Introducing AsmL:
A Tutorial for the Abstract State Machine Language

December 19, 2001

Revised May 23, 2002

Abstract

This paper is an introduction to specifying robust software components using AsmL, the Abstract State Machine Language.

Foundations of Software Engineering -- Microsoft Research
(c) Microsoft Corporation. All rights reserved.

1 Introduction..1

1.1 Audience 1

1.2 Organization of the document 1

1.3 Notation 1

2 Models..2

2.1 Abstract state 2

2.2 Distinct operational steps 3

2.3 The evolution of state variables 3

3 Programs..7

3.1 Hello, world 7

3.2 Reading a file 8

4 Steps 11

4.1 Stopping conditions 11

4.1.1 Stopping for fixed points 11

4.1.2 Stopping for conditions 12

4.2 Sequences of steps 12

4.3 Iteration over collections 13

4.4 Guidelines for using steps 14

5 Updates 15

5.1 The update statement 15

5.2 When updates occur 16

5.3 Consistency of updates 17

5.4 Total and partial updates 18

6 Methods 20

6.1 Functions and update procedures 21

6.2 Local names for values 22

6.3 Method overloading 23

7 Values..24

7.1 What are values? 24

7.2 Structured values 24

7.3 Built-in values 25

8 Constraints 27

8.1 Assertions 27

8.2 Type constraints 27

9 Constants...28

10 Variables 29

11 Classes...32

11.1 Fields defined in a class 32

11.2 New instances of a class 32

11.3 Updates to instance variables 33

11.4 Instances as "references" 33

11.5 Derived classes 34

12 Structured Values 36

12.1 Structure cases 37

13 Sets 39

13.1 Sets as enumerated values 39

13.2 Sets given by value range 39

13.3 Sets described algorithmically 40

13.4 Set Operations 41

13.4.1 Union 41

13.4.2 Intersect 41

13.4.3 Size 41

14 Sequences 43

15 Parallel evaluation 45

15.1 Sequential iteration 46

16 Maps...48

16.1 Map Comprehensions 49

16.2 Maps with multiple arguments 50

16.3 Map Operations **50**
16.3.1 Domain 50
16.3.2 Range 50
16.3.3 Merge 51
16.4 Partial updates of maps **52**
17 Nondeterminism 53
17.1 Nondeterministic choice **53**
17.2 External nondeterminism **54**
17.3 The nondeterminism of "new" **55**
18 Enumerations...56
19 Conditionals and Loops 58
19.1 The If Statement **58**
19.2 Logical Operators **60**
19.2.1 The and operator 60
19.2.2 The not operator 61
19.2.3 The not operator 61
19.2.4 The implies operator 61
20 Patterns..62
21 Predicate Logic...63
21.1 forall..holds **63**
21.2 The exists Quantifier **64**
22 Table of examples...1

1 Introduction

This paper is a basic introduction to AsmL (Abstract State Machine Language) and contains enough information so that, once you've read it, you should be able to write simple models. In addition to this manual, you may also want to read the paper called "Vending Machine Case Study," which uses many of these basic techniques. For a complete description of the AsmL language, see the reference manual, "AsmL: The Abstract State Machine Language."

1.1 Audience

We assume no prior knowledge of the modeling approach used in this paper or any understanding of the mathematical underpinnings that make the language such a precise modeling tool. Even though this paper describes a language based on rigorous mathematical semantics, it is intended for a broad audience. The mathematics is there, but you don't need to concern yourself with it.

AsmL was designed to be accessible to anyone with a basic understanding of programming. A familiarity with one other programming language, such as Visual Basic, is enough to understand this paper.

Readers interested in the details of AsmL, and especially the technical features that provide its mathematical foundations, should refer to http://research.micosoft.com/foundations/AsmL. This Web site includes documentation and a publicly downloadable implementation of the AsmL tool.

1.2 Organization of the document

Section 2 gives the overview. Subsequent sections provide more detail.

1.3 Notation

The document consists of text and lines of AsmL source. AsmL source is formatted using a special style:

```
// This is AsmL source.
```

It is important to use this style when writing AsmL because this is how the AsmL tool knows what to extract from a document. Within the text, we format AsmL literals using a `fixed-width font`.

We use special text colors for terminology that is defined in the document. Such terms are printed in blue when they are defined and dark red when referenced. For example, we define a term as a phrase with special meaning. A term may appear anywhere in the document.

Notes to the reader use a faded color to distinguish them from the rest of the text:

> This is a note.

2 Models

A digital system is one that operates in distinct steps.

Digital systems are *discrete*; that is, they can be decomposed into "bits." The discontinuous, granular nature of digital systems distinguishes them from analog systems. For example, the current passing through an analog circuit may change gradually over time, while a digital switch is either open or closed. A chemical reaction progresses continuously, while the number of documents pending for a laser printer goes down by a discrete quantity as it finishes printing a job.

This paper introduces a language for modeling the structure and behavior of digital systems called AsmL, or the Abstract State Machine Language. Models written in AsmL help us to understand how existing systems work or to specify how new systems will work. To accomplish this, AsmL exploits the fact that digital systems have discrete *operations* as well as discrete *structure*. (The granular, discontinuous nature of digital systems is seen in their structure *and* their operation.)

The modeling approach behind AsmL is very powerful. AsmL can be used to faithfully capture the abstract structure and step-wise behavior of *any* discrete system, including very complex ones such as integrated circuits, software components, and devices that combine both hardware and software.

Two "digital" concepts are at the core of the approach: *abstract state* and *distinct operational steps*.

2.1 Abstract state

An AsmL model is said to be *abstract* because it encodes only those aspects of the system's structure that affect the behavior being modeled. The goal is to use the minimum amount of detail that accurately reproduces (or predicts) the behavior of a system.

For example, instruction caches make a computer chip run faster by allowing memory hardware to run in parallel with the processor hardware. If we were only trying to describe the way the chip interprets its hardware instructions, we might disregard instruction caches entirely, since they don't affect the behavior of the chosen level of detail. However, if we were interested in what messages are sent over the system's internal communication buses, then instruction caches would need to be taken into consideration.

Abstraction helps us reduce complex problems into manageable units and prevents us from getting lost in a sea of detail. AsmL provides a variety of features that allow you to describe (i.e., "model") the relevant state of a system in a very economical, high-level way.

2.2 Distinct operational steps

An AsmL model is *precise* because the behavior of the model matches, *step for step*, the behavior of the system being modeled. This means that the model accurately reflects the result of any step the actual system takes at the model's level of detail. The step-for-step correspondence of model to system behavior is a very important aspect of AsmL. (When you create your own models, you should attempt to factor operations into meaningful steps.)

The technical name for the way AsmL models describe operational steps is "abstract state machines." An abstract state machine is a particular kind of mathematical machine. It is an "operational" machine like the Turing machines you may have already heard of, but unlike a Turing machine, abstract state machines may be defined a very high level of abstraction. (Turing machines operate only on a linear sequence of characters.) You don't need to know much about abstract state machines in order to use AsmL, since AsmL handles many of the details for you.

An easy way to understand abstract state machines is to see them as defining a succession of states that may follow an initial state. Each abstract state machine (or, simply, machine) represents a particular view of the distinct operational steps that occur in the real-world system being modeled. For example, you might open a file, perform individual read or write operations and then close the file. Each of these operations is a step of an abstract machine.

The behavior of a machine (its run) can always be depicted as a sequence of states linked by state transitions. (Moving from state A to state B is a state transition.) A bicycle may initially be red. Then, because someone painted it, it is green. These are two different states and moving from the state of being red to the state of being green is a state transition. The act of painting the bicycle (an "operation") causes the transition.

The behavior of a computer chip is another example of states and state transitions. This behavior can be modeled as steps of an abstract state machine where each step corresponds to the execution of one hardware instruction. A single hardware instruction may result in many changes to the chip's hardware registers.

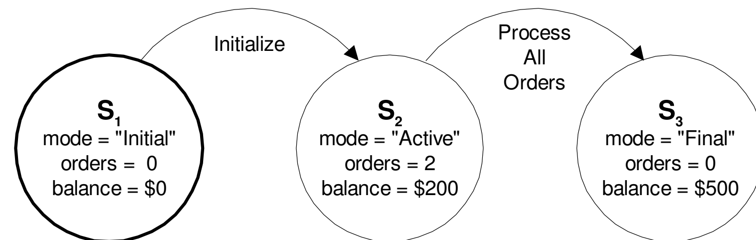
Each state is a particular "configuration" of the machine. The state may be simple, as with our bicycle, or it may be very large, with a complex structure. But no matter how complex the state might be, each step of the machine's operation can be seen as a well-defined transition from one particular state to another.

2.3 The evolution of state variables

We can view any machine's state as a dictionary of `(`*name*`,` *value*`)` pairs, called *state variables.* Our bicycle could have a `(color, red)` variable, where "`color`" is the name of the variable and "`red`" is the value. The names of

variables are given by the machine's symbolic vocabulary. Values are fixed elements, like numbers and strings of characters.

The run of a machine is a series of states and state transitions that results from applying operations to each state in succession. For example, the following diagram shows the run of a machine that models how orders might be processed:

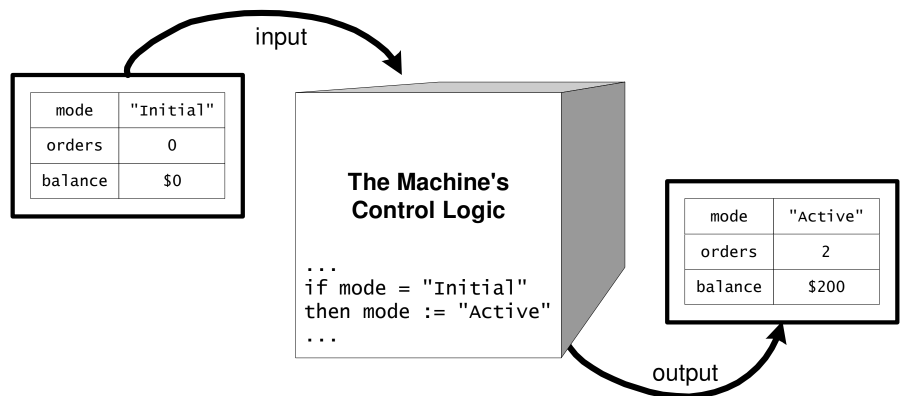


The diagram shows a run consisting of two transitions ("Initialize" and "Process All Orders") and three states, labeled S_1 through S_3. The vocabulary used to represent the machine's state consists of the names "`mode`", "`orders`" and "`balance`." We see that these names are associated in every state with particular values. For example, in state S_1 the value of `mode` is "`Initial`".

Each transition operation (such as "Initialize") can be seen as the result of invoking the machine's control logic on the current state. Each operation calculates the subsequent state as output. In other words, the machine's control logic behaves like a fixed set of transition rules that say how state may evolve. We use the terms control logic and operations interchangeably. (The unqualified term "program" will generally be used to mean the entire definition of a machine, including the names of its state variables. The control logic is that portion of the program made up of the rules governing the state transitions.)

For now, we can think of the control logic as text that precisely specifies, for any given state, what the values of the machine's variables will be in the following step. The typical form of the operational text is "`if` *condition* `then` *update*" where the "condition" is a true/false expression evaluated using the state of the current step and "update" identifies the name-to-value entries in the state-dictionary of the next step. In our order processing example, one rule is: if mode = "Initial" then mode = "Active." If the condition (if mode = "Initial") is true then an update occurs and the new value for mode will be "Active."

Thus, one can think of the machine's control logic as a black box that takes as input a state-dictionary S_1 and gives as output a new dictionary S_2. The two dictionaries S_1 and S_2 have the same set of keys (i.e., variable names specified in the machine's symbolic vocabulary), but the values associated with each variable name may differ between S_1 and S_2.



The run of the machine can be seen as what happens when the control logic is applied to each state in turn. S_1 is given to the black box yielding S_2; processing S_2 results in S_3, and so on. When no more changes to state are possible, the run is complete.

We use the symbol "`:=`" (read as "gets") to indicate the value that a name will have in the resulting state. For example, the diagram shows `mode := "Active"` as the branch of an "`if`" statement. Read as "`mode` gets `Active`", the update can be seen as partially configuring the resulting state (S_2). No change is made to the current state (S_1). The symbol "`:=`" is known as the update operator. Statements in the form "a `:=` b" are update operations.

One consequence of handing states to the control logic unit and observing only the new state is that the changes are not visible as intermediate results—the new state can be seen only during the *following* step. This is in contrast to other programming languages you may be familiar with, such as C or Visual Basic. In these languages, changes take place immediately, in sequential order. In AsmL, all changes happen simultaneously, when you move from one step to another. Then, all the updates happen at once. This is called an atomic transaction.

Treating updates as atomic transactions is an essential feature of our modeling approach. Transactions give meaning to the concept of a step and are a tremendous simplification for subsequent analysis of the model. For example, the order in which update operations are executed within a step has no meaning, since updates are not visible until the next step and therefore can have no influence on the calculations performed in the current step.

Points for review

An AsmL **model** (or **program**) is defined using a fixed **vocabulary** of symbols of our choosing. It has two components: the names of its **state variables** and a fixed set of **operations** of an **abstract state machine**.

Values are simple, immutable elements like numbers and strings.

State can be seen as a particular association of variable names to values, in the style of a dictionary: {($name_1$, val_1), ($name_2$, val_2), … }.

A **run** of the machine is a series of states connected by state transitions.

Each **state transition**, or **step**, occurs when the machine's **control logic** is applied to an input state and produces an output state. "Control logic" is a synonym for the machine's set of **operations**.

The program consists of **statements**. A typical statement is the **conditional update** "`if` *condition* `then` *update.*" Each **update** is in the form "a := b" and indicates that variable name a will be associated with the value b in the output state.

The program never alters the input state. Instead, each update statement adds to a set of **pending updates**. Pending updates are not visible in any program context, but when all program statements have been invoked, the pending updates are merged with a copy of the input state and returned as the output state.

An **inconsistent update error** occurs if the update set contains conflicting information. For example, the program cannot update a variable to two different values in a single step.

3 Programs

In the previous section we said that a machine is defined by a program that has state variables and control logic. In this section we will show syntax and examples of such programs.

The syntax of AsmL is similar to that of a simple procedural programming language. This is intentional, since AsmL models often live inside of specification documents as explanatory pseudo-code that illustrates the concepts given in the text. Such documents have a broad readership and should be easy to understand.

However, programs constructed with this simple syntax have different properties from procedural programs. AsmL programs always describe the possible operations of a machine with a particular set of state variables.

We show two examples to get you started.

3.1 Hello, world

Example 1 Hello, world

```
Main()
  step WriteLine("Hello, world!")
```

The first example defines a one-step machine with no state variables.

AsmL uses indentations to denote block structure, and blocks can be placed inside other blocks. In fact, blocks can be nested, one within the other, to any depth. Statement blocks also affect the scope of variables. This is discussed later in the paper. AsmL's use of indentation to indicate block structure is different from other languages you might be familiar with. Most languages use curly braces ("{" and "}") or special keywords like "`Begin`" and "`End`" to indicate blocks.

Whitespace includes blanks and new-line characters. AsmL does not recognize tabs, so don't use the tab character for indentations. Whitespace separates one part of a statement from another and enables the AsmL compiler to identify where one element in a statement, such as `Integer`, ends and the next statement begins. Apart from its use as a separator, the compiler ignores whitespace. As in other languages, certain symbols like "+" and "-" do not require whitespace separation in order to be recognized.

By convention, an operation named `Main()` gives the top-level operational definition of the model. In this sense, `Main()` is like `main()` in the C programming language. Also, AsmL is case-sensitive. Typing "Main()" instead of `Main()` will cause an error.

The effect of the `WriteLine()` operation is external; that is, it returns no value and changes none of the model's state variables. The following string will be printed: "`Hello, world!`"

The keyword `step` marks the transition from one state to another. In this example there are no state variables, so the initial state and ending state are the same.

3.2 Reading a file

Example 2 Reading a file

```
Main()
  initially F as File? = null
  initially FContents  = ""
  initially Mode       = "Initial"
  step until fixpoint
    if Mode = "Initial" then
      F    := new Open("MyFile.txt")
      Mode := "Reading"

    if Mode = "Reading" and Length(FContents) = 0 then
       FContents := Read(F, 1)

    if Mode = "Reading" and Length(FContents) = 1 then
      FContents := FContents + Read(F, 1)

    if Mode = "Reading" and Length(FContents) > 1 then
      WriteLine(FContents)
      Mode := "Finished"
```

Example 2 introduces a machine whose run consists of five steps.

The syntax "`step until fixpoint`" precedes a block of statements that will be repeatedly run until no changes result. A fixed point occurs when two consecutive states are equal. In other words, the machine runs until a step occurs that does not produce any difference in state from the previous state.

The keyword `initially` identifies state variables and declares their initial values as of the beginning of the run. We will discuss variables in greater length, but, for now, remember that updating variables is the only way to change the state of a machine.

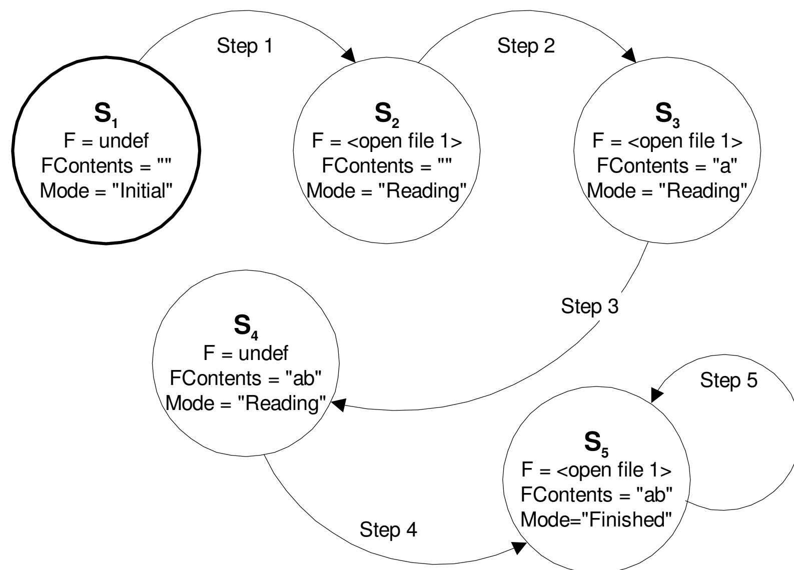
The run of this machine proceeds like this:

- In the first step, the values of all variables are their initial values. For example, the value of `Mode` is `"Initial"`. When we look at the "`if`" statements, only one applies. It causes us to open a file named "myfile.txt" and update the `Mode` variable from `"Initial"` to `"Reading"`.
- In the second step, the `Mode` variable has the value `"Reading"` and the length of the string given by `FContents` is still zero. This matches the conditions given by the second "if" statement. Consequently, we

read a character from the file and update the value of the state variable `FContents` to a string containing just this one character.

- In the third step, we append an additional character to the value of `FContents`. The character added comes from the file being read. As a consequence, `FContents` will be a two-character string.
- In the fourth step, we print the string to the standard output device and see that it is two characters long. (If we printed the contents of `FContents` as part of the third step we would only have seen one character. This is because the update doesn't happen until we transition to the fourth step.) We set Mode to be the value `"Finished"`.
- In the fifth step, none of the conditions of the "if" statements will be true (that is, none of them apply to the state because `Mode` equals `"Finished"`.) The resulting state includes no changes. This signals the machine to stop running because a fixed point has occurred.

We can examine this example to see why four state transitions are needed to bring the system to a fixed point.



This example shows how a program defines the state variables and control logic of a machine and how a run of that program results in a sequence of states and state transitions corresponding to each operational step.

Some readers may be wondering how this approach is different from finite state machines, or other kinds of "automata" that they have encountered previously. Although this topic is beyond the scope of this document, a short answer is that

1) our machines may have state variables with very large (even infinite) ranges as well as complex structure (such as a graph of interconnected nodes) and 2) the operations corresponding to the state transitions may interact with the external environment in a flexible way. Abstract state machines are more general than other kinds of machines and automata. For example, you can model a finite state machine using an abstract state machine that is limited to a single string-valued state variable (a "mode" label) and whose transitions depend only on the current mode.

In an abstract state machine it is possible for state variables to have complex nested data structures as their values, or come from infinite sets like real numbers. Further, an abstract state machine may interact with the external environment and therefore have more than one possible run. For example, we could have made the control logic depend on which character was read from an external file.

In the next section we give some shortcuts that make it more convenient to write the rules that control state transitions.

4 Sequential Processes

In this section, we show several ways that steps can be used for sequential processes.

The general syntax for a sequential process is:

step [*label*] [*stopping-condition*] *statement-block*

where a statement block consists of indented statements that follow, a label is an optional string, number or identifier followed by a colon (":") and a stopping condition is any these forms:

until fixpoint

until *expr*

while *expr*

It is possible for a step to be introduced independently or as part of sequence of steps in the form **step** … **step** …

4.1 Stopping conditions

4.1.1 Stopping for fixed points

The most general form of a step describes iteration until a fixed point is reached. The machine runs until there are no more changes of state. (Example 2 from the previous section showed how this approach can model the states that occur when reading a file.)

The "`until fixpoint`" form is the most general way of specifying state transitions. In fact, all "step" blocks in AsmL can be transformed back into the "`step until fixpoint`" form. The other kinds of steps can be seen as abbreviations for this underlying form.

Example 3 Fixed-point iteration

```
enum EnumMode
  Initial
  Started
  Finished

Main()
  initially Mode = Initial
  initially Count = 0
  step until fixpoint
    if Mode = Initial then
      Mode := Started
      Count := 1
```

```
    if Mode = Started and Count < 10 then
      Count := Count + 1

    if Mode = Started and Count >= 10 then
      Mode := Finished
```

This machine has an initial step, a series of iterated steps and a final step.

Note that in AsmL, types are often implied by context. The type of the variable "`Mode`" is `EnumMode`. The variable "`Count`" is of type `Integer`.

4.1.2 Stopping for conditions

It is also possible to give a specific condition to control the iterated steps of a machine by giving an explicit stopping condition and either the keyword `while` or `until`.

Example 4 Iterating while a condition is true

```
Main()
  initially x = 1
  step while x < 10
    WriteLine(x)
    x := x + 1
```

Running this example produces nine steps. It will print the numbers 1, 2, 3, 4, 5, 6, 7, 8 and 9 in ascending order as its output.

Example 5 Iterating until a condition is met

```
Main()
  initially x = 1
  step until x > 9
    WriteLine(x)
    x := x + 1
```

Either `while` or `until` may be used to give an explicit stopping condition for iterated sequential steps of the machine. Example 5 produces the same output as Example 4

4.2 Sequences of steps

The syntax step … step … indicates a sequence of steps that will be performed in order. It is perfectly valid for one or more of the steps in the sequence to be an iterated step with a stopping condition.

We can restate Example 2 using a sequence of steps:

Example 6 Reading a file using a sequence of steps

```
Main()
  initially F as File?         = null
  initially FContents          = ""
  step 1:   F := Open("MyFile.txt")
  step 2:   FContents := Read(F, 1)
  step 3:   FContents := FContents + Read(F, 1)
  step 4:   WriteLine(FContents)
```

Example 6 and Example 2 have identical meaning. You may notice, however, that Example 6 is much shorter. The reason is that the program can be naturally expressed as a recipe that has four sequential steps. Labels after the "`step`" keyword are optional but sometimes helpful as documentation.

Sequences of steps introduce an implicit "mode" variable for each of the steps, just like the one shown in Example 2 . This is often convenient, but you should be wary of introducing unnecessary steps. This can occur if two operations are really not order-dependent but are still given as two sequential steps. It is very easy to fall into this trap, particularly since most people are used to the statement-by-statement sequential order used by other programming languages.

4.3 Iteration over collections

Another common idiom for iteration is to do one step per element in some finite collection such as a set or sequence. This is possible in AsmL using the following form:

`step foreach` $ident_1$ **`in`** $expr_1$**`,`** $ident_2$ **`in`** $expr_2$ ... *statement-block*

Here is an example:

Example 7 Iteration over items in a collection

```
const myList = [1, 2, 3]

Main()
  step foreach i in myList
    WriteLine(i)
```

This example prints the numbers `1`, `2`, and `3` in sequence. Each invocation of `WriteLine()` occurs in sequence. The order matches that of `myList`.

Sequential, step-based iteration is available for sets as well as sequences, but in the case of sets, the order is not specified. It should not be assumed that sets have any order.

4.4 Guidelines for using steps

Which kind of step you choose for your model will depend on what you are trying to describe. Here are some guidelines.

- If your operations occur in a fixed order, consider using a sequence of steps. This is usually the shortest way and the easiest to read.
- If your operations use an iteration variable like "*i*" or "*nextObj*" **and** each operation must be done in order, you can use an iterated step with a stopping condition that you provide. However, in many cases, order is not important and separate steps are not required for each value of the counter. If the updates don't contradict each other, they can occur as part of just one step of the machine, even though there are many individual changes. See "`forall`" below for more information about this case.
- If you have a collection of values that you want to process (like a set or a sequence) with the same operation, and it is important that the operation be done in sequence, one after another, then a step that iterates on values would be appropriate. However, in many cases, order is not important and separate steps are not required. See "forall" below for more information about this case.
- If you want to repeat an operation until no more changes occur (for instance, swapping out-of-order entries to sort a list), then you might consider using the fixed-point stopping condition.

If you have a choice between using many steps or just a few, you should always opt for economy. There are many benefits to reducing your algorithm to the fewest steps possible.

"Normalizing" your algorithm into the fewest steps that make sense has the same kinds of benefits as organizing your databases into tables with normalized key structures. The structure and relationships will become clearer if you separate the essential aspects of your algorithm from aspects that may be derived.

5 Updates

As we said earlier, a program defines state variables and operations. This section gives a bit more detail about how variables are updated. If you haven't read section 2, which describes how machines run, you should do that now. The most important concept is that state is a dictionary of (*name*, *value*) pairs where each name identifies an occurrence of a state variable. Operations may propose new values for state variables, but the effect of these changes is only visible in subsequent steps.

5.1 The update statement

Updating variables is the only way a system's state can change. This is the update symbol:

`:=`

It is read as "gets." The line `x := 2` would be read as "x gets 2." We interpret an update in the form `x := 2` as adding an entry in the form `(x, 2)` to the update set of the current step.

The following code illustrates how to update a variable and when that update happens.

Example 8 Update statement

```
var x = 0                                      // initialize x
var y = 1                                      // initialize y

Main()
  step
    WriteLine ("In the first step, x = " + x)  // x is 0
    WriteLine ("In the first step, y = " + y)  // y is 1
    x := 2
  step
    // update occurs here
    WriteLine ("In the second step, x = " + x) // x is 2
    WriteLine ("In the second step, y = " + y) // y is 1
```

In this example, we use the syntax "`var`" to introduce global variables `x` and `y`. Either "`initially`" or "`var`" may be used, but as a matter of style one uses "`initially`" to declare local variables that immediately precede a sequence of step and "`var`" for global variables and fields.

In this example, we first set the variables of `x` and `y` to their initial values of `0` and `1`, respectively. Note that we use the equals sign ("=") whenever we initialize and the update operator ("`:=`") whenever we update variables. In AsmL, we view update and initialization as distinct operations.

We then display these values on the screen using the `WriteLine()` library function. (The AsmL documentation set describes the available library functions.) Within this step, we also indicate that the value of `x` should be

updated to the value 2 as of the next step. (This is the meaning of "x := 2"). Note that within a step the order of statements does not matter.

In the next step of the run, we display the new value of x.

Updates don't actually occur until the step following the one in which they are written. This is the reason we need a second step to display the new value of x. Without that second step, the update would never happen.

To demonstrate this, examine the following code:

Example 9 Delayed effect of updates

```
var x = 0
var y = 1

Main()
  step
    WriteLine ("In the first step, X = " + x)
    WriteLine ("In the first step, Y = " + y)
  step
    x := 2
    WriteLine ("In the second step, X = " + x) // x is still 0
```

The initial value of x would be 0, just as in the previous example. However, the value of x printed in the second step is still 0. This is because the update x-gets-2 has been placed in the second step! Its effect is visible only in subsequent steps. Therefore, at the time of the last WriteLine(), the update hasn't occurred yet.

In fact, because the update occurred in the last step of the run, we will never observe the change of x from 0 to 2. (To do so, we would just add another step.)

5.2 When updates occur

All updates given within a single step occur simultaneously at the end of the step. Conceptually, the updates are applied "in between" the steps.

This is similar to the way database transactions work but different from the way most programming languages operate. In programming languages, operations happen sequentially.

To illustrate the difference, we will contrast how a simple swap would be written in Visual Basic and in AsmL. First, here is the Visual Basic code:

```
temp = x                    'copy old value of x to temp
x = y                       'copy old value of y to x
y = temp                    'retrieve original x and copy it to y
```

Because variables are changed sequentially, we need a temporary variable to store the value of x. Without that variable (that is, if we simply said x = y; y = x) we would end up with x being equal to y and y being equal to y.

Translating this to AsmL, we simply write:

```
x := y
y := x
```

We don't need a `temp` variable because the values are swapped simultaneously at the step boundary. For example, if x = 1 and y = 2, then the update statements "x-gets-y" and "y-gets-x" produce the update set {(x, 2), (y, 1)}.

To prove this to yourself, run the following code:

Example 10 Swapping values

```
var x = 1
var y = 2

Main()
  step
    x := y
    y := x
    // updates not yet taken effect
    WriteLine ("In the first step, x = " + x)
    WriteLine ("In the first step, y = " + y)

  step
    // updates have taken effect
    WriteLine ("In the second step, x = " + x)
    WriteLine ("In the second step, y = " + y)
```

All the updates occur simultaneously, in the step after the one in which they appear.

Also, because all updates happen simultaneously, the order in which the updates are written within a step doesn't matter. If we had written

```
y := x
x := y
```

the results would have been the same.

It is recommended that you explicitly introduce locally named values to avoid confusion in readers that are not familiar with the "parallel" update feature of AsmL.

5.3 Consistency of updates

The order of updates within a step does not matter, but all of the updates in the step must be consistent. In other words, we always require that none of the updates given within a step may contradict each other.

For example, you can't update variable "x" to two different values in the same step. That is, `x := 1` and `x := 2` in the same step produce a clash in the update set, since we don't know which of the two should take effect.

If updates do contradict, then they are called "inconsistent updates" and an error occurs.

5.4 Total and partial updates

An update of a variable can either be total or partial. Multiple partial updates to a single variable are allowed, as long as they are consistent.

The updates we have seen so far are all total updates, a simple replacement of a variable's value with a new value.

Partial updates apply to variables that have structure. Examples of structured variables include sets (unordered collections of distinct values) and maps (associative arrays that allow for table lookup).

To illustrate what this means, we will use a class roster as an example. The roster is empty at first but, after registration, contains the names of four students. First, here is an example of a total (or simple) update.

Example 11 Total update of a set-valued variable

```
var Students as Set of String = {}

Main()
  step
    WriteLine ("The initial roster is = " + Students)
    Students := {"Bill", "Carla", "Duncan", "Amy"}
  step
    WriteLine ("The final roster is " + Students)
```

The variable `Students` was, initially, an empty set. It was then updated to contain the names of the four students. The update became visible in the second step as the final roster.

An alternative way to do this is with a series of partial updates:

Example 12 Partial update of a set-valued variable

```
var Students as Set of String = {}

Main()
  step
    WriteLine ("The initial roster is = " + Students)
    add "Bill" to Students
    add "Carla" to Students
    add "Duncan" to Students
    add "Amy" to Students
  step
    WriteLine ("The final roster is " + Students)
```

The result is the same as with the total update. This is because all updates happen simultaneously rather than sequentially.

We can add and remove elements from sets using the following syntax:

remove *expr* **from s***et-variable*

add *expr* **to** *set-variable*

For example, updating the set `Students` with the statement `remove "Bill" from Students` removes `"Bill"` from the set (as of the next step). The update is partial in the sense that other students may be added to the set Students in the same step without contradicting the fact that `"Bill"` has been removed.

Suppose that, during the course, Bill and Duncan drop out of the class, and Carla joins. We can handle this scenario in just one step using the following partial updates:

```
remove "Bill" from Students
add "Carla" to Students
remove "Duncan" from Students
```

Note, as described earlier, the order of updates within a step does not matter because individual updates are not allowed to contradict each other.

6 Methods

Methods are named operations that may be invoked in various contexts. The definition of a method may include parameters that will be associated with particular values each time the method is invoked. Methods may also specify a return value that is available after the operation has been performed.

The general form for a method definition is:

name(*parameter1* **as** *Type***,** *...parametern* **as** *Type* **)** **as** *Type*

where:

- *name* is the name of the method
- (*parameter1* **as** *Type...parametern* **as** *Type*) are the values sent to the method, if there are any
- **as** *Type* is the type of the return value, if there is a return value.

Example 13 Method

```
IsWord(i as Integer) as Boolean
  return (0 <= i and i <= 0xFFFF)

Main()
  step
    if IsWord(123) then
      WriteLine("123 is a word of memory.")
```

Note

An expression in the form `IsWord(`i`)` is an application of name `IsWord` to argument i, which is an `Integer`. This method denotes the value that results by first calculating the value given by i and then substituting this value into the corresponding definition.

This convention is called "strict" evaluation and is familiar to programmers; an alternate approach, which we do not use, is called "lazy" evaluation of arguments. A third approach, which we also do not use, is called "substitution semantics."

The only time that arguments are not "strictly" evaluated is in the case of conditionals like "`if`" and "`match`." For example, in the expression "`if` $p(x)$ `then` a `else` b" either a or b will be evaluated but not both.

Logical operators like "and" and "or" come in two versions. The expression "a and b" may result in the evaluation of either a or b or both a and b. No order is implied. By contrast, given the expression "a `and then` b", b will only be evaluated if a is `true`. In other words, the expression is the same as "`if` a `then` b `else false`". The commutative version of logical disjunction is "a or b". The sequential version is "a `or else` b".

In this example, `IsWord` evaluates an argument i and returns the value `true` if i is greater than or equal to `0` and less than or equal to the hexadecimal number `0xFFFF`. If i doesn't fit these criteria, `IsWord` returns the value `false`.

6.1 Functions and update procedures

We generally distinguish between functions whose applications have no effect on state variables and update procedures that configure the values of state variables for the next sequential step of the machine.

Observing the returned value is the only way to see the effect of applying a function. In contrast, the effect of applying an update procedure may be seen both by observing the value it may (optionally) return and by examining the values of variables in the subsequent step of the abstract machine. (Note that methods such as `WriteLine()` and `read()` are considered to be update procedures, even though from the model's point of view they are don't change any state. This is because they may change external state.)

Functions must return a value; update procedures may optionally return a value. The term "operation" or "method invocation" means that either a function or an update procedure has been applied.

You may use the keywords "function" and "procedure" in front of method declarations to help keep this distinction clear. (The current AsmL tool treats these as comments; a later release will do some consistency checking.)

Here is an example of an update procedure and a function:

Example 14 Update procedures and functions

```
var Count = 0

Increment()                         // update procedure
  Count := Count + 1

CurrentCount() as Integer           // function
  return Count

Main()
  step
    Increment()
  step
    WriteLine("x is " + CurrentCount())
```

The example prints "`x is 1`."

It is recommended as a matter of style that update procedures should not return values. In other words, the effect of an update procedure should be seen

only in changes to state variables, while the effect of a function is seen only by means of a return value.

6.2 Local names for values

Within a method, statements in the form *identifier* = *expr* or `let` *identifier* = *expr* introduce *identifier* as a local name for the value given on the right hand side of the equals sign ("=") by *expr*. Within a given invocation, every reference of the name given by *identifier* denotes the same value, even if expression given on the right hand side is nondeterministic (i.e., uses external functions). In other words, the *identifier* does not substitute for *expr*; instead, it substitutes for the value given by one invocation of *expr*.

Every time the statement block that contains a local name is invoked, a new binding is established that associates the name to a value.

Example 15 Local names

```
Main()
  mySet = {1, 2, 3}
  WriteLine("mySet has " + Size(mySet) + " elements.")
```

Example 15 uses the locally bound name `mySet` to mean the set value `{1, 2, 3}`.

Local names may be introduced within any block of statements. For example, local names could appear after the "`then`" keyword of an `if` … `then` … `else` statement.

A local name can be referenced anywhere in its statement block. However, in a sequence of steps, local names introduced within a `step` block are also available within any following `step` block.

Example 16 Local names in sequences of steps

```
Main()
  step
    mySet1 = {1, 2, 3}
    WriteLine("mySet1 has " + Size(mySet1) + " elements.")
  step
    mySet2 = mySet1 union {4}
    WriteLine("mySet2 has " + Size(mySet2) + " elements.")
```

Example 16 would print "`myset has 3 elements.`" followed by "`mySet2 has 4 elements.`" It is acceptable to use the local name `mySet1` inside the following step block.

6.3 Method overloading

Method overloading allows you to use the same name in different methods and, in each instance, to have the compiler choose the correct instance. Obviously, there must be a clear way for the compiler to decide which method should be called for any particular instance. The key to this is the parameter list. A series of methods with the same name but differentiated by their parameter lists are overloaded methods. Here is an example:

Example 17 Method overloading

```
S = {1, 8, 2, 12, 13, 6}

Max(i as Integer, j as Integer) as Integer
  return (if i > j then i else j)

Max(s as Set of Integer) as Integer
  return (any m | m in S where not (exists n in S where n > m))

Main()
  step WriteLine("The largest number in the set is " + Max(S))
  step WriteLine("The larger of the two integers is " +
                Max(2, 3))
```

In this example, there are two functions called `Max()`. One finds the larger of two integers while the other finds the largest element of a set. The compiler knows which function to use by examining the parameter list.

AsmL's rules for determining if an "overloaded" method name is OK are similar to other languages you might be familiar with.

7 Values

7.1 What are values?

The term value as used in this document has the same meaning as in mathematical sets. A value is an immutable element that supports two operations: equality testing and set membership. In other words, if x and y are values, then we can ask whether x is equal to y. We use the syntax "x = y" for this purpose.

A set is an unordered collection of distinct elements. We use lists of elements within curly braces ("{" and "}") to denote sets. For example, `{1, 2, 3}` is the set containing the values `1`, `2` and `3`. If S is a set, then we can ask whether any value x is an element of S. We use the syntax "x `in` S" to test if x is an element of set S. This test will return `true` or `false`. Sets are themselves values.

We consider certain values to be built-in, for example, the distinguished values `true`, `false` and `null`. Integers are also provided by the types `Integer` and `Long`. (These represent two different finite ranges of mathematical integers.) Other values arise as the result of operations.

7.2 Structured values

Some values are composed of other values. In AsmL, as in mathematics, an ordered pair (x, y) is itself a value, even though its components x and y are also values.

Some structured values are built into AsmL; others may be defined by you.

For example, we use the syntax "`structure` *identifier* ..." to indicate that *identifier* is a type label for tuples of a particular form.

Example 18 User-defined structure type

```
structure Location
  base as Integer
  offset as Integer
```

`Location(1, 4)` is a value of the user-defined structure type `Location`.

Equality testing for structured values is based on the equality of components. Structured values are distinguishable from one other only if their type labels are distinct or if their constituent values are (pair-wise) distinct. For example, `Location(1, 4)` is indistinguishable from `Location(1, 2 + 2)`. Both forms denote the same value. `Location(20, 404)` and `Location(30, 12)` are distinct values.

As with ordered pairs in mathematics, the number of possible values of type `Location` is the product of the number of values possible for each of its components. In this example, there are as many locations as there are distinct pairs of type `Integer`.

User-defined structures are described in more detail later on.

7.3 Built-in values

AsmL provides built-in values using the standard data types that you are already familiar with. It also includes some others that may be new to you. Values of the built-in types are either primitive values (such `true` and `false`) or structured values composed of other values (like sequences).

The following table shows all the data types in AsmL:

Data Type	Meaning
`Boolean`	The type containing the values `true` and `false`
`Null`	The type containing the single value `null`
`Byte`	8-bit unsigned integer type
`Short`	16-bit signed integer type
`Integer`	32-bit signed integer type
`Long`	64-bit signed integer type
`Float`	Single-precision 32-bit floating-point format type as specified in IEEE 754
`Double`	Double-precision 64-bit floating-point format type as specified in IEEE 754
`Char`	Unicode character type
`String`	Unicode string type
`Seq of A`	The type of all sequences formed with elements of type `A`
`Set of A`	The type of all (finite) sets containing elements of type `A`
`Map of A to B`	The type of all tables whose entries map elements of type `A` to type `B`
`(A1, A2, ...)`	The type of all tuples consisting of elements of type `A1`, `A2`, ... Tuple values are written using parentheses. For example, `(1, 2)` is of a value of the built-in type `(Integer, Integer)`.
t?	The type that includes all of the values of type *t* plus the special value `null`. For example, a variable declared as `Boolean`? could contain either of the Boolean values `true` or `false` or the value `null`.

Most of these types are familiar from other programming languages and have the same meaning in AsmL. Maps, sets and sequences are discussed later in this paper.

8 Constraints

8.1 Assertions

`require` and `ensure` statements document constraints placed upon the model. Violating a constraint at runtime is called an assertion failure and indicates a modeling error.

Example 19 Assertions

```
AllTickets = {1..1024}

IsTicketID(i as Integer) as Boolean
  require i > 0
  return (i in AllTickets)

ChooseAnyTicket() as Integer
  ensure IsTicketID(result)
  return (any t | t in AllTickets)
```

Example 19 shows a typical use of assertions. The constant `AllTickets` is a set that contains all of integers from `1` to `1024`. The function `IsTicketID()` returns `true` or `false`, depending on whether its argument is found in the set `AllTickets`. However, the function `IsTicketID()` only handles the case where the argument is greater than zero. The statement "`require i > 0`" documents this assumption.

The function `ChooseAnyTicket()` randomly selects an element of the set `AllTickets`. The statement "`ensure IsTicketID(result)`" documents the fact that the value returned by the function must satisfy the conditions given by `IsTicketID()`. We are "ensuring" that a particular property holds for the result of the function.

The keyword `result` has special meaning inside of `ensure` statement. In this context, `result` denotes the value of the `return` statement.

8.2 Type constraints

We have already shown that types of constants, variables and method parameters may be specified using the syntax *name* **as** *type*.

The meaning of types in AsmL is as constraints on the values of constants, variables and method parameters. The meaning of types in AsmL is very similar to `require` and `ensure` described in section 8.1 above.

Types may be explicitly checked using the "`is`" operator, as in "*value* `is` *type*". Such an expression will be either true or false, depending on whether the value given is an element of the type.

Types are not themselves values. For example, you can't pass a type as an argument.

9 Constants

Constants are fixed, named values. The names of constants are established by declaration in the program.

In AsmL, constants can either be global or local. *Declaring* a constant (or a variables, which we discuss later) means that we give it a name and, optionally, specify the type of data it can store. AsmL employs type inference so explicitly declaring the type may not be necessary. *Initializing* a constant (or variable) means that we give it an initial value. In the case of constants, this initial value is the only value it will ever have, unless you rewrite the program.

Global constants are declared at the beginning of the program. Local constants are declared within a statement block. The general form for declaring and initializing a constant is:

name [**as** Type] = *value*

Here are a few examples

```
x as Integer = 10 // global constant with explicit typing
x = 10            // global constant with implicit typing
```

The general form for declaring and initializing a local constant is the same as for global constants. The following example shows both types of constants:

Example 20 Constants

```
MaxInteger = 100

Main()
  MinInteger = 0
```

Example 20 includes one global constant `MaxInteger` and one local constant `MinInteger`. The scope of a local constant is that of the block in which it is defined.

10 Variables

As we stated earlier, updating variables is the only way to change the state of a machine. Variables are names given to memory locations. The keyword **var** always goes in front of the variable name when the variable is first declared. The variable may also be assigned a type (this may be optional, since AsmL can often deduce the type from the context) and an initial value. Here are some examples of variable declarations:

```
var x as Integer = 10
var y as Boolean = true
var name as String
```

Variables can have complex types such as structures or maps as well as simple types.

Here is the general form for declaring and initializing variables:

var *name* [**as** *Type*] **=** *value*

Here are a few examples:

```
var x as Integer = 10                // shows explicit typing
var x = 10                           // shows implicit typing
var greeting as String = "Hello"     // shows a string variable
var y = [1..6]                       // shows a sequence of
                                     //      integers from 1 to
6
var menu = {"ham", "cheese", "blt"} // shows a set
```

There are three types of variables in AsmL:

- Global variables
- Local variables
- Instance-based variables

Instance-based variables are variables that exist for each instance of a class, and we will discuss them later. Global and local variables are variables with particular scopes. The *scope* of a variable is that part of the program in which the variable name is valid. Within a variable's scope, you can legally refer to it, update its value, or use it in an expression. Referring to a variable outside of its scope results in a compiler error.

Any variable can be understood within the context of the current state. Global variables are accessible from any machine. They are declared at the very beginning of the program:

Example 21 Scope of global variables

```
var x = 20 // global variable
```

```
var y = 10 // global variable
Main()
  step
    x := x + 15
    y := 15
  step
    WriteLine ("X is " + x + " and y is " + y)
```

The result is:

```
X is 35 and y is 15
```

Local variables are only visible within the block where they occur. If a variable is declared within a step, it's a local variable and its scope includes all the steps of that submachine (from the point where it is first declared). However, it's better practice to introduce new methods rather than to introduce new machines. This is shown in the following example:

Example 22 Sequential steps in separate methods

```
var count1 as Integer = 10
var count3 as Integer = 50

Main()
  step
    M1()
  step
    M2()

M1()
  step
    WriteLine ("The value of outer count1 = " + count1)
  step
    var count1 as Integer = 20
    var count2 as Integer = 30
    WriteLine ("The value of inner count1 = " + count1)
    count1 := 3
    count3 := count3 + count2
  step
     WriteLine ("The value of inner count1 = " + count1)

M2()
 step WriteLine ("The value of outer count1 = " + count1)
 step WriteLine ("The value of outer count3 = " + count3)
```

The result of this code is:

```
The value of outer count1 = 10

The value of inner count1 = 20

The value of inner count1 = 3

The value of outer count1 = 10
```

```
The value of outer count3 = 80
```

The first two statements declare and define the two global variables `count1` and `count3`, with initial values 10 and 50, respectively. Both these variables exist from this point until the end of program.

The program then runs a machine with two methods, `M1` and `M2`. The method `M1` prints out the initial value of `count1`, which is 10, and then, in another step, defines two local variables, `count1` and `count2`. The `count1` declared here is different from the first `count1`. The first `count1` still exists, but its name is shadowed by the second `count1`. Any use of the name `count1` within this block refers to the `count1` declared within the block.

We're duplicating the name `count1` only to illustrate how local scope works. Although this code is legal, it's generally poor programming practice to do this.

The second output line shows that we are now using the local variable, `count1` rather than the global variable. We then update both `count1` (local) and `count3`. Notice that both the global variable, `count3`, and the local variable, `count2`, are accessible from within `M1`. In the method's last step, the updates take effect, and we again display the new value of `count1` (local).

Once M1 is finished, its local variables no longer exist. The variables `count1` and `count3` still exist and our output shows that count3 was incremented within `M1`.

11 Classes

Classes in AsmL have many similarities to those in other languages. They are templates for user-defined types that can contain both fields and methods.

A value whose type is a class is an *instance* of the class, or, alternatively, an object.

Classes in AsmL are different from those in other object-oriented languages in that, in AsmL, "objects" have meaning only as identities that distinguish one occurrence of a variable from another. Our objects don't "contain" anything; they are just abstract elements or object-identities. (Seeing each instance as a distinct GUID is a reasonable intuition.) Thus, objects in this model have no state; instead, it is the transition of the machine that provides the model's sole notion of dynamic "state."

11.1 Fields defined in a class

Here is a simple class declaration:

Example 23 Declaration of instance variables

```
class Person
  var name as String
  var age as Integer
```

This is a class called `Person` that has two variables: one called `name` and another called `age`. Variables contained within a class definition are called *instance variables*. If there are two instances of `Person`, there will be two variables called `name` and two variables called `age`. In other words, `name` and `age` occur once per instance of the class.

11.2 New instances of a class

To create an instance of a class, put the keyword new in front of the class name and supply values for any fields whose values were not specified in the class declaration:

Example 24 The "new" operator

```
class Person
  var name as String
  var age as Integer

p1 = new Person("William", 40)
p2 = new Person("Amy", 20)
```

This example creates two new constants (`p1` and `p2`) of type `Person` and initializes the instance variables. Each instance of the class `Person` is distinct from all others – `p1` and `p2` are different entities with their own identity. Creating an instance of a class with the operator `new` augments the model with a new

element. The operator `new` is used only when creating new memory as when we are creating instances of classes. (The operator `new` is not used for structured values, as described later.)

To refer to a member of a specific instance, use any expression whose value is the desired instance, followed by a dot (".") and then by the field name. For example, to refer to the instance variable called "name" that is associated with p1, write:

```
p1.name
```

11.3 Updates to instance variables

Here is an example of updating instance variables:

Example 25 Updating instance variables

```
class Person
  var name as String
  var age as Integer
var p1 = new Person("William", 40)
var p2 = new Person("Amy", 20)

Main()
  step
    p1.name := "Bill"
  step
    WriteLine("Name of p1 is " + p1.name)
  step
    WriteLine("Name of p2 is " + p2.name)
```

This changes the `name` instance variable of `p1` from William to Bill. The result is:

```
Name of p1 is Bill

Name of p2 is Amy
```

The discipline of stepwise updates applies to all variables, even those declared within classes. For example, although the system state includes a `name` instance variable for each variable of type `Person`, the value of each of these occurrences is understood as being relative to the step of the abstract machine that characterizes the run of the entire system.

11.4 Instances as "references"

Instances of classes provide a form of reference-indirection in AsmL. This property becomes important in methods whose arguments are instances of a class:

Example 26 Classes as references (1)

```
class Person
  var name as String
```

```
  var age as Integer

RenamePerson(p as Person, newName as String)
  p.name := newName

Main()
  var p1 = new Person("William", 40)
  step
    RenamePerson(p1, "Bill")
  step
    WriteLine("Name of p1 is " + p1.name)
```

In AsmL, all method arguments and return values are "by value" with the exception of instances of a class. Instances are passed "by reference." (Even instances can be considered as passed by value if you view the argument being passed as the object-id.) In AsmL, classes are the only way to share memory, and they are the only form of aliasing available. There are no pointers in AsmL.

Example 27 Classes as references (2)

```
class Person
  var age as Integer
  var LastName as String

Main()
  var Bob = new Person(30, "Jones")
  var Amy = new Person (20, "Smith")
  var BobsFriend = Amy
  step
    BobsFriend.age := 21
  step
    WriteLine("Amy's age is " + Amy.age)
```

The result is:

```
Amy's age is 21
```

This example shows that, by equating `BobsFriend` with `Amy`, names like `BobsFriend.age` and `Amy.age` refer to the same variable (i.e., the same location in memory). Changing the age of `BobsFriend` is the same as changing the age of `Amy`.

11.5 Derived classes

AsmL supports *inheritance*, which means that you can define one class in terms of another. A class that is defined in terms of another class is called the *derived class.* The class that it is derived from is the *base class*. AsmL supports single inheritance, which means a derived class can only have one base class.

If we see that `class` A **extends** B, then we know that the state variables and operations defined for instances of class B are also available for all instances of class A. Here is an example of inheritance.

Methods may be specialized by derived classes. This means that when a method is invoked, the method body associated with the most specific class will be applied.

Example 28 Derived classes

```
class BasicBox
  var length as Double
  var width as Double
  var height as Double

  Volume() as Double
    return(length * width * height)

class FragileBox extends BasicBox
  Volume() as Double
    return(0.85 * length * width * height)

Main()
  step
    var BasicBox1 = new BasicBox(1.0, 1.0, 0.5)
    var FragileBox1 = new FragileBox(1.0, 1.0, 0.5)
  step
    WriteLine("The volume of the basic box is " +
                                            BasicBox1.Volume())
  step
    WriteLine("The volume of the fragile box is " +
                                            FragileBox1.Volume())
```

The base class is `BasicBox`. It has three instance variables and a method that calculates the volume. Next, we derive a class for a different kind of box called `FragileBox`, which is used for packing fragile items. This requires more packing material so the capacity of `FragileBox` is less than that of `BasicBox`. To reflect this, we use a different version of `Volume`.

From the point of view of variables, each instance of a class is distinct from all other instances. However, the methods available for one instance of a class are available for all other instances of the same class. In other words, while state variables occur for each instance, operations are defined once per class.

12 Structured Values

As described above, a variable—either global or instance-based—is associated with a *value* at each step of the run. The values of some variables have structure; while some (like `Boolean` values) are simple. We gave examples above of cases where a variable contains a value that has distinct components. For example, we showed how total and partial updates can be applied to update a set-valued variable with additional elements. In this section, we give a fuller picture of structured values in AsmL. You can create new types for your own structures.

The keyword `structure` in AsmL declares a new type of compound value.

Example 29 Structured values

```
structure Point
  x as Integer
  y as Integer

var myPosition as Point = Point(0, 0)

Main()
  step
    step
      myPosition := Point(10, 12)
    step
      myPosition.x := 13
```

This example shows a variable `myPosition` whose value is structured as a `Point`. `Point` is an example of a compound-value type.

The run shows two ways that the value of `myPosition` may be updated. The first is a total update where the value `(0, 0)` is replaced with `(10, 12)`. The second step shows a partial update where the "`x`" component of the point is updated to the value `13`. The "y" component is unchanged.

Note that the example includes just one variable, `myPosition`. The fields `x` and `y` are not variables. Instead, we consider them to be indexers into the variable `myPosition`. (AsmL does not allow the variable keyword `var` to be used in structure declarations.)

One way to think of structures is as a generalization of "bit fields". Like bit fields, structures do not provide any memory—they only give a structured interpretation of memory. Instances of classes, on the other hand, provide new memory. (Here, "memory" and "variables" are synonyms.)

Don't confuse structures with classes. Classes contain instance variables and are the only way in AsmL to share memory. Structures contain fields and do not share memory:

Example 30 Structures can't share memory

```
structure Point
  x as Integer
  y as Integer

var point1 as Point = Point(0, 0)
var point2 as Point = point1

run ()
  step
    WriteLine("Point1's coordinates are " + point1)
  step
    WriteLine("Point2's coordinates are " + point2)
    point2.x := 10
  step
    WriteLine("Point1's coordinates are " + point1)
  step
    WriteLine("Point2's coordinates are " + point2)
```

The result of this code is:

```
Point1's coordinates are Point(x=0,y=0)

Point2's coordinates are Point(x=0,y=0)

Point1's coordinates are Point(x=0,y=0)

Point2's coordinates are Point(x=10,y=0)
```

Even though we equate `point1` with `point2`, changing the x-coordinate of `point2` does not change the x-coordinate of `point1`. Unless it's absolutely necessary to share memory, use structures instead of classes. In general, it's always best to keep the amount of state to a minimum.

Structures can inherit members from other structures, using the **extends** keyword and the form *A* **extends** *B*. This means that structure A will include all the members of structure B.

12.1 Structure cases

Structures may incorporate case statements as a way of organizing different variant forms:

Example 31 Structure cases

```
structure InputEvent
  case KeyInput
    key as Char
  case SwitcherInput
    toggle as Boolean
  case CoinInput
```

```
    coin as Integer
  case CoinReturn
    // no data needed
```

The `InputEvent` structure defines different types of input events. An input event can be a key press, a toggle switch, a coin, or a coin return button. Notice that it's possible, as with `CoinReturn`, to leave it undefined.

AsmL structure cases are like unions in C or variants in Visual Basic.

The match operation is used to select cases:

Example 32 Matching against structure cases

```
structure InputEvent
  case KeyInput
    key as Char
  case SwitcherInput
    toggle as Boolean
  case CoinInput
    coin as Integer
  case CoinReturn
    // no data needed

HandleInput(e as InputEvent)
  match e
    KeyInput(k):      WriteLine("Key was pressed: " + k)
    SwitcherInput(t): WriteLine("Switch flipped: " + t)
    CoinInput(c):     WriteLine("Coin inserted: " + c)
    CoinReturn():     WriteLine("Coin return pressed.")

Main()
  step 1: HandleInput(KeyInput('a'))
  step 2: HandleInput(CoinReturn())
  step 3: HandleInput(CoinInput(25))
```

13 Sets

A set is an unordered collection of distinct values that are of the same type (for example, all integers or all characters). These values are called the *elements* or the *members* of the set.

13.1 Sets as enumerated values

To show all the elements of a set, frame the elements between two curly braces and separate one element from the other by commas. Here is a set named `A` whose elements are the letters a, e, i, o, u:

```
A = {'a', 'e', 'i', 'o', 'u'}
```

Elements in a set have no particular order. The set `{'a', 'e', 'i', 'o', 'u'}` is the same value as `{'e', 'o', 'a','i', 'u'}`.

Sets contain no redundant elements (although including them doesn't affect the value of the set or its validity). The set `{'a', 'a', 'e', 'i', 'o', 'u'}` is identical to `{'a', 'e', 'i', 'o', 'u'}`.

A set without any elements is called the empty set and is denoted by a pair of empty braces `{}`. In AsmL (although not in mathematics more generally), sets must be finite. You can't express "the set of all odd numbers" directly in AsmL because there are an infinite number of elements. (You can, of course, test integers using an `IsOdd()` function to give equivalent functionality.)

The following example lists all the elements of a set:

Example 33 Set with enumerated elements

```
x = {1, 2, 3, 4}

Main()
  WriteLine(x)
```

13.2 Sets given by value range

Another way of showing a range of elements is by including the first and last members and separating them by two dots, as shown below:

Example 34 Set range

```
x = {1..4}

Main()
  WriteLine(x)
```

Ranges may be used for integer values (that is, values of type `Byte`, `Short`, `Integer` and `Long`) and `Char` (character) values.

13.3 Sets described algorithmically

To specify a set you can either list all the elements in that set, as we did above, or you can state the properties that characterize the elements of the set. The first method is practical for small sets while the second can be used for sets of any size. In many cases, listing all the elements of a set isn't practical.

Suppose we have a set `A` that includes the integers from `1` to `20`, and we want to find those numbers that, when doubled, still belong to the set. We will call that set `C` and express it this way:

```
C = {i | i in A where 2 * i in A}
```

The "`|`" symbol is read as "such that" and the entire expression is read as "`C` is the set of *i* such that *i* is in `A` where two times *i* is in `A`." The portion of the declaration following the | is called the *binder*. It binds names (in our case, the name "`i`") with values. The binder uses either the word "`in`" or the "=" sign and an expression containing the values that will be associated with the names. It can include a "`where`" clause that constrains those values.

Here is an example:

Example 35 Set comprehension

```
A = {1..20}

C = {i | i in A where 2 * i in A}

Main()
  step
    WriteLine(C)
```

This example finds all the integers in `A` that, when multiplied by `2`, are still in `A`. The result is that `C` = `{1, 2, 3, 4, 5, 6, 7 8, 9, 10}`. The use of *i* may be puzzling because it is referenced before it is defined. This is an example of a bound name, which means that the binder defines it. The bound name immediately precedes its definition. It is associated with values during the parallel evaluation of an expression. Bound names can be used with sets, sequences, and maps.

Here is an example of a binder with multiple names and multiple constraints:

Example 36 Binding multipile names

```
A = {1..5}

C = {(i, j) | i in A where i < 4, j in A where j < i}

Main()
  step
    WriteLine (C)
```

This example finds pairs of numbers where the first number is a member of `A` and less than `4`, while the second number is also in `A` and is less than the first number. The result is that `C` = `{(2, 1) (3, 1) (3, 2)}`.

13.4 Set Operations

In this section we discuss some of the most common operations you may want to perform on sets. For more complete documentation, see the AsmL runtime library documentation.

13.4.1 Union

The `union` operation combines the elements of different sets into a single set:

Example 37 Set union

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6}

Main()
  step
    WriteLine(B union A)
```

The result is the set {1, 2, 3, 4, 5, 6}.

13.4.2 Intersect

The `intersect` operation finds elements that are common to different sets:

Example 38 Set intersection

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6}

Main()
  step
    WriteLine(B intersect A)
```

The result is the set {4, 5}, which includes the two elements the sets have in common. If there are no elements in common, the result is the empty set {}.

13.4.3 Size

The `Size` method returns the number of elements in a set:

Example 39 Set size

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6}

Main()
  step WriteLine(Size(A))
```

The result is 5. The parentheses around A are required because **size** is a method and, like any other method, its arguments must be enclosed in parentheses.

14 Sequences

A sequence is a collection of elements of the same type, just as a set is. However, sequences differ from sets in two ways:

- A sequence is ordered while a set is not
- A sequence can contain duplicate elements while a set does not

Elements of sequences are contained within square brackets:

`[]`

Here are two sequences:

`[1, 2, 3, 4]` and `[4, 3, 2, 1]`

These sequences are not equivalent because they are ordered differently. Here are two more sequences:

`[a, e, i, o, u]` and `[a, a, e, i, o, u]`

Again, these sequences are different because the second sequence contains the letter "a" twice.

The following code shows the difference between sets and sequences when there are duplicate elements:

Example 40 Sequences versus sets

```
X = {1, 2, 3, 4}
Y = {1, 1, 2, 3, 4}
Z = [1, 1, 2, 3, 4]

Main()
  step WriteLine("X = " + X)
  step WriteLine("Y = " + Y)
  step WriteLine("Z = " + Z)
```

Running this example produces these results:

```
X = {1,2,3,4}

Y = {1,2,3,4}

Z = [1,1,2,3,4]
```

Notice that the variables `X` and `Y` are sets while `Z` is a sequence. Even though `Y` contains duplicate elements, AsmL ignores them. The variable `Z`, on the other hand, is a sequence, which allows duplicate elements.

The following piece of code shows the difference between sets and sequences when the order of the elements varies.

Example 41 Ordering of sequences

```
A = {1, 2, 3, 4}
B = {4, 3, 2, 1}
C = [4, 3, 2, 1]
D = [1, 2, 3, 4]

Main()
  step
    if A = B then
      WriteLine("A = B")
    else
      WriteLine ("A <> B")
  step
    if C = D then
      WriteLine("C = D")
    else
      WriteLine("C <> D")
```

Running this program produces these results:

```
A = B

C <> D
```

Even though the order of sets `A` and `B` are different, AsmL considers them to be equivalent. Sequences `C` and `D` are not equivalent, however.

Sequences are zero-based, which means the first element in the sequence is indexed by zero ("0"). To select a specific element from the sequence, use the sequence name followed by the element number, enclosed in parentheses. The following example selects the second element in the sequence:

Example 42 Accessing sequence entries by index

```
m = [1..5]
Main()
  step
    WriteLine(m(1))
```

This code displays the number `2`.

15 Parallel evaluation

It is possible to make updates in parallel using the `forall` statement that evaluates all the members of a set or sequence in a single step.

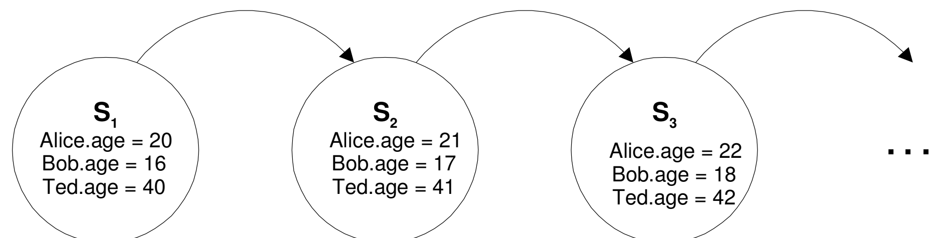
Again, although most programming languages do everything sequentially, AsmL favors parallelism by default. Parallel, rather than sequential, is the norm. (The construct for sequential iteration, **step**, is discussed above in section 4.)

Parallel evaluation is helpful because, as we shall see, it greatly reduces the number of steps required to specify an algorithm.

Example 43 Parallel evaluation

```
class Person
  var age as Integer

Amy    = new Person(20)
Bob    = new Person(16)
Duncan = new Person(40)

People = {Amy, Bob, Duncan}

GrowOlder()
  forall p in People
    p.age := p.age + 1

var year = 2000

Main()
  step while year < 2010
    WriteLine("Amy's age is " + Amy.age + " in " + year)
    WriteLine("Bob's age is " + Bob.age + " in " + year)
    WriteLine("Duncan's age is " + Duncan.age + " in " + year)
    GrowOlder()
    year := year + 1
```

In this example, the ages of Amy, Bob, and Duncan are incremented on a yearly basis for the years 2000 to 2009. Each step of the machine evaluates all three ages at once. This is illustrated in the following diagram:



It would take 9 steps to do the entire calculation, instead of the 27 steps that would be required if the age of each person were incremented sequentially.

It is interesting to count how many variables exist in the example. There are three. `Amy`, `Bob` and `Duncan` are constants; however `Amy.age`, `Bob.age` and `Duncan.age` are variables.

15.1 Sequential iteration

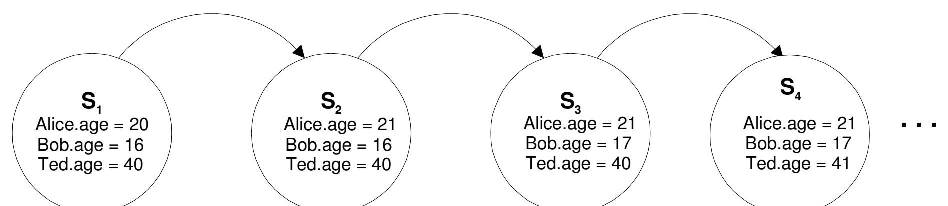
As described above in section 4.3, AsmL also provides for sequential iteration through the elements in a collection using the **`step foreach`** statement.

Here is how sequential iteration could have been used in the previous example. You will note that the end result is the same, but more steps will be used to achieve that result. (Remember that for simplicity and to reduce test cases, it is always desirable to minimize the number of steps.)

Example 44 Sequential iteration over a collection

```
class Person
  var age as Integer

var Amy    = new Person (20)
var Bob    = new Person (16)
var Duncan = new Person (40)

People = {Amy, Bob, Duncan}

GrowOlder()
  step foreach p in People
    p.age := p.age + 1

var year = 2000

Main()
  step while year < 2010
    WriteLine("Amy's age is " + Amy.age + " in " + year)
    WriteLine("Bob's age is " + Bob.age + " in " + year)
    WriteLine("Duncan's age is " + Duncan.age + " in " + year)
    GrowOlder()
    year := year + 1
```

The difference between Example 43 and Example 44 is that it takes many more steps to accomplish the result in Example 44 :



When you use **`forall`**, all the ages are incremented simultaneously for each iteration. When you use **`step foreach`**, the ages are incremented sequentially, with a new step for each update. This results in many more steps. The moral is

that you should never assume that actions must happen sequentially. Instead, it is often the case that changes can happen in parallel without any confusion. If you can remove a sequential dependency, you should do so.

If you need to use `step foreach`, and you are using it with sets, as in this example, remember that sets have no inherent order. This means that the system will perform the sequential evaluations on the set's elements in some arbitrary order, which is not necessarily the order you see written in your code. If the order is important, use sequences rather than sets.

16 Maps

Maps in AsmL are tables that associate keys to values. Maps are similar to associative arrays in Perl or hash tables found in C++ libraries. Like arrays, maps have a set of unique keys and a set of values associated with those keys. The keys are used to retrieve the values. Unlike associative arrays, the keys can be any type of datatype, either simple or complex. A key may be, for example, an integer, a string, or enum or even another map.

A simple example of a map might be a company's telephone book, where a unique key, such as an employee's email name, is used to retrieve information such as the employee's extension. Keys can have more than one argument. Another name for the set of keys is the *domain* of the map, and another name for the set of associated values is the *range* of the map. In our example, the domain would be all the employee email names included in the phone book and the range would be all the extensions.

Here is an example of how to declare a variable as a map:

Example 45 Map declaration

```
var phoneNumber as Map of String to Integer
```

This map will be used to associate employee names with telephone extensions. The keys in the map's domain are of type `String` and the values in the map's range are of type `Integer`. In this example, the key has a single argument.

The "->" symbol associates keys with values. It is read as "maps to."

Example 46 Enumerating map entries

```
phoneNumber = {"Bob" -> 100, "Carla" ->101}
```

The map `phoneNumber` has two elements in it. The first associates the key Bob with the value 100 and the second associates the key Carla with the value 101.

Example 47 Enumerated map

```
var phoneNumber as Map of String to Integer =
                               {"Bob" -> 100, "Carla" -> 101}
Main()
  step
    WriteLine("The entries are " + phoneNumber)
```

This code prints out all the elements of the map, `{"Bob" -> 100, "Carla" -> 101}`.

The next example prints out the extension of one of the employees:

Example 48 Looking up values in a map

```
var phoneNumber as Map of String to Integer =
                               {"Bob" -> 100, "Carla" -> 101}
```

```
Main()
  step
    WriteLine("Carla's extension is " + phoneNumber("Carla"))
```

Notice that the name of the map (in our case, `phoneNumber`, is used the same way that a method name is used and the domain element is used the same way as a method argument. You can't use an element of the range as an argument since these elements aren't unique.

Maps are similar to sets in that we can use binders and constraints to specify particular elements. This example finds telephone extensions that are less than `103`:

Example 49 Map-based binding

```
var phoneNumber as Map of String to Integer =
    {"Bob" -> 100, "Carla" -> 101,
     "Duncan" -> 102, "Amy" -> 103}

Main()
  step
    y = {j | i -> j in phoneNumber where j < 103}
    WriteLine("The set of extensions less than 103 is " + y)
```

16.1 Map Comprehensions

Map comprehensions allow you construct maps using iterated expressions. Iterated expressions are expressions that repeat themselves until a specified set of values has been created. Map comprehensions are most often used to initialize a map so that it has a particular set of values at the beginning of the program. In this example, we create a map whose domain is the set of integers from 1 to 10 and whose range is constructed by adding one to each element of the domain:

Example 50 Map comprehension

```
Main()
  step
    y = {i -> i + 1 | i in {1..10}}
    WriteLine(y)
```

In general, the form of a map comprehension is:

{*i* -> *f(i)* | *i* in *S* where ...}

This denotes a table whose keys are taken from the set S (or a subset of S if a "where" clause filters out some values) and whose lookup values are given by applying the function `f`.

16.2 Maps with multiple arguments

Maps whose keys are tuples can be thought of as multidimensional arrays. Each argument is one of the indexers into the table. When you declare the map, separate each of the argument types with a comma (",") and enclose them all in parentheses ("(" and ")"):

Example 51 Tuples as map keys

```
var phoneNumber as Map of (String, String) to Integer =
         {("Bob", "Home") -> 5550000, ("Bob", "Work") -> 100}
Main()
  step
    WriteLine (phoneNumber)
```

An alternative approach to using multiple arguments is to have multiple maps. The previous example could also be written as:

Example 52 Nested maps

```
var phoneNumber as Map of String to Map of String to Integer =
   {"Bob" -> {"Home" -> 555000, "Work" -> 100}}

Main()
  step
    WriteLine(phoneNumber)
```

Whether you use keys with multiple arguments or maps of maps is a matter of taste. Many people find multidimensional arrays easier to understand than nested maps but there is no difference in terms of results or computation time.

16.3 Map Operations

16.3.1 Indices

We use the `Indices` method to find the domain of a map:

Example 53 Map indices

```
var phoneNumber as Map of String to Integer =
                               {"Bob" -> 100, "Carla" -> 101}
Main()
  step
    WriteLine("The keys are " + Indices(phoneNumber))
```

The following is printed: "`The keys are {Bob, Carla}`".

16.3.2 Values

To find the range of a map, we use the `Values` method:

Example 54 Map values

```
var phoneNumber as Map of String to Integer =
                               {"Bob" -> 100, "Carla" -> 101}
Main()
  step
    WriteLine("The values are " + Values(phoneNumber))
```

This example prints: // prints "`The values are {100,101}`".

16.3.3 Union

The `union` operator combines two disjoint maps:

Example 55 Map merge

```
A = {"Bob" -> 100, "Carla" -> 101,
     "Duncan" -> 102, "Amy" -> 103}

B = {"Jeff" -> 104, "George" -> 105,
     "Ann" -> 106, "Beth" -> 107}

Main()
  step
    WriteLine(A union B)
```

The result is the map `{Amy -> 103, Ann -> 106,Beth -> 107, Bob ->100, Carla -> 101, George ->105, Jeff -> 104, Duncan -> 102}`.

16.3.4 Override (+)

The + operator combines two maps, where the right argument has precedence over the left argument:

Example 56 Map merge

```
A = {"Bob" -> 100, "Carla" -> 101,
     "Duncan" -> 102, "Amy" -> 103}

B = {"Jeff" -> 104, "Carla" -> 105,
     "Ann" -> 106, "Duncan" -> 107}

Main()
  step
    WriteLine(A + B)
```

The result is the map `{Amy -> 103, Ann -> 106, Bob ->100, Carla -> 105, Jeff -> 104, Duncan -> 107}`.

16.4 Partial updates of maps

Along with sets, partial updates are also useful with maps. Suppose we have a telephone book that associates names with office extensions. We could add entries to the phone book with partial updates.

Example 57 Partial update of maps

```
var Extension as Map of String to Integer = {->}

Main()
  step
    Extension("Bob") := 100
    Extension("Carla") := 101
  step
    WriteLine("Bob's extension is " + Extension("Bob"))
    WriteLine("Carla's extension is " + Extension("Carla"))
```

A name (a string) is associated with an extension (an integer). Initially, the map is empty. We can then add entries to it, one person at a time. To find the extension of someone, we apply the name of the map (in our case, Extension) to the element in the map's domain that interests us (in this case, first Bob, and then Carla.)

Partial updates can also be used with nested maps. We can modify our previous example so that the phone book associates a phone number with a location as well as a person. For instance, Bob may have one number for his mobile and another for his office extension.

Example 58 Partial updates of nested maps

```
var phoneNumber as Map of String to (Map of String to Integer)
                                                        = {->}
Main()
  step
    phoneNumber("Bob") := {->}
  step
    phoneNumber("Bob")("Home") := 5555000
    phoneNumber("Carla") := {"Work" -> 101,
                             "Home" -> 5555001}
  step
    WriteLine ("Bob's home phone number is " +
                                    phoneNumber("Bob")("Home"))
```

17 Nondeterminism

Many systems exhibit nondeterministic behavior, which means that there is more than one outcome possible. Flipping a coin, for example, can produce either heads or tails. A CD player may have a "Random" function that selects any one of a number of discs. In neither case do we know what the actual result will be. We do, however, know what the possibilities are. The coin will either be heads or tails. It can't be anything else. The player will pick a CD that is in one of its slots. To summarize, nondeterministic systems exhibit two characteristics:

- There is a finite set of possibilities
- Within that set the result may be any value, but we don't know which one

A good way to understand nondeterminism is to think of it as an interaction between your program and the external environment. For example, when you buy a lottery ticket, you don't know which number will be on your ticket, even though you might know that it must be a seven-digit number. (The lottery agency has a reciprocal but equally nondeterministic view: it knows its algorithm for generating ticket numbers but doesn't know who will buy a ticket.)

In addition to accuracy, nondeterminism provides flexibility. A specification should not limit the possible implementations and nondeterminism can be used to show where multiple options are possible.

Using nondeterminism to model aspects of your system that are outside the chosen level of detail is an important part of keeping your model focused (and not being distracted by detail that does not matter for the chosen view). Successful modelers are conscientious about excluding what does not matter (for the chosen level of abstraction) and including what does.

There are three ways to introduce nondeterminism into your model. We describe each of these in a separate section.

17.1 Nondeterministic choice

The first way to incorporate nondeterminism into a system is to use the **choose** statement or the **any** expression. Here is a simple example:

Example 59 Nondeterministic choice, expression-level

```
A = {1..10}

Main()
  step
    x = any y | y in A
    WriteLine("x is " + x)
```

This selects any element from A. We can add qualifications to limit the possible choices using a **where** clause. Here we show the statement-level version, with the choose keyword:

Example 60 Nondeterministic choice, statement-level

```
S = {1, 6, 3}

Main()
  step
    choose i in S where i > 4
      WriteLine(i + " was chosen.")
```

This example selects some element from A that is greater than 4.

If there are no elements that fit the qualifications, the system generates a runtime error. You can avoid this with **ifnone**:

Example 61 Default choice

```
S = {1, 6, 3}
Main()
  step
    choose i in S where i > 7
      WriteLine(i + " was chosen.")
    ifnone
      WriteLine("There were none to choose.")
```

This program displays "There were none to choose." since there are no elements greater than 7. The ifnone statement is like "else" in "if ... then ... else".

17.2 External nondeterminism

Another kind of nondeterminism occurs when you make a method call outside of AsmL, for example, into an external library. AsmL treats such calls as nondeterministic, since the model doesn't describe the way the external module works.

Note that the sequential steps of the model control the ordering of calls to external routines and not the order of appearance in the program text. You should not assume that the order in which external functions appear in a method body will be the same as the order in which they are invoked at runtime, unless a "step" separates them.

Example 62 External nondeterminism

```
Main()
  step
    WriteLine("This could print second.")
    WriteLine("This might print first.")
```

```
step
  WriteLine("This will print last.")
```

The calls to the external routine `WriteLine()` are only partially ordered in this example.

17.3 The nondeterminism of "new"

The "new" operator that is used to create instances of a class can be seen as an external or nondeterministic function. The reason for this is that "new" expressions (like `new Person("Bill", 40)` in the earlier examples) return a different value every time they are invoked.

In other words, asking for a new instance of a class is like buying a lottery ticket. The lottery agency guarantees that the ticket you get is different from any other they have already sold or will sell in the future, but from your point of view the ticket agent is handing you a ticket with a random number printed on it. You get a new number every time you buy a ticket.

18 Enumerations

Enumerations provide a way of creating symbolic constants. Sometimes you may have variables that can only take on a limited set of values that are best referred to using labels – the days of the week, for example, or the months of the year. If this situation arises, use an enumeration, which is declared with the keyword **enum**. Enumerations are really literals, just like, for example, the number 2, that have labels. Here is an example:

```
enum Color
  Red
  Green
  Blue
```

This statement does two things:

- It declares an enumeration type called `Color`
- It establishes `Red`, `Green`, and `Blue` as symbolic constants called *enumerators*.

You can use an enumeration name to create a variable of that type:

```
var range as Color
```

A variable of type `Color` can only take on one of the three values specified in the enumeration.

By default, enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second, and so on. You can override the default by explicitly assigning integer values:

Example 63 Initialized enumeration

```
enum Punctuation
  Comma = 5
  Period = 3
  Semi = 1

Main()
  step
    if Comma < Period then WriteLine(Period)
    else WriteLine(Comma)
```

This code assigns specific integer values to the enumerators and uses these values to control the screen display. You can also define just some of the enumerators explicitly:

Example 64 Partially initialized enumeration

```
enum Punctuation
  Comma
  Period = 100
  Semi
Main()
  step
```

```
    if Semi < Period then WriteLine(Period)
    else WriteLine(Semi)
```

In this case, `Comma` is 0 by default. Subsequent uninitialized enumerators are larger by one than their predecessors, so `Semi` would have the value 101. This code displays `Semi`, since it is larger than `Period`.

19 Conditionals and Loops

Like other programming languages, AsmL has facilities for allowing you make decisions based on data values and for making programs repeat a set of actions until a specific condition is met. To do either of these, you often need a way of comparing two values. This involves *relational operators*, and they are listed in the following table:

Symbol	Meaning
<	less than
>	greater than
=	equal to
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

Each of these operators compares two values and returns one of two possible Boolean values: **true** if the expression is true, or **false** if the expression is false.

19.1 The If Statement

There are a variety of ways to use the results of a comparison to modify the behavior of a program. The most basic of these is the **if..then** statement. This allows your program to execute a single statement, or a block of statements (denoted by the indentation) if a given condition returns the value **true**. Here is a simple example:

Example 65 If-then

```
S = {1..6}

Main()
  step
    forall x in S
      if (x mod 2 = 1) then
        WriteLine(x + " is odd.")
```

The results of this code are:

```
1 is odd.

3 is odd.

5 is odd.
```

Here we print out only the odd members of the set S. We determine if a number is odd using the modulus operator. If the remainder is 1 then the number is odd and we display it. If the remainder is 0, nothing happens.

By extending the **if** statement to an **if-else** statement, we can perform one action if the condition is true and another if it false:

Example 66 If-then-else

```
S = {1..6}

Main()
  step
    forall x in S
      if (x mod 2 = 1) then
        WriteLine(x + " is odd.")
      else
        WriteLine(x + " is even.")
```

This returns the result:

```
6 is even.

1 is odd.

2 is even.

3 is odd.

4 is even.

5 is odd.
```

(Remember that sets have no intrinsic order. If you run this program, the order of the results may differ from the one shown here.)

You can use **if-then-elseif** statements to handle multiple possible conditions that should be tested in order, one after another:

Example 67 if-then-elseif

```
S = {"ham", "turkey", "blt", "cheese"}

Main()
  step
    x = any y | y in S
    if x = "ham" then
      WriteLine ("Ham sandwich")
    elseif x = "turkey" then
      WriteLine ("Turkey sandwich")
    else
      WriteLine ("Cheese sandwich")
```

This program randomly selects one of the elements in set S, thus determining the program's output.

19.2 Logical Operators

As you can imagine, using **if** statements with complex conditions can be rather clumsy. To simplify the situation, use logical operators that allow you to combine a series of comparisons into a single expression. Here is a table of the most commonly used ones:

Symbol	Meaning
and	Both conditions must be true for a true result
or	At least one condition must be true for a true result
not	Inverts the value of a true or false expression
implies	True unless the first condition is true and the second is false
iff	True when both conditions have the same value

19.2.1 The and operator

Use the **and** operator when you have two conditions that must both be true for a true result:

Example 68 Logical conjunction using "and"

```
S = {1..12}
Main()
  x = any y | y in S
  if (x <= 6) and (x mod 2 = 1) then
    WriteLine (x + " is an odd number between 0 and 6.")
```

The program displays an odd number between 0 and 6, in other words, 1, 3 or 5.

Use the **or** operator when you have two conditions and you want the result to be true if either or both of them are true.

Example 69 Logical disjunction using "or"

```
S = {1..12}

Main()
  step
    x = any y | y in S
    if (x > 6) or (x mod 2 = 1) then
      WriteLine (x + " is greater than six or an odd number")
```

This program displays numbers that are either greater than six, odd, or both. If, for example, it selects the number 6, there will be no output.

19.2.2 The not operator

The **not** operator takes either a true or false result and inverts it. That is, if the original answer is **true**, the final result is **false**. If the original answer is false, the final result is **true**.

19.2.3 The not operator

The **not** operator inverts (or negates) the value of an expression. If the original expression is **true**, the negated expression is **false**. If the original expression is **false**, the negated expression is **true**.

19.2.4 The implies operator

The **implies** operator may be less familiar than **and**, **or**, and **not**, so we will explain it further. This is the truth table, where p is the first condition and q is the second condition:

p	q	p implies q
T	T	T
T	F	F
F	T	T
F	F	T

To put this in more concrete terms, imagine that your boss told you that if you finished a project on time, you would get a raise. If you finished on time and got the raise, your boss told the truth. This is the first case in the truth table. Under what conditions would you say that your boss had lied to you? Clearly, if you finished on time but didn't get the raise, you would call your boss a liar. This is the second case in the truth table. However, what if you failed to finish on time? Either you get the raise anyways (the third case) or you don't get the raise (the fourth case). In neither case, however, can you say that your boss lied.

20 Patterns

21 Predicate Logic

AsmL includes support for predicate logic, a way to systematically describe expressions that are either true or false. Expressions whose value is either true or false are called Boolean expressions.

AsmL's syntax for predicate logic follows accepted mathematical conventions, so the reader who is interested in this subject and wants to learn more about it has a number of standard textbooks to choose from. In this section, we cover the basics to give you a general introduction to the kinds of things that can be described.

A quantifying expression is an important kind of expression used in predicate logic. A quantifying expression returns `true` or `false` depending on whether a given condition has been met *universally* over some collection or *existentially* by at least one example. The universal quantifier is keyword `forall` followed by `holds`. The existential quantifier is the keyword `exists`. They are called quantifiers because they specify how much of the collection (a set, sequence, or map) is included or excluded by the condition.

21.1 forall..holds

Expressions in the form `forall ... holds` are `true` if all members of a collection meet the condition that follows the `holds` keyword:

Example 70 Universal quantification using "forall..holds"

```
S = {1, 2, 4, 6, 7, 8}

IsOdd(i as Integer) as Boolean
  return (1 = i mod 2)

Main()
  test1 = (forall i in S holds IsOdd(i))
  if test1 then
    WriteLine("All odd numbers")
  else
    WriteLine("Not all odd numbers")
```

This prints the result:

```
Not all odd numbers
```

The **forall..holds** quantifer tests whether every element in S is an odd integer. Since S contains two odd elements (the numbers `1` and `7`), the expression is `false`.

You can use filter conditions to further refine the evaluation:

Example 71 Universal quantification using "where"

```
S = {1, 2, 4, 6, 7, 8}
```

```
IsOdd(i as Integer) as Boolean
  return (1 = i mod 2)

Main()
  let test1 = (forall i in S where i > 4 holds IsOdd(i))
  if test1 then
    WriteLine("All odd numbers")
  else
    WriteLine("Not all odd numbers")
```

This returns the same result as the previous example. The filter `where i > 4` restricts the evaluation to the elements `6`, `7`, and `8`.

Universal quantification over the empty set returns the value `true`. In other words, universal quantification is more about the absence of a contradicting counterexample that the presence of positive cases:

Example 72 Universal quantification over the empty set

```
S = {1, 2, 3}

IsOdd(i as Integer) as Boolean
  return (1 = i mod 2)

Main()
  if (forall i in S where i > 3 holds IsOdd(i))
    WriteLine("All elements of S larger than 3 are odd")
```

This will print "All elements of S larger than 3 are odd." (In fact, there are no elements larger than 3.)

21.2 The exists Quantifier

The `exist` quantifier tests whether *at least one* member of a collection meets a specified condition:

Example 73 Existential quantification using exists

```
S = {1, 2, 4, 6, 7, 9, 8}

IsOdd(i as Integer) as Boolean
  return (1 = i mod 2)

Main()
  step
    test1 = (exists i in S where IsOdd(i))
    if test1 then
      WriteLine("Some odd numbers")
    else
      WriteLine("No odd numbers")
```

This returns the result:

`Some odd numbers`

There are three odd numbers present so the result is **true**. You can add filters to refine the evaluation using the keyword `where`:

Example 74 Where conditions

```
S = {1, 2, 4, 6, 7, 9, 8}

IsOdd(i as Integer) as Boolean
  return (1 = i mod 2)

Main()
  test1 = (exists i in S where i > 1 and i < 6 and IsOdd(i))
  step
    if test1 then
      WriteLine("Some odd numbers")
    else
      WriteLine("No odd numbers")
```

This returns the result:

`No odd numbers`

There are no odd numbers in `S` that are greater than `1` and less than `6`.

To find if there is a single occurrence of the stipulated condition, use `exists unique`. This returns `true` only when there is one element that fits the condition given. It returns `false` if there are no elements that fit the condition or if there are multiple elements that fit the condition:

Example 75 Exists unique

```
S = {2, 4, 6, 7, 9}

IsOdd(i as Integer) as Boolean
  return (1 = i mod 2)

Main()
  step
   test1 = exists unique i in S where IsOdd(i)
   if test1 then
     WriteLine("Just one odd number")
   else
     WriteLine("None or more than 1 odd number")
```

The result is: `None or more than 1 odd number`

22 Table of examples

Example 1	Hello, world	7
Example 2	Reading a file	8
Example 3	Fixed-point iteration	11
Example 4	Iterating while a condition is true	12
Example 5	Iterating until a condition is met	12
Example 6	Reading a file using a sequence of steps	12
Example 7	Iteration over items in a collection	13
Example 8	Update statement	15
Example 9	Delayed effect of updates	16
Example 10	Swapping values	17
Example 11	Total update of a set-valued variable	18
Example 12	Partial update of a set-valued variable	18
Example 13	Method	20
Example 14	Update procedures and functions	21
Example 15	Local names	22
Example 16	Local names in sequences of steps	22
Example 17	Method overloading	23
Example 18	User-defined structure type	24
Example 19	Assertions	27
Example 20	Constants	28
Example 21	Scope of global variables	29
Example 22	Sequential steps in separate methods	30
Example 23	Declaration of instance variables	32
Example 24	The "new" operator	32
Example 25	Updating instance variables	33
Example 26	Classes as references (1)	33
Example 27	Classes as references (2)	34
Example 28	Derived classes	35
Example 29	Structured values	36
Example 30	Structures can't share memory	36
Example 31	Structure cases	37
Example 32	Matching against structure cases	38
Example 33	Set with enumerated elements	39
Example 34	Set range	39
Example 35	Set comprehension	40
Example 36	Binding multipile names	40
Example 37	Set union	41
Example 38	Set intersection	41
Example 39	Set size	41
Example 40	Sequences versus sets	43
Example 41	Ordering of sequences	43
Example 42	Accessing sequence entries by index	44
Example 43	Parallel evaluation	45
Example 44	Sequential iteration over a collection	46
Example 45	Map declaration	48

Example 46	Enumerating map entries	48
Example 47	Enumerated map	48
Example 48	Looking up values in a map	48
Example 49	Map-based binding	49
Example 50	Map comprehension	49
Example 51	Tuples as map keys	50
Example 52	Nested maps	50
Example 53	Map domains	50
Example 54	Map range	51
Example 55	Map merge	51
Example 56	Partial update of maps	52
Example 57	Partial updates of nested maps	52
Example 58	Nondeterministic choice, expression-level	53
Example 59	Nondeterministic choice, statement-level	54
Example 60	Default choice	54
Example 61	External nondeterminism	54
Example 62	Initialized enumeration	56
Example 63	Partially initialized enumeration	56
Example 64	If-then	58
Example 65	If-then-else	59
Example 66	if-then-elseif	59
Example 67	Logical conjunction using "and"	60
Example 68	Logical disjunction using "or"	60
Example 69	Universal quantification using "forall..holds"	63
Example 70	Universal quantification using "where"	63
Example 71	Universal quantification over the empty set	64
Example 72	Existential quantification using exists	64
Example 73	Where conditions	65
Example 74	Exists unique	65