Available online at www.sciencedirect.com

SCIENCE @ DIRECT®

Theoretical Computer Science 343 (2005) 370–412

Theoretical Computer Science

www.elsevier.com/locate/tcs

Semantic essence of AsmL

Yuri Gurevich*, Benjamin Rossman, 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 provide static and dynamic semantics for a kernel of the language.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Abstract state machine; Executable specification language

Contents

* Corresponding author.
E-mail address: gurevich@microsoft.com (Y. Gurevich).

0304-3975/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2005.06.017

1. Introduction

Microsoft develops a huge amount of software. But how do Microsoft employees document the requirements, design, data structures, APIs, protocols, etc? Microsoft's development practices are diverse. Seldom do employees use mathematical models. Sometimes they use semi-formal notation like UML, but most of the time they use more or less rigorous English. However, we all know the drawbacks of semi-formal and informal specifications: unintended ambiguity, missing important information, etc. Most importantly, such specifications lack a linkage to code. One cannot run and thus debug them, and it is hard to impose such specifications. In spite of active interaction among architects, developers and testers, the developer's interpretation of an architectural specification may differ from that of the architect, and the tester may not know the precise functionality of the system.We need readable but precise specifications of what the software is supposed to do and we need the specification to be linked to an executable code. We view specifications as models that exhibit the desired behavior on the appropriate level of abstraction. AsmL is a new language for writing such models.

1.1. Language requirements

AsmL is designed to be

- *simple*: easy to use and able to deal naturally with common features like object orientation;
- *precise*: having a simple and uniform mathematical foundation based on abstract state machines (ASMs);
- *executable*: allowing you to validate the model;
- *testable*: with models acting as test oracles for the developed code as well as test case generators;
- *inter-operable*: able to interact with code in the existing Microsoft runtime environments;
- *integrated*: acting properly in the existing Microsoft runtime and tool environments;
- *scalable*: appropriate to write large industrial models;
- *analyzable*: amenable to efficient semantic analysis, like race condition or deadlock detection.

AsmL was designed because no existing language satisfied these criteria; see Section 1.4 in this connection. The group on Foundations of Software Engineering (FSE) at Microsoft Research designed, implemented and integrated AsmL with the Microsoft runtime and tool environment. The FSE group has also built various tools on top of AsmL.

1.2. 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 [9]. These features give rise to a cleaner programming style than is possible with standard imperative programming languages. Synchronous parallelism allows you to perform a collection of parametrized actions in parallel. For example, you may reverse simultaneously all edges of the given finite directed graph. This leads to transactional semantics. The collection of parametrized actions is treated as a single transaction. If something goes wrong, the whole transaction is rolled back. 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 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. Finite choice leads to a simple concept of program refinement: a finer program makes fewer choices and is more defined (and having fewer cases of nontermination or termination with an exception). Finite choice provides also a simple way of interleaving parallel computations that are supposed to be asynchronous, which is good enough for many distributed applications. An extension of AsmL with true asynchrony is in progress.

1.3. AsmL-S, a core of AsmL

AsmL is rich. It incorporates features needed for .NET integration and features needed to support various tools built on top of AsmL. It is also evolving. There are several reasons for this. The Microsoft runtime and tool environments evolve, and AsmL needs to be constantly reintegrated. The FSE group continues to build tools on top of AsmL and needs to be able to support these tools. The group continues to enrich AsmL with new features and revise it from time to time. But there is already a stable and mature core of AsmL.

AsmL-S, where S alludes to “simple”, represents the stable core of AsmL. This paper is a semantical study. So 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 in the language. Sets of type t can be represented as maps from t to the unit type.

1.4. Related work

The semantics of abstract state machines was defined in [9] and elaborated in [10]. The ASMs of [9] have the forall construct and the choose construct but no intra-step sequential composition. Intra-step sequential composition was accounted semantically in [11] (the simple non-iterative form) and in [6] (the iterative form). ASMs with set-theoretic background were studied in [5].

A number of ASM tools preceded AsmL; see Interpreters and Tools at [16] in this connection. None of those tools was sufficient for our purposes, however. Of course, we looked into other tools as well. Precise specification languages like HOL [8], PVS [25], VDM [2], or Z [26] are difficult to use for non-specialists; more importantly they are not inter-operable. Functional languages like Haskell [15] or SML [20] are attractive but they are not state oriented and, in our opinion, do not deal satisfactory with state. Modern object-oriented languages, like C# [14], Java [18], O’Caml [21], or Pizza [22], lack some abstractions of great importance to us. In particular, they do not support synchronous parallelism or non-determinism.

And so the group of Foundations of Software Engineering developed AsmL [1]. This development did not take place in a vacuum, though it is hard to pinpoint all the influences. The object-oriented aspects of AsmL were influenced by mainstream imperative languages like Java [18] and C# [14]. The type system was influenced by mainstream imperative languages as well as functional languages like Haskell [15] or SML [20]. The use of maps was influenced by VDM [2]. An early attempt to consider the semantics of AsmL is found in [12].

1.5. Article organization

This article is organized as follows.

Section 2, that is Section 2, illustrates the design of AsmL by means of examples. For expositional purposes, the language is introduced piecemeal and certain notions get revised along the way. For example, locations are first defined as object fields. Later, maps are introduced and the notion of location is generalized.

In Section 3, we give an abstract syntax for AsmL-S and explore its type system. In Section 4, we present operational semantics for AsmL-S. In Section 5, we prove the type soundness of AsmL-S, discuss semantic refinement and some other issues.

2. Motivating the design

This section serves the purposes of motivation and illustration only. The rest of the paper does not depend on this section.

AsmL is a rich language. One can see it as a fusion of the Abstract State Machine paradigm and the .NET type system, influenced to an extent by other specification languages like VDM or Z. 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 expressions and programs. For the abstract syntax of AsmL-S, see Fig. 1 in Section 3. We stress again that this article is a semantical study. The syntax of the full AsmL, intended to be user friendly and appropriate for substantial programs, was compactified to fit our purposes in this paper.

Remark 1. The "definitions" in this section are provisional, having been simplified for the purpose of explaining examples. The notions of *value*, *type*, *content map*, *store*, etc., are formally defined in Sections 3 and 4.

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, such as 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 the addition operation over *Int* or the conjunction operation over *Bool*.

Exception is an infinite set of exceptions that is disjoint from *Literal*. Think of exceptions as values representing different kinds of errors. We will discuss exceptions further in Section 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

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

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

$$\mathbf{let}\ x = 1\ \mathbf{do}\ x + x \xrightarrow{\mathrm{v}} 2 \tag{3}$$

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

$$\mathbf{if}\ \mathit{true}\ \mathbf{then}\ 0\ \mathbf{else}\ 3 \xrightarrow{\mathrm{v}} 0 \tag{5}$$

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 an 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. *Type* is the collection of AsmL-S types. The types will be introduced as we go; alternatively see Fig. 1 in Section 3. *Values* are either normal values or exceptions.

$$\begin{aligned} \mathit{Nvalue} &= \mathit{ObjectId} \cup \mathit{Literal}, \\ \mathit{Value} &= \mathit{Nvalue} \cup \mathit{Exception}. \end{aligned}$$

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.

$$\begin{aligned} \mathit{TypeMap} &= \mathit{ObjectId} \to \mathit{Type}, \\ \mathit{Location} &= \mathit{ObjectId} \times \mathit{FieldId}, \\ \mathit{ContentMap} &= \mathit{Location} \to \mathit{Nvalue}. \end{aligned}$$

If e is a closed expression, then $e \xrightarrow{\theta,\omega,\mathrm{v}} \theta, \omega, v$ 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. A colon is used to separate the class definitions from the expression that is the body of the program.

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

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.

(One of the referees asked whether "the bindings in the type map ever get 'garbage collected' in the semantics." On the semantical level of this paper, garbage collection is not a semantical issue. In any case, garbage collection is used in the full AsmL but not in AsmL-S.)

$$\begin{aligned}&\textbf{class } A\ \{i \textbf{ as } \mathit{Int}\},\ \textbf{class } B \textbf{ extends } A\ \{b \textbf{ as } \mathit{Bool}\}:\\ &\qquad \textbf{new } B(1, \mathit{true}) \xrightarrow{\theta,\omega,\mathrm{v}} \{o \mapsto B\},\ \{(o,i)\mapsto 1, (o,b)\mapsto \mathit{true}\},\ o. \end{aligned}\tag{7}$$

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.

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

Instance fields can immediately be accessed.

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

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

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

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

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

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

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

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

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

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.

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

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.

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

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

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

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.

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

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

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

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

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-pointer exception or a map-key exception, respectively.

Remark 3. AsmL-S treats maps differently than the full AsmL. The full AsmL is more sophisticated; it treats maps as values which requires partial updates [13]. In AsmL-S, maps are objects. An example illustrating this difference is given in Section 2.10.

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*.

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

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 $e \xrightarrow{\text{s,v}} s, v$ 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.

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

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

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

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

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

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

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

$$\begin{aligned} &\textbf{class } A\ \{F(\mathit{map} \textbf{ as } A{\to}\mathit{Int},\ \mathit{val} \textbf{ as } A) \ \textbf{ as } \mathit{Void} \textbf{ do } \mathit{map}[\mathit{val}] := 0\},\\ &\textbf{class } B \textbf{ extends } A\ \{\} :\\ &\quad \textbf{let } a = \textbf{new } A() \textbf{ do } a.F(\textbf{new } B \to \mathit{Int}\ \{\},\ a)\\ &\xrightarrow{\text{v}} \mathit{maptypeX}. \end{aligned} \tag{24}$$

Map types are covariant in both argument and result types. Since $Int \rightarrow B$ (resp. $B \rightarrow Int$) is a subtype of $Int \rightarrow A$ (resp. $A \rightarrow Int$), it is reasonable for Examples 23 and 24 to type-check successfully at compile time. However, the assignments fails at runtime and throw map-assignment exceptions. Thus, map assignments must be type-checked at runtime. (The same circumstance 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 25–28 show two ways to construct synchronous parallel updates in AsmL-S.

$$\begin{aligned}&\textbf{let } x = \textbf{new } Int \rightarrow Int\ \{\}\ \textbf{do}\\ &\quad \big(x[2] := 4 \ \parallel\ x[3] := 9\big)\\ &\quad \xrightarrow{s,v} \big(\{o \mapsto Int \rightarrow Int\},\ \emptyset,\ \{((o,2),4),((o,3),9)\}\big),\ void. \end{aligned} \tag{25}$$

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.

$$\begin{aligned}&\textbf{let } x = \textbf{new } Int \rightarrow Int\ \{\}\ \textbf{do}\\ &\textbf{let } y = \textbf{new } Int \rightarrow Void\ \{2 \mapsto void, 3 \mapsto void\}\ \textbf{do}\\ &\quad \textbf{forall } i \textbf{ in } y \textbf{ do } x[i] := 2 * i\\ &\qquad \xrightarrow{s,v} \big(\{o_1 \mapsto Int \rightarrow Int,\ o_2 \mapsto Int \rightarrow Void\},\\ &\qquad\quad \{(o_2,2) \mapsto void, (o_2,3) \mapsto void\},\ \{((o_1,2),4),((o_1,3),6)\}\big),\ void. \end{aligned} \tag{26}$$

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.

$$\begin{aligned}&\textbf{let } x = \textbf{new } Int \rightarrow Int\ \{\}\ \textbf{do}\\ &\quad \big(\textbf{forall } i \textbf{ in } x \textbf{ do } x[i] := 1/i\big)\\ &\quad \xrightarrow{s,v} (\{o \mapsto Int \rightarrow Int\}, \emptyset, \emptyset),\ void. \end{aligned} \tag{27}$$

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

$$\begin{aligned}&\textbf{let } x = \textbf{new } Int \rightarrow Int\ \{2 \mapsto 4\}\ \textbf{do}\\ &\quad \textbf{let } y = x[2]\ \textbf{do}\ \big((x[2] := 8) \parallel y\big)\\ &\quad \xrightarrow{s,v} \big(\{o \mapsto Int \rightarrow Int\}, \{(o,2) \mapsto 4\}, \{((o,2),8)\}\big),\ 4. \end{aligned} \tag{28}$$

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 (see the remark after rule E24 in Section 4.3 in this connection), 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 29–32 demonstrate this important feature of AsmL-S.

$$\begin{aligned}&\textbf{let } x = \textbf{new } \mathit{Int}{\rightarrow}\mathit{Int}\ \{2 \mapsto 4\}\ \textbf{do}\\ &\quad \big((x[2] := 8)\ ;\ (x[2] := x[2] * x[2])\big)\\ &\quad \xrightarrow{\text{s,v}} \big(\{o \mapsto \mathit{Int}{\rightarrow}\mathit{Int}\},\ \{(o,2) \mapsto 4)\},\ \{((o,2),64)\}\big),\ \mathit{void}.\end{aligned} \tag{29}$$

The evaluation of a sequential composition of $e_1 ; e_2$ at a state S proceeds as follows. First e_1 is evaluated in S. If no exception is thrown and the resulting update set is consistent, then the update set is fired (or executed) in S. This creates an auxiliary state S'. Then e_2 is evaluated in 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.

$$\begin{aligned}&\textbf{let } x = \textbf{new } \mathit{Int}{\rightarrow}\mathit{Int}\ \{2 \mapsto 4\}\ \textbf{do}\\ &\quad \big(x[2] := 8\ \|\ x[2] := 6\big)\ ;\ x[2] := x[2] * x[2]\\ &\quad \xrightarrow{\text{v}} \mathit{updateX}.\end{aligned} \tag{30}$$

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

$$\begin{aligned}&\textbf{let } x = \textbf{new } \mathit{Int}{\rightarrow}\mathit{Int}\ \{1 \mapsto 2\}\ \textbf{do}\\ &\quad \big(x[2] := 4\ \|\ x[3] := 6\big)\ ;\ x[3] := x[3] + 1\\ &\quad \xrightarrow{\text{s,v}} \big(\{o \mapsto \mathit{Int}{\rightarrow}\mathit{Int}\},\ \{(o,1) \mapsto 2)\},\ \{((o,2),4),\ ((o,3),7)\}\big),\ \mathit{void}.\end{aligned} \tag{31}$$

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 $\{(o,1) \mapsto 2,\ (o,2) \mapsto 4,\ (o,3) \mapsto 6\}$.

$$\begin{aligned}&\textbf{let } x = \textbf{new } \mathit{Int}{\rightarrow}\mathit{Int}\ \{1 \mapsto 3\}\ \textbf{do}\\ &\quad \big(\textbf{while } x[1] > 0\ \textbf{do } x[1] := x[1] - 1\big)\\ &\xrightarrow{\text{s,v}} \big(\{o \mapsto \mathit{Int}{\rightarrow}\mathit{Int}\},\ \{(o,1) \mapsto 3)\},\ \{((o,1),0)\}\big),\ \mathit{void}.\end{aligned} \tag{32}$$

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.

The question arises when are the updates fired? In principle, the updates are collected while the body of the program is executed and fired at the end of the execution. This does not mean that the execution proceeds in the initial state. Consider for instance Example 32. Every round of the while loop is executed in the state resulting from the execution of the previous rounds. Then why should we collect the updates? There is no good reason to collect updates in the case of Example 32. But, as we mentioned already, a while loop may be executed in parallel with some other expression; then the updates need to be reported. Also, something may go wrong with a while loop, in which case it needs to be rolled back.

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 means of an external function $\mathsf{freshid}$. (See Section 4.2 for more about these external functions.)

Evaluation of a program, when convergent, returns one effect and one value. Depending on the environment, different evaluations of the same expression may return different stores and values. Examples 33–37 demonstrate finite choice in AsmL-S.

$$1 \; [] \; 2 \xrightarrow{\mathrm{v}} \mathsf{oneof}\{1, 2\}. \tag{33}$$

An expression e_1 [] e_2 chooses between the given pair of alternatives.

$$\begin{aligned} &\textbf{choose } \; i \textbf{ in } \big(\textbf{new } \mathit{Int}{\to}\mathit{Void} \; \{1 \mapsto \mathit{void}, 2 \mapsto \mathit{void}\}\big) \textbf{ do } i \\ &\xrightarrow{\mathrm{s,v}} \mathsf{oneof}\Big\{\big((\{o \mapsto \mathit{Int}{\to}\mathit{Void}\}, \{(o,1) \mapsto \mathit{void}, (o,2) \mapsto \mathit{void}\}, \emptyset), \; 1\big) \\ &\qquad\qquad \big((\{o \mapsto \mathit{Int}{\to}\mathit{Void}\}, \{(o,1) \mapsto \mathit{void}, (o,2) \mapsto \mathit{void}\}, \emptyset), \; 2\big)\Big\}. \end{aligned} \tag{34}$$

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

$$\begin{aligned} &\textbf{choose } \; i \textbf{ in } \big(\textbf{new } \mathit{Int} \to \mathit{Int} \; \{\}\big) \textbf{ do } i \\ &\xrightarrow{\mathrm{v}} \mathit{choiceX}. \end{aligned} \tag{35}$$

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*.)

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

$$\begin{aligned} &\textbf{class } \mathit{Math}\{\mathit{Double}(x \textbf{ as } \mathit{Int}) \textbf{ as } \mathit{Int} \textbf{ do } 2 * x\} \;\; : \\ &\qquad \textbf{new } \mathit{Math}().\mathit{Double}(1) \;\; [] \;\; \textbf{new } \mathit{Math}().\mathit{Double}(2) \\ &\qquad \xrightarrow{\mathrm{v}} \mathsf{oneof}\{2, 4\}. \end{aligned} \tag{37}$$

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*.

$$\begin{array}{l}\textbf{class}\ A\ \big\{Fact(n\ \textbf{as}\ Int)\ \textbf{as}\ Int\ \textbf{do}\\ \quad\big(\textbf{if}\ n \geqslant 0\ \textbf{then}\big(\textbf{if}\ n=0\ \textbf{then}\ 1\ \textbf{else}\ Fact(n-1)\big)\\ \quad\ \textbf{else throw}\ factorialX\big)\big\}:\\ \quad\textbf{new}\ A.Fact(-5) \xrightarrow{\text{v}} factorialX.\end{array} \tag{38}$$

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 39–41 explain exception handling.

$$\begin{array}{l}\textbf{let}\ x = \textbf{new}\ Int{\rightarrow}Int\ \{\}\ \textbf{do}\\ \quad\textbf{try}\ \big(x[1] := 2\ ;\ x[3] := 4/0\big)\ \textbf{catch}\ argX : 5\\ \quad\xrightarrow{\text{s,v}}\ \big(\{o \mapsto Int{\rightarrow}Int\}, \emptyset, \emptyset\big),\ 5\end{array} \tag{39}$$

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.)

$$\textbf{try}\ \big(\textbf{throw}\ fooX\big)\ \textbf{catch}\ barX, bazX\ :\ 1 \xrightarrow{\text{v}} fooX. \tag{40}$$

Uncaught exceptions propagate up.

$$\textbf{throw}\ fooX\ \parallel\ \textbf{throw}\ barX \xrightarrow{\text{v}} \mathsf{oneof}\{fooX, barX\}. \tag{41}$$

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

$$\textbf{throw}\ fooX\ []\ 1 \xrightarrow{\text{v}} \mathsf{oneof}\{fooX, 1\}. \tag{42}$$

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 1–42 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.

$$x + y \xrightarrow{\mathrm{v}}_{\{x\mapsto 7,\ y\mapsto 11\},\ (\emptyset,\emptyset,\emptyset)} 18 \tag{43}$$

$$\ell[2] \xrightarrow{\mathrm{v}}_{\{\ell\mapsto o\},\ (\{o\mapsto Int\to Bool\},\{(o,2)\mapsto false\},\emptyset)} \mathit{false}. \tag{44}$$

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.

2.10. Maps as objects

This subsection expands Remark 3. It was prompted by a question of Robert Stärk who raised the following example.

$$\begin{aligned}
&\textbf{class } A\ \{f \textbf{ as } Int \to Bool,\ g \textbf{ as } Int \to Bool\}\ :\\
&\quad \textbf{let } a = \textbf{new } A(\textbf{new } Int \to Bool\ \{1 \mapsto true,\ 2 \mapsto true\},\\
&\quad \textbf{new } Int \to Bool\ \{\})\ \textbf{do}\\
&\qquad a.g := a.f\ \ ;\ \ a.x(2) := false\\
&\quad \xrightarrow{\mathrm{s,v}}\ \big(\{o_1 \mapsto A,\ o_2 \mapsto Int \to Bool,\ o_3 \mapsto Int \to Bool\},\\
&\qquad\quad \{(o_1, f) \mapsto o_2,\ (o_1, g) \mapsto o_3\},\ \{((o_1, g), o_2),\ (o_2, 2), false)\}\big),\ void.
\end{aligned} \tag{45}$$

In this example, the first assignment $a.g := a.f$ is responsible for the update $((o_1, g), o_2)$; the second assignment gives rise to the update $((o_2, 2), \mathit{false})$. Thus, $a.g[2]$ has value *false* after all updates are executed.

This same program has a different semantics in the full AsmL, where maps are treated as values rather than objects. In AsmL, the assignment $a.g := a.f$ has the effect of updating $a.g$ to the value of $a.f$, i.e., the map $\{1 \mapsto \mathit{true},\ 2 \mapsto \mathit{false}\}$. The second assignment, $a.f[2] := \mathit{false}$, has no bearing on $a.g$. Thus, $a.g[2]$ has value *true* after all updates are executed.

In treating maps as objects in AsmL-S, we avoid having to introduce the machinery of partial updates [13], which is necessary for the treatment of maps as values in AsmL. This causes a discrepancy between the semantics of AsmL-S and of AsmL. Fortunately, there is an easy AsmL-S expression that updates the value of a map m_1 to the value of another map m_2 (without assigning m_2 to m_1):

$$\textbf{forall } i \textbf{ in } m_1 \textbf{ do remove } m_1[i]\ \ ;\ \ \textbf{forall } i \textbf{ in } m_2 \textbf{ do } m_1[i] := m_2[i]$$

The first forall expression erases m_1; the second forall expression copies m_2 to m_1 at all keys i in the domain of m_2.

3. Syntax and static 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 $x_1 \propto y_1, \ldots, x_n \propto y_n$ can be abbreviated to $\bar{x} \propto \bar{y}$, where $\propto$ represents a binary operator. This allows us, for instance, to describe an argument sequence ℓ_1 **as** $t_1, \ldots, \ell_n$ **as** t_n more succinctly as $\bar{\ell}$ **as** $\bar{t}$. The empty vector is denoted by ε.

Fig. 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 super-class, 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-value 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

pgm	$=$	$\overline{cls}\ :\ e$	*programs*
cls	$=$	$\textbf{class}\ c\ \textbf{extends}\ c\ \{\overline{fld}\ \overline{mth}\}$	*class declarations*
fld	$=$	$f\ \textbf{as}\ t$	*field declarations*
mth	$=$	$m(\overline{\ell}\ \textbf{as}\ \overline{t})\ \textbf{as}\ t\ \textbf{do}\ e$	*method declarations*
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*
	$\mid$	$\textbf{let}\ \ell = e\ \textbf{do}\ e$	*local binding*
	$\mid$	$\textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$	*case distinction*
	$\mid$	$\textbf{new}\ c\ (\overline{e})$	*creation of class instances*
	$\mid$	$\textbf{new}\ t \to t\ \{\overline{e} \mapsto \overline{e}\}$	*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*
	$\mid$	$e[e] := e \quad \mid \quad \textbf{remove}\ e[e]$	*index update*
	$\mid$	$e\ \textbf{is}\ t$	*type test*
	$\mid$	$e\ \textbf{as}\ t$	*type cast*
	$\mid$	$e \parallel e \quad \mid \quad \textbf{forall}\ \ell\ \textbf{in}\ e\ \textbf{do}\ e$	*parallel composition*
	$\mid$	$e\ [\,]\ e \quad \mid \quad \textbf{choose}\ \ell\ \textbf{in}\ e\ \textbf{do}\ e$	*nondeterministic composition*
	$\mid$	$e\ ;\ e \quad \mid \quad \textbf{while}\ e\ \textbf{do}\ e$	*sequential composition*
	$\mid$	$\textbf{try}\ e\ \textbf{catch}\ \overline{exc}\ :\ e$	*exception handling*
	$\mid$	$\textbf{throw}\ exc$	*explicit exception generation*

Fig. 1. Abstract Syntax of AsmL-S.

will be addressed later. **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.

3.2. Class table

It is convenient to view a program as a class table together with the expression to be evaluated [17]. We assume that no class name is declared more than once and that there is no declaration for *Object*. The class table associates class names different from *Object* with the corresponding declarations.

Proviso 4. *For the remainder of this paper, we restrict attention to an arbitrary but fixed class table. In particular,* classes *will mean declared classes.*

If c is a class other than *Object*, then $\mathit{parent}(c)$ is the class c' extended by c according to the declaration of c. We assume that $\mathit{parent}(c)$ either equals *Object* or is declared earlier than c. $\mathit{addf}(c)$ is the sequence of distinct field names appearing in the declaration of c. The sequence of all fields of a class is defined by induction using the concatenation operation.

$$\begin{aligned}&\mathit{fldseq}(\mathit{Object}) = \varepsilon\\ &\mathit{fldseq}(c) = \mathit{addf}(c)\cdot \mathit{fldseq}(\mathit{parent}(c)).\end{aligned}$$

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

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

The situation is slightly more complicated with methods because, unlike fields, methods can be overridden. Let $\mathit{addm}(c)$ be the set of method names included in the declaration of c. We presume for simplicity that different method declarations of any class c have different names. We define inductively the set of all method names of a class.

$$\begin{aligned}&\mathit{mthset}(\mathit{Object}) = \emptyset\\ &\mathit{mthset}(c) = \mathit{addm}(c) \cup \mathit{mthset}(\mathit{parent}(c))\end{aligned}$$

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

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

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 $\mathit{dclr}(m, c)$ is the declaration of m in the class $\mathit{home}(m, c)$ defined as follows:

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

3.3. Subtyping

The subtype relation $\leqslant$ (relative to the underlying class table) is defined inductively by the following rules, where t, t', t'', τ, τ' are arbitrary types and c, c' are arbitrary classes.

- $t \leqslant t, \quad \dfrac{t \leqslant t' \quad t' \leqslant t''}{t \leqslant t''}$ $\qquad$ $\leqslant$ *is a partial order*
- $\dfrac{\mathit{parent}(c) = c'}{c \leqslant c'}$ $\qquad$ $\leqslant$ *extends the parent relation over classes*
- $\tau \to t \leqslant \mathit{Object}$ $\qquad$ *maps are objects*

- $\dfrac{\tau \leqslant \tau' \quad t \leqslant t'}{(\tau \to t) \leqslant (\tau' \to t')}$ *maps types are covariant in argument and result types*
- $\dfrac{t \leqslant Object}{Null \leqslant t}$ *Null lies beneath all object types*
- $Thrown \leqslant t$ *Thrown lies beneath all other types*.

Note that map types are covariant in both argument and result types which is consistent with the type system of AsmL and which fits many purposes. For example, maps are often used as lookup tables e.g. to represent dynamic functions of abstract state machines [9]. (In Section 5.3.4 we discuss the advantages and disadvantages of changing our type system such that map types are contravariant in argument types.)

The subtype relation is a partial order of a relatively simple form described in the following proposition. Call two types *comparable* if one of them is a subtype of the other; otherwise call them *incomparable*.

Proposition 5.

1. *The primitive types form an anti-chain with respect to* $\leqslant$ (*i.e. they are pairwise incomparable*). *No primitive type compares to Null*.
2. *Restricted to classes, the subtype relation is a* (*reflexive transitive*) *tree relation. The class tree is rooted at Object and lies above Null*. *No class compares to any primitive type*.
3. *The map types are located below Object and above Null*. *No map type compares to primitive types or subclasses of Object*.
4. *Below all these types is located Thrown*.
5. *For all map types* $t_1 \to t_2$ *and* $\tau_1 \to \tau_2$, *we have*

$$(t_1 \to t_2) \leqslant (\tau_1 \to \tau_2) \iff (t_1 \leqslant \tau_1) \wedge (t_2 \leqslant \tau_2).$$

The proof is straightforward. □

Corollary 6. *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* $t_1 \sqcup t_2$.

3.4. Well-typed expressions

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, $optype(and) = (Bool, Bool) \to Bool$.

A *type context T* is a function mapping local variables, possibly including *me*, to types. $\mathfrak{T}_T$ is a function associating certain expressions *e* with types. If $\mathfrak{T}_T(e)$ is defined, then *e* is said to be *well-typed* with respect to *T*.

The definition of $\mathfrak{T}_T(e)$ is inductive. The induction step splits into many rules, most of them self-explanatory. A comment, if any, follows the rule. As a notational shorthand, we write $\mathfrak{T}_T(e_1, \dots, e_n) = (t_1, \dots, t_n)$ to mean that $\mathfrak{T}_T(e_i) = t_i$ for all $i = 1, \dots, n$. The same applies to inequalities.

Note that, in the following rules, types t and τ may equal *Thrown*, but remember that *Thrown* is not available in the syntax and thus cannot occur in expressions.

Literals and local variables

T1. $\mathfrak{T}_T(lit) = littype(lit)$.

T2. $\dfrac{\ell \in \mathrm{dom}(T)}{\mathfrak{T}_T(\ell) = T(\ell)}$.

$\mathfrak{T}_T(\ell)$ is undefined when $\ell \notin \mathrm{dom}(T)$. It will follow that an expression e is well-typed with respect to T only if $\mathrm{dom}(T)$ contains all free variables in e.

Operations

T3. $\dfrac{optype(op) = \bar{\tau} \to t \qquad \mathfrak{T}_T(\bar{e}) \leqslant \bar{\tau}}{\mathfrak{T}_T(op(\bar{e})) = t}$.

Local binding

T4. $\dfrac{\mathfrak{T}_T(e_1) = t}{\mathfrak{T}_T(\textbf{let } \ell = e_1 \textbf{ do } e_2) = \mathfrak{T}_{T \circledleq \{\ell \mapsto t\}}(e_2)}$.

Here $T \circledleq \{\ell \mapsto t\}$ is the type context obtained from T either by adding $\ell \mapsto t$, if $\ell \notin \mathrm{dom}(T)$, or else by replacing $\ell \mapsto T(\ell)$ with $\ell \mapsto t$. The *override* operation $\circledleq$ is defined formally in Section 13.

Case distinction

T5. $\dfrac{\mathfrak{T}_T(e_1) = Bool}{\mathfrak{T}_T(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) = \mathfrak{T}_T(e_2) \sqcup \mathfrak{T}_T(e_3)}$.

Thus, **if** e_1 **then** e_2 **else** e_3 is well-typed with respect to T only if the least upper bound of $\mathfrak{T}_T(e_2)$ and $\mathfrak{T}_T(e_3)$ exists.

Class instances

T6. $\dfrac{fldinfo(c) = \bar{f} \textbf{ as } \bar{t} \qquad \mathfrak{T}_T(\bar{e}) \leqslant \bar{t}}{\mathfrak{T}_T(\textbf{new } c(\bar{e})) = c}$.

T7. $\dfrac{\mathfrak{T}_T(e) = c}{\mathfrak{T}_T(e.f) = fldtype(f, c)}$.

T8. $\dfrac{\mathfrak{T}_T(e_1) = c \qquad dclr(m, c) = m(\bar{\ell} \textbf{ as } \bar{\tau}) \textbf{ as } t \textbf{ do } e_3 \qquad \mathfrak{T}_T(\overline{e_2}) \leqslant \bar{\tau}}{\mathfrak{T}_T(e_1.m(\overline{e_2})) = t}$.

T9. $\dfrac{\mathfrak{T}_T(e_2) \leqslant \mathfrak{T}_T(e_1.f)}{\mathfrak{T}_T(e_1.f := e_2) = Void}$.

Maps

T10. $\dfrac{\mathfrak{T}_T(\overline{e_1}) \leqslant t_1 \qquad \mathfrak{T}_T(\overline{e_2}) \leqslant t_2}{\mathfrak{T}_T(\textbf{new } t_1 \to t_2 \ \{\overline{e_1} \mapsto \overline{e_2}\}) = t_1 \to t_2}$.

T11. $\dfrac{\mathfrak{T}_T(e_1) = \tau \to t \qquad \mathfrak{T}_T(e_2) \leqslant \tau}{\mathfrak{T}_T(e_1[e_2]) = t}$.

T12. $\dfrac{\mathfrak{T}_T(e_1) = \tau\to t \qquad \mathfrak{T}_T(e_2)\leqslant\tau \qquad \mathfrak{T}_T(e_3)\leqslant t}{\mathfrak{T}_T(e_1[e_2] := e_3) = \mathit{Void}}.$

T13. $\dfrac{\mathfrak{T}_T(e_1) = \tau\to t \qquad \mathfrak{T}_T(e_2)\leqslant\tau}{\mathfrak{T}_T(\textbf{remove } e_1[e_2]) = Void}.$

Note that map assignments require runtime type checking (for the same reason that array assignments of C# or Java require runtime type checking). For example, we may have a method that, given a map x of type $\mathit{Int} \to \mathit{Point}$, performs assignment $x[3] := \textbf{new } \mathit{Point}()$, which is statically correct. Later on, we extend *Point* to *ColoredPoint* so that the type $\mathit{Int} \to \mathit{ColoredPoint}$ is a subtype of $\mathit{Int} \to \mathit{Point}$. But passing a map of type $\mathit{Int} \to \mathit{ColoredPoint}$ to our method causes a problem. See also examples 23 and 24.

Type test and type cast

T14. $\dfrac{t < \mathfrak{T}_T(e)}{\mathfrak{T}_T(e \textbf{ is } t) = \mathit{Bool}}.$

T15. $\dfrac{t < \mathfrak{T}_T(e)}{\mathfrak{T}_T(e \textbf{ as } t) = t}.$

Casting into a subtype is viewed valid at compile time but may turn out to be invalid at runtime. Thus, casts must be rechecked at runtime.

The premise $t < \mathfrak{T}_T(e)$ requires an explanation. Why do we restrict type casting to this one case? If $\mathfrak{T}_T(e)\leqslant t$, then, by type soundness (theorem 18), e **is** t must evaluate to *true* unless an exception occurs. If $\mathfrak{T}_T(e)$ and t have no lower bound other than *Thrown*, then type soundness implies that e **is** t must evaluate to *false*. In either case, the expression e **is** t is superfluous and can harmlessly (perhaps usefully) be disallowed. There is a third possibility: $\mathfrak{T}_T(e)$ and t are incomparable but have a lower bound $t' > \mathit{Thrown}$. In this case, we can replace e **is** t with the more reasonable e **is** $t \sqcap \mathfrak{T}_T(e)$. (Note that the greatest lower bound exists by corollary 6.)

Parallel, nondeterministic and sequential composition

T16. $\dfrac{\mathfrak{T}_T(e_1) \text{ is defined}}{\mathfrak{T}_T(e_1 \parallel e_2) = \mathfrak{T}_T(e_2)}.$

This reflects the intention that an expression $e_1 \parallel e_2$ outputs the value produced by e_2 unless an exception is thrown. There are good ways to restore the symmetry of the parallel composition. This issue will be discussed later on.

T17. $\dfrac{\mathfrak{T}_T(e_1) = \tau\to t \qquad \mathfrak{T}_{T \otimes\{\ell\mapsto\tau\}}(e_2) \text{ is defined}}{\mathfrak{T}_T(\textbf{forall } \ell \textbf{ in } e_1 \textbf{ do } e_2) = \mathit{Void}}.$

T18. $\mathfrak{T}_T(e_1 \,[]\, e_2) = \mathfrak{T}_T(e_1) \sqcup \mathfrak{T}_T(e_2).$

T19. $\dfrac{\mathfrak{T}_T(e_1) = \tau\to t}{\mathfrak{T}_T(\textbf{choose } \ell \textbf{ in } e_1 \textbf{ do } e_2) = \mathfrak{T}_{T \otimes\{\ell\mapsto\tau\}}(e_2)}.$

The part **choose** ℓ **in** e_1 refers to choosing an element in the domain of e_1 (rather than choosing an entire maplet).

T20. $\dfrac{\mathfrak{T}_T(e_1) \text{ is defined}}{\mathfrak{T}_T(e_1\,;\,e_2) = \mathfrak{T}_T(e_2)}.$

T21. $\dfrac{\mathfrak{T}_T(e_1) = \mathit{Bool} \qquad \mathfrak{T}_T(e_2) \text{ is defined}}{\mathfrak{T}_T(\textbf{while } e_1 \textbf{ do } e_2) = \mathit{Void}}.$

Exception generation and handling

T22. $\mathfrak{T}_T(\textbf{throw } \mathit{exc}) = \mathit{Thrown}.$

T23. $\mathfrak{T}_T(\textbf{try } e_1 \textbf{ catch } \overline{\mathit{exc}} : e_2) = \mathfrak{T}_T(e_1) \sqcup \mathfrak{T}_T(e_2).$

Remark 7. Typing rules T1–T23 could be strengthened so as to filter out certain degenerate expressions like $7 + (\textbf{throw } \mathit{fooX})$ which always evaluates to an exception even though it is well-typed. See Section 5.3.6 in this connection.

3.5. Well-formed programs

A program is *well-formed* if all of its classes are well-formed and its body is well-typed in the empty type context. A class c is *well-formed* if every method $m \in \mathit{mthset}(c)$ is well-formed relative to c, symbolically m ok in c.

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

- $\dfrac{m \in \mathit{addm}(c) - \mathit{mthset}(c') \qquad \mathfrak{T}_T(e) \leqslant t}{m \text{ ok in } c}.$
- $\dfrac{m \in \mathit{mthset}(c) - \mathit{addm}(c) \qquad m \text{ ok in } c'}{m \text{ ok in } c}.$
- $\dfrac{\begin{pmatrix} m \in \mathit{addm}(c) \cap \mathit{mthset}(c') \qquad \mathfrak{T}_T(e) \leqslant t \qquad m \text{ ok in } c' \\ \mathit{dclr}(m, c') = m(\ell'_1 \textbf{ as } \tau'_1, \ldots, \ell'_n \textbf{ as } \tau'_n) \textbf{ as } t' \textbf{ do } e' \qquad \bar{\tau} \to t \leqslant \bar{\tau}' \to t' \end{pmatrix}}{m \text{ ok in } c}.$

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

Proviso 8. *In the sequel, we assume that all classes in the underlying class table are well-formed.*

3.6. Analysis: type contexts

The results in this subsection are not used until Section 5. We include them here because they belong naturally in the present section on static semantics.

3.6.1. Induced type contexts

Let $\circ$ be a distinguished element of *LocalVar*. A *punctured expression* is an expression e with a unique free occurrence of variable $\circ$; this is sometimes written as $e(\circ)$. For any expression e', let $e(e'/\circ)$ denote the expression obtained from e by substituting the unique free occurrence of $\circ$ with e'.

For every punctured expression $e(\circ)$ and type context T, we define the *induced type context* $T \circledast e(\circ)$ at $\circ$ in e with respect to T.

- If e is $\circ$ then $T \circledast e(\circ) = T$.
- If e has any of the forms

$$\textbf{let } \ell = e' \textbf{ do } e''(\circ), \quad \textbf{forall } \ell \textbf{ in } e' \textbf{ do } e''(\circ), \quad \textbf{choose } \ell \textbf{ in } e' \textbf{ do } e''(\circ),$$

  then $T \circledast e(\circ) = (T \circledleq \{\ell \mapsto \mathfrak{T}_T(e')\}) \circledast e''(\circ)$. For example,
  $T \circledast \big(\textbf{let } \ell = 7 \textbf{ do } (11 + \circ)\big) = T \circledleq \{\ell \mapsto \mathit{Int}\}$.
- Otherwise $T \circledast e(\circ) = T \circledast e_0(\circ)$ where $e_0(\circ)$ is the unique maximal proper punctured subexpression of e. For example, if $e = e' \parallel e''(\circ)$ then $T \circledast e(\circ) = T \circledast e''(\circ)$.

Proposition 9. *If $\mathfrak{T}_{T \circledast e(\circ)}(e') = \mathfrak{T}_{T \circledast e(\circ)}(e'')$ then $\mathfrak{T}_T(e(e'/\circ)) = \mathfrak{T}_T(e(e''/\circ))$.*

The proposition is proven by a straightforward induction on $e(\circ)$.

We will not need the concepts of punctured expressions or induced type contexts until Section 5.2.

3.6.2. Dominating type contexts

Let T and T' be any type contexts. We say T' *dominates* T, written $T \preccurlyeq T'$, if $T(\ell) \leqslant T'(\ell)$ for all $\ell \in \mathrm{dom}(T)$.

Theorem 10. *If $T \preccurlyeq T'$ and both $\mathfrak{T}_T(e)$, $\mathfrak{T}_{T'}(e)$ are defined, then $\mathfrak{T}_T(e) \leqslant T_{T'}(e)$.*

Proof. Proof is by induction on e. Assume that the statement hold for all proper subexpressions of e. By examination of typing rules T1–T23, we show that the statement holds for e as well.

T1–T3, *T6*, *T9*–T10, *T12–T17*, *T20–T22*: These cases are obvious. For instance, if e is of the form **remove** $e_1[e_2]$ then $\mathfrak{T}_T(e) = \mathfrak{T}_{T'}(e) = \mathit{Void}$ by rule T13.

T4: Suppose e is of the form **let** $\ell = e_1$ **do** e_2. Let $t = \mathfrak{T}_T(e_1)$ and $t' = \mathfrak{T}_{T'}(e_1)$. Then $t \leqslant t'$ by the induction hypothesis. Therefore, $T \circledleq \{\ell \mapsto t\} \preccurlyeq T' \circledleq \{\ell \mapsto t'\}$. Using the induction hypothesis again, we have

$$\mathfrak{T}_T(e) = \mathfrak{T}_{T \circledleq \{\ell \mapsto t\}}(e_2) \leqslant \mathfrak{T}_{T' \circledleq \{\ell \mapsto t'\}}(e_2) = \mathfrak{T}_{T'}(e).$$

T5, *T18*, *T23*: Suppose e is any of the following: **if** e_0 **then** e_1 **else** e_2, e_1 [] e_2, or **try** e_1 **catch**$\overline{exc}$: e_2. Then $\mathfrak{T}_T(e_i) \leqslant \mathfrak{T}_{T'}(e_i)$ for $i = 1, 2$ by the induction hypothesis. Therefore,

$$\mathfrak{T}_T(e) = \mathfrak{T}_T(e_1) \sqcup \mathfrak{T}_T(e_2) \leqslant \mathfrak{T}_{T'}(e_1) \sqcup \mathfrak{T}_{T'}(e_2) = \mathfrak{T}_{T'}(e).$$

T7: Suppose e is of the form $e_0.f$. Let $c = \mathfrak{T}_T(e_0)$ and $c' = \mathfrak{T}_{T'}(e_0)$. Then f is a field of both c and c'. Since $c \leqslant c'$, well-formedness of the underlying class table implies that f is declared in a unique common ancestor of c and c'. Therefore,

$$\mathfrak{T}_T(e) = \mathit{fldtype}(f, c) = \mathit{fldtype}(f, c') = \mathfrak{T}_{T'}(e).$$

T8: Suppose e is of the form $e_0.m(\bar{e})$. Let $c = \mathfrak{T}_T(e_0)$, $c' = \mathfrak{T}_{T'}(e_0)$, $\bar{\tau} \to t = \mathit{mthtype}(m, c)$, and $\bar{\tau}' \to t' = \mathit{mthtype}(m, c')$. Since $c \leqslant c'$, the well-formedness of m relative to c and c' implies that $\bar{\tau} \to t \leqslant \bar{\tau}' \to t'$. In particular, $\mathfrak{T}_T(e) = t \leqslant t' = \mathfrak{T}_{T'}(e)$.
T11: Suppose e is of the form $e_1[e_2]$. Let $\tau \to t = \mathfrak{T}_T(e_1)$ and $\tau' \to t' = \mathfrak{T}_{T'}(e_1)$. Invoking the induction hypothesis, we have $\mathfrak{T}_T(e) = t \leqslant t' = \mathfrak{T}_{T'}(e)$.
T19: Suppose e is of the form **choose** ℓ **in** e_1 **do** e_2. Again, let $\tau \to t = \mathfrak{T}_T(e_1)$ and $\tau' \to t' = \mathfrak{T}_{T'}(e_1)$. By the induction hypothesis, $\tau \leqslant \tau'$. It follows that $T \circledleq \{\ell \mapsto \tau\} \leqslant T' \circledleq \{\ell \mapsto \tau'\}$. Using the induction hypothesis again, we have

$$\begin{aligned}\mathfrak{T}_T(\textbf{choose } \ell \textbf{ in } e_1 \textbf{ do } e_2) &= \mathfrak{T}_{T \circledleq \{\ell \mapsto \tau\}}(e_2) \\ &\leqslant \mathfrak{T}_{T' \circledleq \{\ell \mapsto \tau'\}}(e_2) = \mathfrak{T}_{T'}(\textbf{choose } \ell \textbf{ in } e_1 \textbf{ do } e_2). \quad \square\end{aligned}$$

4. Operational semantics

By induction on expressions e, we define the effect $\mathfrak{E}_{b,s}(e)$ of executing e (starting) at a given store s under a binding b for the free variables of e. This allows us to define the effect of executing a program.

Our semantics is structural operational semantics (SOS) in the sense that it is operational and is defined by induction on syntactical structure. It is thus similar to Plotkin's structural operational semantics [24]. People distinguish between small-step and big-step styles of structural operational semantics [23]. The latter is sometimes called natural semantics [19]. Our semantics is of the big-step variety.

However, we break the SOS tradition as far as the interaction with the outside world is concerned. To query the outside world, we use *external functions*; we use them the same way they are used in abstract state machines [9]; we do not presume any familiarity with abstract state machines, however. The question arises why to break the tradition. (One of our referees insisted that we address this question.) Well, there are two aspects of AsmL-S that require the intervention of outside world. One is nondeterminism[1] and the other is the creation of new objects. Traditional SOS deals elegantly with nondeterminism. It is more awkward to account for new-object creation in traditional SOS, especially when, as in our case, multiple new objects are created in parallel. More importantly, the full AsmL is highly interactive, and so our semantics should scale up with respect to additional kinds of interaction with the outside world.

Remark 11. We speak here about *intra-step interaction*, a kind of interaction that occurs within one step of a program. The resolution of nondeterminism and new object creation are examples of such intra-step interaction. Other examples include calling library routines or foreign methods. Without loss of generality, intra-step interaction can be conducted by issuing queries and receiving replies [3]. Call an interactive algorithm *ordinary* if it

[1] The point that an algorithm needs an outside world to resolve nondeterminism is argued in [11, Section 9.1].

completes a step only after all the queries from that step have been answered and if it uses no information from the outside world except for the answers to its queries. An axiomatization of ordinary sequential algorithms with intra-step interaction is found in [3]. It turns out that external functions are sufficient to support ordinary interaction with the outside world [4].

The rest of this section is as follows. In Section 4.1, we define some semantic domains and functions that are needed in the definition of $\mathfrak{E}_{b,s}(e)$. In Section 4.2, we introduce evaluation contexts, effects and two external functions that take care of object construction and nondeterminism. Section 4.3 is devoted to a recursive definition of $\mathfrak{E}_{b,s}(e)$ and a definition of the effect $\mathit{Effect}(\pi)$ of a program π. A type soundness theorem is formulated in Section 4.2 and proved in Section 5.1.

4.1. Stores

Let *Literal*, *LocalVar*, *ObjectId*, *Class*, *FieldId*, *MapType*, *Exception* be the following disjoint sets: the AsmL-S literals, an infinite set of local variables including *me*, a pool of potential object ids, the classes of the underlying class table, the field names of these classes, the map types generated by these classes, the set of built-in exceptions plus an infinite set of potential custom exceptions. Let *DEL* be a fresh symbol, not occurring anywhere in the AsmL-S syntax.

We define a few additional sets of interest. Some of them have been described—in a preliminary way—in Section 2. If α, β are sets, then $\alpha \to \beta$ denotes the set of partial functions from α to β. $\wp(\alpha)$ denotes the powerset of α.

$$\begin{aligned} \mathit{Nvalue} &= \mathit{ObjectId} \cup \mathit{Literal}, \\ \mathit{Value} &= \mathit{Nvalue} \cup \mathit{Exception}. \end{aligned}$$

Elements of *Nvalue* are called *normal values*.

$$\begin{aligned} \mathit{TypeMap} &= \mathit{ObjectId} \to (\mathit{Class} \cup \mathit{MapType}) \\ \mathit{Index} &= \mathit{FieldId} \cup \mathit{Nvalue} \\ \mathit{ContentMap} &= \mathit{ObjectId} \to (\mathit{Index} \to \mathit{Nvalue}) \\ \mathit{Update} &= (\mathit{ObjectId} \times \mathit{Index}) \times (\mathit{Nvalue} \cup \{\mathit{DEL}\}) \\ \mathit{UpdateSet} &= \wp(\mathit{Update}). \end{aligned}$$

If θ is a type map and t is a type, then we define

$$\mathit{Nvalue}_\theta(t) = \{o : \theta(o) \leqslant t\} \cup \{\mathit{lit} : \mathit{littype}(\mathit{lit}) \leqslant t\}.$$

States of a computation are represented by stores. Formally, a *store* is a triple $s = (\theta, \omega, u)$, where θ is a type map, ω is a content map and u is an update set, that satisfies the following three conditions.

(a) $\mathrm{dom}(\theta) = \mathrm{dom}(\omega) \supseteq \{o : ((o, i), v) \in u\}$ where v could be *DEL*

(b) if $\theta(o) = c \in \mathit{Class}$ and $\mathit{fldinfo}(c) = \bar{f}$ **as** $\bar{t}$ then

- $\mathrm{dom}(\omega(o)) = \{f_1, \ldots, f_n\}$
- $\omega(o)(\bar{f}) \in \mathit{Nvalue}_\theta(\bar{t})$
- $((o, i), v) \in u \implies i \in \{f_1, \ldots, f_n\}$ and $v \in \mathit{Nvalue}_\theta(\mathit{fldtype}(i, c))$

(c) if $\theta(o) = \tau \to t \in \mathit{MapType}$ then
- $\mathrm{dom}(\omega(o)) \subseteq \mathit{Nvalue}_\theta(\tau)$
- $\mathrm{rng}(\omega(o)) \subseteq \mathit{Nvalue}_\theta(t)$
- $((o, i), v) \in u \implies i \in \mathit{Nvalue}_\theta(\tau)$ and $v \in \mathit{Nvalue}_\theta(t) \cup \{\mathit{DEL}\}$.

If s is a store, we will sometimes write θ_s, ω_s and u_s to denote the components of s. To clarify our intentions, here are explanations in plain language.

The domain of θ and ω is the set of object ids allocated prior to the evaluation of the expression. Once an object is created, its id persists until the end of the run of the program. That is, unless the object becomes unreachable and is garbage-collected; however, for purposes of semantics, garbage collection can be ignored.

θ maps allocated objects to their runtime types. Once declared, an object's runtime type never changes. The content map ω associates objects with functions representing their, well, contents. If o is an instance of class c, then $\omega(o)$ maps the field names of c to their values in o. If o is an object of type $\tau \to t$, then $\omega(o)$ is the map represented by o.

Remark 12. Alternatively (and closer to the traditional ASM paradigm), let $\mathit{Location} = \mathit{ObjectId} \times \mathit{Index}$. Then *ContentMap* can be defined as $\mathit{Location} \to \mathit{Nvalue}$. This explains why updates are represented as pairs in $\mathit{Location} \times (\mathit{Nvalue} \cup \{\mathit{DEL}\})$.

There are two kinds of updates: modifications and removals. A modification update puts a new value into a given location. Formally, this is a pair $((o, f), v)$, where o is a class instance, f is a field of o and v is a value of the appropriate type, or else $((o, v_1), v_2)$, where o is an object of map type and v_1, v_2 are values of the appropriate domain and codomain types, respectively. It is not required that the new value differs from the old one. Since updates may be performed simultaneously, a trivial update, where the new value equals the old one, may have semantical significance: it may clash with another update of the same location. A removal update, formally a pair $((o, v), \mathit{DEL})$, removes a given map location. We say u is *inconsistent* if it contains distinct updates of the same location; otherwise, it is *consistent*. A consistent update set thus gives rise to a content map in the alternative sense: from $\mathit{ObjectId} \times \mathit{Index}$ to $\mathit{Nvalue} \cup \{\mathit{DEL}\}$.

Notation 13. Let R_1, R_2 be any binary relations and m_1, m_2 any maps.
- The *override of R_1 by R_2* is defined by

$$R_1 \lessdot R_2 = \left\{(x, y) \in R_1 : \nexists z \,.\, (x, z) \in R_2\right\} \cup R_2.$$

- The *override of m_1 by m_2* is defined by

$$(m_1 \lessdot m_2)(x) = \begin{cases} m_1(x) & \text{if } x \in \mathrm{dom}(m_1) - \mathrm{dom}(m_2) \\ m_2(x) & \text{if } x \in \mathrm{dom}(m_2). \end{cases}$$

- The *union of R_1 and R_2* is defined, in the usual way, as $R_1 \cup R_2$.
- If $m_1 \lessdot m_2 = m_2 \lessdot m_1$, then the *union of m_1 and m_2* is defined by

$$(m_1 \cup m_2)(x) = \begin{cases} m_1(x) & \text{if } x \in \mathrm{dom}(m_1) \\ m_2(x) & \text{if } x \in \mathrm{dom}(m_2). \end{cases}$$

If $m_1 \lessdot m_2 \neq m_2 \lessdot m_1$, then $m_1 \cup m_2$ is undefined.

- By extension, for any stores $s_1 = (\theta_1, \omega_1, u_1)$ and $s_2 = (\theta_2, \omega_2, u_2)$ we define

$$s_1 \cup s_2 = (\theta_1 \cup \theta_2,\ \omega_1 \cup \omega_2,\ u_1 \cup u_2)$$

provided that maps $\theta_1 \cup \theta_2$ and $\omega_1 \cup \omega_2$ are defined. ($u_1 \cup u_2$ is always defined, since u_1 and u_2 are binary relations.) Check that if $s_1 \cup s_2$ is defined then it meets the definition of a store.
- We say s_2 *extends* s_1, written $s_1 \subseteq s_2$, if $s_1 \cup s_2 = s_2$.

Remark 14. If $G(m) = \{(x, y) : m(x) = y\}$ denotes the graph of a map m, then $G(m_1 \lessdot m_2) = G(m_1) \lessdot G(m_2)$ and $G(m_1 \cup m_2) = G(m_1) \cup G(m_2)$ when $m_1 \cup m_2$ is defined.

Remark 15. Representing *ContentMap* in the form $\alpha \to (\beta \to \gamma)$, rather than $(\alpha \times \beta) \to \gamma$, allows us to use the convenient override operation $\lessdot$.

Firing updates. Let $s = (\theta, \omega, u)$ be any store. If u is consistent, then it gives rise to a new store $\hat{s} = (\theta,\ \hat{\omega},\ \emptyset)$ where content map $\hat{\omega}$ is defined by

$$\hat{\omega}(o)(i) = \begin{cases} v & \text{if } ((o, i), v) \in u \text{ and } v \neq \mathit{DEL}, \\ \omega(o)(i) & \text{if } ((o, i), \mathit{DEL}) \notin u. \end{cases}$$

$\hat{s}$ is the store obtained from s by "firing" all updates in u. If u is inconsistent, then $\hat{s}$ is undefined.

4.2. Evaluation contexts, effects, and external functions

An *evaluation context* is a pair (b, r) consisting of a store r and a *binding* b, which is a partial function from *LocalVar* to $\mathrm{dom}(\theta_r) \cup \mathit{Literal}$. Every evaluation context (b, r) gives rise to a type context $[b, r]$ where

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

Check that, if a store s extends r, then (b, s) is an evaluation context and $[b, r] = [b, s]$. (We will use this fact extensively in the type soundness proof.)

An expression 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.

An *effect* is a pair $\langle s, v\rangle$ (the angular brackets are used only for the purpose of visual distinction) consisting of a store s and a value v in $\mathrm{dom}(\theta_s) \cup \mathit{Literal} \cup \mathit{Exception}$. The type of effect $\langle s, v\rangle$ is defined in the obvious way:

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

In the next subsection, we define an operator $\mathfrak{E}_{b,r}$ over (b, r)-typed expressions. The computation of $\mathfrak{E}_{b,r}(e)$ is in general nondeterministic and it may diverge. If it converges, it produces an effect $\mathfrak{E}_{b,r}(e) = \langle s, v\rangle$. In Section 5.1 we prove:

Theorem 18 (*Type soundness*). *$type(\mathfrak{E}_{b,r}(e)) \leqslant \mathfrak{T}_{[b,r]}(e)$ for all b, r, e and any converging computation of $\mathfrak{E}_{b,r}(e)$.*

The definition of the effect operator $\mathfrak{E}_{b,r}$ utilizes two external functions. One of them is a nullary function freshid. Every evaluation of freshid produces a new object id. Different invocations of freshid produce different objects. The other external function is a unary function oneof(X). It takes a nonempty set X as an argument and returns one of its elements. We presume that the outside environment guarantees that the two external functions work properly.

The effect $\mathfrak{E}_{b,r}(e)$ is nondeterministic only because of the use of external functions. Due to the use of external functions, the effect depends on the outside environment. We keep the dependence of the effect $\mathfrak{E}_{b,r}(e)$ on the environment implicit. The equality $\mathfrak{E}_{b,r}(e) = \langle s, v\rangle$ means that some convergent computation of $\mathfrak{E}_{b,r}(e)$ produces the effect $\langle s, v\rangle$. The equality $\mathfrak{E}_{b,r}(e) = \mathfrak{E}_{b',r'}(e')$ means that

- the range of possible convergent effects of $\mathfrak{E}_{b,r}(e)$ equals the range of possible convergent effects of $\mathfrak{E}_{b',r'}(e')$, and
- there is a divergent computation of $\mathfrak{E}_{b,r}(e)$ if, only if, there is a divergent computation of $\mathfrak{E}_{b',r'}(e')$.

The range of possible effects of $\mathfrak{E}_{b,r}(e)$ does not depend on the environment.

4.3. Definition of the effect operator

This section is devoted to a recursive definition of the effect operator $\mathfrak{E}_{b,r}(e)$ over (b, r)-typed expressions. The recursion reflects the inductive definition of the abstract syntax of AsmL-S.

Proviso. *In rules* E1–E32, *the symbols $v, v', v_1, v_2, \ldots$ stand for normal values, not exceptions. In rules* E33–E40 (*dealing with exception generation, handling and propagation*), *these same symbols represent any values* (*normal or exceptional*). *In this way, we separate the rules for normal evaluations from those for exception handling and propagation.*

Literals and local variables

E1. $\mathfrak{E}_{b,r}(lit) = \langle r, lit\rangle$.

E2. $\mathfrak{E}_{b,r}(\ell) = \langle r, b(\ell)\rangle$.

Operations

E3. $$\frac{\mathfrak{E}_{b,r}(\bar{e}) = \langle \bar{s}, \bar{v}\rangle \qquad op(\bar{v}) \text{ is defined}}{\mathfrak{E}_{b,r}(op(\bar{e})) = \langle \bigcup \bar{s}, op(\bar{v})\rangle}.$$

E4. $$\frac{\mathfrak{E}_{b,r}(\bar{e}) = \langle \bar{s}, \bar{v}\rangle \qquad op(\bar{v}) \text{ is undefined}}{\mathfrak{E}_{b,r}(op(\bar{e})) = \langle r, argX\rangle}.$$

Local Binding

E5. $$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s, v\rangle}{\mathfrak{E}_{b,r}(\mathbf{let}\ \ell = e_1\ \mathbf{do}\ e_2) = \mathfrak{E}_{b\ \otimes\{\ell \mapsto v\},\ s}(e_2)}.$$

Case distinction

E6. $$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s, \mathit{true}\rangle}{\mathfrak{E}_{b,r}(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) = \mathfrak{E}_{b,s}(e_2)}.$$

E7. $$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s, \mathit{false}\rangle}{\mathfrak{E}_{b,r}(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) = \mathfrak{E}_{b,s}(e_3)}.$$

Null exceptions

E8. $$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s, \mathit{null}\rangle}{\mathfrak{E}_{b,r}(e_1.f) = \mathfrak{E}_{b,r}(\textbf{forall } \ell \textbf{ in } e_1 \textbf{ do } e_2) = \mathfrak{E}_{b,r}(\textbf{choose } \ell \textbf{ in } e_1 \textbf{ do } e_2) = \langle r, \mathit{nullX}\rangle}$$

$$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s, \mathit{null}\rangle \qquad \mathit{type}(\mathfrak{E}_{b,r}(e_2)) \neq \mathit{Thrown}}{\mathfrak{E}_{b,r}(e_1.f := e_2) = \mathfrak{E}_{b,r}(e_1[e_2]) = \mathfrak{E}_{b,r}(\textbf{remove } e_1[e_2]) = \langle r, \mathit{nullX}\rangle}$$

$$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s, \mathit{null}\rangle \qquad \mathit{type}(\mathfrak{E}_{b,r}(e_{2,3})) \neq \mathit{Thrown}}{\mathfrak{E}_{b,r}(e_1[e_2] := e_3) = \langle r, \mathit{nullX}\rangle}$$

$$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s, \mathit{null}\rangle \qquad \mathit{type}(\mathfrak{E}_{b,r}(\overline{e_2})) \neq \mathit{Thrown}}{\mathfrak{E}_{b,r}(e_1.m(\overline{e_2})) = \langle r, \mathit{nullX}\rangle}.$$

Class instances

E9. $$\frac{\mathfrak{E}_{b,r}(\overline{e}) = \langle \overline{s}, \overline{v}\rangle \qquad \mathsf{freshid}() = o}{\mathfrak{E}_{b,r}(\textbf{new } c(\overline{e})) = \langle r \cup \bigcup \overline{s} \cup \big(\{o \mapsto c\}, \{o \mapsto \{\overline{f} \mapsto \overline{v}\}\}, \emptyset\big),\ o\rangle}.$$

We include "$r \cup$" in case $\mathit{fldseq}(c)$ is the empty sequence ε.

E10. $$\frac{\mathfrak{E}_{b,r}(e) = \langle s, v\rangle \qquad v \neq \mathit{null}}{\mathfrak{E}_{b,r}(e.f) = \langle s,\ \omega_s(v)(f)\rangle}.$$

E11. $$\frac{\left(\begin{array}{c}\mathfrak{E}_{b,r}(e_1) = \langle s_1, v_1\rangle \qquad \mathfrak{E}_{b,r}(\overline{e_2}) = \langle \overline{s_2}, \overline{v_2}\rangle \qquad \mathit{type}(\langle s_1, v_1\rangle) = c \\ \mathit{dclr}(m, c) = m(\overline{\ell} \textbf{ as } \overline{\tau}) \textbf{ as } t \textbf{ do } e_3\end{array}\right)}{\mathfrak{E}_{b,r}(e_1.m(\overline{e_2})) = \mathfrak{E}_{\{\mathit{me} \mapsto v_1,\, \overline{\ell} \mapsto \overline{v_2}\},\, s_1 \cup \bigcup \overline{s_2}}(e_3)}.$$

Note that both e_1 and $\overline{e_2}$ are evaluated in store r. A similar remark applies to a number of other rules. The binding in the latter evaluation context is $\{\mathit{me} \mapsto v_1,\ \overline{\ell} \mapsto \overline{v_2}\}$ rather than $b \otimes \{\mathit{me} \mapsto v_1,\ \overline{\ell} \mapsto \overline{v_2}\}$ since the free variables in e_3 are contained among $\overline{\ell} \cup \{\mathit{me}\}$ as a consequence of the well-formedness of m relative to c.

Remark 16. Almost every rule in the recursive definition of $\mathfrak{E}_{b,r}(e)$ reduces $\mathfrak{E}_{b,r}(e)$ to effects $\mathfrak{E}_{b,s}(e')$ where e' is a proper subexpression of e. Rules E11 (method calls) and E32 (while-expressions) are the only exceptions. Consequentially, these rules are the only reasons that computation of $\mathfrak{E}_{b,r}(e)$ may diverge.

E12. $$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s_1, v_1\rangle \qquad \mathfrak{E}_{b,r}(e_2) = \langle s_2, v_2\rangle \qquad v_1 \neq \mathit{null}}{\mathfrak{E}_{b,r}(e_1.f := e_2) = \langle s_1 \cup s_2 \cup \big(\emptyset, \emptyset, \{((v_1, f), v_2)\}\big),\ \mathit{void}\rangle}.$$

The empty type map as well as the empty context map are denoted by ∅.

Maps

E13. $$\frac{\begin{pmatrix} \mathfrak{E}_{b,r}(\overline{e_1}) = \langle \overline{s_1}, \overline{v_1}\rangle & \mathfrak{E}_{b,r}(\overline{e_2}) = \langle \overline{s_2}, \overline{v_2}\rangle \\ \mathit{consistent}(\overline{v_1}, \overline{v_2}) & \mathsf{freshid}() = o \end{pmatrix}}{\begin{array}{c}\mathfrak{E}_{b,r}(\mathbf{new}\ \tau \to t\ \{\overline{e_1} \mapsto \overline{e_2}\}) \\ = \langle r \cup \bigcup \overline{s_1} \cup \bigcup \overline{s_2} \cup \big(\{o \mapsto \tau \to t\}, \{o \mapsto \{\overline{v_1} \mapsto \overline{v_2}\}\}, \emptyset\big),\ o\rangle\end{array}},$$
where $\mathit{consistent}(a_1, \ldots, a_n, b_1, \ldots, b_n) = \bigwedge_{1 \leqslant i,j \leqslant n} (a_i = a_j) \leftrightarrow (b_i = b_j)$.

E14. $$\frac{\mathfrak{E}_{b,r}(\overline{e_1}) = \langle \overline{s_1}, \overline{v_1}\rangle \qquad \mathfrak{E}_{b,r}(\overline{e_2}) = \langle \overline{s_2}, \overline{v_2}\rangle \qquad \neg\mathit{consistent}(\overline{v_1}, \overline{v_2})}{\mathfrak{E}_{b,r}(\mathbf{new}\ \tau \to t\ \{\overline{e_1} \mapsto \overline{e_2}\}) = \langle r,\ \mathit{argconsistencyX}\rangle}.$$

E15. $$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s_1, v_1\rangle \qquad \mathfrak{E}_{b,r}(e_2) = \langle s_2, v_2\rangle \qquad v_2 \in \mathrm{dom}(\omega_{s_1}(v_1))}{\mathfrak{E}_{b,r}(e_1[e_2]) = \langle s_1 \cup s_2,\ \omega_{s_1}(v_1)(v_2)\rangle}.$$

E16. $$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s_1, v_1\rangle \qquad \mathfrak{E}_{b,r}(e_2) = \langle s_2, v_2\rangle \qquad v_2 \notin \mathrm{dom}(\omega_{s_1}(v_1))}{\mathfrak{E}_{b,r}(e_1[e_2]) = \langle r,\ \mathit{mapkeyX}\rangle}.$$

E17. $$\frac{\begin{pmatrix} \mathfrak{E}_{b,r}(e_1) = \langle s_1, v_1\rangle \qquad \mathfrak{E}_{b,r}(e_2) = \langle s_2, v_2\rangle \qquad \mathfrak{E}_{b,r}(e_3) = \langle s_3, v_3\rangle \\ \mathit{type}(s_2, v_2) \to \mathit{type}(s_3, v_3) \leqslant \mathit{type}(s_1, v_1) \end{pmatrix}}{\mathfrak{E}_{b,r}(e_1[e_2] := e_3) = \langle s_1 \cup s_2 \cup s_3 \cup \big(\emptyset, \emptyset, \{((v_1, v_2), v_3)\}\big),\ \mathit{void}\rangle}.$$

E18. $$\frac{\begin{pmatrix} \mathfrak{E}_{b,r}(e_1) = \langle s_1, v_1\rangle \qquad \mathfrak{E}_{b,r}(e_2) = \langle s_2, v_2\rangle \qquad \mathfrak{E}_{b,r}(e_3) = \langle s_3, v_3\rangle \\ \mathit{type}(\langle s_2, v_2\rangle) \to \mathit{type}(\langle s_3, v_3\rangle) \nleqslant \mathit{type}(\langle s_1, v_1\rangle) \end{pmatrix}}{\mathfrak{E}_{b,r}(e_1[e_2] := e_3) = \langle r,\ \mathit{maptypeX}\rangle}.$$

E19. $$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s_1, v_1\rangle \qquad \mathfrak{E}_{b,r}(e_2) = \langle s_2, v_2\rangle \qquad v_1 \neq \mathit{null}}{\mathfrak{E}_{b,r}(\mathbf{remove}\ e_1[e_2]) = \langle s_1 \cup s_2 \cup \big(\emptyset, \emptyset, \{((v_1, v_2), \mathit{DEL})\}\big),\ \mathit{void}\rangle}.$$

Type test and type cast

E20. $$\frac{\mathfrak{E}_{b,r}(e) = \langle s, v\rangle \qquad \mathit{type}(\langle s, v\rangle) \leqslant t}{\mathfrak{E}_{b,r}(e\ \mathbf{is}\ t) = \langle s, \mathit{true}\rangle}.$$

E21. $$\frac{\mathfrak{E}_{b,r}(e) = \langle s, v\rangle \qquad \mathit{type}(\langle s, v\rangle) \nleqslant t}{\mathfrak{E}_{b,r}(e\ \mathbf{is}\ t) = \langle s, \mathit{false}\rangle}.$$

E22. $$\frac{\mathfrak{E}_{b,r}(e) = \langle s, v\rangle \qquad \mathit{type}(\langle s, v\rangle) \leqslant t}{\mathfrak{E}_{b,r}(e\ \mathbf{as}\ t) = \langle s, v\rangle}.$$

E23. $$\frac{\mathfrak{E}_{b,r}(e) = \langle s, v\rangle \qquad \mathit{type}(\langle s, v\rangle) \nleqslant t}{\mathfrak{E}_{b,r}(e\ \mathbf{as}\ t) = \langle r, \mathit{castX}\rangle}.$$

Parallel, nondeterministic and sequential composition

E24. $$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s_1, v_1\rangle \qquad \mathfrak{E}_{b,r}(e_2) = \langle s_2, v_2\rangle}{\mathfrak{E}_{b,r}(e_1 \parallel e_2) = \langle s_1 \cup s_2,\ v_2\rangle}.$$

The do-in-parallel operation $\|$ returns the combined stores of both effects together with the value of the second effect. (An alternative semantics in which $e_1 \parallel e_2$ returns *void* is discussed in Section 5.3.2.)

E25. $$\frac{\begin{pmatrix} \mathfrak{E}_{b,r}(e_1) = \langle s, v\rangle \\ \mathfrak{E}_{b \otimes \{\ell \mapsto \rho\}, s}(e_2) = \langle s_\rho, v_\rho\rangle \text{ for each } \rho \in \mathrm{dom}(\omega_s(v)) \end{pmatrix}}{\mathfrak{E}_{b,r}(\textbf{forall } \ell \textbf{ in } e_1 \textbf{ do } e_2) = \left\langle s \cup \bigcup_{\rho \in \mathrm{dom}(\omega_s(v))} s_\rho,\ \mathit{void} \right\rangle}.$$

A forall-expression computes the combined store of multiple parallel executions of e_2 with respect to evaluation contexts which vary as the local variable ℓ ranges over the domain of the map given by e_1. The value returned is *void*.

E26. $$\frac{\mathsf{oneof}\{\mathit{left}, \mathit{right}\} = \mathit{left}}{\mathfrak{E}_{b,r}(e_1 \,[]\, e_2) = \mathfrak{E}_{b,r}(e_1)}, \qquad \frac{\mathsf{oneof}\{\mathit{left}, \mathit{right}\} = \mathit{right}}{\mathfrak{E}_{b,r}(e_1 \,[]\, e_2) = \mathfrak{E}_{b,r}(e_2)}.$$

Recall that $\mathsf{oneof}(X)$ is an external function computed by the outside world. Different calls to $\mathsf{oneof}(X)$ can give different results.

E27. $$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s, v\rangle \qquad \mathrm{dom}(\omega_s(v)) = \emptyset}{\mathfrak{E}_{b,r}(\textbf{choose } \ell \textbf{ in } e_1 \textbf{ do } e_2) = \langle r, \mathit{choiceX}\rangle}.$$

E28. $$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s, v\rangle \qquad \mathrm{dom}(\omega_s(v)) \neq \emptyset \qquad \mathsf{oneof}(\mathrm{dom}(\omega_s(v))) = \rho}{\mathfrak{E}_{b,r}(\textbf{choose } \ell \textbf{ in } e_1 \textbf{ do } e_2) = \mathfrak{E}_{b \otimes \{\ell \mapsto \rho\}, s}(e_2)}.$$

In choose-expressions, like in forall-expressions, ℓ is bound to a value in the domain of the map given by e_1.

E29. $$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s, v\rangle \qquad u_s \text{ is inconsistent}}{\mathfrak{E}_{b,r}(e_1 \,;\, e_2) = \langle r, \mathit{updateX}\rangle}.$$

E30. $$\frac{\begin{pmatrix} r' = (\theta_r, \omega_r, \emptyset) & \mathfrak{E}_{b,r'}(e_1) = \langle s_1, v_1\rangle & s_1 = (\theta_1, \omega_1, u_1) \\ u_1 \text{ is consistent} & \mathfrak{E}_{b,\widehat{s_1}}(e_2) = \langle s_2, v_2\rangle & s_2 = (\theta_2, \omega_2, u_2) \end{pmatrix}}{\mathfrak{E}_{b,r}(e_1 \,;\, e_2) = \langle \big(\theta_2,\ \omega_2 \otimes \omega_1,\ u_r \cup (u_1 \otimes u_2)\big),\ v_2\rangle}.$$

Recall how we compute the update set of $e_1 \,;\, e_2$. First we evaluate e_1 *in the present store*, then we evaluate e_2 *in the modified store obtained by firing all updates generated by* e_1 and we return the specially combined store *in which updates generated by* e_2 *override updates generated by* e_1. We compute e_1 in the evaluation context r', rather than r, in order to isolate updates generated by e_1 from those accumulated in u_r. We then compute e_2 in the store $\widehat{s_1}$ obtained from s_1 by firing u_1 (see the end of Section 4.1 in this connection) and return $u_r \cup (u_1 \otimes u_2)$.

We return the type map θ_2 since, by monotonicity, $\theta_r \subseteq \theta_1 = \theta_{\widehat{s_1}} \subseteq \theta_2$. Thus, $\mathrm{dom}(\theta_2)$ includes all existing objects as well as all objects created by e_1 and e_2. Also by monotonicity, $\omega_r \subseteq \omega_1$ and $\omega_{\widehat{s_1}} \subseteq \omega_2$. However, it can happen that $\omega_1 \neq \omega_{\widehat{s_1}}$. It remains to explain the content map $\omega_2 \otimes \omega_1$. One may have an impression that the content map should be just ω_2.

But this is not necessarily so. Recall that e_2 is evaluated in the auxiliary store $\widehat{s_1}$ obtained from s_1 by firing u_1. After the evaluation of e_2, the auxiliary store is thrown away. The objects altered by firing u_1 should be returned to their virgin status. This is achieved by overriding ω_2 with ω_1.

E31. $$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s, v\rangle \qquad u_s \text{ is inconsistent}}{\mathfrak{E}_{b,r}(e_1\ ;\ e_2) = \langle r, \mathit{updateX}\rangle},$$

$$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s, \mathit{false}\rangle \qquad u_s \text{ is consistent}}{\mathfrak{E}_{b,r}(\textbf{while } e_1 \textbf{ do } e_2) = \langle s, \mathit{void}\rangle}.$$

E32. $$\frac{\begin{pmatrix} r' = (\theta_r, \omega_r, \emptyset) \quad \mathfrak{E}_{b,r'}(e_1) = \langle s_1, \mathit{true}\rangle \qquad s_1 = (\theta_1, \omega_1, u_1) \\ u_1 \text{ is consistent} \qquad \mathfrak{E}_{b,\widehat{s_1}}(e_2) = \langle s_2, v_2\rangle \qquad s_2 = (\theta_2, \omega_2, u_2) \\ s = \big(\theta_2,\ \omega_2 \oslash \omega_1,\ u_r \cup (u_1 \oslash u_2)\big) \end{pmatrix}}{\mathfrak{E}_{b,r}(\textbf{while } e_1 \textbf{ do } e_2) = \mathfrak{E}_{b,s}(\textbf{while } e_1 \textbf{ do } e_2)}.$$

If the evaluation of e_1 creates no updates, then Rule E32 can be simplified to contain only the premises $\mathfrak{E}_{b,r}(e_1) = \langle s_1, \mathit{true}\rangle$ and $\mathfrak{E}_{b,s_1}(e_2) = (s, v)$. In general, however, the evaluation of e_1 does produce updates, and they need to be taken care of. Further, if the guard e_1 is deterministic, Rule E32 can be simplified to contain only the premises $\mathfrak{E}_{b,r}(e_1) = \langle s_1, \mathit{true}\rangle$ and $\mathfrak{E}_{b,r}(e_1; e_2) = (s, v)$. But if e_1 contains calls to the external function oneof then the simplified form is not appropriate: we have to ensure that e_1 is not evaluated twice.

Rule E32 is one reason that computation of $\mathfrak{E}_{b,r}(e)$ may diverge (see Remark 16 following rule E11).

Exception generation and handling

In the remaining rules, $v, v', v_1, v_2, \ldots$ represent any values, normalor exceptional.

E33. $\mathfrak{E}_{b,r}(\textbf{throw } \mathit{exc}) = \langle r, \mathit{exc}\rangle$.

E34. $$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s, v\rangle \qquad v \notin \overline{\mathit{exc}}}{\mathfrak{E}_{b,r}(\textbf{try } e_1 \textbf{ catch } \overline{\mathit{exc}} : e_2) = \langle s, v\rangle}.$$

E35. $$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s, v\rangle \qquad v \in \overline{\mathit{exc}}}{\mathfrak{E}_{b,r}(\textbf{try } e_1 \textbf{ catch } \overline{\mathit{exc}} : e_2) = \mathfrak{E}_{b,r}(e_2)}.$$

Here e_2 is evaluated in store r. The updates produced during the evaluation of e_1 are lost. (In fact, $s = r$ by the part 3 of Theorem 17.)

Exception propagation

E36. $$\frac{\mathfrak{E}_{b,r}(e_1, \ldots, e_n) = \langle s_1, v_1\rangle, \ldots, \langle s_n, v_n\rangle \qquad \{v_1, \ldots, v_n\} \cap \mathit{Exception} \neq \emptyset}{\mathfrak{E}_{b,r}(e_0) = \big\langle r,\ \mathsf{oneof}(\{v_1, \ldots, v_n\} \cap \mathit{Exception})\big\rangle}$$

where e_0 is any of the following:

$\textbf{new } c(e_1, \ldots, e_n)$	$\textbf{new } \tau \to t\ \{e_1 \mapsto e_2,\ \ldots,\ e_{n-1} \mapsto e_n\}$	$e_1 \textbf{ is } t$
$e_1.f$	$e_1[e_2]$	$e_1 \textbf{ as } t$
$e_1.m(e_2, \ldots, e_n)$	$e_1[e_2] := e_3$	$e_1 \parallel e_2$
$e_1.f := e_2$	$\textbf{remove } e_1[e_2]$	

E37. $$\frac{\mathfrak{E}_{b,r}(e_1) = \langle s, v\rangle \qquad v \in \mathit{Exception}}{\mathfrak{E}_{b,r}(e_0) = \langle r, v\rangle},$$

where e_0 is any of the following:

$$\begin{array}{lll} \textbf{let } \ell = e_1 \textbf{ do } e_2 & \textbf{forall } \ell \textbf{ in } e_1 \textbf{ do } e_2 & e_1 \,;\, e_2 \\ \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 & \textbf{choose } \ell \textbf{ in } e_1 \textbf{ do } e_2 & \textbf{while } e_1 \textbf{ do } e_2. \end{array}$$

This is different from E36: e_1 is evaluated first.

E38. $$\frac{\begin{pmatrix} r' = (\theta_r, \omega_r, \emptyset) & \mathfrak{E}_{b,r'}(e_1) = \langle s_1, v_1\rangle & v_1 \notin \mathit{Exception} \\ u_{s_1} \text{ is consistent} & \mathfrak{E}_{b,\widehat{s_1}}(e_2) = \langle s_2, v_2\rangle & v_2 \in \mathit{Exception} \end{pmatrix}}{\mathfrak{E}_{b,r}(e_1 \,;\, e_2) = \langle r, v_2\rangle}.$$

E39. $$\frac{\begin{pmatrix} r' = (\theta_r, \omega_r, \emptyset)\mathfrak{E}_{b,r'}(e_1) = \langle s_1, \mathit{true}\rangle \qquad u_{s_1} \text{ is consistent} \\ \mathfrak{E}_{b,\widehat{s_1}}(e_2) = \langle s_2, v_2\rangle \qquad v_2 \in \mathit{Exception} \end{pmatrix}}{\mathfrak{E}_{b,r}(\textbf{while } e_1 \textbf{ do } e_2) = \langle r,\ v_2\rangle}.$$

E40. $$\frac{\begin{pmatrix} \mathfrak{E}_{b,r}(e_1) = \langle s, v\rangle \qquad v \notin \mathit{Exception} \\ \mathfrak{E}_{b \circledleq \{\ell \mapsto \rho\}, s}(e_2) = \langle s_\rho, v_\rho\rangle \text{ for each } \rho \in \operatorname{dom}(\omega_s(v)) \\ \{v_\rho : \rho \in \operatorname{dom}(\omega_s(v))\} \cap \mathit{Exception} \neq \emptyset \end{pmatrix}}{\begin{array}{l}\mathfrak{E}_{b,r}(\textbf{forall } \ell \textbf{ in } e_1 \textbf{ do } e_2) = \\ \qquad \big\langle r,\ \mathsf{oneof}(\{v_\rho : \rho \in \operatorname{dom}(\omega_s(v))\} \cap \mathit{Exception})\big\rangle \end{array}}.$$

This concludes the definition of $\mathfrak{E}_{b,r}$.

Check that the premises of rules E1–E40 are mutually exclusive. However, the premises are not complete, i.e. they do not cover all possibilities. If e is (b, r)-typed but does not satisfy any premise, then $\mathfrak{E}_{b,r}(e)$ is said to *diverge*. If $\mathfrak{E}_{b,r}(e) = \langle s, v\rangle$ converges, then check that E1–E40 guarantee that $\langle s, v\rangle$ is indeed an effect (i.e. s is a store and $v \in \operatorname{dom}(\theta_s) \cup \mathit{Literal} \cup \mathit{Exception}$.

The following theorem describes an important property of $\mathfrak{E}_{b,r}$.

Theorem 17 (*Monotonicity of stores*). *Suppose* $\mathfrak{E}_{b,r}(e) = \langle s, v\rangle$.
1. $r \subseteq s$
2. $[b, r] = [b, s]$
3. $v \in \mathit{Exception} \implies r = s$

Proof. Statements 1 and 3 are easily verified by inspection of effect rules E1–E40. Statement 2 follows trivially from 1. □

Effect of the program. Programs π are also evaluated (or executed) for its effect. Let e be the body of π. By abuse of notation, we write $\emptyset$ for both the empty binding and the initial store (with no objects or updates). The effect of π is defined as follows. Recall that $\hat{s}$ is the store resulting from firing updates u_s at store s.

- $$\frac{\mathfrak{E}_{\emptyset,\emptyset}(e) = \langle s, v\rangle \qquad v \in \mathit{Nvalue} \qquad u_s \text{ is consistent}}{\mathit{Effect}(\pi) = \langle \widehat{s}, v\rangle}.$$
- $$\frac{\mathfrak{E}_{\emptyset,\emptyset}(e) = \langle s, v\rangle \qquad v \in \mathit{Nvalue} \qquad u_s \text{ is inconsistent}}{\mathit{Effect}(\pi) = \langle \emptyset, \mathit{updateX}\rangle}.$$
- $$\frac{\mathfrak{E}_{\emptyset,\emptyset}(e) = \langle s, v\rangle \qquad v \in \mathit{Exception}}{\mathit{Effect}(\pi) = \langle \emptyset, v\rangle}.$$

5. Analysis

The precise semantics of a programming language allows one to prove various properties of the language. In Section 5.1 we prove the type soundness of AsmL-S. In Section 5.2, we prove a refinement theorem. Some additional issues are discussed in Section 5.3.

5.1. Type soundness

This subsection is devoted to a proof of type soundness for AsmL-S.

Theorem 18 (*Type soundness*). *For every evaluation context (b, r) and every (b, r)-typed expression e, we have*

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

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

Proof. Proof is by induction on e. Assume that the statement holds for any b', r', e' where e' is a proper subexpression of e. By examination of effect rules E1–E40, we will show that the statement also holds for b, r, e.
E4, E8, E14, E16, E18, E23, E27, E29, E33, E36–E40: Each of these rules produces an exceptional value with type *Thrown*. Thus, if e satisfies the premise of any of these rules then

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

since *Thrown* lies below every other type.

Proviso. In all cases below except E34 and E35, $v, v', v_1, v_2, \ldots$ represent normal values (not exceptions).

E12, E17, E19, E20, E21, E25, E31: Each of these rules returns a particular literal: $E20$, $E21$ return *true*, *false*, respectively; the other rules return *void*. The corresponding typing rules assign *Bool* or *Void*, accordingly. Therefore, type soundness holds with equality.
E1–E3: Type soundness follows immediately from T1–T3. To wit:

$$\mathit{type}(\mathfrak{E}_{b,r}(\mathit{lit})) \overset{\text{E1}}{=} \mathit{type}(\langle r, \mathit{lit}\rangle) = \mathit{littype}(\mathit{lit}) \overset{\text{T1}}{=} \mathfrak{T}_{[b,r]}(\mathit{lit}),$$

$$type(\mathfrak{E}_{b,r}(\ell)) \overset{\text{E2}}{=} type(\langle r, b(\ell)\rangle) = [b,r](\ell) \overset{\text{T2}}{=} \mathfrak{T}_{[b,r]}(\ell).$$

If $e = op(\bar{e})$ where $\mathfrak{E}_{b,r}(\bar{e}) = \langle \bar{s}, \bar{v}\rangle$, $optype(op) = \bar{\tau} \to t$ and $op(\bar{v})$ is defined, then

$$type(\mathfrak{E}_{b,r}(e)) \overset{\text{E3}}{=} type(\langle \bigcup \bar{s}, op(\bar{v})\rangle) = littype(op(\bar{v})) = t \overset{\text{T3}}{=} \mathfrak{T}_{[b,r]}(e).$$

E5: Suppose e is of the form **let** $\ell = e_1$ **do** e_2 where $\mathfrak{E}_{b,r}(e_1) = \langle s, v\rangle$ and $\mathfrak{T}_{[b,r]}(e_1) = t$. By the induction hypothesis, $type(\langle s, v\rangle) \leqslant t$. By theorem 17 (statement 2), $[b, s] = [b, r]$. Thus, we have

$$\begin{aligned}[b \circledleq \{\ell \mapsto v\}, s] &= [b, s] \circledleq \{\ell \mapsto type(\langle s, v\rangle)\} \preccurlyeq [b, s] \circledleq \{\ell \mapsto t\} \\ &= [b, r] \circledleq \{\ell \mapsto t\}.\end{aligned}$$

By effect rule E5, $\mathfrak{E}_{b,r}(e) = \mathfrak{E}_{b \circledleq \{\ell \mapsto v\}, s}(e_2)$. By the induction hypothesis,

$$type(\mathfrak{E}_{b \circledleq \{\ell \mapsto v\}, s}(e_2)) \leqslant \mathfrak{T}_{[b \circledleq \{\ell \mapsto v\}, s]}(e_2).$$

By typing rule T4, $\mathfrak{T}_{[b,r]}(e) = \mathfrak{T}_{[b,r] \circledleq \{\ell \mapsto t\}}(e_2)$. Theorem 10 yields the inequality

$$\mathfrak{T}_{[b \circledleq \{\ell \mapsto v\}, s]}(e_2) \leqslant \mathfrak{T}_{[b,r] \circledleq \{\ell \mapsto t\}}(e_2).$$

We conclude that $type(\mathfrak{E}_{b,r}(e)) \leqslant \mathfrak{T}_{[b,r]}(e)$.
E6, *E7*: Suppose $e =$ **if** e_1 **then** e_2 **else** e_3 where $\mathfrak{E}_{b,r}(e_1) = \langle s, true\rangle$. Then

$$\begin{aligned}type(\mathfrak{E}_{b,r}(e)) \overset{\text{E6}}{=} type(\mathfrak{E}_{b,s}(e_2)) &\overset{\text{hyp.}}{\leqslant} \mathfrak{T}_{[b,s]}(e_2) \\ &= \mathfrak{T}_{[b,r]}(e_2) \leqslant \mathfrak{T}_{[b,r]}(e_2) \sqcup \mathfrak{T}_{[b,r]}(e_3) \overset{\text{T5}}{=} \mathfrak{T}_{[b,r]}(e),\end{aligned}$$

where the middle equality is by Theorem 17 (statement 2). The argument for E7 works the same way.
E9, *E13*: Type soundness follows immediately from T6, T10.

$$type(\mathfrak{E}_{b,r}(\textbf{new } c(\bar{e}))) \overset{\text{E9}}{=} type(\langle \ldots \cup (\{o \mapsto c\}, \ldots), o\rangle) = c \overset{\text{T6}}{=} \mathfrak{T}_{[b,r]}(\textbf{new } c(\bar{e}))$$

$$\begin{aligned}type(\mathfrak{E}_{b,r}(\textbf{new } \tau \to t \ \{\overline{e_1} \mapsto \overline{e_2}\})) &\overset{\text{E13}}{=} type(\langle \ldots \cup (\{o \mapsto \tau \to t\}, \ldots), o\rangle) \\ &= \tau \to t \overset{\text{T10}}{=} \mathfrak{T}_{[b,r]}(\textbf{new } \tau \to t \ \{\overline{e_1} \mapsto \overline{e_2}\}).\end{aligned}$$

E10: Suppose $e = e_1.f$ where $\mathfrak{E}_{b,r}(e_1) = \langle s, v\rangle$ and $v \neq null$. By typing rule T7, $\mathfrak{T}_{[b,r]}(e_1) = c$ for some class c with field f. By the induction hypothesis (and the fact that $v \neq null$), $type(\langle s, v\rangle) = c'$ for some class $c' \leqslant c$. Thus, f is a field of c' and $fldtype(f, c') = fldtype(f, c)$.

By definition of store, $f \in \mathrm{dom}(\omega_s(v))$ and $\omega_s(v)(f) \in Nvalue_{\theta_s}(fldtype(f, c'))$. Consequentially, $type(\langle s, \omega_s(v)(f)\rangle) \leqslant fldtype(f, c')$. Putting it all together, we get

$$type(\mathfrak{E}_{b,r}(e)) \overset{\text{E10}}{=} type(\langle s, \omega_s(v)(f)\rangle) \leqslant fldtype(f, c') = fldtype(f, c) \overset{\text{T7}}{=} \mathfrak{T}_{[b,r]}(e).$$

E11: Suppose $e = e_1.m(\overline{e_2})$, $\mathfrak{E}_{b,r}(e_1) = \langle s_1, v_1\rangle$, $\mathfrak{E}_{b,r}(\overline{e_2}) = \langle \overline{s_2}, \overline{v_2}\rangle$, $type(\langle s_1, v_1\rangle) = c$ and $dclr(m, c) = m(\bar{\ell}$ **as** $\bar{\tau})$ **as** t **do** e_3.

By E11, $\mathfrak{E}_{b,r}(e) = \mathfrak{E}_{\{me \mapsto v_1, \bar{\ell} \mapsto \overline{v_2}\}, s_1 \cup \bigcup \overline{s_2}}(e_3)$. By the induction hypothesis,

$$type(\mathfrak{E}_{\{me \mapsto v_1, \bar{\ell} \mapsto \overline{v_2}\}, s_1 \cup \bigcup \overline{s_2}}(e_3)) \leqslant \mathfrak{T}_{[\{me \mapsto v_1, \bar{\ell} \mapsto \overline{v_2}\}, s_1 \cup \bigcup \overline{s_2}]}(e_3).$$

It suffices to show that the latter type is dominated by $\mathfrak{T}_{[b,r]}(e)$. The induction hypothesis also implies $type(\langle s_1, v_1\rangle) \leqslant \mathfrak{T}_{[b,r]}(e_1)$ and $type(\langle \overline{s_2}, \overline{v_2}\rangle) \leqslant \mathfrak{T}_{[b,r]}(\overline{e_2})$. We thus obtain the following dominance relation among type contexts:

$$[\{me \mapsto v_1, \overline{\ell} \mapsto \overline{v_2}\}, s_1 \cup \bigcup \overline{s_2}] \preccurlyeq \{me \mapsto \mathfrak{T}_{[b,r]}(e_1), \overline{\ell} \mapsto \mathfrak{T}_{[b,r]}(\overline{e_2})\}.$$

Theorem 10 now yields:

$$\mathfrak{T}_{[\{me \mapsto v_1, \overline{\ell} \mapsto \overline{v_2}\}, s_1 \cup \bigcup \overline{s_2}]}(e_3) \leqslant \mathfrak{T}_{\{me \mapsto \mathfrak{T}_{[b,r]}(e_1), \overline{\ell} \mapsto \mathfrak{T}_{[b,r]}(\overline{e_2})\}}(e_3).$$

The well-formedness of m relative to c implies

$$\mathfrak{T}_{\{me \mapsto \mathfrak{T}_{[b,r]}(e_1), \overline{\ell} \mapsto \mathfrak{T}_{[b,r]}(\overline{e_2})\}}(e_3) \leqslant t.$$

It suffices to show that $t \leqslant \mathfrak{T}_{[b,r]}(e)$. Let $c' = \mathfrak{T}_{[b,r]}(e_1)$ and suppose $dclr(m, c') = m(\overline{\ell}'$ **as** $\overline{\tau}')$ **as** t' **do** e_3'. Then $c \leqslant c'$ by the induction hypothesis. The well-formedness of m relative to c and c' implies $t \leqslant t'$. By typing rule T8, $\mathfrak{T}_{[b,r]}(e) = t'$. Thus $t \leqslant \mathfrak{T}_{[b,r]}(e)$.
E15: Suppose $e = e_1[e_2]$, $\mathfrak{E}_{b,r}(e_i) = \langle s_i, v_i\rangle$ for i=1, 2, $type(\langle s_1, v_1\rangle)$=$t$, $\mathfrak{T}_{[b,r]}(e_1) = t'$, and $v_2 \in \mathrm{dom}(\omega_{s_1}(v_1))$. By E15, $type(\mathfrak{E}_{b,r}(e)) = \langle s_1 \cup s_2, v_2\rangle$.

By the induction hypothesis, $t \leqslant t'$. Letting $t = \tau_1 \to \tau_2$ and $t' = \tau_1' \to \tau_2'$, we have $\tau_2 \leqslant \tau_2'$ and $\mathfrak{T}_{[b,r]}(e) = \tau_2'$ by T11.

The definition of store implies $v_2 \in Nvalue_{\theta_{s_1}}(t_1)$, since $\omega_{s_1}(v_1) = v_2$. Consequentially, $type(\langle s_1, v_2\rangle) \leqslant \tau_2$. Clearly, $type(\langle s_1, v_2\rangle) = type(\langle s_1 \cup s_2, v_2\rangle)$. Putting it all together, we get

$$type(\mathfrak{E}_{b,r}(e)) = type(\langle s_1 \cup s_2, v_2\rangle) = type(\langle s_1, v_2\rangle) \leqslant \tau_2 \leqslant \tau_2' = \mathfrak{T}_{[b,r]}(e).$$

E22: If $e = e_1$ **as** t where $\mathfrak{E}_{b,r}(e_1) = \langle s, v\rangle$ and $v \leqslant t$, then

$$type(\mathfrak{E}_{b,r}(e)) \overset{\text{E22}}{=} type(\langle s, v\rangle) \leqslant t \overset{\text{T15}}{=} \mathfrak{T}_{[b,r]}(e).$$

E24: Suppose $e = e_1 \parallel e_2$ where $\mathfrak{E}_{b,r}(e_i) = \langle s_i, v_i\rangle$ for $i = 1, 2$. Then

$$type(\mathfrak{E}_{b,r}(e)) \overset{\text{E24}}{=} type(\langle s_1 \cup s_2, v_2\rangle) = type(\langle s_2, v_2\rangle) \overset{\text{hyp.}}{\leqslant} \mathfrak{T}_{[b,r]}(e_2) \overset{\text{T16}}{=} \mathfrak{T}_{[b,r]}(e_2),$$

where the middle equality follows from the observation that $v_2 \in \mathrm{dom}(\theta_{s_2})$.
E26: Suppose $e = e_1$ [] e_2 and $\mathsf{oneof}\{left, right\} = left$. Then

$$type(\mathfrak{E}_{b,r}(e)) \overset{\text{E26}}{=} type(\mathfrak{E}_{b,r}(e_1)) \overset{\text{hyp.}}{\leqslant} \mathfrak{T}_{[b,r]}(e_1) \leqslant \mathfrak{T}_{[b,r]}(e_1) \sqcup \mathfrak{T}_{[b,r]}(e_2) \overset{\text{T18}}{=} \mathfrak{T}_{[b,r]}(e).$$

The case where $\mathsf{oneof}\{left, right\} = right$ is handled the same way.
E28: Suppose $e =$ **choose** ℓ **in** e_1 **do** e_2, $\mathfrak{E}_{b,r}(e_1) = \langle s, v\rangle$, $\mathrm{dom}(\omega_s(v)) \neq \emptyset$, and $\mathsf{oneof}(\omega_s(v)) = \rho$. By theorem 17 (statement 2), $[b, s] = [b, r]$.

Let $type(\langle s, v\rangle) = \tau \to t$. Then $\rho \in Nvalue_{\theta_s}(\tau)$ by definition of store. Consequentially, $type(\langle s, \rho\rangle) \leqslant \tau$. Letting $\mathfrak{T}_{[b,r]}(e_1) = \tau' \to t'$, the induction hypothesis implies $(\tau \to t) \leqslant (\tau' \to t')$ and therefore $\tau \leqslant \tau'$. Thus, we have

$$\begin{aligned}[b \otimes \{\ell \mapsto \rho\}, s] &= [b, s] \otimes \{\ell \mapsto type(\langle s, \rho\rangle)\} \preccurlyeq [b, s] \otimes \{\ell \mapsto \tau'\} \\ &= [b, r] \otimes \{\ell \mapsto \tau'\}.\end{aligned}$$

We conclude

$$\begin{aligned} \mathit{type}(\mathfrak{E}_{b,r}(e)) &\overset{\text{E28}}{=} \mathit{type}(\mathfrak{E}_{b \olessthan \{\ell \mapsto \rho\},s}(e_2)) \\ &\overset{\text{hyp.}}{\leqslant} \mathfrak{T}_{[b \olessthan \{\ell \mapsto \rho\},s]}(e_2) \leqslant \mathfrak{T}_{[b,r] \olessthan \{\ell \mapsto \tau'\}}(e_2) \overset{\text{T19}}{=} \mathfrak{T}_{[b,r]}(e), \end{aligned}$$

where the latter inequality is by theorem 10.

*E*30: Suppose $e = e_1 \,;\, e_2$ and let $r' = (\theta_r, \omega_r, \emptyset)$, $\mathfrak{E}_{b,r'}(e_1) = \langle s_1, v_1 \rangle$, $s_1 = (\theta_1, \omega_1, u_1)$, u_1 be consistent, $\mathfrak{E}_{b,\widehat{s_1}}(e_2) = \langle s_2, v_2 \rangle$, and $s_2 = (\theta_2, \omega_2, u_2)$. Then

$$\begin{aligned} \mathit{type}(\mathfrak{E}_{b,r}(e)) &\overset{\text{E30}}{=} \mathit{type}(\langle(\theta_2, \omega_2 \olessthan \omega_1, u_r \cup (u_1 \olessthan u_2)), v_2\rangle) \\ &= \mathit{type}(\langle s_2, v_2 \rangle) \overset{\text{hyp.}}{\leqslant} \mathfrak{T}_{[b,\widehat{s_1}]}(e_2) = \mathfrak{T}_{[b,r]}(e) \overset{\text{T20}}{=} \mathfrak{T}_{[b,r]}(e_2). \end{aligned}$$

Both unmarked equalities above are trivial to establish. The first follows from the observation that $\theta_{s_2} = \theta_2$. The second follows from the observation that $\theta_r \subseteq \theta_{\widehat{s_1}}$ and therefore $[b, r] = [b, \widehat{s_1}]$.

E31, E32: Suppose $e =$ **while** e_1 **do** e_2. Recall that we consider a converging computation of $\mathfrak{E}_{b,r}(e)$. Eventually it returns value *void* by rule E31, assuming the recursion implied by rule E32 is well-founded. Type soundness clearly holds, since $\mathfrak{T}_{[b,r]}(e) = \mathit{Void}$ by typing rule T21.

E34, E35: Suppose $e =$ **try** e_1 **catch** $\overline{\mathit{exc}} : e_2$ and $\mathfrak{E}_{b,r}(e_1) = \langle s_1, v_1 \rangle$ where the value v_1 may be exceptional. If $v_1 \notin \overline{\mathit{exc}}$ then

$$\mathit{type}(\mathfrak{E}_{b,r}(e)) \overset{\text{E34}}{=} \mathit{type}(\langle s_1, v_1 \rangle) \overset{\text{hyp.}}{\leqslant} \mathfrak{T}_{[b,r]}(e_1) \leqslant \mathfrak{T}_{[b,r]}(e_1) \sqcup \mathfrak{T}_{[b,r]}(e_2) \overset{\text{T23}}{=} \mathfrak{T}_{[b,r]}(e).$$

On the other hand, if $v_1 \in \overline{\mathit{exc}}$ and $\mathfrak{E}_{b,r}(e_2) = \langle s_2, v_2 \rangle$ then

$$\mathit{type}(\mathfrak{E}_{b,r}(e)) \overset{\text{E35}}{=} \mathit{type}(\langle s_2, v_2 \rangle) \overset{\text{hyp.}}{\leqslant} \mathfrak{T}_{[b,r]}(e_2) \leqslant \mathfrak{T}_{[b,r]}(e_2) \sqcup \mathfrak{T}_{[b,r]}(e_2) \overset{\text{T23}}{=} \mathfrak{T}_{[b,r]}(e).$$

□

5.2. *Semantic refinement*

First we formalize the idea that one expression semantically refines the other with respect to a given type context *T*. Then we prove that an expression e_1 semantically refines an expression e_2 with respect to *T* if e_1 is obtained from e_2 by replacing a subexpression e'' of e_2 with some e' that semantically refines e'' with respect to the appropriate type context T'.

Let *s* be any store and let $V \subseteq \mathit{Value}$. The *s-span of V*, symbolically $\mathit{span}_s(V)$, is the least superset *O* of $V \cap \mathrm{dom}(\theta_s)$ satisfying the following conditions.

- If $o \in O$ then $\big(\mathrm{dom}(\omega_s(o)) \cup \mathrm{rng}(\omega_s(o))\big) \cap \mathrm{dom}(\theta_s) \subseteq O$.
- If $o \in O$, $((o, x), y) \in u_s$ and u_s is consistent, then $\{x, y\} \cap \mathrm{dom}(\theta_s) \subseteq O$.

Think of $\mathit{span}_s(V)$ as the set of objects reachable in *s* from *V*.

The triple $s_V = (\tilde{\theta}, \tilde{\omega}, \tilde{u})$ is defined as follows, where $\rightarrow\leftarrow$ is a fresh symbol connoting inconsistency.

$$\begin{aligned} \tilde{\theta} &= \theta_s \restriction \mathit{span}_s(V) \\ \tilde{\omega} &= \omega_s \restriction \mathit{span}_s(V) \end{aligned}$$

$$\tilde{u} = \begin{cases} \{((o, x), y) \in u_s \ : \ o \in span_s(V)\} & \text{if } u_s \text{ is consistent} \\ \rightarrow\leftarrow & \text{if } u_s \text{ is inconsistent} \end{cases}$$

s_V is called the *V-essential part of s*. Check that s_V is a store if u_s is consistent. Think of s_V as the result of garbage-collecting all unreachable objects and irrelevant updates in *s*, where *V* is a set of accessible values (such as those named by local variables). If u_s is inconsistent, then the updates in u_s can be ignored.

Lemma 19. *Suppose that r, s, s' are stores and U, V are subsets of value.*
(a) $s_V = s'_V \iff s_{V \cap ObjectId} = s'_{V \cap ObjectId}$.
(b) $s_{U \cup V} = s'_{U \cup V} \iff (s_U = s'_U \wedge s_V = s'_V)$.
(c) *If $U \subseteq V$ then $s_V = s'_V \Longrightarrow s_U = s'_U$.*
(d) *If $r \subseteq s$ and $r \subseteq s'$ then $s_{\mathrm{dom}(\theta_r)} = s'_{\mathrm{dom}(\theta_r)}$.*

Proof. (a)–(c) are obvious. (d) follows from the definition of the inclusion relation over states. $\square$

For any (b, r)-typed expression *e*, the set

$$\mathfrak{E}^{\star}_{b,r}(e) = \bigcup \begin{array}{l} \{\langle s, v\rangle \ : \text{ some convergent computation of } \mathfrak{E}_{b,r}(e) \text{ returns } \langle s, v\rangle\} \\ \{\infty \quad\ : \text{ some computation of } \mathfrak{E}_{b,r}(e) \text{ diverges}\} \end{array}$$

is *the set of possible effects of e*. It is presumed that the symbol ∞ is not used for anything else.

Now suppose *e* and e' are arbitrary expressions. We say that *e refines* e' with respect to type context *T* (written $e \lesssim_T e'$) if $\mathfrak{T}_T(e) = \mathfrak{T}_T(e') \in$ *Type* and each evaluation context (b, r) with $[b, r] = T$ satisfies I and II, below.
I. If $\infty \in \mathfrak{E}^{\star}_{b,r}(e)$ then $\infty \in \mathfrak{E}^{\star}_{b,r}(e')$.
II. For every effect $\langle s, v\rangle \in \mathfrak{E}^{\star}_{b,r}(e)$, there exists an effect $\langle s', v\rangle \in \mathfrak{E}^{\star}_{b,r}(e')$ such that $s_{\mathrm{rng}(b)\cup\{v\}} = s'_{\mathrm{rng}(b)\cup\{v\}}$.

We say that *e* and e' are *semantically equivalent* with respect to *T* (written $e \approx_T e'$) if $e \lesssim_T e'$ and $e' \lesssim_T e$.

As expected, refinement is transitive.

Proposition 20. *If $e \lesssim_T e'$ and $e' \lesssim_T e''$ then $e \lesssim_T e''$.* $\square$

Another important property of refinement is monotonicity with respect to subexpressions. Recall punctured expressions defined above in Section 3.6.1.

Theorem 21 (*Refinement*). *Suppose $e(\circ)$ is a punctured expression and e', e'' are expressions such that $e(e'/\circ)$ is T-well-typed and $e'' \lesssim_{T \circledast e(\circ)} e'$. Then*

$$e(e''/\circ) \lesssim_T e(e'/\circ).$$

Proof. Let $e(\circ)$, e' and e'' be as in the hypothesis. Proof of the statement $e(e''/\circ) \lesssim_T e(e'/\circ)$ is by induction on the *depth* of the variable $\circ$ in the punctured expression $e(\circ)$, defined recursively by the rules:

- $\mathit{depth}(\circ) = 0$,
- if $e \neq \circ$ then $\mathit{depth}(e(\circ)) = 1 + \mathit{depth}(e_0(\circ))$ where $e_0(\circ)$ is the unique maximal proper punctured subexpression of e.

The case of $\mathit{depth}(e(\circ)) = 0$, i.e. the case $e = \circ$, is trivial.

It suffices to prove the theorem in the case of $\mathit{depth}(e(\circ)) = 1$. Indeed, suppose that $\mathit{depth}(e) > 1$. Let e_0 be the maximal proper punctured subexpression of e and let e_1 be the result of replacing e_0 by $\circ$ in e so that $e(\circ) = e_1(e_0(\circ)/\circ)$ and $\mathit{depth}(e_1(\circ)) = 1$. Check that $T \circledast e(\circ) = (T \circledast e_1(\circ)) \circledast e_0(\circ)$. Invoking the induction hypothesis twice, we have

$$\begin{array}{llll} & e'' & \lesssim_{(T \circledast e_1(\circ)) \circledast e_0(\circ)} & e' \\ \Longrightarrow & e_0(e'') & \lesssim_{T \circledast e_1(\circ)} & e_0(e') \\ \Longrightarrow & e_1(e_0(e'')) & \lesssim_{T} & e_1(e_0(e')). \end{array}$$

So we restrict attention on the case of $\mathit{depth}(e(\circ)) = 1$. Since $e'' \lesssim_{T \circledast e(\circ)} e'$, we have $\mathfrak{T}_{T \circledast e(\circ)}(e') = \mathfrak{T}_{T \circledast e(\circ)}(e'') \in \mathit{Type}$. Thus, by Proposition 9, $\mathfrak{T}_T(e(e'/\circ)) = \mathfrak{T}_T(e(e''/\circ)) \in \mathit{Type}$. Let (b, r) be any evaluation context such that $[b, r] = T$. We must show that statements I and II in the definition of the refinement relation hold.

There are numerous (sub)cases to consider, for instance, three cases where e is of the forms **if** $\circ$ **then** e_1 **else** e_2, **if** e_0 **then** $\circ$ **else** e_2 and **if** e_0 **then** e_1 **else** $\circ$. In most cases, proof follows straightforwardly from the typing and effect rules. When $\circ$ falls under the scope of a fresh binding (i.e. is the body of a let-, forall- or choose-expression), proof is a matter of definition-chasing. We consider here a single case where e is of the form **let** $\ell = e_1$ **do** $\circ$.

Claim 22. I *and* II *hold if e is of the form* **let** $\ell = e_1$ **do** $\circ$.

For any expression e_2 such that **let** $\ell = e_1$ **do** e_2 is T-well-typed, effect rules E5 and E37 (those mentioning let-expressions) imply that $\mathfrak{E}_{b,r}(\textbf{let } \ell = e_1 \textbf{ do } e_2)$ diverges if, and only if, either $\mathfrak{E}_{b,r}(e_1)$ diverges or else $\mathfrak{E}_{b,r}(e_1) = \langle s, v\rangle$ and v is normal and $\mathfrak{E}_{b \otimes \{\ell \mapsto v\}, s}(e_2)$ diverges. Since $e'' \lesssim_{T \circledast e(\circ)} e'$ and $[\{\ell \mapsto v\}, s] = T \circledast e(\circ)$, we have

$$\infty \in \mathfrak{E}^{\star}_{\{\ell \mapsto v\}, s}(e'') \Longrightarrow \infty \in \mathfrak{E}^{\star}_{\{\ell \mapsto v\}, s}(e').$$

We conclude $\infty \in \mathfrak{E}^{\star}_{b,r}(e(e''/\circ)) \Longrightarrow \infty \in \mathfrak{E}^{\star}_{b,r}(e(e'/\circ))$, confirming statement I.

As for statement II, suppose $\langle s, v\rangle \in \mathfrak{E}^{\star}_{b,r}(e(e''/\circ))$. By E5 and E37, it follows that there is an effect $\langle s_1, v_1\rangle \in \mathfrak{E}^{\star}_{b,r}(e_1)$ such that

$$\big(v_1 \in \mathit{Exception} \wedge v_1 = v\big) \vee \big(v_1 \in \mathit{Nvalue} \wedge \langle s, v\rangle \in \mathfrak{E}^{\star}_{b \otimes \{\ell \mapsto v_1\}, s_1}(e'')\big).$$

If $v_1 \in \mathit{Exception}$, then E37 implies $\langle s, v\rangle \in \mathfrak{E}^{\star}_{b,r}(e(e_2/\circ))$ for any expression e_2 such that $e(e_2/\circ)$ is $[b, r]$-typed, so in particular for $e_2 = e'$. We will therefore assume that $v_1 \in \mathit{Nvalue}$.

Since $[\{\ell \mapsto v_1\}, s_1] = T \circledast e(\circ)$, the refinement relation $e'' \lesssim_{T \circledast e(\circ)} e'$ means that there is an effect $\langle s', v\rangle \in \mathfrak{E}^{\star}_{b \otimes \{\ell \mapsto v_1\}, s_1}(e')$ such that

$$s_{\mathrm{rng}(b \otimes \{\ell \mapsto v_1\}) \cup \{v\}} = s'_{\mathrm{rng}(b \otimes \{\ell \mapsto v_1\}) \cup \{v\}}.$$

Note that $\mathrm{rng}(b) \cap \mathit{ObjectId} \subseteq \mathrm{dom}(\theta_r)$. By monotonicity of stores (Theorem 17), $r \subseteq s$ and $r \subseteq s'$. Lemma 19 implies $s_{\mathrm{dom}(\theta_r)\cup\{v_1,v\}} = s'_{\mathrm{dom}(\theta_r)\cup\{v_1,v\}}$. Hence, by Lemma 19(c),

$$s_{\mathrm{rng}(b)\cup\{v\}} = s'_{\mathrm{rng}(b)\cup\{v\}}.$$

Effect rule E5 implies $\langle s', v\rangle \in \mathfrak{E}^{\star}_{b,r}(e(e'/\circ))$, so we are done. $\square_{(\mathit{Theorem}\ 21)}$

We now describe a few canonical refinements involving nondeterministic expressions of the forms $e_1 [] e_2$ and **choose** ℓ **in** e_1 **do** e_2.

Proposition 23. *Suppose $e_1 [] e_2$ is T-well-typed and $e_0 \lesssim_T$ true [] false. Then*

$$\big(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2\big) \lesssim_T e_1 [] e_2.$$

Furthermore, for $i = 1, 2$, if $\mathfrak{T}_T(e_i) = \mathfrak{T}_T(e_1 [] e_2)$ then $e_i \lesssim_T e_1 [] e_2$.

The proof is straightforward. $\square$

In order to state a similar proposition for choose-expressions, we must first define a specialized notion of refinement. We say that *e choice-domain refines e′* with respect to T (written $e \lesssim_T^{\mathrm{c.d}} e'$) if $\mathfrak{T}_T(e) = \mathfrak{T}_T(e') \in \mathit{MapType}$ and each evaluation context (b, r) with $[b, r] = T$ satisfies $\mathrm{I}^{\mathrm{c.d.}}$ and $\mathrm{II}^{\mathrm{c.d.}}$, below.

$\mathrm{I}^{\mathrm{c.d.}}$. If $\infty \in \mathfrak{E}^{\star}_{b,r}(e)$ then $\infty \in \mathfrak{E}^{\star}_{b,r}(e')$.

$\mathrm{II}^{\mathrm{c.d.}}$. For every effect $\langle s, v\rangle \in \mathfrak{E}^{\star}_{b,r}(e)$, there exists an effect $\langle s', v'\rangle \in \mathfrak{E}^{\star}_{b,r}(e')$ such that $\mathit{type}(\langle s, v\rangle) = \mathit{type}(\langle s', v'\rangle) = t$ and one of the following holds:

i. $t \in \{\mathit{Null}, \mathit{Thrown}\}$ and $v = v'$,

ii. $t \in \mathit{MapType}$ and $\omega_s(v) = \omega_{s'}(v') = \emptyset$,

iii. $t \in \mathit{MapType}$, $\emptyset \neq \omega_s(v) \subseteq \omega_{s'}(v')$ and $s_{\mathrm{rng}(b)} = s'_{\mathrm{rng}(b)}$.

We are now able to state the following:

Proposition 24. *Suppose* **choose** ℓ **in** e_1 **do** e_2 *is T-well-typed and $e_1' \lesssim_T^{\mathrm{c.d}} e_1$. Then*

$$\big(\textbf{choose } \ell \textbf{ in } e_1' \textbf{ do } e_2\big) \lesssim_T \big(\textbf{choose } \ell \textbf{ in } e_1 \textbf{ do } e_2\big).$$

The statement follows straightforwardly from the relevant definitions, including typing rule T19 and effect rules E8, E27, E28, E37 (i.e. those mentioning choice-expressions).

5.3. Discussion

There are numerous additional issues related to the analysis of AsmL-S. Here we touch on some of them without developing them in depth.

5.3.1. Simultaneous let

Currently the let bindings are evaluated sequentially. Consider for example the expression

$$\textbf{let } \ell_1 = e_1 \textbf{ do } \big(\textbf{let } \ell_2 = e_2 \textbf{ do } e_3\big)$$

where ℓ_1 does not occur in e_2. It would not hurt to evaluate e_1, e_2 in parallel (unless e_1 produces an exception) but our rules dictate to evaluate e_1 first and e_2 second. In the spirit of ASMs with its emphasis on parallelism, we generalize the current let construct to a new simultaneous let construct:

$$\textbf{let } \ell_1 = e_1, \ \ldots, \ \ell_n = e_n \textbf{ do } e_{n+1},$$

where $\ell_1, \ldots, \ell_n$ are distinct local variables. To evaluate the above expression, evaluate all binding bodies $e_1, \ldots, e_n$ at the present evaluation context. Let us presume that all n computations converge. If all n computations return normal values, then proceed to evaluation e_{n+1} in the new evaluation context. Otherwise, return one of the exceptional values nondeterministically. The only reason we did not introduce this simultaneous let construct in AsmL-S above was to simplify notation.

5.3.2. Parallel composition

$\mathfrak{E}_{b,r}(e_1 \parallel e_2)$, when convergent, returns the value of $\mathfrak{E}_{b,r}(e_2)$, as defined in rule E24. Why do we want that the expression $e_1 \parallel e_2$ returns anything? Because the return value may be useful in programming. Unfortunately, our decision to return the value of e_2 breaks the symmetry between e_1 and e_2, that is, $e_1 \parallel e_2 \not\approx e_2 \parallel e_1$. Furthermore, this contrasts with the symmetry of forall-expressions, which always return *void*. The operator $\parallel$ can be made symmetric, and consistent with the semantics of forall-expressions, by modifying rule E24 so that $\mathfrak{E}_{b,r}(e_1 \parallel e_2)$ always returns *void*. (Note that this can be simulated in the present semantics by writing $(e_1 \parallel e_2) \parallel void$, though this does not change fundamental asymmetry of the $\parallel$.)

Making $\parallel$ symmetric exacts some price. Suppose that we would like to simulate the asymmetric version of $\parallel$ that returns the value of the second expression. This would be possible to achieve, but awkward, in the present syntax. For example, we could write

$$\big(\textbf{new } \mathit{Bool} \rightarrow \ T \ \{\mathit{false} \mapsto e_1, \ \mathit{true} \mapsto e_2\}\big)[\mathit{true}]$$

for the appropriate T. The asymmetric $e_1 \parallel e_2$ could be expressed more naturally by means of simultaneous let as $\textbf{let } \ell_1 = e_1, \ \ell_2 = e_2 \textbf{ do } \ell_2$.

5.3.3. Coverage

Our definition of $\mathfrak{E}^\star_{b,r}(e)$ in Section 5.2 tacitly assumes that the definition of $\mathfrak{E}_{b,r}$ is complete and covers all the cases, so that every finite computation of $\mathfrak{E}_{b,r}(e)$ returns a value, possibly exceptional. The assumption is not immediately obvious but can be proven.

5.3.4. Covariance vs. contravariance in argument types

In the type system of AsmL-S, maps are covariant in both argument and result types (see Section 3.3). This is consistent with the type system of the full AsmL. On the other hand, in functional languages, functions are conventionally contravariant in argument types [23]. The rationale for contravariance in argument types is that a function of type $\tau \rightarrow t$ could be safely placed in any context expecting a map of type $\tau' \rightarrow t$ where $\tau' \leqslant \tau$. One can argue that maps should be contravariant in argument types. Either variant has benefits and drawbacks. Ultimately the most important consideration is how maps are supposed to be used. In AsmL they have been used more as look-up tables than as functions in functional programming, and so covariance is appropriate. For those familiar with abstract state machines, let us mention that maps are often used to represent dynamic functions of ASMs; in that role they are essentially look-up tables.

We make a couple of technical points related to the controversy. One benefit of contravariance in argument types is preclusion of the *maptypeX* exception that arises in computing $\mathfrak{E}_{b,r}(e_1[e_2] := e_3)$ when $type(\mathfrak{E}_{b,r}(e_1)) = T \rightarrow T'$ and $type(\mathfrak{E}_{b,r}(e_2)) \nleq T$, as in example 24. Obviously, we want to have as few built-in exceptions as necessary. But contravariance in argument types has a price. The problem with contravariance lies in computing the forall- and choose-expressions of AsmL-S

$$\mathfrak{E}_{b,r}(\textbf{forall } \ell \textbf{ in } e_1 \textbf{ do } e_2) \quad \mathfrak{E}_{b,r}(\textbf{choose } \ell \textbf{ in } e_1 \textbf{ do } e_2),$$

where the binding of ℓ ranges over the domain of the map given by e_1 (which, as a set, should be viewed as naturally covariant).

Suppose for a moment that map types are contravariant in argument types but the type system of AsmL-S is otherwise unchanged. We encounter problematic programs such as the following:

```
**class** $A$, **class** $B$ **extends** $A$ $\{i$ **as** $Int\}$ :
    **let** $f = \big($**new** $A \rightarrow Int$ $\{$**new** $A() \rightarrow 0\}\big)$ **do**
        **let** $g = \big($**if** $true$ **then** $f$ **else new** $B \rightarrow Int$ $\{\}\big)$ **do**
            **choose** $\ell$ **in** $g$ **do** $\ell.i$
```

We check that this program is well-typed: the static type of g is the least upper bound of $A \rightarrow Int$ and $B \rightarrow Int$, that is, $B \rightarrow Int$; the static type of ℓ is therefore B; the body $\ell.i$ of the choose-expression is thus well-typed. However, in evaluating this program we run into the problem of computing $\ell.i$ when ℓ has runtime type A and i is not a field of A. This calls for a new exception (for undefined object fields) precluded in the current semantics of AsmL-S. An alternative fix is to require explicit type casting in forall- and choose-expressions, as e.g. **forall** ℓ **as** t **in** e_1 **do** e_2.

5.3.5. Side-effect-free expressions

Since AsmL is primarily a specification language, it may be reasonable to require that expressions e_0 in

let-expressions	**let** $\ell = e_0$ **do** e_1,
conditional expressions	**if** e_0 **then** e_1 **else** e_2,
class-field expressions	$e_0.f$,
type tests	e_0 **is** t.

be side-effect free. The list is not exhaustive list. Our purpose here is just to illustrate the idea.

The requirement that e be side-effect free means that no evaluation of e_0 can produce updates, and it can be enforced by simple syntactical constraints.

It is less reasonable to impose such restrictions of the full AsmL because it is also used as a programming language. The guard of a conditional expression could be instrumented for example to collect certain data. In this semantical study, we have not been opposed in principle to restrictions of that kind. It turns out, however, that the fact that we did not impose such restrictions did not cause any problems. If one takes a route of imposing such restrictions on AsmL-S, one should consider enriching the language with additional constructs to compensate for the lost expressivity.

5.3.6. Well-typed expressions with subexpressions of static type Thrown

If an expression *e* has static type *Thrown* in some type context *T*, then type soundness (Theorem 18) implies that the value of $\mathfrak{E}_{b,r}(e)$ is exceptional for all evaluation contexts (b, r) such that $[b, r] = T$. Most of such expressions *e* are nonsense expressions, with the obvious exception when *e* is a throw-expression **throw** *exc*. For example, the map-creation expression

$$\textbf{new}\ \mathit{Int} \to \mathit{Bool}\ \{(\textbf{throw}\ \mathit{fooX}) \mapsto \mathit{true}\}$$

is well-typed in the present semantics. Its static type is *Int* → *Bool*, even though any evaluation will return *fooX*. This does not contradict type soundness, since $\mathit{type}(\mathit{fooX}) = \mathit{Thrown} < (\mathit{Int} \to \mathit{Bool})$. However, the fact that this expression will always result in an exception can be recognized—and prevented—at compile time. In this particular example, we can change typing rule T10 from its present formulation

$$\frac{\mathfrak{T}_T(\overline{e_1}) \leqslant t_1 \qquad \mathfrak{T}_T(\overline{e_2}) \leqslant t_2}{\mathfrak{T}_T(\textbf{new}\ t_1 \to t_2\ \{\overline{e_1} \mapsto \overline{e_2}\}) = t_1 \to t_2}$$

to the following:

$$\frac{\mathit{Thrown} < \mathfrak{T}_T(\overline{e_1}) \leqslant t_1 \qquad \mathit{Thrown} < \mathfrak{T}_T(\overline{e_2}) \leqslant t_2}{\mathfrak{T}_T(\textbf{new}\ t_1 \to t_2\ \{\overline{e_1} \mapsto \overline{e_2}\}) = t_1 \to t_2}.$$

The effect of this change is that the above degenerate map-creation expression is no longer well-typed. A similar observation applies to several other type rules with explicit premises. Note that such a strengthening of type rules does not jeopardize type soundness, or any other theorem, as the only consequence is that fewer expressions are well-typed.

Notice that the qualifier “static” in the heading of this discussion item is there for good reason. It is undecidable whether a given subexpression produces only exceptions. These improvements in type checking catch only the most egregious offenders.

Acknowledgements

Without the support from the FSE group this work would not be possible. In particular, thanks to Wolfgang Grieskamp and Nikolai Tillmann for developing the runtime mechanism of AsmL. Thanks to Robert Stärk for raising an issue clarified in Section 2.10. Thanks to Klaus Havelund for going through the paper with a magnifying glass and thus helping us very much to debug and clarify the paper.

References

[1] The AsmL webpage, http://research.microsoft.com/foundations/AsmL/.

[2] D. Bjoerner, C.B. Jones (Eds.), Formal Specification and Software Development, Prentice-Hall International, Englewood Cliff, NJ, 1982.

[3] A. Blass, Y. Gurevich, Ordinary interactive small-step algorithms, I, ACM Trans. Comput. Logic, to appear. See Microsoft Research Tech. Report MSR-TR-2004-16.

[4] A. Blass, Y. Gurevich, Ordinary Interactive Small-Step Algorithms, II, Microsoft Research, Tech. Report MSR-TR-2004-88.

[5] A. Blass, Y. Gurevich, S. Shelah, Choiceless polynomial time, Ann. Pure Appl. Logic 100 (1999) 141–187.

[6] E. Boerger, J. Schmid, Composition and submachine concepts for sequential ASMs, in: P. Clote, H. Schwichtenberg (Eds.), Computer Science Logic (Proc. CSL 2000), Lecture Notes in Computer Science, Vol. 1862, Springer, Berlin, 2000, pp. 41–60.

[8] M.J.C. Gordon, T.F. Melham, Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, Cambridge University Press, Cambridge, 1993.

[9] Y. Gurevich, Evolving algebra 1993: Lipari guide, in: E. Boerger (Ed.), Specification and Validation Methods, Oxford University Press, Oxford, 1995, pp. 9–36.

[10] Y. Gurevich, May 1997 draft of the ASM guide, Tech. Report CSE-TR-336-97, EECS Department, University of Michigan, 1997 (available at the author’s website).

[11] Y. Gurevich, For every sequential algorithm there is an equivalent sequential abstract state machine, ACM Trans. Comput. Logic 1 (1) (2000) 77–111.

[12] Y. Gurevich, W. Schulte, M. Veanes, Toward Industrial Strength Abstract State Machines, Tech. Report MSR-TR-2001-98, Microsoft Research, October 2001.

[13] Y. Gurevich, N. Tillmann, Partial updates: exploration, Springer J. Universal Comput. Sci. 7 (11) (2001) 918–952.

[14] A. Hejlsberg, S. Wiltamutch, P. Golde, The C# Programming Language, Addison-Wesley, Reading, MA, 2003.

[15] P. Hudak, S.P. Jones, P. Wadler, B. Boutel, J., Fairbairn, J. Fasel, M.M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, J. Peterson, Report on the Programming Language Haskell, Version 1.2, SIGPLAN Notices 27 (5) May 1992.

[16] J.K. Huggins, ASM Michigan web page, http://www.eecs.umich.edu/gasm.

[17] A. Igarashi, B.C. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM Trans. Program. Languages Systems (TOPLAS) 23 (3) (2001) 396–450.

[18] B. Joy, G. Steele, J. Gosling, G. Bracha, Java (TM) Language Specification, 2nd ed., Addison-Wesley, Reading, MA, 2000.

[19] G. Kahn, Natural semantics, in: Proc. Symp. on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, Vol. 247, 1987, pp. 22–39.

[20] R. Milner, M. Tofte, R. Harper, D. MacQueen, The Definition of Standard ML (Revised), MIT Press, Cambridge, MA, 1997.

[21] Objective Caml Website http://www.ocaml.org/.

[22] M. Odersky, P. Wadler, Pizza into Java: translating theory into practice, Proc. 24th ACM Symp. on Principles of Programming Languages, Paris, France, January 1997.

[23] B.C. Pierce, Types and Programming Languages, MIT Press, Cambridge, MA, 2002.

[24] G.D. Plotkin, Structural approach to operational semantics, Tech. Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, September 1981.

[25] PVS Website http://pvs.csl.sri.com/.

[26] J.M. Spivey, The Z Notation: A Reference Manual, 2nd ed., Prentice-Hall, New York, 1992.