\ ((o,3),7)\}\big),\ \mathit{void}\end{array} \tag{30}$$

In this example, the update $((o,3),6)$ from the first expression of the sequential pair is overridden by the update $((o,3),7)$ from the second expression, which is evaluated in the state with content map $\{(o,1) \mapsto 2,\ (o,2) \mapsto 4,\ (o,3) \mapsto 6\}$.

$$\begin{array}{l}\textbf{let } x = \textbf{new } \mathit{Int} \to \mathit{Int}\ \{1 \mapsto 3\} \textbf{ do } \big(\textbf{while } x[1] > 0 \textbf{ do } x[1] := x[1] - 1\big) \\ \quad \xrightarrow{\text{s,v}}\ \big(\{o \mapsto \mathit{Int} \to \mathit{Int}\},\ \{(o,1) \mapsto 3)\},\ \{((o,1),0)\}\big),\ \mathit{void}\end{array} \tag{31}$$

While loops behave as in usual sequential languages, except that a while loop may be executed in parallel with other expressions and the final update set is reported rather than executed.

2.7 Finite Choice

AsmL-S supports choice between a pair of alternatives or among values in the domain of a map. The actual job of choosing a value from a given set X of alternatives is delegated to the environment. On the abstraction level of AsmL-S, an external function $\mathsf{oneof}(X)$ does the job. This is similar to delegating to the environment the duty of producing fresh object identifiers, by mean of an external function $\mathsf{freshid}$.

Evaluation of a program, when convergent, returns one effect and one value. Depending on the environment, different evaluations of the same program may return different effects and values. Examples 32–36 demonstrate finite choice in AsmL-S.

$$1 \,[\!]\, 2 \xrightarrow{\mathrm{v}} \mathsf{oneof}\{1, 2\} \tag{32}$$

An expression $e_1 \,[\!]\, e_2$ chooses between the given pair of alternatives.

$$\begin{aligned} &\textbf{choose } i \textbf{ in } \big(\textbf{new } \mathit{Int} \to \mathit{Void}\ \{1 \mapsto \mathit{void}, 2 \mapsto \mathit{void}\}\big) \textbf{ do } i \\ &\xrightarrow{\mathrm{s,v}} \mathsf{oneof}\big\{\big((\{o \mapsto \mathit{Int} \to \mathit{Void}\}, \{(o,1) \mapsto \mathit{void}, (o,2) \mapsto \mathit{void}\}, \emptyset),\ 1\big) \\ &\qquad\qquad \big((\{o \mapsto \mathit{Int} \to \mathit{Void}\}, \{(o,1) \mapsto \mathit{void}, (o,2) \mapsto \mathit{void}\}, \emptyset),\ 2\big)\big\} \end{aligned} \tag{33}$$

Choice-expressions choose from among values in the domain of a map.

$$\textbf{choose } i \textbf{ in } \big(\textbf{new } \mathit{Int} \to \mathit{Int}\ \{\}\big) \textbf{ do } i \xrightarrow{\mathrm{v}} \mathit{choiceX} \tag{34}$$

If the choice domain is empty, a choice exception is thrown. (The full AsmL distinguishes between choose-expressions and choose-statements. The choose-expression throws an exception if the choice domain is empty, but the choose-statement with the empty choice domain is equivalent to *void*.)

$$\begin{aligned} &\textbf{class } \mathit{Math}\{\mathit{Double}(x \textbf{ as } \mathit{Int}) \textbf{ as } \mathit{Int} \textbf{ do } 2 * x\} \ : \\ &\quad \textbf{new } \mathit{Math}().\mathit{Double}(1 \,[\!]\, 2) \qquad\qquad \xrightarrow{\mathrm{v}} \mathsf{oneof}\{2, 4\} \end{aligned} \tag{35}$$

$$\begin{aligned} &\textbf{class } \mathit{Math}\{\mathit{Double}(x \textbf{ as } \mathit{Int}) \textbf{ as } \mathit{Int} \textbf{ do } 2 * x\} \ : \\ &\quad \textbf{new } \mathit{Math}().\mathit{Double}(1) \,[\!]\, \textbf{new } \mathit{Math}().\mathit{Double}(2) \xrightarrow{\mathrm{v}} \mathsf{oneof}\{2, 4\} \end{aligned} \tag{36}$$

Finite choice distributes over function calls.

2.8 Exception Handling

Exception handling is mandatory for a modern specification language. In any case, it is necessary for AsmL because of the integration with .NET. The parallel execution of AsmL-S means that several exceptions can be thrown at once. Exception handling behaves as a finite choice for the specified caught exceptions. If an exception is caught, the store (including updates) computed by the try-expression is rolled back.

In AsmL-S, exceptions are special values similar to literals. For technical reasons, it is convenient to distinguish between literals and exceptions. Even though exceptions are values, an exception cannot serve as the content of a field, for example. (In the full AsmL, exceptions are instances of special exceptional classes.) There are several built-in exceptions: *argX*, *updateX*, *choiceX*, etc. In addition, one may use additional exception names e.g. *fooX*.

$$\begin{array}{l}\textbf{class } A\ \{\mathit{Fact}(n\ \textbf{as}\ \mathit{Int})\ \textbf{as}\ \mathit{Int}\ \textbf{do}\ (\textbf{if } n \geq 0\ \textbf{then}\\ \qquad\qquad (\textbf{if } n = 0\ \textbf{then } 1\ \textbf{else } \mathit{Fact}(n-1))\ \textbf{else throw } \mathit{factorialX})\}: \\ \quad \textbf{new } A.\mathit{Fact}(-5) \xrightarrow{\mathrm{v}} \mathit{factorialX}\end{array} \tag{37}$$

Custom exceptions may be generated by means of a throw-expression. Built-in exceptions may also be thrown. Here, for instance, **throw** *argX* could appropriately replace **throw** *factorialX*.

Examples 38–40 explain exception handling.

$$\begin{array}{l}\textbf{let } x = \textbf{new } \mathit{Int} \to \mathit{Int}\ \{\}\ \textbf{do}\ (\textbf{try}\ (x[1] := 2\ ;\ x[3] := 4/0)\ \textbf{catch}\ \mathit{argX} : 5) \\ \xrightarrow{\mathrm{s,v}}\ (\{o \mapsto \mathit{Int} \to \mathit{Int}\}, \emptyset, \emptyset), 5\end{array} \tag{38}$$

The argument exception triggered by $4/0$ in the try-expression is caught, at which point the update $((x, 1), 2)$ is abandoned and evaluation proceeds with the contingency expression 5. In general, the catch clause can involve a sequence of exceptions: a "catch" occurs if the try expression evaluates to any one of the enumerated exceptions. Since there are only finitely many built-in exceptions and finitely many custom exceptions used in a program, a catch clause can enumerate *all* exceptions. (This is common enough in practice to warrant its own syntactic shortcut, though we do not provide one in the present paper.)

$$\textbf{try}\ (\textbf{throw}\ \mathit{fooX})\ \textbf{catch}\ \mathit{barX}, \mathit{bazX}\ :\ 1 \xrightarrow{\mathrm{v}} \mathit{fooX} \tag{39}$$

Uncaught exceptions propagate up.

$$\textbf{throw}\ \mathit{fooX}\ \ \|\ \ \textbf{throw}\ \mathit{barX} \xrightarrow{\mathrm{v}} \mathsf{oneof}\{\mathit{fooX}, \mathit{barX}\} \tag{40}$$

If multiple exceptions are thrown in parallel, one of them is returned nondeterministically.

$$\textbf{throw}\ \mathit{fooX} \,[\!]\, 1 \xrightarrow{\mathrm{v}} \mathsf{oneof}\{\mathit{fooX}, 1\} \tag{41}$$

Finite choice is "demonic". This means that if one of the alternatives of a choice expression throws an exception and the other one converges normally the result might be either that the exception is propagated or that the value of the normally terminating alternative is returned.

2.9 Expressions with Free Variables

Examples 1-41 illustrate operational semantics for closed expressions (containing no free variables). In general, an expression e contains free variables. In this case, operational semantics of e is defined with respect to an *evaluation context* (b, r) consisting of a binding b for the free variables of e and a store $r = (\theta, \omega, u)$ where for each free variable x, $b(x)$ is either a literal or a object identifier in $\mathrm{dom}(\theta)$. We write $e \xrightarrow{\mathrm{v}}_{b,r} v$ if computation of e in evaluation context (b, r) produces value v.

$$x + y \xrightarrow[\{x \mapsto 7,\, y \mapsto 11\},\ (\emptyset,\emptyset,\emptyset)]{\mathrm{v}} 18 \tag{42}$$

$$\ell[2] \xrightarrow[\{\ell \mapsto o\},\ (\{o \mapsto \mathit{Int} \to \mathit{Bool}\}, \{(o,2) \mapsto \mathit{false}\}, \emptyset)]{\mathrm{v}} \mathit{false} \tag{43}$$

A more general notation $e \xrightarrow{\mathrm{s,v}}_{b,r} s, v$ means that a computation of e in evaluation context (b, r) produces new store s and value v.

3 Syntax and Semantics

The syntax of AsmL-S is similar to but different from that of the full AsmL. In this semantics paper, an attractive and user-friendly syntax is not a priority but brevity is. In particular, AsmL-S does not support the offside rule of the full AsmL that expresses scoping via indentation. Instead, AsmL-S uses parentheses and scope separators like ':'.

3.1 Abstract Syntax

We take some easy-to-understand liberties with vector notation. A vector $\bar{x}$ is typically a list $x_1 \ldots x_n$ of items possibly separated by commas. A sequence $x_1 \, \alpha \, y_1, \ldots, x_n \, \alpha \, y_n$ can be abbreviated to $\bar{x} \, \alpha \, \bar{y}$, where α represents a binary operator. This allows us, for instance, to describe an argument sequence ℓ_1 **as** t_1, ..., ℓ_n **as** t_n more succinctly as $\bar{\ell}$ **as** $\bar{t}$. The empty vector is denoted by ϵ.

Figure 1 describes the abstract syntax of AsmL-S. The meta-variables c, f, m, ℓ, *prim*, *op*, *lit*, and *exc*, in Fig. 1 range over disjoint infinite sets of class names (including *Object*), field names, method names, local variable names (including *me*), primitive type symbols, operation symbols, literals, and exception names (including several built-in exceptions: *argX*, *updateX*, ...). Sequences of class names, field names, method names and parameter declarations are assumed to have no duplicates.

An AsmL-S program is a list of class declarations, with distinct class names different from *Object*, followed by an expression, the body of the program. Each class declaration gives a superclass, a sequence of field declarations with distinct field names, and a sequence of method declarations with distinct method names.

AsmL-S has three categories of types — primitive types, classes and map types — plus two auxiliary types, *Null* and *Thrown*. (*Thrown* is used in the static semantics, although it is absent from the syntax.) Among the primitive types, there are *Bool*, *Int* and *Void*. Ironically, *Void* isn't void but contains one element. There could be additional primitive types; this makes no difference in the sequel.

Objects come in two varieties: class instances and maps. Objects are created with the **new** operator only; more sophisticated object constructors have to be programmed in AsmL-S. A new-class-instance expression takes one argument for each field of the class, thereby initializing all fields with the given arguments. A new-map expression takes a (possibly empty) sequence of key-values pairs, called *maplets*, defining the initial map. Maps are always finite. A map can be overridden, extended or reduced (by removing some of its maplets). AsmL-S supports the usual object-oriented expressions for type testing and type casting.

The common sequential programming languages have only one way to compose expressions, namely the sequential composition $e_1 \, ; e_2$. To evaluate $e_1 \, ; e_2$, first evaluate e_1 and then evaluate e_2. AsmL-S provides two additional compositions: the parallel composition $e_1 \parallel e_2$ and the nondeterministic composition $e_1 \, [\!] \, e_2$. To evaluate $e_1 \parallel e_2$, evaluate e_1 and e_2 in parallel. To evaluate $e_1 \, [\!] \, e_2$ evaluate either e_1 or e_2. The related semantical issues will be addressed later. The **while**, **forall** and **choose** expressions generalize the two-component sequential, parallel and nondeterministic compositions, respectively.

AsmL-S supports exception handling. In full AsmL, exceptions are instances of special exception classes. In AsmL-S, exceptions are atomic values of type *Thrown*. (Alternatively, we could have introduced a whole hierarchy of exception types.) There are a handful of built-in exceptions, like *argX*; all of then end with "X". A user may use additional exception names. There is no need to declare new exception names; just use them. Instead of prescribing a particular syntactic form to new exception names, we just presume that they are taken from a special infinite pool of potential exception names that is disjoint from other semantical domains of relevance.

pgm	$=$	$\overline{cls} : e$	programs
cls	$=$	**class** c **extends** c $\{\overline{fld}\ \overline{mth}\}$	classes
fld	$=$	f **as** t	fields
mth	$=$	$m(\overline{\ell}$ **as** $\overline{t})$ **as** t **do** e	methods
lit	$=$	$null \mid void \mid true \mid 0 \mid \ldots$	literals
op	$=$	$+ \mid - \mid / \mid = \mid < \mid and \mid \ldots$	primitive operations
$prim$	$=$	$Bool \mid Int \mid Void \mid \ldots$	primitive types
t	$=$	$prim \mid Null \mid c \mid t \to t$	normal types
exc	$=$	$argX \mid updateX \mid choiceX \mid \ldots$	exceptions
e	$=$		expressions
		$lit \quad \mid \quad \ell$	literals/local variables
	$\mid$	$op(\overline{e})$	built-in operations
	$\mid$	**let** $\ell = e$ **do** e	local binding
	$\mid$	**if** e **then** e **else** e	case distinction
	$\mid$	**new** $c\,(\overline{e})$	creation of class instances
	$\mid$	**new** $t \to t\ \{\overline{e} \mapsto \overline{e}\}$	creation of maps
	$\mid$	$e.f \quad \mid \quad e\,[e] \quad \mid \quad e.m(\overline{e})$	field/index/method access
	$\mid$	$e.f := e$	field update
	$\mid$	$e[e] := e \quad \mid$ **remove** $e[e]$	index update
	$\mid$	e **is** t	type test
	$\mid$	e **as** t	type cast
	$\mid$	$e \parallel e \quad \mid$ **forall** ℓ **in** e **do** e	parallel composition
	$\mid$	$e \,[\!]\, e \quad \mid$ **choose** ℓ **in** e **do** e	nondeterministic composition
	$\mid$	$e\,;e \quad \mid$ **while** e **do** e	sequential composition
	$\mid$	**try** e **catch** $\overline{exc} : e$	exception handling
	$\mid$	**throw** exc	explicit exception generation

Fig. 1. Abstract Syntax of AsmL-S

3.2 Class Table and Subtypes

It is convenient to view a program as a class table together with the expression to be evaluated [8]. The class table maps class names different from *Object* to the corresponding class definitions. The class table has the structure of a tree with edge relation $c \lhd c'$ meaning that **extends** c' is in the declaration of c; we say c' is the *parent* of c. *Object* is of course the root of class tree.

Remark 3. Whenever the "**extends** c" clause is omitted in examples 1-43, there is an implicit **extends** *Object*.

The subtype relation $\leq$ corresponding to a given class table is generated recursively by the rules in Fig. 2, for arbitrary types t, t', t'', τ, τ' and classes c, c':

- $t \leq t, \quad \dfrac{t \leq t' \quad t' \leq t''}{t \leq t''}$ — $\leq$ *is a partial order*
- $\dfrac{c \lhd c'}{c \leq c'}$ — $\leq$ *extends the parent relation over classes*
- $t \to t' \leq \mathit{Object}$ — *maps are objects*
- $\dfrac{t \leq \tau \quad t' \leq \tau'}{(t \to t') \leq (\tau \to \tau')}$ — *maps types are covariant in argument and result types*
- $\dfrac{t \leq \mathit{Object}}{\mathit{Null} \leq t}$ — *Null lies beneath all object types*
- $\mathit{Thrown} \leq t$ — *Thrown lies beneath all other types*

Fig. 2. Inductive Definition of the Subtype Relation

Call two types *comparable* if one of them is a subtype of the other; otherwise call them *incomparable*. Primitive types compare the least. If t is a primitive type, then $t \leq t$ and $\mathit{Thrown} \leq t$ are the only subtype relations involving t.

The following proposition is easy to check.

Proposition 1 *Every two types t_1, t_2 have a greatest lower bound $t_1 \sqcap t_2$. Every two subtypes of Object have a least upper bound $t_1 \sqcup t_2$.* □

Remark 4. One may argue that map types should be contravariant in the argument, like function types [11]. In the full paper, we discuss pros and cons of such a decision.

If c is a class different from *Object*, then $\mathit{addf}(c)$ is the sequence of distinct field names given by the declaration of c. These are the new fields of c, acquired in addition to those of $\mathit{parent}(c)$. The sequence of all fields of a class is defined by induction using the concatenation operation.

$$\mathit{fldseq}(\mathit{Object}) = \epsilon$$
$$\mathit{fldseq}(c) = \mathit{addf}(c) \cdot \mathit{fldseq}(\mathit{parent}(c))$$

We assume that $\mathit{addf}(c)$ is disjoint from $\mathit{fldseq}(\mathit{parent}(c))$ for all classes c. If f is a field of c of type t, then $\mathit{fldtype}(f, c) = t$. If $\mathit{fldseq}(c) = (f_1, \ldots, f_n)$ and $\mathit{fldtype}(f_i, c) = t_i$, then

$$\mathit{fldinfo}(c) = \bar{f} \textbf{ as } \bar{t} = (f_1 \textbf{ as } t_1, \ldots, f_n \textbf{ as } t_n).$$

The situation is slightly more complicated with methods because, unlike fields, methods can be overridden. Let $addm(c)$ be the set of method names included in the declaration of c. We define inductively the set of all method names of a class.

$$mthset(Object) = \emptyset$$
$$mthset(c) = addm(c) \cup mthset(parent(c))$$

For each $m \in mthset(c)$, $dclr(m, c)$ is the declaration

$$m(\ell_1 \textbf{ as } \tau_1, \ldots, \ell_n \textbf{ as } \tau_n) \textbf{ as } t \textbf{ do } e$$

of m employed by c. We assume, as a syntactic constraint, that the variables ℓ_i are all distinct and different from me. The declaration $dclr(m, c)$ is the declaration of m in the class $home(m, c)$ defined as follows:

$$\frac{m \in addm(c)}{home(m, c) = c} \qquad \frac{m \in mthset(c) - addm(c)}{home(m, c) = home(parent(c))}$$

In the sequel, we restrict attention to an arbitrary but fixed class table.

3.3 Static Semantics

We assume that every literal *lit* has a built-in type $littype(lit)$. For instance, $littype(2) = Int$, $littype(true) = Bool$ and $littype(null) = Null$. We also assume that a type function $optype(op)$ defines the argument and result types for every built-in operation op. For example, $optype(and) = (Bool, Bool) \rightarrow Bool$.

Suppose e is an expression, possibly involving free variables. A *type context* for e is a total function T from the free variables of e to types.

$\mathfrak{T}_T(e)$ is a partial function from expressions and type contexts to types. If $\mathfrak{T}_T(e)$ is defined, then e is said to be *well-typed* with respect to T, and $\mathfrak{T}_T(e)$ is called its *static type*.

The definition of $\mathfrak{T}_T(e)$ is inductive, given by rules in Fig. 3. See the full paper for a more thorough exposition.

3.4 Well-Formedness

We now make an additional assumption about the underlying class table: *for each class c and each method $m \in mthset(c)$, m is* well-formed *relative to c (symbolically: m ok in c).*

The definition of m ok in c is inductive. Suppose $dclr(m, c) = m(\ell_1 \textbf{ as } \tau_1, \ldots, \ell_n \textbf{ as } \tau_n) \textbf{ as } t \textbf{ do } e$ and $c \lhd c'$. Let T denote the type context $\{me \mapsto c\} \cup \{\ell_1 \mapsto \tau_1, \ldots, \ell_n \mapsto \tau_n\}$.

- $$\frac{m \in addm(c) - mthset(c') \qquad \mathfrak{T}_T(e) \leq t}{m \text{ ok in } c}$$
- $$\frac{m \in mthset(c) - addm(c) \qquad m \text{ ok in } c'}{m \text{ ok in } c}$$
- $$\frac{\begin{array}{c} m \in addm(c) \cap mthset(c') \qquad \mathfrak{T}_T(e) \leq t \qquad m \text{ ok in } c' \\ dclr(m, c') = m(\ell'_1 \textbf{ as } \tau'_1, \ldots, \ell'_n \textbf{ as } \tau'_n) \textbf{ as } t' \textbf{ do } e' \qquad \bar{\tau} \rightarrow t \leq \bar{\tau}' \rightarrow t' \end{array}}{m \text{ ok in } c}$$

The statement $\bar{\tau} \rightarrow t \leq \bar{\tau}' \rightarrow t'$, in the final premise, abbreviates the inequalities $\tau_1 \leq \tau'_1, \ldots, \tau_n \leq \tau'_n$ and $t \leq t'$.

- $\mathfrak{T}_T(lit) = littype(lit)$
- $\dfrac{optype(op) = \bar{\tau} \to t \qquad \mathfrak{T}_T(\bar{e}) \le \bar{\tau}}{\mathfrak{T}_T(op(\bar{e})) = t}$
- $\dfrac{\mathfrak{T}_T(e_1) = t}{\mathfrak{T}_T(\textbf{let } \ell = e_1 \textbf{ do } e_2) = \mathfrak{T}_{T \oplus \{\ell \mapsto t\}}(e_2)}$
- $\dfrac{\mathfrak{T}_T(e_1) = Bool}{\mathfrak{T}_T(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) = \mathfrak{T}_T(e_2) \sqcup \mathfrak{T}_T(e_3)}$
- $\dfrac{fldinfo(c) = \bar{f} \textbf{ as } \bar{t} \qquad \mathfrak{T}_T(\bar{e}) \le \bar{t}}{\mathfrak{T}_T(\textbf{new } c(\bar{e})) = c}$
- $\dfrac{\mathfrak{T}_T(e) = c}{\mathfrak{T}_T(e.f) = fldtype(f, c)}$
- $\dfrac{\mathfrak{T}_T(e_1) = c \qquad \mathfrak{T}_T(\overline{e_2}) \le \overline{\tau} \qquad dclr(m, c) = m(\bar{\ell} \textbf{ as } \overline{\tau}) \textbf{ as } t \textbf{ do } e_3}{\mathfrak{T}_T(e_1.m(\overline{e_2})) = t}$
- $\dfrac{\mathfrak{T}_T(e_2) \le \mathfrak{T}_T(e_1.f)}{\mathfrak{T}_T(e_1.f := e_2) = Void}$
- $\dfrac{\mathfrak{T}_T(\overline{e_1}) \le t_1 \qquad \mathfrak{T}_T(\overline{e_2}) \le t_2}{\mathfrak{T}_T(\textbf{new } t_1 \to t_2 \ \{\overline{e_1} \mapsto \overline{e_2}\}) = t_1 \to t_2}$
- $\dfrac{\mathfrak{T}_T(e_1) = \tau \to t \qquad \mathfrak{T}_T(e_2) \le \tau}{\mathfrak{T}_T(e_1[e_2]) = t}$
- $\dfrac{\mathfrak{T}_T(e_1) = \tau \to t \qquad \mathfrak{T}_T(e_2) \le \tau \qquad \mathfrak{T}_T(e_3) \le t}{\mathfrak{T}_T(e_1[e_2] := e_3) = Void}$
- $\dfrac{\mathfrak{T}_T(e_1) = \tau \to t \qquad \mathfrak{T}_T(e_2) \le \tau}{\mathfrak{T}_T(\textbf{remove } e_1[e_2]) = Void}$
- $\mathfrak{T}_T(\ell) = T(\ell)$
- $\dfrac{t < \mathfrak{T}_T(e)}{\mathfrak{T}_T(e \textbf{ is } t) = Bool}$
- $\dfrac{t < \mathfrak{T}_T(e)}{\mathfrak{T}_T(e \textbf{ as } t) = t}$
- $\dfrac{\mathfrak{T}_T(e_1) \text{ is defined}}{\mathfrak{T}_T(e_1 \parallel e_2) = \mathfrak{T}_T(e_2)}$
- $\dfrac{\mathfrak{T}_T(e_1) = \tau \to t \qquad \mathfrak{T}_{T \oplus \{\ell \mapsto \tau\}}(e_2) \text{ is defined}}{\mathfrak{T}_T(\textbf{forall } \ell \textbf{ in } e_1 \textbf{ do } e_2) = Void}$
- $\mathfrak{T}_T(e_1 \parallel e_2) = \mathfrak{T}_T(e_1) \sqcup \mathfrak{T}_T(e_2)$
- $\dfrac{\mathfrak{T}_T(e_1) = \tau \to t}{\mathfrak{T}_T(\textbf{choose } \ell \textbf{ in } e_1 \textbf{ do } e_2) = \mathfrak{T}_{T \oplus \{\ell \mapsto \tau\}}(e_2)}$
- $\dfrac{\mathfrak{T}_T(e_1) \text{ is defined}}{\mathfrak{T}_T(e_1 \,;\, e_2) = \mathfrak{T}_T(e_2)}$
- $\dfrac{\mathfrak{T}_T(e_1) = Bool \qquad \mathfrak{T}_T(e_2) \text{ is defined}}{\mathfrak{T}_T(\textbf{while } e_1 \textbf{ do } e_2) = Void}$
- $\mathfrak{T}_T(\textbf{throw } exc) = Thrown$
- $\mathfrak{T}_T(\textbf{try } e_1 \textbf{ catch } \overline{exc} : e_2) = \mathfrak{T}_T(e_1) \sqcup \mathfrak{T}_T(e_2)$

Fig. 3. Static Types of Expressions in AsmL-S

3.5 Operational Semantics

Suppose (b, r) is an evaluation context (subsection 2.9) for an expression e, where $r = (\theta, \omega, u)$. Then (b, r) gives rise to a type context $[b, r]$ defined by

$$[b,r](\ell) = \begin{cases} \theta_r(b(\ell)) & \text{if } b(\ell) \in \mathrm{dom}(\theta_r) \\ \mathit{littype}(b(\ell)) & \text{if } b(\ell) \in \mathit{Literal}. \end{cases}$$

We say e is (b, r)-*typed* if it is well-typed with respect to the type context $[b, r]$, that is, if $\mathfrak{T}_{[b,r]}(e)$ is defined.

In the full paper we define an operator $\mathfrak{E}_{b,r}$ over (b, r)-typed expressions. The computation of $\mathfrak{E}_{b,r}$ is in general nondeterministic (as it relies on external functions freshid and oneof) and it may diverge (as it is recursive but not necessarily well-founded). If it converges, it produces an *effect* $\mathfrak{E}_{b,r}(e) = (s, v)$ where s is a store and v is a value, that is, $e \xrightarrow{\mathrm{s,v}}_{b,r} s, v$ in the notation of subsection 2.9.

After seeing the examples in section 2, the reader should have a fairly good idea how $\mathfrak{E}_{b,r}$ is defined for most types of expression. See the full paper for a complete set of rules defining the effect operator.

4 Analysis

The effect operator is monotone with respect to stores: if $\mathfrak{E}_{b,r}(e) = (s, v)$ then r is a substore of s. Furthermore, if v is an exception then $r = s$, meaning that the store is rolled back whenever an exception occurs.

In addition to these properties, the static-type and effect operators satisfy the usual notions of type soundness and semantic refinement. See the full paper for precise statements and proofs of the theorems mentioned in this section.

The type of an effect (s, v), where $s = (\theta, \omega, u)$, is defined as follows:

$$\mathit{type}(s,v) = \begin{cases} \theta(v) & \text{if } v \in \mathrm{dom}(\theta) \\ \mathit{littype}(v) & \text{if } v \in \mathit{Literal} \\ \mathit{Thrown} & \text{if } v \in \mathit{Exception}. \end{cases}$$

Theorem 2 (Type Soundness) *For every evaluation context (b, r) and every (b, r)-typed expression e, we have*

$$\mathit{type}(\mathfrak{E}_{b,r}(e)) \leq \mathfrak{T}_{[b,r]}(e)$$

for any converging computation of $\mathfrak{E}_{b,r}(e)$.

In the full paper we define a relation $\lesssim$ of *semantic refinement* among expressions. (More accurately, a relation $\lesssim_T$ is defined for each type context T.) The essential meaning of $e_1 \lesssim e_2$ is that, for all evaluation contexts (b, r),

- computation of $\mathfrak{E}_{b,r}(e_1)$ potentially diverges only if computation of $\mathfrak{E}_{b,r}(e_2)$ potentially diverges, and
- the set of "core" effects of convergent computations of $\mathfrak{E}_{b,r}(e_1)$ is included in the set of "core" effects of convergent computations of $\mathfrak{E}_{b,r}(e_2)$.

Roughly speaking, the "core" of an effect (s, v) is the subeffect (s', v') that remains after a process of garbage-collection relative to the binding b: we remove all but the objects reachable from values in $\mathrm{rng}(b)$.

The refinement relation $\lesssim$ has the following property.

Theorem 3 (Refinement) *Suppose e_0', e_0, e_1 are expressions where e_0 is a subexpression of e_1 and e_0' refines e_0. Let e_1' be the expression obtained from e_1 by substituting e_0' in place of a particular occurrence of e_0. Then e_1' refines e_1.*

Here is a general example for refining binary choice expressions:

$$e_0 \lesssim (\mathit{true} \,[\!]\, \mathit{false}) \Longrightarrow \big(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2\big) \lesssim e_1 \,[\!]\, e_2$$

To give a similar general example involving the choose construct, we need a relation $\lesssim^{\text{c.d.}}$ of *choice-domain refinement* defined in the full paper.

Proposition 4

$$e_1' \lesssim^{\text{c.d.}} e_1 \Longrightarrow \big(\textbf{choose } \ell \textbf{ in } e_1 \textbf{ do } e_2\big) \lesssim \big(\textbf{choose } \ell \textbf{ in } e_1' \textbf{ do } e_2\big)$$

The full paper appears as a Microsoft Research Technical Report.

References

1. The AsmL webpage, `http://research.microsoft.com/foundations/AsmL/`.
2. Dines Bjoerner and Cliff B. Jones (Editors), "Formal Specification and Software Development", Prentice-Hall International, 1982.
3. Egon Boerger and Robert Staerk, "Abstract State Machines: A Method for High-Level System Design and Analysis", Springer, Berlin Heidelberg 2003.
4. Foundations of Software Engineering group, Microsoft Research, http://research.microsoft.com/fse/
5. Yuri Gurevich, "Evolving Algebra 1993: Lipari Guide", in "Specification and Validation Methods", Ed. E. Boerger, Oxford University Press, 1995, 9–36.
6. Yuri Gurevich, "For every Sequential Algorithm there is an Equivalent Sequential Abstract State Machine", ACM Transactions on Computational Logic 1:1 (2000), pages 77–111.
7. James K. Huggins, ASM Michigan web page, `http://www.eecs.umich.edu/gasm`.
8. Atsushi Igarashi, Benjamin C. Pierce and Philip Wadler, "Featherweight Java: a minimal core calculus for Java and GJ", ACM Transactions on Programming Languages and Systems (TOPLAS) 23:3 (May 2001), 396–450.
9. Gilles Kahn, "Natural semantics", In Proc. of the Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 247 (1987), 22–39.
10. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen, "The Definition of Standard ML (Revised)", MIT Press, 1997.
11. Benjamin C. Pierce, "Types and Programming Languages", MIT Press, Cambridge, Massachusetts, 2002
12. Gordon D. Plotkin, "Structural approach to operational semantics", Technical report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, September 1981
13. J. M. Spivey, "The Z Notation: A Reference Manual", Prentice-Hall, New York, Second Edition, 1992.