The action semantics of object-oriented languages

This item was submitted to Loughborough University's Institutional Repository by the/an author.

Additional Information:


Metadata Record: https://dspace.lboro.ac.uk/2134/28283

Publisher: © Matthew J.A. Caswell

Rights: This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 2.5 Generic (CC BY-NC-ND 2.5) licence. Full details of this licence are available at: http://creativecommons.org/licenses/by-nc-nd/2.5/

Please cite the published version.
This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/
The Action Semantics of Object-Oriented Languages

by

Matthew J. A. Caswell

A Doctoral Thesis

Submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of Loughborough University 14 September 1998

© by Matthew J. A. Caswell, 1998
Abstract

Action Semantics is a framework for defining the semantics of languages. It is intended to be accessible to a wider audience of Computer Scientists than traditional semantics frameworks (such as Denotational Semantics). There has been little work carried out to date on the techniques required to define object-oriented languages with Action Semantics.

The work presented in this thesis examines four potential approaches to defining the Action Semantics of object-oriented languages. In order to illustrate the four approaches a simple language EIL (Example Inheritance Language) is given, and described using these four approaches. The language Smalltalk-80 has been selected for a case study of a practical application of one of the techniques described above.

It is important to be able to relate Action Semantics definitions of object-oriented languages to similar definitions given in other frameworks. It is described how this can be achieved. An example is given for the Action Semantics and Denotational Semantics of Smalltalk.

This thesis concludes that it is feasible to produce Action Semantics definitions of object-oriented languages.

Keywords: Action Semantics; Denotational Semantics; Object-Oriented; Programming Languages; Smalltalk; Equivalence
To Rachel,
with all my love and gratitude
for making me finish this thesis
Acknowledgements

My thanks should go to my supervisor, Dr Roger Stone. Our many discussions have helped me organise my thoughts into something understandable! Also thank you for reading all of those long documents that I produced throughout my study!

My thanks should also go to Dr John Cooke, who read through and gave very helpful criticism of draft versions of this thesis.

Finally thank you to Rachel. You kept me going!
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.1</td>
<td>Using Cells to Model Objects</td>
<td>56</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Using Agents to Model Objects</td>
<td>57</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Copy Down Semantics</td>
<td>58</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Look Up Semantics</td>
<td>60</td>
</tr>
<tr>
<td>3.5</td>
<td>The Semantics of EIL</td>
<td>61</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Cell Based Copy Down Semantics of EIL</td>
<td>61</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Cell Based Look Up Semantics of EIL</td>
<td>63</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Agent Based Copy Down Semantics of EIL</td>
<td>66</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Agent Based Look Up Semantics of EIL</td>
<td>69</td>
</tr>
<tr>
<td>3.6</td>
<td>Conclusions</td>
<td>72</td>
</tr>
<tr>
<td>4</td>
<td>AN ACTION SEMANTICS OF SMALLTALK</td>
<td>74</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>74</td>
</tr>
<tr>
<td>4.2</td>
<td>A Restricted SMALLTALK</td>
<td>75</td>
</tr>
<tr>
<td>4.3</td>
<td>Abstract Syntax</td>
<td>77</td>
</tr>
<tr>
<td>4.4</td>
<td>The Semantics</td>
<td>78</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Semantic Entities</td>
<td>78</td>
</tr>
<tr>
<td>4.4.1.1</td>
<td>Standard Objects</td>
<td>79</td>
</tr>
<tr>
<td>4.4.1.2</td>
<td>The &quot;new&quot; method</td>
<td>81</td>
</tr>
<tr>
<td>4.4.1.3</td>
<td>Variable Initialisation</td>
<td>81</td>
</tr>
<tr>
<td>4.4.1.4</td>
<td>Calling Methods</td>
<td>83</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Semantic Functions</td>
<td>85</td>
</tr>
<tr>
<td>4.4.2.1</td>
<td>Creating User Defined Classes</td>
<td>85</td>
</tr>
<tr>
<td>4.4.2.2</td>
<td>Constructing Methods</td>
<td>86</td>
</tr>
<tr>
<td>4.4.2.3</td>
<td>Expressions</td>
<td>88</td>
</tr>
<tr>
<td>4.5</td>
<td>Conclusions</td>
<td>90</td>
</tr>
<tr>
<td>5</td>
<td>Denotational Semantics</td>
<td>92</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>92</td>
</tr>
<tr>
<td>5.2</td>
<td>Denotational Semantics</td>
<td>92</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Semantic Functions</td>
<td>92</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Environments</td>
<td>94</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Storage</td>
<td>96</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Continuations</td>
<td>99</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Recursion and Iteration</td>
<td>101</td>
</tr>
<tr>
<td>5.3</td>
<td>Generation of a Continuation Style Denotational Semantics from an Action Semantics</td>
<td>104</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Termination States and Continuations</td>
<td>104</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Action Combinators</td>
<td>106</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Primitive actions and yielders</td>
<td>110</td>
</tr>
<tr>
<td>5.4</td>
<td>Conclusions</td>
<td>111</td>
</tr>
<tr>
<td>6</td>
<td>On a Correspondence Between Formal Semantics Definition Methods</td>
<td>112</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>112</td>
</tr>
<tr>
<td>6.2</td>
<td>A Simple Language</td>
<td>113</td>
</tr>
<tr>
<td>6.3</td>
<td>Equivalence of Semantics Definition Methods</td>
<td>117</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Equivalence between the Denotational and Action Semantics of the Example Language</td>
<td>118</td>
</tr>
<tr>
<td>6.4</td>
<td>Equivalence of the Two Semantics Definitions of SMALLTALK</td>
<td>132</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Defining Equivalence</td>
<td>133</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Converting Environments and Stores</td>
<td>134</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Semantics of Method Calls</td>
<td>134</td>
</tr>
<tr>
<td>6.5</td>
<td>Conclusions</td>
<td>140</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>142</td>
</tr>
<tr>
<td>7.1</td>
<td>The Aims of This Thesis</td>
<td>145</td>
</tr>
<tr>
<td>7.2</td>
<td>Future Work</td>
<td>146</td>
</tr>
</tbody>
</table>
F.1.3  Case: enacting the abstraction completes ................................................................. 212
F.1.4  Case: enacting the abstraction escapes, \(Y_2\) is nothing, \(Y_4\) is true ...................... 213
F.1.5  Case: enacting the abstraction escapes, \(Y_2\) is true, \(Y_4\) is nothing ...................... 216
Chapter
1

Introduction

1.1 Motivations and Aims

Action Semantics is a framework for defining the semantics of languages that has been developed largely by Peter Mosses at Aarhus University. One of the main advantages that it offers over other frameworks is its good readability. A computer scientist who is not an expert in formal semantics should be able to read Action Semantics definitions and gain some understanding (at least at a superficial level) of what has been written. It is hoped therefore that Action Semantics will open formal semantics to a wider audience of computer scientists.

In order for Action Semantics to succeed as a practical tool for defining the semantics of languages it needs to be explored in a wide range of paradigms to ensure that it is both feasible and appropriate to use. To date there has been very little work carried out on exploring the suitability of Action Semantics for defining the semantics of object-oriented languages.

Research has already been carried out on defining the semantics of object-oriented languages in other frameworks (for example Denotational Semantics). It would be advantageous to be able to relate any Action Semantics definition of a language, with previous definitions of that language in other frameworks. This might allow us to assess the success or failure of the Action Semantics version to provide a suitable and accessible definition. It may also allow ideas and techniques used in other frameworks to be applied to Action Semantics.
The work presented in this thesis is intended to add to the body of knowledge on the practical application of Action Semantics to defining languages. Specifically object-oriented languages and their definition by Action Semantics are explored.

The aims of the work presented in this thesis are therefore:

i) To demonstrate that it is possible to describe object-oriented languages using Action Semantics.

ii) To develop generic models that can be used within Action Semantics to define languages that exhibit object-oriented features.

iii) To provide examples of how object-oriented languages can be described using Action Semantics.

iv) To suggest methods by which Action Semantics Definitions of object-oriented languages can be related and compared to the existing body of knowledge on the semantics of such languages.

v) To form a basis for future research into Action Semantics and object-oriented languages.

vi) To investigate the usefulness and flexibility of Action Semantics.

1.2 Overview

Chapter 2 of this thesis introduces the background concepts that are required in order to be able to read this thesis. The concept of formal semantics is introduced, and two formal semantics frameworks are discussed, i.e. Denotational Semantics and Action Semantics. The terminology and ideas behind object-oriented languages are then introduced. Finally a number of papers from other authors which investigate the semantics of object-oriented languages are examined in turn.

The concepts of encapsulation and inheritance are central to object-oriented languages. These concepts are examined in Chapter 3 and four potential methods of defining languages that exhibit these properties are discussed. An example language is given that has these properties and it is defined using each of the four methods.
Chapter 1: Introduction

A case study of the techniques developed in Chapter 3 is presented in Chapter 4. A restricted version of the popular object-oriented language Smalltalk-80 is described. A complete semantics of this restricted language is then given.

Chapter 5 introduces the concepts of Denotational Semantics. In particular we examine how it might be possible to generate a Denotational Semantics from an Action Semantics of a language. The techniques that are discussed are then applied to the Action Semantics of Smalltalk to produce a Denotational version. This is useful in that it enables us to compare the semantics of Smalltalk that is developed in this thesis with other authors versions (which are largely written in Denotational Semantics).

Chapter 6 examines a small part of the semantics of Smalltalk. It shows that the two versions of that portion of the semantics that were developed in Chapters 4 and 5, are in fact equivalent. This chapter introduces and demonstrates the techniques that might be used to construct similar proofs between Action Semantics definitions and Denotational Semantics definition.

The conclusions of this work are presented in Chapter 7.
Background and Literature Review

2.1 The Semantics of Languages

Formal semantics is the discipline by which we can define the semantics of computer languages in a precise and unambiguous manner. There are many reasons why this is desirable. Language designers can use formal semantics as a tool in the design process, and also to enable comparison between other languages. It also allows the communication of the concepts of the language to implementors.

Implementors can use the semantics to ensure that the compilers that they write are correct and consistent with other implementations of the same language. Finally programmers can use the semantics as an ultimate reference manual, and to ensure that their programs conform to rigorous standards.

Research into defining the formal semantics of computer languages has been carried out for many years. There have been numerous frameworks that have been suggested for carrying out this task. There has been no single framework that is clearly better than any other, although Denotational Semantics is probably the most frequently used. Hoare and Lauer in 1974 [HoL74] argued that a language should be defined using several different methods; with each method targeted towards a particular class of reader.

There now follows a discussion on two important frameworks for defining semantics: Denotational Semantics and Action Semantics.
2.1.1 Denotational Semantics

Denotational Semantics is one of the oldest of the established frameworks for defining the semantics of computer languages. It is based on the use of the λ-notation, and uses higher order functions and Scott-domains to define semantics. It was developed largely by Robert Milne, Christopher Strachey and Dana Scott [MiS76, Sco76].

One of the main advantages of Denotational Semantics is the ease with which theoreticians can prove properties about languages that have been defined using it. There has been a lot of research carried out into Denotational Semantics, and consequently the underlying theory is very strong.

The semantics are presented as functions from an abstract syntax into a Scott-domain. For example consider the following.

<table>
<thead>
<tr>
<th>Store : Identifier → Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute : Statement → Store → Store</td>
</tr>
<tr>
<td>execute ( [S_1 ; S_2] ) ( st = ) execute ( S_2 ) ( execute ( S_1 ) ( st ) )</td>
</tr>
</tbody>
</table>

Let us assume that we have defined suitable values for the domains used in the above example. The denotation of a statement sequence is shown. A syntactic entity which represents this sequence is enclosed in semantic brackets and is passed as an argument to the function "execute". The result is defined to be a function that maps a store to a new store. The new store in this function is calculated by passing the first statement to the "execute" function. The resulting function is then passed the original store. This will then produce an intermediate store as a result. The process is then repeated with the second statement and the intermediate store to produce the store that is the result of the whole function.

A point to notice about this is that the store for the execution of this statement is passed as an explicit argument to the semantic function. If we later decided that
statements needed to additionally have some form of additional environment information, then we would need to reformulate these rules, in order to include an explicit environment argument.

Let us now consider another example. This example has been extended with an environment as discussed above. The following statement repeatedly executes the sub-statement "S" until the Boolean expression "B" evaluates to false. Assume that a suitable value has been assigned to the function "the_value_of".

\[
\text{execute} \left[ \text{while } B \text{ do } S \right] \quad \text{env} = \quad \text{fix ( } \Gamma \text{ )}
\]
where \[ \Gamma : (\text{Store } \rightarrow \text{ Store}) \rightarrow \text{ Store } \rightarrow \text{ Store} \]
\[ \Gamma = \lambda x.\lambda \text{st.} \text{the_value_of}(B, \text{env}) = \text{true} \rightarrow \]
\[ x \left( \text{execute } S \text{ env st} \right), \quad \text{st} \]

This example demonstrates a problem with Denotational Semantics definitions. It is written in a style that might be considered as cryptic to a reader unfamiliar with Denotational Semantics. However this style is common. The use of notation such as the fixed point operator (\text{fix}), lambda notation ($\lambda a.b$) and McCarthy conditionals ($a \rightarrow b, c$) all make this very difficult to interpret for a reader unfamiliar with these notations. The reader should not be concerned at this stage with the exact meaning of this example and the interpretation of this notation. An introductory discussion of Denotational Semantics is given in Chapter 5.

2.1.2 Action Semantics

Action Semantics has been developed by Peter Mosses in collaboration with David Watt. The starting point was Denotational Semantics. Mosses attempted to identify those operations that were frequently carried out in defining the semantics of languages. These actions could then be used as standard building blocks, and assembled as required to define a language. This work on Abstract Semantic Algebras
Among the claimed benefits of Action Semantics are:

- comprehensibility
- modularity
- reusability
- compositionality (i.e., phrases are specified in terms of their component parts)
- easy to scale up from small languages to large
- easy to add new constructs to a language
- easy to read, even to the uninitiated
- the action combinators have desirable algebraic properties to allow reasoning about the semantics

Action Semantics has been applied to a number of different languages. An early attempt was carried out by Watt in his semantic definition of the functional language ML [Wat88]. This was written using an earlier version of Action Semantics than that detailed in [Mos92].

Watt has also been responsible, in collaboration with Mosses, for producing a semantic definition of the common imperative language PASCAL [MoW93]. Additionally Mosses illustrates Action Semantics through defining an imperative language known as AD in [Mos92]. AD is in fact a sub language of ADA.

Action Semantics definitions are written in Action Notation. This notation makes use of a number of different actions, each with different purposes and properties. Actions can be split into a number of categories, depending on their purpose. The basic Action Notation concerns itself with flow of control. In addition to the basic Action Notation, there are four different facets of actions for specifying the flow of information. The functional facet specifies the flow of transient (i.e., temporary) information. The declarative facet specifies the flow of scoped information. The intended use of this facet is obviously for specifying the bindings that may occur within a language. This facet could be compared with the introduction of environments in Denotational Semantics. The imperative facet is used to handle stable information. This could be
compared to the introduction of *stores* in Denotational Semantics. Finally the *communicative* facet deals with *permanent* information, and is used for specifying communication between processes. This could be useful, for example, in specifying a language designed to be implemented on a parallel machine. In addition to the four facets, there is also *reflective* Action Notation, which is used for specifying abstractions, and their subsequent application.

There is a rich selection of data types used within the notation that are fully specified under a framework known as *unified algebras* [Mos89, Mos94a]. These data types, because of the reusability feature of Action Notation, are fully available to use within specifications. This therefore means that introducing new data types within languages becomes easier since there is already available a large selection of data types on which to model them.

The motivation for the development of Action Semantics was Peter Mosses’ dissatisfaction with some of the pragmatic aspects of Denotational Semantics. He felt that the problem lay in the direct use of the \( \lambda \)-notation. The semantic equations that are used are directly dependent on the exact structure of the denotations that are used. For example consider again the example presented in section 0. Here the semantic function "execute" requires the current store and produces a new store. If we were to change the language to make statements additionally dependant on an environment, then the semantic equation "execute" would have to be completely rewritten.

Continuing this example, let us examine the Action Semantics of statement composition. Here the statements are composed in order by the "and then" action combinator. A point to note about this is that there is no explicit reference to a store or an environment (these are handled implicitly). This definition would look the same for statements that required reference to an enviroment, and for those that did not.
execute \[ S_1 ; S_2 \] =
execute \( S_1 \) and then execute \( S_2 \).

In order to define a while...do statement each iteration in a loop is *unfolded* by using the "unfold" action primitive, cf. section 0. The start of a loop is marked by the action combinator "unfolding_".

execute \[ \text{while} \; B \; \text{do} \; S \] =
unfolding
\[
\begin{align*}
\text{check the value of } B \; \text{is true and then} \\
\text{execute } S \; \text{and then unfold} \\
\text{or} \\
\text{check the value of } B \; \text{is false and then rebind}.
\end{align*}
\]

In the author's opinion Action Semantics has great potential for language specifications. In addition to the benefits already listed Action Semantics also makes specification of non-determinism easy. However, although it is easy to read Action Notation, it is quite difficult for a beginner to write. There is a large amount of notation to read and understand before useful specifications can be produced. This however could be regarded as a necessary evil in view of the benefits of the notation.

A second problem that has been encountered by the author in preparing this thesis is the fact that in cases where explicit references to environment and storage variables are required, it is necessary to refer to the Operational Semantics of the Action Notation as given in [Mos92, Appendix C]. The Operational Semantics makes formidable reading, and reasoning about specifications using it is made exceptionally difficult and long winded.

2.2 Object-Oriented Languages

An object in an object-oriented language consists of a private data area, and a set of publicly accessible procedures or methods. The set of public methods represents the interface between external objects and the internals of an object. The internal structure
of objects should be invisible to other objects, i.e. they are *black boxes*. Objects in a system interact by sending messages to each other. These messages request that a particular method in an object be executed, and also provide any arguments for that method.

In addition to the above there are also usually the following facilities in a language in order for it to be an object-oriented language:

- **Classes.** A class organises objects into sets that share the same operations. For example the class of Integers would include all the objects that represent individual integers. An integer would for example have operations for addition and subtraction. These methods would be defined within the class.

- **Inheritance.** This mechanism organises object classes into a hierarchy. A new subclass is defined by copying the structure of some existing class (its super class). The new class then defines amendments to the existing classes, or new methods and instance variables.

When a message is sent to an object it must find the appropriate method to execute. One of the most common methods of doing this is by using the standard message "look-up" algorithm. When an object receives a message it examines the message to find the name of the method that is required. The class of the object is then examined to see if a method of that name is defined in the class. If there is such a method then it is executed with the appropriate arguments. If there is no such method then the parent class of that class is examined in a similar fashion. This continues until either a method is found and executed, or until the root of the class tree is reached. If the root is reached then normally an error message is produced.

When a new class is defined, that class may overwrite some of the methods of the class on which it was based. For example consider the following code fragment from some hypothetical object-oriented language.
This code defines two new classes, "shape" and "circle". The class "shape" contains a variable that is private to it called "location", and one method that is publicly accessible method, "set". The class "circle" is based on "shape", but defines one new private variable "radius". It also inherits the private variable "location" from "shape" and redefines the method "set".

In most object-oriented languages there is normally some method to allow access to the method that has been overwritten by a newer version. For example in the above code fragment the method "set" is redefined by the class "circle". However access is still allowed to the original version of "set" via the "super" expression. An object that executes the "super" expression sends a message to itself, requesting that the given method is executed. However the search for the method starts in the parent class of the current class, so that previous versions of the method are accessible.

A reader wishing to study the principles of object-oriented languages further should see [Str88, Wat90].

2.2.1 Smalltalk-80

One particular object-oriented language that will be examined in this thesis is Smalltalk-80 [GoR83]. Smalltalk was developed during the 70's and early 80's at the
Chapter 2: Background and Literature Review

Xerox Palo Alto Research Center. It is a very pure object-oriented language - with all entities within a system modelled as objects. All of the "built-in" types are just objects that are provided in a standard library. The language itself is essentially procedural in nature, although all of the control statements that one might expect in a normal procedural language are absent (e.g. "if...then...else..." etc). Instead these operations are carried out by sending messages to standard objects.

For example consider the following C++ code fragment.

```cpp
if (a == b) then
    y.do_one();
else
    y.do_two();
```

The above gives a good example of a typical "if...then...else..." construct that one would not be surprised to see (except for minor syntactic variations) in almost any procedurally based language. Informally the semantics of this is that the expression "(a == b)" is first evaluated. If the result of this is true then the "y.do_one()" statement is evaluated, otherwise the "y.do_two()" statement is evaluated.

A similar statement in Smalltalk-80 would look like the following.

```smalltalk
(a = b)
  ifTrue: [ y do_one ]
  ifFalse: [ y do_two ]
```

The semantics of this is very different to the C++ version. First the expression "(a == b)" is evaluated. This produces a truth value object as a result. This object is then sent an "ifTrue...ifFalse..." message, with the "[ y do_one ]" block as a first argument and "[ y do_two ]" as the second. If the truth value object is "true" then the first argument block is executed. Otherwise the second argument block is executed.

All objects in Smalltalk are instances of some class. Classes themselves are also modelled as objects. All classes (with the exception of the class "Object"), must
inherit behaviour from some other class. The class from which the behaviour is inherited is called its superclass or parent. A class in Smalltalk can have only one parent class, and so this forms classes into a tree structure. The class "Object" (which has no parent) is at the root of this tree.

Every class object in the Smalltalk system, must itself be an instance of a class. To deal with this Smalltalk has metaclasses. Every class object is an instance of a metaclass. Each metaclass only has one instance. In addition metaclasses are also modelled as objects. All the metaclasses are instances of just one class, called "Metaclass". See Figure 1 for the class structure (based on diagrams in [GoR83]).

In Figure 1, boxes represent classes and circles represent instances of classes. The metaclass of "Object" is "Object class", the metaclass of "ClassDescription" is "ClassDescription class" and so on. Each metaclass box has a corresponding circle in Metaclass. There are two example classes that have been defined. The first is "Example1". The class "Example1" has two instances. The metaclass is "Example1 class". Every metaclass only ever has one instance. The second example class is a subclass of "Example1" and is called "Example2". This has three instances, and its metaclass is called "Example2 class".

There is some circularity in these definitions. For example "Metaclass" is an instance of "Metaclass class", and "Metaclass class" is an instance of "Metaclass". Similarly the class "Object" has a subclass called "Object class". The only instance of "Object class" is the class "Object". A reader interested in metaclasses should see the book [GoR83] for more details.
Figure 1. The class structure in Smalltalk-80

As with the hypothetical language discussed in the previous section, Smalltalk-80 has a super expression which allows access to methods that have been overwritten by newer versions.

2.3 The Semantics of Object-Oriented Languages

This section reviews some of the most relevant research on the semantics of Object-Oriented languages in chronological order.

2.3.1 The work of Wolczko

Wolczko produced two versions of the semantics of Smalltalk-80 in a denotational style, with differing methods of accessing methods [Wol87, Wol88].
2.3.1.1 Look up

In [WoI87] a semantics of Smalltalk is presented that uses the standard "look-up" algorithm to find the correct method to execute after a message has been sent to an object. He defines a program to be a map from class names to class bodies.

\[
\text{Program = Class\_map} \\
\text{Class\_map = map Class\_name to Class\_body}
\]

Objects in the system are stored in an object memory. Each object is assigned a unique ordinary object pointer (Oop). The term Oop (instead of Op for object pointer) is used here for historical reasons (this is explained further in [WoI88]). The object memory is defined as a map from these Oops to the details of the objects.

\[
\text{Object\_memory = map Oop to Object}
\]

In addition to this an environment domain is defined. This environment contains the name of the class of the current object, and the program map.

\[
\text{SEnv :: Class : Class\_name} \\
\text{P : Program}
\]

Wolczko [WoI87] then defines a function called "find" that is used to find the appropriate method to execute. (Note the function "P" is used to return the "Program" element from the "SEnv" argument).

\[
\text{Search\_function = Selector \to [Method]} \\
\text{find : [Class\_name] \times SEnv \to Search\_function} \\
\text{find(class, \rho) sel} \triangleq \\
\text{if class = nil then nil else if sel \in \text{dom Methods}(P(\rho)(\text{class})) then ... else find(Super(P(\rho)(\text{class})), \rho) sel}
\]

This function works by checking to see if the method name (sel) that has been supplied is in the domain of the map from selectors to methods. If it is in the domain then the method has been found and the appropriate action is taken (this has been
elided here). Otherwise the parent class of the class of the current object is then checked, and so on, until the method is found.

2.3.1.2 Copy Down

In [Wol88] an amended technique is used. Here methods are "copied down" at creation into all of the classes that inherit them. This means that when searching for a method, it is only necessary to look at one class.

\[
P_{program} : Program' \rightarrow Program \\
P_{program} [\text{p}] \Delta \\
\{ \text{class} id \mapsto P_{class} [\text{class} id] \text{p} | \text{class} id \in \text{dom p} \} \\
P_{class} : \text{Class name} \rightarrow \text{Class map'} \rightarrow \text{Class body} \\
P_{class} [\text{class} id] \text{class map} \Delta \\
\text{mk_class body}(\text{inst vars}(\text{class} id, \text{class map}), \\
\text{all_methods_of}(\text{class} id, \text{class map}))
\]

Here a program is translated from a version that has inheritance, to one that does not. Every class in the system is converted by the "PProgram" function. This maps "class_ids" to the new version of the Class body. The new version has no inheritance. These new Class bodies are generated by the "PClass" function. The function "inst_vars" checks the current class, and all of its super classes to find all of the instance variables that are required by instances of this class. Similarly the function "all_methods_of" checks the current class and all of its super classes to find all of the methods that are required by this class. By combining the results of these two functions a new Class body is created that does not inherit anything from any other class.

Most semantics of object-oriented languages appear to treat classes as a special case semantic entity. Wolczko outlines how his model could be modified in order to allow classes to be treated as objects [Wol88, pp.58-64], which is closer to the intuitive semantics of Smalltalk.
The main problem with Wolczko's work is that it would be very difficult for a non-expert reader to understand the semantics. The concepts that are involved in object-oriented languages are inherently complicated. However Wolczko's use of a Denotational Semantics style makes understanding the semantics even more difficult.

2.3.2 The work of Kamin

The primary innovation that Kamin [Kam88] introduced was the use of fixed points to define inheritance. Readers unfamiliar with fixed points should refer to section 5.2.5.

\[ H \in \text{Hier} = (\text{ClassName} \rightarrow \text{ClassDef}) \]

In a method inspired by Wolczko, a class hierarchy is defined by Kamin to be a map from class names to class definitions. The syntax of a class definition ("ClassDef") is as given below. The definitions of "ClassName", "ClassVar", "InstVar", "Message" and "Method" are not given here, but are not required in order to understand the general concepts.

\[ \text{ClassDef} = \text{Name Name Var* InstVar* Methods} \]
\[ F \in \text{Methods} = (\text{Message} \rightarrow \text{Method + {no-def}}) \]

A set of methods within a class is represented as a mapping from messages to the individual representations of methods. Kamin also defines an environment as follows.

\[ \rho \in \text{Env} = \text{Name} \times \text{Message} \rightarrow \text{MethodVal} \]

These definitions lead to the definition of inheritance given below. Note the use of lambda notation. This is used as follows:

\[ \lambda x. x + 3 \]

This defines a function, that takes an argument "x" and produces a result "x+3". Note also that Kamin uses the least fixed point operator \( Y \), which is referred to elsewhere within this thesis as \( \text{fix} \).
Chapter 2: Background and Literature Review

<table>
<thead>
<tr>
<th>D : Hier → Env</th>
</tr>
</thead>
<tbody>
<tr>
<td>C : Hier → Env → Env</td>
</tr>
</tbody>
</table>

\[
D \[ H \] = Y(C \[ H \]) \\
\]
\[
C \[ H \] \rho = \\
\lambda <c, m>. \\
\]
\[
\text{let } H(c) = C \, S \, w \rightarrow x \rightarrow F \\text{ in if } F(m) = \text{no-def then } \rho <S, m> \text{ else } M \[ F(m) \] \rho \\
\]
\[
where \rightarrow \text{ represents a list} \\
\]

The semantic function "D" is defined as the least fixed point of the function "C". The function "C" takes as arguments the class hierarchy and an environment. It produces an environment as a result, which is a function from a class name and message pair to a method value. The class name and message pair is represented as "<c, m>". Using the supplied class hierarchy "H" and class name "c", a class definition is found, "C \, S \, w \rightarrow x \rightarrow F". The method definition for the given message is found and the method value is given by the semantic function, "M" (undefined here). If the given message is not found for the current class definition then the super class is checked. This is achieved by looking up the method value for the super class and given message in the environment provided to "C". When "C" is fixed this amounts to recursion.

This method of finding the appropriate method value for a given message is similar to the "copy down" approach that was used by Wolczko. The work of searching the class tree for inherited methods is carried out in one go at the creation of the environment. This means that the algorithm for finding methods is static as opposed to dynamic.

Classes in Kamin's semantics, are not treated as normal objects, but as special semantic entities. The "new" message which is normally sent to class objects is treated as a special expression instead of as a normal message. Kamin states his motivation for doing this as follows.

"This is to avoid having to introduce class objects, which would complicate the definition".
We feel that treating classes as a special case does not simplify the understanding of the model. It may in fact have the opposite effect and add confusion. It would not be too difficult to integrate classes as normal objects.

2.3.3 The work of Yelland

Yelland [Ye189] places heavy emphasis on the value of *fully abstract* semantics. Yelland argues that the denotations of objects should only give enough detail to explain their *externally observable behaviour*. This effectively treats objects as *black boxes*. The view is taken that an "observer" is someone who is writing software that interacts with an existing system of objects. It is this existing system that is being observed. In order to define what is meant by an observation, Yelland make the following definition.

"To observe a system from a new object, load all of the variables of that object with a set of values, execute some sequence of statements, and then take a note of the resulting contents of the variables."

A formal version of this definition is also given.

A fully abstract semantics of an example language is derived from a Denotational Semantics of that language. Concepts from universal algebra are used in order to achieve this. A second fully abstract semantics is also given which is considered to be more "natural". This is developed by constructing state-transition graphs for systems.

It is interesting to compare the semantics developed by Yelland, with the Action Semantics developed for object-oriented languages. Action Semantics puts the emphasis on making definitions readable and modular (and hence easily extensible). This approach introduces what Yelland would regard as extraneous detail. Yelland takes the opposite approach and sacrifices all attempts for semantics definitions to be readable or extensible in order to produce fully abstract semantics.
2.3.4 The work of Cook and Palsberg

Cook and Palsberg [CoP94] consider two different methods of defining inheritance. They compare the traditional "look up" algorithm with a new method that they propose based on "generators" and "wrappers". The framework that they use is Denotational Semantics.

A generator is a function, the least fixed point of which represents an object. They model objects as record values (where the notation \{ \ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n \} is a record that associates the value \( v_i \) with the label \( \ell_i \)). They give the following example of a generator that is associated with a class, "Point".

\[
\text{MakeGenPoint}(a, b) = \lambda \text{self.} \\
\{ \\
\quad x \mapsto a, \\
\quad y \mapsto b, \\
\quad \text{distFromOrig} \mapsto \sqrt{\text{self}.x^2 + \text{self}.y^2}, \\
\quad \text{closerToOrg} \mapsto \lambda p.(\text{self}.\text{distFromOrig} < p.\text{distFromOrig}) \\
\}
\]

\[p = \text{fix}(\text{MakeGenPoint}(3, 4)) = \{ \\
\quad x \mapsto 3, \\
\quad y \mapsto 4, \\
\quad \text{distFromOrig} \mapsto 5, \\
\quad \text{closerToOrg} \mapsto \lambda p.(5 < p.\text{distFromOrig})
\]

In this example instances of the class "Point" have four members: "x", "y", "distFromOrig" and "closerToOrg". The construction of a new "Point" object requires two parameters: "a" and "b". Applying the generator function to these parameters gives a function that takes some argument "self", and produces a record as a result. By taking the fixed point of this function, the reference to "self" can be removed, leaving just a record. The example shows the application of the generator for "Point" to the two parameters 3 and 4. The result is fixed to produce a record where there is no references to "self".
A wrapper is a function which is used to define the differences between a class and a sub-class. Continuing the previous example they define a wrapper for a class "Circle" which is a sub-class of "Point".

```java
CircleWrapper(a, b, r) = λself.λsuper.
    { radius 🔄 r,
      distFromOrig 🔄 max(super.distFromOrig - self.radius, 0)
    }
```

A generator for the new class, "Circle", is then created as shown.

```java
MakeGenCircle(a, b, r) = λself.
    (CircleWrapper(a, b, r)(self)(MakeGenPoint(a, b)(self)) ⊕
     MakeGenPoint(a, b)(self))

where
    M ⊕ O
is an operation where any method defined in M replaces the corresponding method in O
```

Instances of the class "Circle" have 5 members: "x", "y", "distFromOrig", "closerToOrg" and "radius". The definitions of "x", "y" and "closerToOrg" are inherited from the super class "Point". The definition of "distFromOrig" overrides the version in "Point". The wrapper function takes three parameters: "a", "b" and "r". Passing the arguments "self" and "super" to the result of the wrapper function produces a record.

The generator function for the "Circle" class is then defined by combining the wrapper function for "Circle" with the generator for the super class "Point".

The idea behind this approach is to capture the concept of differential or incremental programming. A new class is defined in terms of the differences between itself and its parent class. The wrappers define the differences, whilst the generators define the complete classes. Cook and Palsberg state that they believe that their semantics is a more intuitive explanation of inheritance than the standard method "look up" algorithm. We believe that because the model as it stands requires a knowledge of
fixed points this makes understanding the model more difficult. The model will only be accessible to those few experts who understand fixed points. This point is acknowledged by the authors themselves.

"It may even be argued that it [the new model] is a great deal more complex, because it requires an understanding of fixed points."

2.3.5 The work of Golubski and Lippe

Golubski and Lippe [GoL95] present the most recent work on the semantics of Smalltalk-80. The method used is to syntactically transform programs into a form that eliminates all message sending. Messages are instead simulated by other expressions. The authors call this the "copy rule" semantics. The formal basis for the syntactic transformations is unclear.

Some extra expressions are added to the language to enable message sends to be simulated. These include a "block", a "trap" and a "returnjump". The "block" expression works in a similar manner to blocks in other languages, and enables the use of temporary variables that are local to that block. The "trap" expression executes its body until either the execution completes successfully, or a "returnjump" expression is encountered. If a "returnjump" is found then the trap executes a second exception expression.

Additionally two types of new variable are added. The first of these types enable direct access to the instance variables of objects. The second type is used in resolving messages to "super".

Methods and instance variables that are inherited by classes are "copied down" into those classes statically at creation, instead of dynamically, using a "look up" algorithm. However those methods that are overloaded are not copied down. This means that calls to the "super" expression have to be resolved dynamically, and this presents some problems. The method given by Golubski and Lippe in [GoL95] for
resolving "super" expressions is erroneous due to this. Additionally the exact method by which the "copying down" occurs is not given. This is an unfortunate omission.

It is our opinion that this method of defining the semantics of object-oriented languages is insufficiently motivated. It does not appear to add any advantage over methods proposed in previous papers, and is counter intuitive. The semantics definitions produced are very hard to read and would not be accessible to a non expert.

2.3.6 The work of Palma et al.

The work by Palma et al. [PMM95] is important in relation to the research presented in this thesis because it gives an Action Semantics of a simple object-oriented language. The emphasis of the work is very much on improving the readability of the semantics.

The language that they have chosen to define is called POOL (Parallel Object-Oriented Language). POOL is a very simple language and does not exhibit the property of inheritance that one would expect to see in most object-oriented languages. It does however have the concept of encapsulation.

In a sequential Object-oriented language (such as Smalltalk-80) there is only ever one object which is active (i.e. executing) at any one time. However because POOL is a parallel language it enables all objects in the system to be active at the same time. In order for two objects to send a message between each other, they must first rendezvous. The receiving object must enter a state that enables it to receive a message, and the sending object must enter a state that enables it to send a message. Only when they are both in these respective states can the message send occur.

Objects are modelled within the semantics by the Action Semantics concept of an agent. An agent can be thought of as similar to a "process". Agents in Action
Semantics can send messages between themselves, and this idea is used to model message sending between objects in POOL.

The semantics of a message send is given as follows.

\[
\text{evaluate } \left( E_1: \text{Expression} \right) \text{!} \ M: \text{Meth-ID} \ \left( E_2: \text{Expression} \right) \ \text{=} \ \\
\text{evaluate } E_1 \ \text{and evaluate } E_2 \ \\
\text{then} \ \\
\text{send a} \ \\
\text{message[to the given agent\#1][containing msg(M, the given entity\#2)]} \ \\
\text{and then receive a message[from the given agent\#1]} \ \\
\text{then give contents of the given message.}
\]

First the two sub-expressions "E1" and "E2" are both evaluated. The expression "E1" should evaluate to an agent to which the message should be sent, and the expression "E2" to an entity that will be used as an argument to the method. The message is then sent and a result is awaited by the "receive" action. The result of the whole expression is the contents of the message that was sent back from the receiving object.

The semantics given by Palma et al. is interesting, but unfortunately does not give us any insight into how to handle inheritance and the various complexities connected with it. The work does however suggest one possible way in which encapsulation can be modelled, i.e. by the use of agents. Most traditional models of encapsulation would use cells in a store to model objects. The semantics that is presented is very readable, when compared to the other semantics definitions that have been discussed here. This suggests that Action Semantics may in general produce more understandable and accessible semantics definitions.

### 2.3.7 The work of Watt

Watt [Wat97] defines an Action Semantics of a subset (known as JOOS) of the object-oriented language Java.
In Watts semantics of JOOS, classes are represented as records. Each class record contains four elements: a mapping from tokens to types (to represent the fields declared by that class); a mapping from tokens to method abstractions (to represent the methods defined by that class); an abstraction (representing the constructor of the class); and a reference to the super class of the class (optional: not present if this is the root class).

Objects are also represented as records in Watts semantics. An object record consists of: a reference to the class that defined this object; bindings from tokens to variables (cells) to store the values of the instance variables; and an identity. The identity uniquely identifies the object. An object \( o_1 \) is the same object as \( o_2 \) if and only if they have the same identity.

Methods are discovered dynamically when they are called.

The following fragment of Watts semantics shows the definition of an expression to send a message to an object.

\[
\text{evaluate} \left[ \text{E:Expression} \ "\" \ \text{I:Identifier}\ "\(" \text{A:Arguments} \ "\"\)\] = \\
\text{evaluate E and}
\text{respectively evaluate A}
\text{then}
\text{enact the application of the method I of the class of the given object to the given (object, value')}
\text{or}
\text{check the given reference is null then}
\text{escape with the null-reference-exception .}
\]

This expression is used to send a method to an object. The expression consists of a sub-expression \( E \) to identify the object to which the message is to be sent; an identifier to identify the message name; and a list of arguments to be evaluated. The expression identifying the object can evaluate to "null", in which case an error condition is indicated. Otherwise the method \( I \) is searched for in the class of the object that the expression evaluated to.
The method to be used is determined through the following definitions.

- method-bindings = map[token to method].

1. \( c = \text{class of } (t:\text{type-bindings}, m:\text{method-bindings}, k:\text{constructor}) \Rightarrow \) method-bindings \( c = m \).
2. \( c = \text{class of } (t:\text{type-bindings}, m:\text{method-bindings}, k:\text{constructor}, c':\text{class}) \Rightarrow \) method-bindings \( c = \text{overlay}(m, \text{method-bindings } c') \).

- method _ of _ :: token, class \( \rightarrow \) method (partial).
3. method \( t:\text{token} \) of \( c:\text{class} \) = method-bindings \( c \) at \( t \).

The method bindings for any particular class are made up by overlaying the methods defined by the class over the methods bindings defined by the super class. It is then a simple matter to discover the appropriate method for any given token.

JOOS has a "super" expression to access methods defined in super classes of the current class. The semantics of this is defined by Watt as follows.

\[
\text{evaluate } [[] \text{"super" "." } I:\text{Identifier }\(" A:\text{Arguments }\)" ] ] = \\
\text{respectively evaluate } A \text{ then} \\
\text{enact the application of} \\
\text{the method } I \text{ of the superclass of the class of} \\
\text{the object bound to "this"} \\
to (\text{the object bound to "this", the given value}).
\]

The definition of this semantics is very similar to the definition of normal message sending. The main difference is that the search for the method skips the class of the current object, and starts in the super class of that class.
Chapter 3: The Action Semantics of an Object-Oriented Language

3.1 A Introduction to Action Semantics

This section briefly introduces the concepts of Action Semantics, and may be skipped by a reader already familiar with this area. For a more complete introduction the reader is referred to [Wat91 (pp. 184-268)] or [Mos94b].

3.1.1 Actions

An "action" could be described as a computational unit that is to be performed. A computation will take some form of data as input and have an outcome. Some of the allowable outcomes that might occur as a result of a performing an action would be: completing, the action finishes normally; failing, the action finishes abnormally; diverging, the action does not finish at all or escaping, the action finishes abnormally after some exceptional condition.

Action Notation defines different forms of data that can be used by "actions". These are the different facets of Action Notation. Some of the facets defined by Mosses include:

The functional facet. This facet involves the flow of transient data. Transient data received as an input to an action must be used by that action immediately, or it is lost.
Normally transient data will be produced as the result of performing one action, and then passed as an input to the next action.

The *declarative* facet. This facet describes the flow of scoped data. A common concept in computer languages is the idea of bindings that have a scope. Once the flow of control falls outside of the scope the bindings are no longer available. This is the same for Action Notation. Data in this facet has a scope, and is available to all actions that fall within that scope. The data is unavailable to actions outside the scope.

The *imperative* facet. This facet describes the flow of storable data. All computers have some form of store associated with them that has the capability of "remembering" data. A store has a number of locations or *cells*. Each cell in the store can be either defined, and has a given value, or it is undefined. The state of a store is only changed as a result of an explicit computation upon that store. Action Notation also has the concept of a store. All actions have access to the data within the store. Actions that operate on the data within the store operate in the imperative facet.

The Action Notation defined by Mosses in [Mos92] specifies two types of actions: *primitive* and *compound*. A primitive action is a simple action that operates within one of the facets. An action combinator can take one or more actions as arguments to produce a new one. In this way much more complicated actions can be built up from putting together the primitive actions and the action combinators.

### 3.1.2 Basic

Before examining some of the actions that operate on the different types of data within the various facets, it is first necessary to introduce some basic actions and action combinators.
3.1.2.1 Actions

As discussed above there are a number of different possible outcomes from performing an action. An action may complete normally; it may complete abnormally (e.g. maybe having encountered some recoverable error); it may fail; or it may diverge (i.e. it never completes, for example in the case of an infinite loop). For each outcome that an action may have there is an associated primitive-action. For example there is a complete action that simply completes normally. There is also an escape action that completes abnormally. This is all that these actions do - they have no other function. In themselves they are not particularly useful, but when combined with other actions and action combinators they can be very useful.

- complete : primitive-action
- escape : primitive-action
- fail : primitive-action
- diverge : primitive-action

3.1.2.2 Action Combinators

Different actions can be combined together to produce compound actions using action combinators.

First we will examine the action combinator or. The following is the signature for or. The signature tells us what the combinator or looks like, i.e. what arguments it takes, and what kind of a result it produces. This signature also tells us some additional information about the properties of or, i.e. that it is total, associative, commutative and idempotent; the identity (or unit) is the action fail.

\[
\text{or} :: \text{action, action} \rightarrow \text{action} \quad (\text{total, associative, commutative, idempotent, unit is fail})
\]

The or action combinator can be used to represent non-deterministic choice. It takes two action arguments to produce a single compound action. The arguments themselves may of course also be compound actions. When this action is performed
one of its action arguments is non-deterministically selected and performed. The outcome of the whole action is the outcome of the action that was selected. If the selected action fails then the other action is performed instead. This makes the primitive action fail the unit for this action combinator.

For example consider a situation where you wish to non-deterministically choose between executing a statement "A", and executing a statement "B". You have already defined an action "execute" to model the execution of statements. This choice would be written as follows:

\[
\begin{align*}
|& \text{execute A} \\
\text{or} & \\
| & \text{execute B}.
\end{align*}
\]

Note the use of vertical lines to indicate grouping. This is the same as:

\[
(\text{execute A}) \text{ or } (\text{execute B}).
\]

This notation is common in Action Semantics Definitions.

The or combinator can be particularly useful when used in conjunction with the check primitive action (described below). Used with check, or can be used to model deterministic choice.

- \[ \text{and} :: \text{action, action} \rightarrow \text{action} \text{ (total, associative, unit is complete)} \]

The and combinator can be used to model an implementation dependent ordering of performance of the two sub-actions. Both of the argument actions are performed, but one could be entirely performed before the other, or some interleaving of performance could take place. If the sub-actions were to produce some data as a result of their performance, and combines these two data items into a pair.

More properties of and are discussed in section 3.1.3.3.


• \texttt{\_ and then\_} :: action, action \rightarrow action (\textit{total, associative, unit} is complete).

This combinator is used to model sequential ordering of performance. The first action argument is completely performed before the second action argument is performed.

More properties of \texttt{and then are discussed in section 3.1.3.3}.

• \texttt{\_ trap\_} :: action, action \rightarrow action (\textit{total, associative, unit} is escape).

This combinator is used where it might be expected that an action will escape. The first action argument is performed. The outcome of the entire action, is the outcome of performing the first action (if it does not escape). The second action is only ever performed if the first escapes. If this occurs then the outcome of the entire action is the outcome of performing the second action. For example consider a situation where you have an action called "execute". Normally this will just complete. In some situations an error may occur and the action will escape. If this occurs then a second action "report error" should be performed. This would be written as follows:

\[
\begin{align*}
| & \text{execute} \\
& \text{trap} \\
& \text{report error}. \\
\end{align*}
\]

It is possible for an action to escape and give some transient data to the second sub-action. See \texttt{escape with} below.

\section*{3.1.3 The Functional Facet}

\subsection*{3.1.3.1 Yielders}

A yielder is an entity that can be evaluated to produce some form of data. Ordinary data is a special case of a yielder which simply evaluates to itself.

A commonly used yielder is given below:

• \texttt{given\_} :: data \rightarrow yielder (\textit{strict}).
This yielder evaluates to the data that is provided to it as transient data, provided that that data is a subsort of its argument. For example:

the given truth-value

This will evaluate to true if the incoming transient data is true, or false if the incoming transient data is false. If the incoming transient data is not a subsort of truth-value then this will evaluate to nothing (where nothing is a vacuous sort that never contains any individuals). Note the function the used here is an identity function. It is optional but can often help to improve the readability of Action Semantics definitions.

The word "strict" in the signature of the "given" function implies that if the data argument evaluates to nothing then the value of the whole function is nothing.

3.1.3.2 Actions

• give_ :: yielder → primitive-action .

The primitive action give takes as an argument a yielder.

The give action simply produces as transient data the data obtained by evaluating its yielder argument. For example:

give true.

This simple action will complete normally and produce as transient data the value true. It does not use any incoming transient, scoped or storable data. It does not produce any scoped data as an output, nor does it change the store in any way.

• regive : primitive-action .

This is a very simple action that produces a transient data outcome that is a copy of its transient data input.
• escape with _ :: yielder → primitive-action.

This is another primitive action that takes a single yielder as an argument. It is very similar to give, except that it does not finish normally, but instead escapes. An action that has an *escaping* outcome will propagate this outcome to all compound actions that have this action as a component. An action that *escapes* can have this propagation *trapped* by the use of a **trap** action combinator. Once an *escaping* action has been trapped it can produce transient data. The transient data is that data given by the argument of **escape with**. For example:

```
| escape with true
| trap
| regive .
```

This compound action will give the transient value **true** as a result of performing it. The **escape with** action is first performed, which leads to an *escaping* outcome. This is trapped via the **trap** combinator. The transient data passed to the **regive** action is the data that was given by the **escape with** action (i.e. **true**). Therefore the whole action gives **true** as a result.

• check _ :: yielder → primitive-action.

This action takes as an argument a yielder that evaluates to a **truth-value**. If the yielder evaluates to **true** then this action completes normally. If the yielder does not evaluate to true then this action fails. For example:

```
check the given integer is 1.
```

This action would examine the value of the incoming transient data. If this evaluates to 1 then the action completes normally. Otherwise the action fails.

This is particularly useful when used in conjunction with the **or** action combinator. Deterministic choice can be modelled using a mixture of the two. For example:
check the given truth-value is true and then give 1
or
check the given truth-value is false and then give 2.

In this example the entire action expects a truth-value to be given to it (otherwise it fails). If the truth-value is true then the result of the entire action is to give 1. If the truth-value is false then the result of the entire action is to give 2.

The or combinator gives a non-deterministic choice. However if the "wrong" branch is taken then the check action will fail, and the other branch will then be performed. Since there will only ever be one branch that does not fail, this gives us a deterministic choice.

3.1.3.3 Action Combinators

- _ then _ :: action, action \rightarrow action (total, associative, unit is regive).

In order to give the transient data produced from performing one action into another action the then combinator is used. The two argument actions are performed sequentially with the data produced from one being used by the second. The transient data produced as a result of performing the second sub action is the transient data given as a result of the whole compound action. Although this is sequential performance it is different from the and then combinator because and then does not pass the data from one action to the next. The and then combinator will simply combine into a tuple the data produced from its two sub-actions and give that as a result. The and combinator also has this effect of combining its two sub-action results into a tuple.

3.1.3.4 Example

In order to define the semantics of an if...then...else language construct this might be done as follows:
execute [ "if" E:Expression "then" S_1:Statement "else" S_2:Statement ] =
    evaluate E
    then
        check the given truth-value is true and then execute S_1
    or
        check the given truth-value is false and then execute S_2.

This example gives part of the definition of an *execute* semantic function, which is used for executing the statements of a particular language. It is assumed that a suitable definition of *evaluate* exists for evaluating expressions. Note the use of the emphatic brackets to denote a syntactic structure.

The action combinator *then* is used here to sequentialy order the performance of this action. First the sub action *evaluate* E is carried out. It is assumed that the result of this evaluation will be a *truth-value*. If it is not then the entire action will fail (because both branches of the *or* will fail). Once E has been evaluated, one of the branches of the *or* is chosen non-deterministically. However since the two branches are guarded by a *check* action, this amounts to a deterministic choice. If the result of evaluating E was *true* then the first branch will be taken. If it was *false* then the second branch will be taken. Depending on the branch selected either S_1 or S_2 will then be executed.

### 3.1.4 The Declarative Facet

#### 3.1.4.1 Yielders

- the _ bound to _ :: bindable, yielder \(\rightarrow\) yielder.

This yielder can be used to examine the scoped information currently available. Scoped information is represented in the form of a mapping from identifiers to data. The data is said to be "bound to" the identifiers. For example:

the truth-value bound to X
This yielder will examine the current scoped information to find the data that is associated with the identifier X. If there is no data associated with X or it is not a sub-sort of truth-value then this will evaluate to nothing.

### 3.1.4.2 Actions

- **bind _ to _ :: yielder, yielder → primitive-action**.

  This action is used to produce new scoped information. It takes two arguments: a token and a data value. The result is scoped information that has the given data bound to the given token.

- **produce _ :: yielder → primitive-action**.

  The produce action takes a yielder as an argument. This yielder should evaluate to a mapping of tokens to data. This action produces this mapping as scoped information.

- **rebind : primitive-action**.

  In a similar manner to regive in the functional facet, rebind simply reproduces the scoped information that is passed to it.

### 3.1.4.3 Action Combinators

- **_ hence _ :: action, action → primitive-action (total, associative, unit is rebind)**.

  This combinator is very similar to the _ then _ combinator of the functional facet. The two argument actions are performed in sequence. The scoped information passed as a result of performing the first action is passed to the second. For example:

```
| bind tok to true  
| hence            
| | give the truth-value bound to tok  
| | and                   
| | give not the truth-value bound to tok.  
```
This very simple action will cause the transient data (true, false) to be given as a result of performing it. The value true is first bound to the token tok. The action combinator hence then passes this scoped information to the next sub-action. This sub-action is itself a compound action. Note that the and combinator passes the scoped information it was passed to both of its sub-actions, so that both can see that true is bound to tok.

In a similar way to and, the hence combinator combines the transient data produced by its sub actions into a tuple.

• moreover :: action, action → primitive-action (total, associative, unit is complete).

The moreover combinator can be used to combine two input scoped information mappings provided by its sub-actions. For example:

\[
\begin{align*}
| & \text{produce bindings1} \\
| & \text{moreover} \\
| & \text{produce bindings2} .
\end{align*}
\]

The two sub-actions both produce scoped information mappings. This action then produces as a result a single mapping that is a combination of the two input mappings. For every token that has a data item bound to it in each of the two input mappings, there is an equivalent binding in the output mapping. In the case of any conflict (i.e. the same token has a value bound to it in both bindings1 and bindings2) then the bindings from the second sub-action are used in preference. For example:

\[
\begin{align*}
| & \text{bind tok1 to false} \\
| & \text{and} \\
| & \text{bind tok2 to false} \\
| & \text{moreover} \\
| & \text{bind tok1 to true} .
\end{align*}
\]

This action will produce the scoped information where true is bound to tok1, and false is bound to tok2. Note that and has a similar effect on scoped bindings to moreover in that it combines the two. However and will fail if there is any conflict.

37
Chapter 3: The Action Semantics of an Object-Oriented Language

• **furthermore** :: action → action \((\text{total})\).

`furthermore` is used to add new bindings to the current scoped information. For example:

```
bind tok1 to false
hence
furthermore bind tok2 to false.
```

The first sub-action of `hence` produces scoped information where `false` is bound to `tok1`. The `furthermore` action, takes this information and adds to it the binding of `tok2` to `false`.

• **before** :: action, action → action \((\text{total, associative, unit is complete})\).

This action is similar to `moreover`, except that it has an accumulative effect. Bindings produced by the first sub-action are overlaid on the bindings provided to the entire action and are passed to the second sub-action. For example:

```
bind tok1 to false
before
bind tok2 to the truth-value bound to tok1.
```

This compound action produces bindings where `tok1` has `false` bound to it, and `tok2` also has `false` bound to it. The bindings produced by the first sub-action are available within the second sub-action. An equivalent form of this example could in fact be constructed using the `hence` and `furthermore` action combinators:

```
bind tok1 to false
hence
furthermore bind tok2 to the truth-value bound to tok1.
```
3.1.5 The Imperative Facet

3.1.5.1 Yielders

• the _ stored in _ :: storable, yielder → yielder.

Unlike the transient and scoped information, the imperative facet deals with stable information that remains static and available until it is explicitly changed. This yielder is used to examine the contents of a given cell in the store. It evaluates to the contents of that cell. For example:

the truth-value stored in a.

This will examine the cell a and obtain its contents. If the content is a truth-value then this yielder will evaluate to that truth-value, otherwise it will evaluate to nothing.

3.1.5.2 Actions

• store _ in _ :: yielder, yielder → primitive-action.

This primitive action is used to change the current store. Its two yielder arguments evaluate to a storable value, and a cell in the store. After this action is performed the cell in the store is updated to hold the given value so that subsequent inspection by the yielder the _ stored in _ will evaluate to that value. For example:

\[
\text{store true in a} \\
\text{and then} \\
\text{give the truth-value stored in a.}
\]

This action stores the truth-value, true in the cell a. Next that cell is inspected to give the truth-value stored in it. Therefore this whole compound action, gives as a result of performing it the truth-value, true and also stores that value in a cell a.
3.1.6 An Example

The second half of this chapter will examine various methods of modelling the semantics of a "toy" object-oriented language known as EIL (Example Inheritance Language). Parts of the definition of this language are relatively straightforward, and we will now examine these sections as an example of the use of Action Semantics. The more complicated aspects of the definition of this language will be examined later in this chapter.

EIL is a very simple language. It does however have objects, instance variables, classes and inheritance like most object-oriented languages. It has expressions for referring to the current object; examining the contents of instance variables; referring to names classes (which are treated in the same way as objects); instance variable assignment; method calling; and a "super" expression (similar to the "super" expression found in the Smalltalk language).

The syntax of EIL is shown below. Mosses in [Mos92] shows how abstract syntaxes of languages should be written in Action Notation. The syntax of EIL has been written in this style. The keywords such as "needs", "closed" and "grammar" can be ignored by the reader at this stage. The main part of the syntax itself is written in a BNF like style.
Abstract Syntax

needs: [Mos92]/Data Notation/Characters/ASCII (letter, digit)
closed.
grammar:

Identifiers

• User-Object-Identifier = □.
• Object-Identifier = User-Object-Identifier | "object" (disjoint).
• Variable-Identifier = □.
• Method-Identifier = □.

Expressions

• Expression = [["self"] | [Variable-Identifier] | [Object-Identifier] |

Classes

• Class = [["class" Object-Identifier ClassBody]] .
• ClassBody = [["inherits" Object-Identifier InstVar* Method+]] .
• InstVar = [["instance" Variable-Identifier]] .
• Method = [["method" Method-Identifier "is" Expression+]] .

3.1.6.1 Expressions

The value of an expression in this language is modelled by the transient information given as a result of performing the evaluate semantic function on that expression.

(1) evaluate < E₁:Expression E₂:Expression+ > =
    | evaluate E₁
    then
    | evaluate E₂.

The definition of a method in EIL is a sequence of expressions. The return value of the method is the value of the last evaluated expression. This sequential evaluation is modelled using the action combinator then. Using this combinator means that the
Chapter 3: The Action Semantics of an Object-Oriented Language

overall value of evaluating a sequence of expressions is the same as the value obtained from evaluating the last expression.

(2) evaluate \([ "self" ]\) =

give the object bound to self.

The EIL language is object-oriented. As in many such languages there is an expression which simply evaluates to the current object. In our definition of EIL we use an internal token self which has bound to it in the current scoped information the value of the current object. This expression examines the scoped information to obtain the value of this object, and then gives it as transient information. This transient information represents the result of evaluating this expression.

(3) evaluate \([ \text{VarNam}:\text{Variable-Identifier} ]\) =

give the object stored in the cell bound to VarNam.

This expression is used to find the value of a variable. Variable identifiers are represented by the syntactic entity Variable-Identifier. The definition of a Variable-Identifier is left unspecified in the definition of the syntax of EIL (the box character \(\Box\), can be used in Action Notation to indicate something that is to be filled in later). The syntactic definition of Variable-Identifiers is not however required in order to be able to understand and define the semantics of expressions.

Variables in EIL, as in most languages have a scope. They are modelled using scoped information. Since these variables are the instance variables of objects, the values of these variables have a life beyond the current scope, and are stable. The values of variables are stored within cells of a store.

This expression gives as transient information the value of a specified variable. In order to find the value of a specific variable, it is necessary to first find the cell within which the value is stored. The cell is bound to the token for the name of the variable that we are interested in. The yielder, "the cell bound to VarNam", gives us the cell where the value is stored. All values in this language are objects, and therefore the value of the entire expression is given as the object stored in the cell.
Chapter 3: The Action Semantics of an Object-Oriented Language

(4) evaluate $\llbracket \text{ObId:Object-Identifier} \rrbracket =$

give the object bound to ObId.

In addition to instance variables, it is also possible in this language to reference other objects. This language models all classes as objects, and therefore it is possible to use a class name at any point where an object would be expected. Class names are bound directly to the objects that model them in this semantics. Therefore this expression simply gives as transient information the object that is bound to the given object name.

(5) evaluate $\llbracket \text{VarNam:Variable-Identifier} \text{"is" E:Expression} \rrbracket =$

evaluate E
then
store the given object in the cell bound to VarNam
and
regive.

This is an assignment expression. Assignment occurs by changing the contents of a cell in the store. The new value for the variable is calculated via a recursive call to the semantic function evaluate on the sub-expression E. Once the new value for the variable has been calculated it is passed as transient information to the remainder of the action for this expression via the then combinator. This value is stored in the cell that is bound to the selected variable in the current scope.

The value of the entire expression is the value of E. The primitive action regive is used to pass on this transient information. This can occur concurrently with the store operation, and therefore the and combinator is used.

(6) evaluate $\llbracket \text{E}_1:Expression \text{Sel:Method-Identifier E}_2:Expression \rrbracket =$

evaluate $\text{E}_1$
and then
evaluate $\text{E}_2$
then
call the method Sel with argument the given object#2 at the given object#1.

This expression represents method calls in our object-oriented language. There are two sub-expressions, $\text{E}_1$ and $\text{E}_2$, which evaluate to the target object for the method
call, and an argument (object) for that call respectively. The evaluation of these expressions must occur sequentially (since they may have side-effects). This is done by using the \textbf{and then} action combinator. Each expression should evaluate to an object and therefore the combined result of evaluating these two expressions is an object pair, that is passed as transient information to the final part of this action via \textbf{then}. The method call itself is carried out by an action (unspecified here) called \textbf{call the method \_ with argument \_ at \_}. Informally this action is intended to represent the semantics of calling a named method with a given argument. The named method is defined at the specified object. Note that the individual components of the object pair are accessed using the notation \textbf{the given object#1}, for the first object, and \textbf{the given object#2} for the second object.

### 3.1.6.2 Elaborating Variables

Classes within EIL specify the methods and instance variables that instances of those classes will have. The methods may reference the instance variables. When a new object instance is created, the instance variables for that object must also be created, or \textit{elaborated}. Instance variables are elaborated in this semantics using the \textbf{elaborate} semantic function.

\begin{enumerate}
\item \texttt{elaborate < I_1:InstVar I_2:InstVar^+ > =}
\item \texttt{\hspace{2em} \texttt{\vert \text{elaborate I_1}}}
\item \texttt{\hspace{2em} \texttt{\vert \text{and}}}
\item \texttt{\hspace{2em} \texttt{\vert \text{elaborate I_2}}}.
\end{enumerate}

A list of instance variables can be elaborated concurrently. This is modelled by the \textbf{and} action combinator. Note the recursive call to the \textbf{elaborate} semantic function.

\begin{enumerate}
\item \texttt{elaborate [ "instance" id:Variable-Identifier ] =}
\item \texttt{\hspace{2em} \texttt{\vert \text{allocate a cell}}}
\item \texttt{\hspace{2em} \texttt{\vert \text{then}}}
\item \texttt{\hspace{2em} \texttt{\vert \text{bind id to the given cell}}}.
\end{enumerate}

An individual instance variable is elaborated by first allocating a cell where the value for the variable will be stored. Next the token for the variable is bound to that cell to produce scoped information.
3.1.7 Creating Classes

Classes in this language are created using the semantic function create. The definition of the semantics of creating individual classes is dependent on the model used. However we can show here how a list of classes can be created.

(1) create \(< C_1: \text{Class} \ C_2: \text{Class}^+ > =
\begin{align*}
&\text{create } C_1 \\
&\text{before} \\
&\text{create } C_2 .
\end{align*}

The first stage of creating a list of classes is to create the first class in that list. The binding produced by this sub-action will be in the form of a class name bound to an object (remember that classes are modelled as objects in this language). The bindings produced by this need to be available to the action that creates the remaining classes (in case those classes reference the earlier classes), and therefore the before combinator has been used.

3.1.8 Other Facets

There are a few other facets to Action Semantics, two of which will now be discussed.

3.1.8.1 Reflective Facet

The reflective facet deals with the creation of abstraction from actions, and the subsequent reflection of those abstractions back into actions.

3.1.8.1.1 Data

\begin{itemize}
\item abstraction \leq \text{datum} .
\item abstraction of \_ :: \text{action} \rightarrow \text{abstraction} (\text{total}).
\end{itemize}

An abstraction is a sub-sort of the datum sort. An abstraction can be constructed using the abstraction of function. This takes as an argument an action, and produces an abstraction as a result. Since this abstraction is a sub-sort of datum it can be passed
around in the same way as any other datum (e.g. it can be given as transient data, or stored in a cell).

3.1.8.1.2 Yielders

- **application** _ to _ :: yielder, yielder → yielder.
- **closure** _ :: yielder → yielder.

These two yielders can be used to modify an abstraction so that the enclosed action receives certain data. For example consider a situation where we have an abstraction `myfunc` defined as follows:

```
myfunc = abstraction of
    store the given truth-value in the cell bound to tok.
```

In the following situation the **closure** yielder is used to modify the abstraction so that it is passed the current scoped information:

```
bind tok to mycell
hence
give closure of myfunc
```

This gives as a result an abstraction that is similar to `myfunc` as defined above, except that the action contained within the abstraction is passed the binding of `tok` to `mycell`, i.e. it looks as follows:

```
myfunc' = abstraction of
    bind tok to mycell
    hence
    store the given truth-value in the cell bound to tok.
```

The **application** _ to _ yielder evaluates to an abstraction that is the same as the one provided as the first argument but it is given the second argument as transient information. This could be used as show below:

```
application of true to myfunc'.
```
This would give as a result an abstraction similar to \texttt{myfunc}', except that the action contained within the abstraction is first passed the transient data \texttt{true}, i.e. it looks as follows:

\[
\text{myfunc}'' = \begin{aligned}
\text{abstraction of} \\
\text{give true} \\
\text{then} \\
\text{bind tok to mycell} \\
\text{hence} \\
\text{store the given truth-value in the cell bound to tok}.
\end{aligned}
\]

\subsection{Actions}

- \texttt{enact :: yielder \rightarrow primitive-action}.

This action has the effect of immediately enacting the abstraction that is provided to it as an argument. Continuing the previous example:

\[
enact \text{myfunc}''.
\]

This would perform the action contained within the \texttt{myfunc}'' abstraction, i.e. the value \texttt{true} would be stored in the cell \texttt{mycell}.

\subsection{Communicative Facet}

The communicative facet deals with agents and communications between those agents. An agent can be thought of as a process, similar to the processes that you might find on a computer. They operate independently and in parallel, and can communicate with each other by sending messages. One common use for agents is to model semantic concepts based on parallel processing.

A communication between two agents takes the form of a message being sent from one to the other. Communication is asynchronous. Once a message has arrived at an agent it is placed into an unbounded buffer. The message remains there until the agent inspects the buffer to see if any messages have arrived for it. A message contains
information about the identity of the sending agent and the identity of the target agent as well as the message data itself.

In any Action Semantics definition there is initially only one active agent. All other agents are inactive. An active agent may offer a contract to an inactive agent, requesting that it perform some task. A contract contains an abstraction that the inactive agent should perform.

3.1.8.2.1 Data

Shown below are the signatures for numerous functions and data types that are used within the communicative facet. The reader should not be concerned with the exact meanings of the properties total, strict, linear and partial; although the interested reader may find them defined in [Mos92, Appendix F].

- communication $\leq$ distinct-datum.
- communication $=$ message $|$ contract.
- contents _ :: message $\rightarrow$ sendable (total), contract $\rightarrow$ abstraction (total).
- sender _ :: communication $\rightarrow$ agent (total).
- receiver _ :: message $\rightarrow$ agent (total), contract $\rightarrow$ agent (strict, linear).
- _[containing _ ] :: message, sendable $\rightarrow$ message (partial), contract, abstraction $\rightarrow$ contract (partial).
- _[from _ ] :: communication, agent $\rightarrow$ communication (partial).
- _[to _ ] :: message, agent $\rightarrow$ message (partial), contract, agent $\rightarrow$ contract (strict).

The functions sender and receiver take a message or contract as an argument and evaluate to the sending or receiving agent for the message. Similarly the contents function evaluates to the data contents of a given message.

By convention in Action Semantics functions with arguments in square brackets, specify subsorts. Therefore the function _[to _ ] can be used as following:

message[to a]
This specifies the sub-sort of messages that are addressed to the agent \textit{a}. The function \texttt{[from _]}, is used to specify the sub-sort of messages from a particular agent. The function \texttt{[containing _]}, specifies the sub-sort of messages containing some given data.

### 3.1.8.2.2 Actions

- **send _** :: yielder $\rightarrow$ primitive-action.

  This action is used to send a given message. Within the message itself is the data to be sent and the target agent for the message.

- **receive _** :: yielder $\rightarrow$ action.

  This action is a compound action (i.e. it is not a primitive-action). Its purpose is to await an incoming message. Normally it would be used specifying particular attributes of the message that is expected (for example who it is from). For example:

  \begin{verbatim}
  receive a message[from a][containing a truth-value]
  \end{verbatim}

  This action waits for a message to arrive from the agent \textit{a} that has as its contents a \texttt{truth-value}.

- **subordinate _** :: yielder $\rightarrow$ action.

  This action is also not a primitive action. Its purpose is to set up a contract with an agent and give the chosen agent as transient information. The contract is an abstraction that simply awaits an incoming message, with an abstraction content. This new abstraction is then enacted. For example:

  \begin{verbatim}
  | subordinate an agent
  | then
  | send a message[to the given agent][containing abstraction of do-some-work]
  \end{verbatim}
This action contracts an agent to carry out an abstraction. The abstraction is sent within a message. In this case the abstraction the agent should carry out, contains an action \textbf{do-some-work}. Upon receipt of this message the contracted agent will extract the abstraction from the message and enact it, thus performing the \textbf{do-some-work} action.

### 3.2 Defining Object-Oriented Languages using Action Semantics

The remainder of this chapter explores possible methods by which the concepts of object-oriented languages (such as encapsulation and inheritance) can be modelled using Action Semantics. Four different approaches to modelling object-oriented languages are discussed, and these approaches are illustrated through reference to the semantics of an example language called EIL. Appendix A provides the complete semantics of the example language EIL using these four different approaches.

Encapsulation is a concept usually associated with object-oriented languages. As with many terms within the field of object-orientation, encapsulation is not clearly defined and is open to different interpretations by different people. Within the context of this thesis encapsulation is the concept that data owned by an object is accessible only to that object. Changes to the data owned by an object are only possible via the services provided by that object. This is the same definition, in effect, as that given by Snyder in [Sny93]. Stroustrup [Str88] calls this concept data abstraction.

Inheritance is considered to be a system in which objects can copy definitions of instance variables and methods from a parent class. Methods can also be redefined. Often a facility will exist within a language for accessing methods that have been overwritten by later versions. This facility exists for example in Smalltalk-80, [GoR83], where the programmer can access overwritten methods via the \textbf{super} expression.
3.3 The EIL Language

EIL has been designed to be very simple in order not to obscure the important underlying models that will be used in its semantic definition. There now follows a brief, informal description of the language in order to familiarise the reader with it before attempting to define it formally. The semantics of EIL presented here are based on its abstract syntax. This abstract syntax is given in Appendix A. Some examples of the EIL syntax can be seen below. There are of course many concrete syntaxes which could be used that would be compatible with the abstract syntax. In the examples presented in this section a concrete syntax (not defined here) has been selected that is as suggestive of the abstract syntax as possible.

A program in EIL consists of an expression to evaluate within the context of a set of class definitions. Classes have a similar function in EIL as classes in most object-oriented languages such as C++ [Sch92], or Smalltalk-80 [GoR83]. They can be considered to be templates, from which objects can be generated. A class defines the methods and instance variables that are available within object instances of the class. A method is a function that acts as an external interface for the object. An object can communicate with another object by invoking one of its methods. The instance variables represent the internal state of an object. Instance variables store the data associated with an object, and can only be accessed via the objects methods.

All classes must inherit information from some other class (except the root class object). A new class inherits all of the methods and instance variables that are defined in the parent class, and then adds its own new definitions (possibly overriding methods).

```
class examp inherits object   // start of class definition
    instance instv
    method meth1 is ...  // end of class definition
```
A class definition starts with the keyword "class" followed by an Identifier representing the name of the class. Next comes the keyword "inherits" followed by the name of a class which this class is based on.

After the class name has been declared the class body is given. A class body starts with a complete list of all of the instance variables available within the class. Each instance variable declaration consists of the keyword "instance" followed by the name of the instance variable.

The definitions of the methods provided by the class come next. A method definition is made up of the keyword "method", then an Identifier for the name of the method, the keyword "is", and finally a list of expressions to be executed when the method is enacted.

The definition of an example method is shown below. This example shows the different types of expressions that can be used within the EIL language. The method is called `addAndDisplay`. Its purpose is to add to some instance variable `counter` the argument sent to the method. It then displays this result using some other method on the current object called `print`. Finally this example method has overridden some previous version of it (defined in the parent class). This previous version is enacted through the use of a `super` expression.

```plaintext
method addAndDisplay is
  counter is (counter plus arg);
  self print counter;
  super addAndDisplay arg;
```

Expressions come in 5 forms.

1) A reference to `self`. This expression always evaluates to the current object.
2) An identifier on its own. There are two types of identifiers:
   a) An instance variable identifier. This evaluates to the value of that instance variable (e.g. `counter` in the method above).
b) An object name. This will either be a class name, or the special object "arg" which evaluates to the argument of the current method.

3) An instance variable assignment, i.e. the instance variable is assigned the result of evaluating the given expression, e.g. ("counter is ...") in the above method.

4) A method invocation, consisting of a sub-expression, an identifier and a second sub-expression. The first sub-expression evaluates to the target object (i.e. the object which is having its method invoked), the identifier gives the method name and the result of the second sub-expression is bound to "arg" during the invocation of the method. The result that a method returns to it's caller is simply the value of the last expression to be evaluated within the body of that method. For example the method above invokes the method plus on the object stored in the counter instance variable, "counter plus arg".

5) A method invocation via super. This expression invokes a method on the current object. The method executed will be the one defined by the parent class of the class that defined the current method. This allows a programmer to access methods that have just been overridden.

Classes are treated as objects in their own right. All class objects provide only one method, i.e. the "new" method. For the sake of syntactic and semantic simplicity all methods in EIL have exactly one argument; however in this case the argument is ignored and simply returns a new object which is an instance of the class. There is one pre-defined class which is called "object". Instances of class "object" have no methods. When a new class is defined it must specify a class name that is to be its parent. This therefore produces a hierarchy of classes with "object" at the top. The new class inherits all of the properties of the parent, but can redefine them if necessary. This can be seen in the following example program fragment.
The class hierarchy defined here can be seen in the following diagram.

This example illustrates how classes interact. The class "shape" defined above inherits all of the methods provided by "object" (i.e. none). It also provides the method "initialise". It declares an instance variable "location" (which can only be examined and changed via the methods). The method "initialise" updates the value of "location". The class "circle" inherits all of the methods and instance variables from "shape", i.e. it inherits "initialise" and "location". It declares the new instance variable "radius". The method "initialise" is redefined to additionally update the variable "radius".
Notice that in this example both versions of the method "initialise" change the value of "location". If we assume that the intent is to change "location" to the same value for each version then this program could be rewritten to make use of the "super" call.

```plaintext
class shape inherits object
classbody
    instance location

    method initialise is
        location is ...

class circle inherits shape
classbody
    instance radius

    method initialise is
        radius is ...
        super initialise ...
```

The method "initialise" in the class "circle" must set the value of the instance variable "radius". In all other respects it does exactly the same as the "initialise" in the class "shape", and so the original version is called via the expression "super".

### 3.4 Models for Object-Oriented Features

There are a number of ways that object-oriented features such as encapsulation and inheritance could be modelled. Four potential methods are explored in this chapter.

Two different strategies are used to model objects:
- Objects are modelled as cells in a store. Each cell contains a record that represents the state of the object (e.g. the values of the instance variables).
- Objects are modelled as agents. Action Semantics provides a Communicative facet that allows the creation of a system of agents. Each agent could be considered to be similar to a process on a computer. Messages between objects are modelled by sending messages between agents.
Most semantics of object-oriented languages to date have utilised the model of cells in a store, e.g. see [Wol88]. Action Semantics provides agents as an "in-built" facility. The similarity between agents and objects suggests that it is worth exploring whether using agents would make a suitable model for semantic definitions. This strategy was utilised by Palma et al. in their definition of the language POOL [PMM95].

Two different strategies are used to model the process required to search for a particular method:

- Methods are "copied down" to an object from its class at the creation time of the object. This has the advantage that in order to search for a particular method the search does not need to look at any other object than the one that was sent the original message. This is very similar to the approach taken by Wolczko in [Wol88].
- Methods are "looked up" dynamically. When an object receives a request to execute a particular method the class of that object is inspected to find the appropriate method.

The "look up" algorithm is the traditional method of finding the appropriate method to be executed. For example see [GoR83]. The "copy down" approach described here has been inspired by Wolczko [Wol88].

Combining the above two sets of strategies in different ways produces four methods of defining the semantics of object-oriented languages, i.e. "Cell based copy down", "Cell based look up", "Agent based copy down" and "Agent based look up".

3.4.1 Using Cells to Model Objects

A cell is a single element within a store. Each object in the system is modelled by exactly one cell in the store. This of course can also include class objects. Standard, pre-defined objects that often exist within languages (e.g. such as those that represent the constants of a type provided by the language) will also have to have a cell representing each object.
Cells that represent objects will contain a record. This record will give information on the instance variables available within that object, and the values of those instance variables, as well as information about the methods available within the object.

3.4.2 Using Agents to Model Objects

In Action Semantics, language definitions can be created using a system of agents. Agents represent individual processes and communicate with each other using messages. If in this last sentence we replace "objects" for "processes" and "methods" for "messages", it is easy to see the basis for which we can use the communicative facet of Action Semantics to model an object-oriented language. There appears to be (at least superficially) some similarity between the concepts of agents and objects.

Every object that is created will have a corresponding agent to model its behaviour, including class objects and standard pre-defined objects. The starting point for a network of agents in Action Semantics is the "user-agent". New agents can be contracted out from any other agent. A network of agents can thus be created from the user-agent starting point. A new agent is created for every object that is created.

Method requests are represented by message passing between agents. There are in effect a number of steps involved in a method invocation.

1) The source object/agent encodes and sends a message to the target object/agent with details of the method to be executed and any arguments that are required by the method. The source object/agent then suspends and awaits a response.

2a) The target object/agent will be in a suspended state until a message arrives. It then decodes from the incoming message the details of which method to execute and the arguments that are needed. The appropriate method is then executed.

2b) The target object/agent then encodes a message to be sent to the source object/agent with details of the result of executing the method. Once the message has been sent the target object/agent suspends.
3) The source object/agent receives the message from the target object/agent and decodes from it the result of the method. The object/agent then resumes.

It is interesting to note that the models in the agent based system suspend the source object's execution during the method invocation. This guarantees that we only ever have one object executing at any one time, and is what we would expect in a normal sequential style language. It is easy to see how we might extend the model to deal with parallel style languages such as America and Rutten's POOL [AmR92]. It would be possible to amend the model so that the source object does not suspend it's execution after sending the message to invoke a method.

3.4.3 Copy Down Semantics

An object that is created from a class that inherits information from previous class definitions must obtain the information about its behaviour from all of these classes. In order to specify a "super" expression it becomes necessary for objects not only to have access to information about the most up to date version of methods, but also previous versions as well. This causes two major problems that must be overcome for the "copy down" strategy:

- How do you distinguish between different versions of the same method?
- How do you determine which version of a method should be executed?

The approach taken in this thesis has been that, during the elaboration of a method definition, it is provided with information about the methods of the current superclass. This information is bound to a special token in the semantics definitions presented in Appendix A called "super-class-methods". For example consider the following situation:
Chapter 3: The Action Semantics of an Object-Oriented Language

class A inherits object
    method meth1 is ...
    method meth2 is ... //version 1

class B inherits A
    method meth2 is ... //version 2
    method meth3 is ... //version 1

class C inherits B
    method meth1 is ... //version 2
    method meth2 is ... //version 3

For Class A:
There are two methods provided by this class "meth1" and "meth2". Since this class inherits from "object", and that class defines no methods for its instances, the token "super-class-methods" is not bound to anything during the elaboration of these two methods.

For Class B:
This class provides 3 methods: "meth1", "meth2" (version 2), and "meth3". This class inherits from class A.

"meth1" is inherited directly from A, and therefore its definition is exactly as it is for A. The token "super-class-methods" was not bound to anything during its elaboration, and so any references to "super" in this method would fail.

"meth2" is a new (overridden) method for this class. Therefore the methods available via a call to "super" in that method are the methods defined in A. Hence the token "super-class-methods" is bound to a mapping for the methods "meth1" and "meth2" (version 1) exactly as they are defined in class A.
"meth3" is also a new method for this class, and therefore "super-class-methods" is bound to a mapping for the methods "meth1" and "meth2" (version 1) exactly as they are defined in class A.

For Class C:
In this class there are three methods provided "meth1" (version 2), "meth2" (version 3) and "meth3".

"meth1" is a new (overridden) method for this class. The token "super-class-methods" is bound to a mapping for the methods available within class B, i.e. "meth1" (version 1), "meth2" (version 2) and "meth3".

"meth2" is also a new (overridden) method for this class. The token "super-class-methods" is bound a mapping for the methods available within class B, i.e. "meth1" (version 1), "meth2" (version 2) and "meth3".

"meth3" is an inherited method from class B. Its definition is exactly the same as it is in class B. The token "super-class-methods" is therefore bound to a mapping for the methods "meth1" (version 1) and "meth2" (version 1).

3.4.4 Look Up Semantics

The look up semantics stores information about the methods that are provided by objects in the classes that defined those objects. When an object receives a message requesting that a particular method be executed, then this method must first be "looked up" in the appropriate class. If the method that is being looked up is not stored in the class because it has been inherited, then this looking up procedure gets chained back to the parent class. This process continues until the method has been found.

Modelling the semantics of "super" type expressions is relatively straight forward. with this approach. When a "super" expression is executed it is simply a matter of
starting the look up process in the super class of the class that defined the current method. Notice however that this is not necessarily the super class of the class of this object. Continuing the example in the previous section, if a "super" expression was present in "meth3" of an instance of class C, the look up search should start in class A. This is because class A is the super class of class B, where "meth3" was defined.

3.5 The Semantics of EIL

The language EIL was introduced earlier in this chapter. The four different models for defining the semantics of object-oriented languages discussed have been applied to the EIL language. The different versions of the semantics are presented in Appendix A. The remainder of this section examines in detail the features of these different semantics definitions.

3.5.1 Cell Based Copy Down Semantics of EIL

In the copy down approach, we must store a method overridden by new classes so it can be accessed by the "super" expression. This is done by binding the methods defined in the super class of the new class to the special token "super-class-methods". For example consider the following fragment from the semantics of EIL.

```
(1) create 
  [ "class" id:Object-Identifier 
    [ "inherits" iid:Object-Identifier ivs:InstVar* methods:Method+ ] ] =
    ... 
    furthermore bind super-class-methods to the methods defined by the object bound to iid
    hence
    produce the bindings bound to super-class-methods
    moreover
    construct the methods 
  ...
```

The super class of the new class is called "iid". The two lines of this fragment beginning "furthermore...", bind the methods defined by this super class to the special token "super-class-methods". The methods themselves are constructed in the call to
the semantic function "construct". Methods are defined as a set of bindings from method names to method-abstractions. The method-abstractions are "closed" during their definition. This means that the bindings available at the time of definition of the method, will be available within the methods itself, i.e. the value of "super-class-method" will be accessible.

Note that the complete set of methods for this class is built up by first producing the methods defined by the super class and then constructing the new ones. Any methods that are overridden will take precedence due to the use of the "moreover" action combinator.

A "super" call is defined as follows:

(1) evaluate \[ \text{"super" id:Method-Identifier E:Expression} \] =
    
give the object bound to self and evaluate E
then
enact application of the abstraction yielded by
  (the methods bound to super-class-methods) at id to
  (the given object#1, the given object#2).

Here the abstraction for the required method is found by referring to the methods that are bound to the special token "super-class-methods". As discussed above these are the methods defined by the super-class of the class that defined this method.

Objects are represented as a record stored in an cell. There are two essential attributes of an object that need to be stored: the instance variables for this object and the methods for this object.

- methods = map[token to method-abstraction].
- instance-variables = map[token to cell].
- object-record = (instance-variables, methods) | ...

There is one special case of objects. A class is viewed as a type of object with some extra properties. In fact a class could be modelled in exactly the same manner as an object, with the extra properties stored as special case private instance variables of the
object. However for the sake of readability this approach is not taken here. The extra properties for classes are: the instance variables defined by this class, and the methods defined by this class. These are required so that new instances of the class can be created based on those definitions.

- methods = map[token to method-abstraction].
- instance-variables = map[token to cell].
- instance-variable-tokens = list of token°.
- object-record = (instance-variables, methods) | (instance-variables, methods, instance-variable-tokens, methods).

It is worth noting that the instance variables for classes are in fact not used. If EIL had the concept of class variables then they would be useful. They have been left in this semantics for the sake of readability, and also so that it would be easy to extend the language with class variables in the future. It is also worth noting that there is only ever one method defined for any class: the "new" method. Some languages allow the definition of class methods, and this is where they would be stored if EIL had that concept.

3.5.2 Cell Based Look Up Semantics of EIL

Methods in the look up approach are not stored in the object. When a method is called it must be searched for. The search occurs through the use of a function called "look up the abstraction for _ in _". This works by checking for the existence of the required abstraction in the class given as an argument. If it exists then that abstraction is given as transient information. If it does not exist then the search continues via a recursive call to the function, passing the super class of the current class as an argument.
Chapter 3: The Action Semantics of an Object-Oriented Language

(1) call the method `meth` with argument `arg` at `ob:object` =

```
look up the abstraction for `meth` in the class `object` of `ob`
then
enact application of the given method-abstraction to `(ob, arg)`.  
```

(2) look up the abstraction for `meth:Method-Identifier` in `ob:object` =

```
give (the methods defined in `ob`) at `meth`
of
  | check not `meth` is in the mapped-set of the methods defined by `ob`
  and then
  | look up the abstraction for `meth` in the super class of `ob`.  
```

In the look up approach it is necessary for each method to have a reference to the class that created it. This is necessary so that the look up search for "super" expressions can start at the right place. The reference is stored by binding the special token "super-class" to the super class of the class object, before the construction of the methods. The method abstractions are "closed" during construction, and so "super-class" will be available within the methods themselves. Note however that inherited methods will have "super-class" set to the super class of the class that defined them.

(1) create 
```
"class" id:Object-Identifier
  [ "inherits" iid:Object-Identifier ivs:InstVar* methods:Method+ ] ] =
  allocate an object
then
  furthermore bind super-class to the super class of the given object
  hence
  construct the methods

...  
```

The "super" expression uses the current value of "super-class" to determine in which class the look up search for the methods should begin.

(1) evaluate 
```
"super" id:Method-Identifier E:Expression ] =
  look up the abstraction for `id` in the object bound to super-class
and
  evaluate `E`
then
  enact application of the given abstraction#1 to
  (the object bound to self, the given object#2).  
```
Objects in this approach are stored as a record in a cell. Two items of information are required to represent an object in the cell based look up approach. Firstly a reference to the class object that this object is an instance of is required. This is so that a search for a method can begin in that class object. Secondly the instance variables for the object are required.

- instance-variables = map[token to cell].
- object-record = (object, instance-variables) | ... 

In the same way as for the cell based copy down approach, this approach requires some extra information to be stored for classes. Again, this information could be stored as special case private instance variables of the object, but for the sake of readability are stored in the object record. The extra information required is a reference to the super class of this class; the instance variables defined by this class, and the methods defined by this class. Unlike the copy down approach, the item of the object record that specifies the methods defined by this class does not include the inherited methods. These are found by following the reference to the super class object.

- methods = map[token to method-abstraction].
- instance-variables = map[token to cell].
- instance-variable-tokens = list of token'.
- object-record = (object, instance-variables) | (object, instance-variables, object, instance-variable-tokens, methods).

In the same way as for the cell based copy down approach, the instance variables for class objects are in fact always empty because there is no concept of a class variable in EIL. They are left in for the sake of readability and to allow for the future extension of EIL with that capability.

Since all methods must be looked up in the class of the current object, and classes are treated as objects, it follows that classes must themselves have a class. In this semantics the concept of a meta-class has been introduced (which is an instance of
Chapter 3: The Action Semantics of an Object-Oriented Language

itself). This is where the class method "new" is stored. All classes are instances of the meta-class.

3.5.3 Agent Based Copy Down Semantics of EIL

In the same way as for the cell based copy down approach, the agent based copy down approach also needs to store information about the overridden methods in a class as well as all of the normal methods. This is achieved by binding the special token "super-class-methods" to the methods defined by the super class of the current class. Compare the following fragment of the semantics of EIL with the equivalent section for the cell based copy down approach.

```
(1) create [ "class" id:Object-Identifier
            send a message[to the object bound to iid][containing behaviour-request]
            then
            receive a message[from the object bound to iid]
            [containing (instance-variable-tokens,methods)]
            then
            furthermore bind super-class-methods to
            component\#2 of the contents of the given message
            hence
            produce the bindings bound to super-class-methods
            moreover
            construct the methods
```

Notice how the methods for the super class are discovered. In the cell based approach it was a simple matter to examine the contents of the record representing the super class. In the agent based approach this is not possible, and so a message has to be sent to the agent representing that class object, requesting that it provide the new class with the information.

During the construction of methods it is necessary for the instance variable bindings to be produced before the expression sequence that represents the method is executed. This was also true for the cell based approaches. In both the cell based and the agent based approaches it is necessary to send as arguments to the method, the current
object executing the method, and the value to be bound to "arg". The agent based approach additionally requires that the instance variables for the current object be sent. In the cell based approach it was a simple matter to examine the object record for the given object to discover the instance variables. However with the agent based approach this is not possible (without sending a message to and receiving a message from the object) and so they are sent as arguments to the method.

(2) construct \[ "method" id:Method-Identifier "is" E:Expression^+ \] =
bind id to closure of abstraction of
bind self to the given object#1
and
furthermore produce the given instance-variables#2
and
bind instance-vars to the given instance-variables#2
and
bind "arg":Object-Identifier to the given object#3
hence
evaluate E.

A call to "super" is implemented in the same way as for the cell based copy down approach. The methods for the super class of the class of this object are bound to the special token "super-class-methods", and so it is a simple matter to refer to this in order to discover the method to be executed. Compare this with the same section for the cell based copy down semantics of EIL.

(1) evaluate \[ "super" id:Method-Identifier E:Expression \] =
evaluate E
then
enact application of the abstraction yielded by
(the methods bound to super-class-methods) at id to
(the performing agent, the instance-variables bound to instance-vars, the given object#1).

Objects are represented in this semantics as agents. In the same way as for the cell based approach an object has certain information stored about it. This information is the instance variables for the object, and the methods available within the object. Unlike the cell based approach there is no record to store the information in. Instead it is accessible within the agent as scoped information. The instance variables are bound
to a special token "instance-variables". The methods are directly available as normal bindings. The fragment of the semantics below shows how a class object constructs a new instance of itself.

(2) make a new class =

```
| subordinate an object
| and
| elaborate the instance-variable-tokens bound to class-instances
| hence
| bind instance-variables to the current bindings
| and
| produce the methods bound to class-methods
| hence
| give closure of abstraction of activate message loop

then
| submit the given abstraction#2 to the given object#1
| and
| give the given object#1
```

Note that for a class object there is extra information stored, in the same way as for the cell based method. It is necessary for class objects to know about the instance variables that they define, and the methods that they define. The defined methods are bound to the special token "class-methods" and the defined instance variables are bound to the special token "instance-variable-tokens". Both of these can be seen in the above fragment.

An agent representing an object must be able to respond to incoming messages from other agents. Incoming messages take the form of requests for methods to be executed. In the case of classes they could also be requests for information about the methods and instance variables defined by the class.
Chapter 3: The Action Semantics of an Object-Oriented Language

3.5.4 Agent Based Look Up Semantics of EIL

Methods are looked up in this approach by sending a special message to class objects requesting that they return the method abstraction for the given method. If the method does not exist in the current class object, then the message is propagated up to the super class object.

<table>
<thead>
<tr>
<th>activate message loop =</th>
</tr>
</thead>
<tbody>
<tr>
<td>unfolding</td>
</tr>
<tr>
<td>receive a message[from an object][containing (token, object)]</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>enact application of the abstraction bound to</td>
</tr>
<tr>
<td>component#1 of the contents of the given message</td>
</tr>
<tr>
<td>to (the performing agent, the instance-variables bound to instance-vars,</td>
</tr>
<tr>
<td>component#2 of the contents of the given message)</td>
</tr>
<tr>
<td>and</td>
</tr>
<tr>
<td>give the sender of the given message</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>send a message[to the given object#2][containing the given object#1]</td>
</tr>
<tr>
<td>and</td>
</tr>
<tr>
<td>receive a message[from an object][containing a behaviour-request]</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>send a message[to the sender of the given message][containing</td>
</tr>
<tr>
<td>(the instance-variable-tokens bound to class-instances,</td>
</tr>
<tr>
<td>the methods bound to class-methods)]</td>
</tr>
<tr>
<td>and</td>
</tr>
<tr>
<td>unfold .</td>
</tr>
</tbody>
</table>
Chapter 3: The Action Semantics of an Object-Oriented Language

(1) look up the abstraction for meth:token in ob:object =
   send a message[to ob][containing meth]
   then
   receive a message[from ob][containing a method-abstraction]
   then
   give the contents of the given message.

(1) activate message loop =

   receive a message[from an object][containing a Method-Identifier]
   then
   give the method-abstraction yielded by
   (the method-bindings bound to class-methods) at
   the contents of the given message
   or
   check not the contents of the given message is in
   the mapped-set of the method-bindings bound to class-methods
   and then
   look up the abstraction for the contents of the given message
   in the object bound to super-class
   and give the sender of the given message
   then
   send a message[to the given object#2]
   [containing the given method-abstraction#1]

In order that methods in the look up approach can handle the "super" expression, they need to have a reference to the super class of the class that defined them. This is stored as scoped information bound to the special token "super-class", in the same way as for the cell based look up method.

(2) make all the methods and variables of
   ["inherits" id:Object-Identifier ivs:InstVar* methods:Method*]

   bind super-class to the object bound to id
   hence
   construct the methods.

When a method encounters the "super" expression the method is looked up starting at the class that is bound to the "super-class" token, as shown below.
Chapter 3: The Action Semantics of an Object-Oriented Language

(1) evaluate \[ \langle \text{"super" id:Method-Identifier E:Expression} \rangle = \]
    evaluate E
    and
    look up the abstraction for id in the object bound to super-class
    then
    enact application of
    the closure of the given abstraction
    to the given object.

Objects are modelled as agents. An object in the look up method needs to know about
the instance variables for that object, and the class that defined it (in order that it can
look up methods in that class). The instance variables for this method are stored as
normal bindings in the agent. The object representing the class of this object is bound
to the special token "this-class". The following fragment of the semantics shows how
the method "new" found in classes constructs new objects.

(4) the new method =
    closure of abstraction of
    subordinate an object
    and
    give the given object
    and
    produce the given bindings
    thence
    initialise the items of the flat-list bound to class-instances
    and
    bind this-class to the given object
    hence
    give closure of abstraction of activate message loop
    then
    submit the given abstraction to the given object
    and
    give the given object.

Classes need additional information about the instance variables defined by them, the
methods defined by them, and the super class of the class. This information is bound
to the special tokens "class-instances", "class-methods" and "super-class" respectively.

Agents modelling look up style objects need to be able to respond to incoming
messages from other agents. In particular they need to be able to respond to messages
requesting that a particular method be executed. Classes also need to be able to
respond to messages requesting information about the instance variables defined by
them, and also messages requesting that a particular method be looked up and the
abstraction returned.

\[
(1) \text{activate message loop = unfolding}
\]

\[
\text{receive a message[from an object][containing (Method-Identifier, object)]}
\]

\[
\text{then}
\]

\[
\text{... /* semantics for executing a method */}
\]

\[
\text{and}
\]

\[
\text{receive a message[from an object][containing a behaviour-request]}
\]

\[
\text{then}
\]

\[
\text{... /* semantics for returning information about the defined instance}
\]

\[
\text{variables */}
\]

\[
\text{and}
\]

\[
\text{receive a message[from an object][containing a Method-Identifier]}
\]

\[
\text{then}
\]

\[
\text{... /* semantics for looking up a particular method abstraction */}
\]

\[
\text{and}
\]

\[
\text{unfold}.
\]

In the same way as for the cell based look up method, every object needs to have a
class. Class objects are instances of a meta-class (which is an instance of itself). It is
the meta-class that defines the method "new" available within all classes.

3.6 Conclusions

This chapter has introduced the concepts of Action Semantics. A toy object-oriented
language, ENIL, has been examined in detail, and different methods of defining its
semantics have been explored.

The traditional method of modelling objects has been to use cells in a store. Action
Semantics also presents the possibility of using agents to model objects. Messages
sent between objects can be modelled as messages between agents. Agents execute in
parallel, which opens up the possibility of using them to define parallel languages that
have object-oriented features such as POOL.
Chapter 3: The Action Semantics of an Object-Oriented Language

It has been the experience of this author that the apparent similarities between agents and objects is largely superficial. In the opinion of this author the semantics of EIL based on the agent model are not as intuitively clear as the cell based versions. This is largely due to the fact that agents enforce encapsulation very strongly. It is not possible for one agent based object to inspect the state of another agent based object directly. A message must be sent. For example consider the situation where a new class is defined which must inherit the instance variables of its super class. In the cell based approach it is a simple matter to examine the cell containing the relevant information about the super class. In the agent based approach however it is necessary to send a message to the agent representing the class, requesting that the information be supplied. This adds unnecessary complexity to the semantics definitions.

Agents do have the advantage that they can be used to model parallel execution. This property may make the agent based approach attractive for defining some languages (such as POOL). For sequential style languages however there is little advantage to using the agent based approach.

Two different method searching systems have been used in this chapter. The first one is the traditional method "look up" system. In this approach an object must refer to its class in order to find the definitions of its methods. The "copy down" approach was inspired by Wolczko. This system copies down the definitions of methods into objects from classes at the time of their creation.

There is little to separate these two systems in their relative intuitiveness and complexity. Both have been used to effectively define the semantics of the language EIL. The look up approach has the small advantage that methods are looked up dynamically at the time they are called. This leaves open the possibility of languages where the definition of methods within classes changes during the execution of a program. This could not be modelled using the copy down system. However most languages do not provide such a facility. It is unclear to this author whether such a facility would in fact be useful or desirable.
Chapter 4: An Action Semantics of Smalltalk

An Action Semantics of Smalltalk

4.1 Introduction

This chapter presents an Action Semantics of the Object-Oriented language Smalltalk. The complete semantics is given in Appendix B.

Action Semantics has not been used extensively to describe Object-Oriented languages (see Chapter 2). The aim of this chapter is to provide an Action Semantics of a full Object-Oriented language and to act as a case study for one of the methods discussed in the previous chapter. The language that has been chosen for this is a cut down version of Smalltalk-80 [GoR83]. This language has a number of advantages for our purpose:

- Minimalistic syntax. There are very few basic expressions provided within the language, which makes it easier to define, because there is less work to carry out.
- Other definitions already completed to compare against. There have been a number of previous attempts to define Smalltalk including [GoL95, Kam88, Wol87, Wol88].
- The language itself is commonly used and well understood by a large number of people.

Most of the previous attempts at defining Smalltalk have used the framework of Denotational Semantics. Both Kamin [Kam88] and Golubski and Lippe [GoL95] make the "simplification" of making classes a special semantic entity distinct from objects. Smalltalk-80 itself uses objects to model everything including classes. The major step that is required to allow classes to be treated as objects is to make the
operation to create new objects a primitive method instead of a special case expression. This is not difficult to achieve. Wolczko [WoI87] makes the same simplification as above but in [WoI88] sketches how his semantics can be amended to allow classes to be treated as objects.

The semantics presented in this chapter does not treat classes as special cases, and so in the opinion of this author brings us closer to the intuitive semantics of Smalltalk-80. The semantics is heavily based on the work presented in Chapter 3 of this thesis.

### 4.2 A Restricted Smalltalk

The Smalltalk that is defined here differs in a number of ways from standard Smalltalk-80. The language has been restricted in a number of ways for the sake of brevity. The facilities that have been omitted are not essential to the language.

- There are no class, global or pool variables. Wolczko [WoI87] makes the same simplification and suggests that none of these are essential since they could be simulated by sending messages to the appropriate dictionary.

- Literals are restricted to Integers and Arrays. The full version of Smalltalk-80 provides many more literals than these. However the purpose of this chapter is simply to demonstrate how the semantics of Smalltalk can be produced. It is not necessary to examine in detail every type of literal; two examples should be sufficient.
There are no metaclasses. The class structure of the Smalltalk presented in this chapter has been simplified, and metaclasses have been removed. Instead all classes are instances of the class "Class". "Class" is also an instance of itself. This structure is outlined in Figure 1. In this figure boxes represent classes and circles instances of classes. Note that for every box in the figure there is a corresponding circle in "Class". It should not be difficult to extend the semantics to include metaclasses. This restriction does not of course interfere with our insistence that classes are treated as objects.
4.3 Abstract Syntax

grammar:

**Lexical elements**
- Selector = "new" |
- Integer-Constant = .
- Non-Root-Class = "Array" | "BlockContext" | "Class" | "Integer" | .
- Class-Name = Non-Root-Class | "Object". (disjoint)
- Instance-Variable = .
- Arg-Variable = .
- Temporary-Variable = .

**Variables and Literals**
- Variable-Name = [ Instance-Variable ] | [ Arg-Variable ] | [ Temporary-Variable ] (disjoint).
- Literal = [ Integer-Constant ] | [ "array" Literal* ] .

**Expressions**
- Expression = [ Variable-Name ] | [ Class-Name ] | [ Literal ] | [ Expression Expression* ] | [ "[" Expression ] | [ "[" Arg-Variable* Expression "]"] ] | [ Variable-Name "←" Expression ] | [ Expression Selector Expression* ] | [ "super" Selector Expression* ] | [ "self" ] .

**Classes**
- Method = [ Sel:Selector Method-Def ] .
- Method-Def = [ Arg-Variable* Temporary-Variable* Expression ] | [ Arg-Variable* Temporary-Variable* "primitive" Integer-Constant Expression ] .
- Class-Def = [ Class-Name "inheriting" Class-Name Instance-Variable* Method* ] .

In Action Notation, the box character □ represents something to be filled in later. In this case it is used for the lexical elements of the abstract syntax. For example we do not need to know the exact representation of Instance-Variables. It is merely enough to know that they exist as an independent entity within the syntax. The box character
works in the same way for Selectors, except in this instance there is one known member of Selector, i.e. "new".

Normally a program in Smalltalk is an expression that is evaluated in the context of a list of user defined class definitions. This semantics of Smalltalk only gives a definition of classes and not programs.

4.4 The Semantics

Each object in the system is represented as a cell in a store that contains a record. The record contains a set of bindings that map instance variable names to cells (which then store values), and a set of bindings that map method names to the abstractions that represent those methods. Every object is uniquely identified by the cell where the object record is stored. The methods for an object are copied from the class of that object at its creation, and placed in the object record along with the appropriate instance variables that represent the new object. This is the cell based copy down approach that was discussed in Chapter 3. Any of the other three methods could also have been used. This method has been selected since it is the approach that was chosen by Wolczko [Wol88].

4.4.1 Semantic Entities

- object \leq cell.
- method-abstraction = abstraction
  [given an object | storing | escaping with (object, integer) | diverging | failing]
  [using the given (object, integer, argument-list) | current storage].
- argument-list = list of object*.
- methods = map[token to method-abstraction].
- instance-variables = map[token to cell].
- instance-variable-tokens = list of token*.
- object-record = object record of (instance-variables, methods) | object record of (instance-variables, methods, instance-variable-tokens, methods).

As has already been stated an object is represented as a record stored in a cell. The cell uniquely identifies the object. There are two variants of object record. Both variants have as their first two components the instance variables for the object, and the
methods for the object. Currently our restricted semantics of Smalltalk does not have class variables. This means that for class objects the instance variables component never actually contains any instance variables. However since the full version of Smalltalk does have class variables this redundancy has been accepted to allow for future expansion of this semantics definition.

The second variant of the object record is used to store information about class objects. Notice that this variant can be used at any place in the language where a normal object is expected (i.e. classes are treated as objects). The extra information required for classes is a list of instance variables defined by this class, and a list of the methods defined by this class. This information is required so that new instances of the class can be created. When a new object instance is created its instance-variables and methods are copied down from the extra fields in the class object record.

- reserved-tokens = super-class-methods | sender | block-contents | object-value | self (individual).
- token = string of ( letter, ( letter | digit )*) | reserved-tokens (disjoint).

Tokens are defined as a letter followed by a string of letters and digits. There are also some reserved tokens which are used internally within the semantics definition.

### 4.4.1.1 Standard Objects

All objects are instances of a class. All classes ultimately inherit from the class "Object". All classes are also instances of the class "Class". These two classes are special, in that they must be defined in order to be able to define any new classes. Two actions are defined within the semantics definition presented in Appendix B for creating these two class objects.
Chapter 4: An Action Semantics of Smalltalk

(1) create object called class =
    allocate an object
    then
    store the object-record of (empty-map, the new method, empty-list, the new method) in the given object
    and
    bind "Class" to the given object.

The object "Class" is created by allocating a cell, storing its object-record in that cell, and binding the token "Class" to that cell. The object-record contains four elements (i.e. it is the variant of object-record used for storing classes). The object has no instance variables, and provides one method (the new method). Instances of this class also have no instance variables, and have one method (the new method). In fact this class is an instance of itself.

(1) create object called object =
    allocate an object
    then
    store object-record of (empty-map, the new method, empty-list, empty-map) in the given object
    and
    bind "Object" to the given object.

The object "Object" is created in the same way as for the object "Class". The difference is in the object-record that is stored. This object has no instance variables, and provides one method (the new method). Instances of this class have no instance variables and no methods. This class object is an instance of the class "Class".
4.4.1.2 The "new" method

The method "new" that is provided by classes, consists of a map from the token "new" to a method-abstraction. A method-abstraction is an abstraction that expects certain parameters to be given to it. The first of these is the target object for this method. When the method is invoked this first object is the class object for which a new instance is required. A new instance is made by allocating a new cell for the object, storing the object-record for the object in the cell, and returning a reference to the object as a result of the method.

The object-record contains the instance-variables for the new object, which are initialised and created based on the instance-variables defined by the class. It also contains the methods for the object which are copied from the methods defined by the class.

4.4.1.3 Variable Initialisation

There are three different types of variables used within Smalltalk: instance variables, argument variables and temporary variables. All are represented in the semantics in the same way, i.e. as maps from tokens to cells. The cells store objects that represent the values of the variables.

Instance variables represent the state of an object. They are stored within the object-record for that object. Argument variables are defined for individual methods, and
only have a scope within that method. They are initialised on creation to the arguments that were sent to the method. Temporary variables also only have a scope within individual methods. They are created for temporary storage of data during the execution of a method.

Initialisation of variables takes two forms. All variables when they are initialised have a cell created to represent their value. That cell is then bound to the token for the variable. Argument variables additionally have a value set for them on creation, i.e. an object is automatically stored in the cell.

1. initialise <> =
   complete.

2. initialise <Var:token Var\textsubscript{list}:token\textsuperscript{*}> =
   allocate a cell
   then
   bind Var to the given cell
   and
   initialise Var\textsubscript{list}:

3. initialise <> to <> =
   complete.

4. initialise <Arg:token Arg\textsubscript{list}:token\textsuperscript{*}> to lst:argument-list =
   allocate a cell
   then
   store the head of lst in the given cell
   and
   bind Arg to the given cell
   and
   initialise Arg\textsubscript{list} to tail of lst.
### 4.4.1.4 Calling Methods

| (1) call the method Sel:token with arguments args:argument-list at ob:object = |
| enact application of |
| the method-abstraction yielded by ((the methods of ob) at Sel) |
| to (ob, the successor of the integer bound to sender, args) |
| trap |
| check the given integer#2 is the successor of the integer bound to sender |
| and then |
| give the given object#1 |
| or |
| check not the given integer#2 is |
| the successor of the integer bound to sender and then |
| escape with the given tuple. |

Methods are called using the action "call the method _ with arguments _ at _". This action takes three arguments: the name of the method to call, a list of the argument values for the method and the target object for the method call. The method is executed through a simple process of finding the method-abstraction for the given token and enacting it.

Enaction of the method requires that three arguments are sent to it. The first is the target object for the method. The second is an integer that represents the current context. The last is a list of the argument values.

The integer representing the current context is used in order to correctly handle the execution of a return ("↑") expression in a method. In its simplest form this expression simply forces the method to return to the calling object with a given value. It is possible in Smalltalk however to create a BlockContext object using the square brackets "[" and "]". If a return expression is contained within the square brackets then the handling of a return is different. A BlockContext object is enacted by calling the "value" method on that object. If a return expression is encountered, then the return relates to the context in which the BlockContext was created, i.e. it does not return to the location that called the "value" method, it returns to the location that called the method where the BlockContext was defined.
The context for the current method is represented as an integer bound to the special token "sender". Each time a new method is added to the call stack this integer is incremented.

A method completes by either finishing the execution of all the expressions in the method, or by executing a return expression. If no return expression is encountered then the enactment of the method-abstraction simply completes normally and gives as transient data its result. If a return expression is encountered then the enactment of the method-abstraction escapes giving the context that the return was called in, and the data that is being returned. This escape is trapped and the context examined. If the context is not the current context then the escape is propagated back. Otherwise the result of the return is given as transient data.
4.4.2 Semantic Functions

4.4.2.1 Creating User Defined Classes

Classes are defined using the "create" semantic function.

```
(1) create < C1:Class-Def C2:Class-Def+ > =
    | create C1
    | before
    | create C2 .

(2) create [ ClsNam:Class-Name "inheriting" iid:Class-Name
    ivs:Instance-Variable' mthds:Method' ] =
    | allocate an object
    | and furthermore bind super-class-methods to
      the methods defined by the object bound to iid
      hence
      | produce the methods bound to super-class-methods
      moreover
      | construct the mthds
      hence
      | give the current bindings
    then
    | store the object-record of( 
      empty-map,
      the new method,
      concatenation( 
        the instance variables defined by the object bound to iid,
        list of ivs),
      the given methods#2) in the given object#l
    and
    | bind ClsNam to the given object#l.
```

A class is created by allocating a cell for the new class object; constructing the methods defined by the class; creating an object-record for the class object; storing the object record in the new cell and finally binding the name for the class to the cell.

The methods for the class are constructed by first finding the methods that have been inherited from the super-class. These methods are bound to the special token "super-class-methods", so that they are available if a method executes a "super" expression. The complete set of methods for the class are found by combining the inherited
methods with the new methods. New methods override existing inherited methods of the same name due to the use of the "moreover" action combinator.

The object-record for the class object contains no instance-variables. As discussed above this field is technically redundant for classes because there are no class variables in this restricted version of Smalltalk. However it has been left in for simplicity and to allow for the future expansion of the language.

There is one method for the class object, which is the "new" method. The instance variables defined by the class are found by concatenating the list of instance variables defined by the super class of this class, with the newly defined ones. The final item in the object record is the list of methods defined by this class, which are constructed as discussed above.

4.4.2.2 Constructing Methods

(1) construct <> =
     complete.

(2) construct < M₁:Method M₂:Method > =
     construct M₁
     and
     construct M₂.

(3) construct [ Sel:Selector MethDef:Method-Def ]
     bind Sel to make method MethDef.

Methods are constructed by creating a set of bindings for those methods from the name of the method to the method-abstraction that represents that method. There are two form of method definition. The first is the simplest case and consists of a list of arguments, a list of temporary variables and an expression. The second form is the same as the first but additionally specifies a primitive method. [GoR83] lists a large number of primitive methods available within Smalltalk. Appendix B defines just one of those in the Semantic Entities section.
The action enclosed within a method-abstraction does a number of things. It expects to receive three arguments. The first is the target object for this method. This is bound to the special token "self", and is used to evaluate the "self" expression. The second argument is an integer representing the context of this method call (see section 4.4.1.4 for a discussion on this). This context is bound to the special token "sender". The third argument is a list of the argument values for this method. These values are used to initialise the list of argument variables.

The temporary variables for the method are initialised, and the instance variables for this object are produced. Finally the expression for the method is evaluated. The environment therefore during the evaluation of the expression is a combination of the instance-variables, the temporary variables, the argument variables, and settings for the special tokens "self" and "sender".

The second form of method definition specifies a primitive method to be used. If that primitive method fails for some reason then a non-primitive method can be used instead.
4.4.2.3 Expressions

The complete list of available expressions is defined in Appendix B. Some of the more interesting expressions are discussed below.

(3) evaluate \[
\text{Int:Integer-Constant} =
\]
call the method "new" with arguments empty-list at the object bound to "Integer"
then store the decimal of Int in the cell yielded by (the instance variables of the given object) at object-value and regive.

Integer literals are evaluated by creating a new instance of the class object "Integer" to represent that literal. Note that the definition of the class "Integer" has been left unspecified in this semantics. It is assumed however that instances of that class have a special instance variable "object-value", which holds the integer value.

(4) evaluate \[
\text{"array" Arr:Literal*} =
\]
call the method "new" with arguments empty-list at the object bound to "Array"
and then evaluate the list Arr
then store the given flat-list#2 in the cell yielded by (the instance variables of the given object#1) at object-value and give the given object#1.

Array literals are dealt with in a similar manner to integer literals. A new instance of the class "Array" is created to represent that literal. Internally the value is stored as a list of object references within the special instance variable "object-value". Again the definition of the class "Array" has been left unspecified in this semantics.

(6) evaluate \[
\text{"↑" E:Expression} =
\]
evaluate E then escape with (the given object, the integer bound to sender).
Chapter 4: An Action Semantics of Smalltalk

The return expression simply evaluates its sub-expression and escapes. It escapes with two data items, i.e. the value of the evaluated expression, and the integer value for the context that is bound to the token "sender".

(7) evaluate 
\[
\begin{array}{l}
\text{[" Arg:Arg-Variable* E:Expression "] =}
\end{array}
\]
\[
\begin{array}{l}
call the method "new" with arguments empty-list at
the object bound to "BlockContext"
then
store closure of abstraction of
| furthermore initialise Arg to the given argument-list
| hence
| evaluate E
| in the cell yielded by
| (the instance variables of the given object) at block-contents
and
regive.
\end{array}
\]

Blocks are represented by instances of the "BlockContext" class, which is left unspecified in this semantics. The value of the block is stored in a special instance variable called "block-contents". The value is represented as a closure of an abstraction, which simply initialises some argument variables and evaluates the expression. Note that because the abstraction is closed, the expression is evaluated in the current environment, e.g. it has access to the current instance variables, and the current context value bound to "sender".

(9) evaluate 
\[
\begin{array}{l}
\text{[ E:Expression Sel:Selector EL:Expression* ] =}
\end{array}
\]
\[
\begin{array}{l}
evaluate E
\end{array}
\]
\[
\begin{array}{l}
and then
\end{array}
\]
\[
\begin{array}{l}
evaluate the list EL
\end{array}
\]
\[
\begin{array}{l}
then
\end{array}
\]
\[
\begin{array}{l}
call the method Sel with arguments the given argument-list#2 at
the given object#1.
\end{array}
\]

Method calls occur by evaluating an expression that identifies the target object for the method, and then evaluating a list of expressions that represent the argument values. The calculated objects are then used in the action "call the method _ with arguments _ at _".
Chapter 4: An Action Semantics of Smalltalk

(10) evaluate \[ "self" \] =
    give the object bound to self.

The current object is bound to the special token "self".

(11) evaluate \[ "super" Sel:Selector EL:Expression* \] =
    evaluate the list EL
    then
    enact application of the abstraction yielded by
    ((the methods bound to super-class-methods) at Sel) to
    (the object bound to self, the successor of the integer bound to sender,
    the given argument-list#3).
    trap
    check the given integer#2 is the successor of the integer bound to sender
    and then
    give the given object#1
    or
    check not the given integer#2 is
    the successor of the integer bound to sender
    and then
    escape with the given tuple.

The "super" expression is very similar to a normal method call (compare with the
definition of "call the method _ with arguments _ at _"). The main difference is that
the method definition is found by examining the methods bound to the special token
"super-class-methods". The methods bound to that token are those defined by the
super class of the class that defined the current method. Therefore overridden methods
can be accessed via this expression.

4.5 Conclusions

An Action Semantics of a cut down version of the Object-Oriented language
Smalltalk-80 has been presented. This semantics could be used as a basis for
producing the semantics of other Object-Oriented languages.

One potential method of defining the semantics of Object-Oriented languages has
been investigated and found feasible. Our knowledge of defining these languages
using Action Notation has therefore been increased. An effective demonstration of how Action Semantics can be used in practice has been given.

The method that has been chosen to define the semantics uses cells in a store to represent objects. All the methods for an object are copied down from the class of that object at its creation time. This is only one possible method of defining the semantics. Other potential approaches were given in Chapter 3.

The semantics that has been given differs from other approaches in that it treats classes as objects, and does not have a special "new" expression. This semantics is closer to the intuitive semantics of the language.

In the opinion of the author the use of Action Semantics for the definition of the Smalltalk language has succeeded in its aim of making the semantics more readable. It should be possible for a computer scientist with no knowledge of Action Semantics to get at least a superficial feel for how the semantics of the language works. It would be an interesting experiment to attempt to demonstrate that this is in fact the case.
Chapter

5

Denotational Semantics

5.1 Introduction

This chapter considers the popular method for defining the semantics of languages that is known as Denotational Semantics. An introduction is given to the main concepts of Denotational Semantics. In addition this chapter will examine methods by which a Denotational Semantics of a language can be generated from a given Action Semantics for that language. This process has been carried out for the definition of Smalltalk that was given in the previous chapter.

5.2 Denotational Semantics

A summary of the main concepts of Denotational Semantics that are important for this chapter is given here. For a complete tutorial on the subject see [Gor79], [Ten76], or [Wat91].

5.2.1 Semantic Functions

In denotational semantics every phrase within a language is assigned a meaning called a denotation. The semantics of the language is defined by semantic functions that map syntactic entities to their denotations. For example consider the following language syntax:
In order to specify the semantics of this language we must define semantic functions from each of the syntactic entities to a denotation to represent the meaning of that entity. This language might be defined as follows:

<table>
<thead>
<tr>
<th>value:</th>
<th>Digit → Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>totalvalue:</td>
<td>Numeral → Integer</td>
</tr>
<tr>
<td>evaluate:</td>
<td>Expression → Integer</td>
</tr>
</tbody>
</table>

value ["0"] = 0
value ["1"] = 1

totalvalue [D] =
    value D

totalvalue [ND] =
    ((totalvalue N) * 2) + value D

evaluate [N] =
    totalvalue N

evaluate [E1 "+", E2] =
    (evaluate E1) + (evaluate E2)

evaluate [E1 "-", E2] =
    (evaluate E1) - (evaluate E2)

Here three semantic functions have been defined. Each of the three types of syntactic entity have been assigned a denotation in the domain of Integers. Each semantic function maps one type of syntactic entity to its denotation. For example the "value" function maps "Digit" syntactic entities to Integers. The symbol "0" is mapped to the
integer 0. Note the distinction here between the symbol "0" and the value 0. The symbol has no value in itself, we could have defined the language such that:

<table>
<thead>
<tr>
<th>Digit = &quot;A&quot;</th>
<th>&quot;B&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>value [&quot;A&quot;] = 0</td>
<td></td>
</tr>
<tr>
<td>value [&quot;B&quot;] = 1</td>
<td></td>
</tr>
</tbody>
</table>

This change does not affect the underlying semantics of the language, merely the symbols used to represent it.

### 5.2.2 Environments

The concept of an environment is common within programming languages. An environment is a set of bindings from identifiers to values. An environment will normally have a scope. This is similar to the concept of scoped information in Action Semantics.

As an example of the use of environments in Denotational Semantics we will extend the example language given in the previous section. Firstly we will extend the definition of expressions.

\[
\text{Expression} = \text{Numeral} \\
\quad | \text{Constant-Identifier} \\
\quad | \text{Expression} \ "+" \ \text{Expression} \\
\quad | \text{Expression} \ "-" \ \text{Expression} \\
\quad | \ "let" \ \text{Constant-Identifier} \ "=" \ \text{Expression} \ "in" \ \text{Expression}
\]

Here two extra kinds of expressions have been added to the language. The first is a simple constant identifier. The concept is that an identifier has a constant value associated with it. This expression should then evaluate to the value associated with the identifier. The second new expression is "let...in...". This allows us to associate a value with a given identifier. The value of the identifier has a scope which extends throughout the second sub-expression. For example the expression "let x=2 in x+3" would evaluate to 5. Note that we have not defined here what an identifier is.
In this new version of the semantics, the semantic function "evaluate" has been amended so that expressions now map to a function that maps environments to integers. In other words the value of an expression is an entity that given an environment will produce an integer. An environment in this language is an entity that maps identifiers to integers. Note the use of the square bracket notation, to signify a change to the environment:

\[ \text{env}[a \mapsto \text{val}] \]

This produces a new environment where all identifiers map to the same values that they did in the original environment "env", except that "a" maps to "val".

In this new version of the "evaluate" semantic function the environment is passed as an explicit argument. Since the environment is just a function, in order to look up the value of an identifier it is a simple matter of providing the identifier as an argument to the environment. The "let...in..." expression is defined by producing a new version of the environment and passing that as an argument to the recursive "evaluate" call for the second sub-expression. Notice that the new environment will only have a scope during the evaluation of the second sub-expression.
5.2.3 Storage

All computers have some form of store. Stores have locations or cells, which hold values. The state of a store remains the same until it is explicitly changed. Denotational semantics deals with storage snapshots. Updating the value stored in a cell can be modelled as a function from one storage snapshot to another.

Continuing the example from the previous section, we could introduce statements into our language that allow the use of updateable variables.

Statement = "declare" Variable-Identifier
| Variable-Identifier "=" Expression
| Statement ";" Statement

Expression = Numeral
| Constant-Identifier
| Variable-Identifier
| Expression "+" Expression
| Expression "-" Expression
| "let" Constant-Identifier "=" Expression "in" Expression

Here "Variable-Identifiers" have been introduced. Statements take the form of an assignment to a variable. Expressions have been extended so that variables can be accessed.

Stores can be defined as follows:

| Store: Cell → Integer
| unused-cell : Store → Cell

For this language a store is a function from the domain of cells to the domain of integers. We will use a function "unused-cell" (left unspecified here), which given a store will evaluate to a cell that is as yet unused in that store. We will also reuse our notation for updating environments for stores, i.e.
This represents a store "st", that has been updated so that the cell "c" is associated with the value "v".

We also update our concept of an environment, so that it can map constant identifiers to integers, and variable identifiers to cells.

Using this notation we can formulate the semantic functions for the new version of our language.
A new semantic function, "execute", has been introduced here to deal with the semantics of statements. Since statements can be composed sequentially it is important that the store produced as a result of executing the first statement is available to the second statement. The "declare" statement allows the declaration of new variable identifiers, and allocates a cell to be used for storing the value of that variable. Declarations in this language can occur anywhere where a statement occur (and they are therefore treated as a normal statement), and do not have to occur at the
beginning of a block. The "declare" statement also changes the current environment. Therefore the result of executing a statement is an updated environment and store.

5.2.4 Continuations

The examples that have been presented so far have been in the direct style. Some language constructs are difficult to model using this style, for example exceptions and "goto" statements. Languages that contain these types of constructs can be modelled using continuations. A continuation represents the subsequent action of a program.

We will now introduce an "escape" statement into our language, and a "begin...end" statement. Informally the idea behind this is that "begin...end" marks the start and end of a block of statements. An "escape" statement executed within such a block will immediately move the flow of control to the statement immediately following the next "end". This is the kind of construct that is difficult to define using the direct style. The abstract syntax now looks like this.

| Statement =   | "declare" Variable-Identifier       |
|              | Variable-Identifier "=" Expression |
|              | Statement ";" Statement             |
|              | "begin" Statement "end"             |
|              | "escape"                             |

| Expression = | Numeral                              |
|             | Constant-Identifier                 |
|             | Variable-Identifier                 |
|             | Expression "+" Expression          |
|             | Expression "+" Expression          |
|             | "let" Constant-Identifier "=" Expression "in" Expression |

An execution of a statement is defined as a function requiring information about the current environment and the store, and ultimately produces an environment and store pair as a result. A continuation for our language will therefore be based on this.

| Continuation: Environment → Store → Environment × Store |
Chapter 5: Denotational Semantics

The signature of the "execute" function will be changed to use continuations. This function will take a statement; a continuation representing the subsequent action of the program if no "escape" statement is encountered (represented in the semantics below by the "cont" variable), and a second continuation representing the subsequent action of the program if an "escape" statement is encountered (represented by the "econt" variable). The result will be a continuation that represents the result of executing this statement followed by the rest of the program.

Non-compound statements (except for "escape") all use the normal completion continuation (cont) to pass the results of executing the statement to the "rest" of the program. During the execution of a "begin...end" statement, the enclosing statement is executed. Any escape executed within the enclosing statement should cause the flow of control to move to the rest of the program following the "begin...end". Since this is represented by the argument "cont", this is passed as the exceptional continuation argument (normally represented by "econt") to the execution of the enclosed statement.

An "escape" statement simply uses the exceptional continuation, thus bypassing all of the rest of the statements in this block.

Note the use of *lambda-notation* in the semantics below to define an anonymous function. This is used as follows:

\[ \lambda x. (x + 3) \]

This defines a function takes a value as an argument. The argument is given the name "x". The result is obtained by substituting all occurrences of "x" in the expression "x+3", with the argument value. For example:

\[ (\lambda x. (x + 3)) 5 \]

by substituting "5" for "x" gives:

\[ (5 + 3) \]
execute:  \[\text{Statement} \rightarrow \text{Continuation} \rightarrow \text{Continuation} \rightarrow \text{Continuation}\]

\[
\text{execute } [\text{"declare" } V ] \text{ cont } econt \text{ env } st = \\
\text{let } c = \text{ unused-cell } st \\
in \\
\text{cont } env[V \mapsto c] \text{ st[c \mapsto 0]}
\]

\[
\text{execute } [V \text{ ":=" } E ] \text{ cont } econt \text{ env } st = \\
\text{cont } env \text{ st[(env V) \mapsto (evaluate } E \text{ env st)]}
\]

\[
\text{execute } [S_1 \text{ ";" } S_2 ] \text{ cont } econt = \\
\text{let } cont' = \lambda env'.\lambda st'.\text{execute } S_2 \text{ env'} st' \text{ cont } econt \\
in \\
\text{execute } S_1 \text{ cont'} econt
\]

\[
\text{execute } [\text{"begin" } S \text{ "end" } ] \text{ cont } econt = \\
\text{execute } S \text{ cont } cont
\]

\[
\text{execute } [\text{"escape" } ] \text{ cont } econt = \\
\text{econt}
\]

### 5.2.5 Recursion and Iteration

Recursion in denotational semantics is important because it can be used to model the effects of iterative constructs. A recursive function, \( f \), can be manipulated into the form:

\[ f = F f \]

where \( F \) is a functional. A functional, is a function that maps a function in one domain to some other function in that same domain. In the above \( f \) is called a fixed point. The least defined definition for \( f \) is called the least fixed point. In this section we make use of the function \text{fix} which is the least fixed point of its argument. Dana Scott [Sco76] quotes the following theorem about least fixed points:
Chapter 5: Denotational Semantics

Every continuous function $F: \mathcal{P} \rightarrow \mathcal{P}$ has a least fixed point given by the formula

$$\text{fix}(F) = \bigcup \{ F^n(\emptyset) \mid n \in \omega \}$$

where $\emptyset$ is the empty set and $F^n$ is the $n$-fold composition of $F$ with itself.

[$\mathcal{P}$ is the domain of all subsets of the set $\omega$ of nonnegative integers]

To demonstrate the use of $\text{fix}$ in the definition of an iterative construct we will introduce a "while...do...endwhile" loop into our example language.

<table>
<thead>
<tr>
<th>Statement =</th>
<th>&quot;declare&quot; Variable-Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Variable-Identifier &quot;=&quot; Expression</td>
</tr>
<tr>
<td></td>
<td>Statement &quot;;&quot; Statement</td>
</tr>
<tr>
<td></td>
<td>&quot;begin&quot; Statement &quot;end&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;escape&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;while&quot; Expression &quot;do&quot; Statement &quot;endwhile&quot;</td>
</tr>
</tbody>
</table>

Informally the idea behind this construct is that whilst the expression evaluates to a non-zero value the statement will be repeatedly executed. The definition of this statement is given below. This definition uses the notation "...→...". This checks the value before the "→" to see if it is true or false. If it is true then the whole thing evaluates to the value immediately after the "→". If it is false then the whole thing evaluates to the value immediately after the ",". It can be thought of as similar to an "if...then...else" statement found in many common languages.

execute \[ "while" E "do" S "endwhile" \] cont econt env st =

$$\text{fix}(\lambda f. \text{let})$$

\[
\text{let}
\]

\[ \text{cont'} = f \text{ cont econt} \]

\[ \text{in} \]

\[ \text{(evaluate E env st)}=0 \rightarrow \text{ cont env st, execute S cont'} \text{ econt env st} \]
Chapter 5: Denotational Semantics

The definition of "while...do...endwhile" is best explained through the use of an example. Consider a situation where the first time through the loop, the expression E evaluates to a non-zero value. The second time it is evaluated it evaluates to zero. Therefore:

\[ \text{let } F = \lambda f. (\text{evaluate } E) = 0 \rightarrow \text{cont, execute } S \text{ econt} \]

in

\[ \text{execute } [ \text{"while" } E \text{ "do" } S \text{ "endwhile" }] \text{ cont econt env st} \]

\[ = f_{\text{ix}}(F) \text{ env st} \]

(from definition of execute)

\[ = F(f_{\text{ix}}(F)) \text{ env st} \]

(from property of \( f_{\text{ix}} \))

\[ = ((\text{evaluate } E) = 0 \rightarrow \text{cont, execute } S f_{\text{ix}}(F) \text{ econt}) \text{ env st} \]

(from definition of \( F \))

\[ = \text{execute } S f_{\text{ix}}(F) \text{ econt env st} \]

(from assumption)

Assuming the execution of \( S \) completes normally (i.e. it does not escape), the continuation \( f_{\text{ix}}(F) \) will be called with some new environment \( \text{env}' \) and store \( \text{st}' \). This represents the second iteration through the loop (i.e. this time through the expression will be 0):

\[ f_{\text{ix}}(F) \text{ env}' \text{ st}' \]

\[ = F(f_{\text{ix}}(F)) \text{ env}' \text{ st}' \]

(from property of \( f_{\text{ix}} \))

\[ = ((\text{evaluate } E) = 0 \rightarrow \text{cont, execute } S f_{\text{ix}}(F) \text{ econt}) \text{ env}' \text{ st}' \]

(from definition of \( F \))

\[ = \text{cont env}' \text{ st}' \]

(from assumption)
"undefined" symbol, \( \bot \). This will propagate in the same way that failure propagates in Action Semantics. The "completed" and "escaped" states can both be represented via continuations. For example consider the definitions of a Smalltalk expression given in fig. 1 and fig. 2 which we shall now discuss:

\[
\text{evaluate \{ VarNam:Variable-Name \} = give the object stored in the cell bound to VarNam.}
\]

**Figure 1. Action Semantics of a Smalltalk expression.**

\[
\text{evaluate \{ VarNam:Variable-Name \} ecnt retcnt env st = ecnt (st (env VarNam)) st}
\]

**Figure 2. Denotational Semantics of a Smalltalk expression.**

In this example the semantics of a particular type of expression is given. This type of expression can only either fail, or complete normally. Normal completion is represented by the expression continuation, "ecnt". Failure would occur if the environment function "env" returned, \( \bot \). This failure would then be propagated by "ecnt". Smalltalk has a "return" expression (shown below), that immediately exists the currently executing method, returning a value. This requires the use of a "return" continuation, which has been called "retcnt". Notice that for this particular expression "retcnt" is not used. However the Denotational Semantics still contains a reference to it.

Now consider the following semantics of an expression from the Smalltalk language:

\[
\text{evaluate \{ "↑" E:Expression \} =}
\]
\[
| \text{evaluate E} \\
| \text{then} \\
| \text{escape with (the given object, the integer bound to sender).}
\]

**Figure 3. Action Semantics of an expression with an escape**
Figure 4. Denotational Semantics of an expression that simulates an escape

The purpose of this example is to demonstrate how an Action Semantics "escape" can be modelled in Denotational Semantics. This expression represents a return from a method, and is defined in the Action Semantics via the use of an "escape". In the Denotational Semantics the escape is represented by the continuation function "retcnt". The expression "E" is evaluated as normal, and then the value returned by that evaluation is returned along with the current "sender" using "retcnt". The "sender" represents the object that called this method.

Continuations and expression continuations are defined as follows:

\[
\text{Cont: } \text{State} \rightarrow \text{Answer} \\
\text{ECont: } \text{Object} \rightarrow \text{Cont}
\]

A continuation takes as an argument a state in order to produce an answer. An expression continuation is similar, but we are also interested in a data value (i.e. an object). For this particular semantics we do not need any other type of continuation. However, we could imagine a semantics that would require other types of continuations, for example if we were interested in the bindings produced by statements. We may choose to model this by an extra argument in a continuation:

\[
\text{EnvCont: } \text{Environment} \rightarrow \text{Cont}
\]

5.3.2 Action Combinators

This section examines how some selected action combinators might be modelled in Denotational Semantics.
Chapter 5: Denotational Semantics

First we will look at the action combinator "then". Assume we have some actions, \(X\), \(A\) and \(B\). \(X\) has been defined such that the action \(A\) produces some form of data output, which is then used as an input to the action \(B\). \(B\) itself produces an output which is then used as the output of the whole action \(X\).

\[
X = \begin{cases} 
  A \\
  \text{then} \\
  B.
\end{cases}
\]

Let us also assume that \(X\), \(A\) and \(B\) can be modelled in Denotational Semantics by the functions \(X'\), \(A'\) and \(B'\) respectively such that they have the following types:

- \(X' : E\text{Cont} \rightarrow \text{Cont}\)
- \(A' : E\text{Cont} \rightarrow \text{Cont}\)
- \(B' : E\text{Cont} \rightarrow E\text{Cont}\)

where \(\text{Cont}\) is some form of continuation function and,

\[
E\text{Cont} = \text{Data} \rightarrow \text{Cont}
\]

In the Denotational semantics \(A'\) is defined such that it takes as an input an expression continuation. The result of executing \(A'\) can then be passed to this expression continuation. \(B'\) is defined so that it has an explicit argument that represents the data input that is implicit in the Action Semantics. We can now define \(X'\) as follows.

\[
X' \text{ cnt} = A' (B' \text{ cnt})
\]

\(X'\) calls the function \(A'\), and passed to it an expression continuation that expects some data argument and passes it to \(B'\). Compare this with the Action Semantics definition of \(X\), where an action \(A\) is first performed, and the data result is passed to the performance of the action \(B\).

For a second example assume we have the three actions \(W\), \(C\) and \(D\) which are modelled by \(W'\), \(C'\) and \(D'\) respectively. Let us further assume that \(W\) produces a pair
as a result of executing it which is formed by combining the data results of C and D, i.e. we have,

\[
W = \begin{cases} 
C \\
\text{and} \\
D.
\end{cases}
\]

W', C' and D' have the following types.

\[
\begin{align*}
W' & : \text{ECont} \rightarrow \text{Cont} \\
C' & : \text{ECont} \rightarrow \text{Cont} \\
D' & : \text{ECont} \rightarrow \text{Cont}
\end{align*}
\]

We can define W' as follows.

\[
W' \text{ cnt} = \\
C' (\lambda d_1. D' (\lambda d_2. \text{cnt} (d_1, d_2))) \\
or \\
W' \text{ cnt} = \\
D' (\lambda d_2. C' (\lambda d_1. \text{cnt} (d_1, d_2)))
\]

In Action Semantics the combinator "and" takes two sub-actions (here represented by C and D). The order of performance of these sub-actions is not defined. C could be completely performed before D; D could be completely performed before C; or there could be some interleaving of performance. This concept is difficult to represent in Denotational Semantics. The two versions of W' that have been presented only show the two cases of completely performing C' before D', and completely performing D' before C'. In cases where the outcome of performing W is not changed by the order of performance of C and D, it is reasonable to model W by one of the two forms of W' that have been shown.
Let us now consider the case where we have an action that iterates using the *unfold* action. In the following example we are only concerned with the flow of data (i.e. we ignore bindings etc). We define \( F \) such that:

\[
F = \text{unfold.}
\]

We can model \( F \) by a function \( F' \) so that:

\[
F' : (\text{Data} \rightarrow \text{ECont} \rightarrow \text{Cont}) \rightarrow \text{Data} \rightarrow \text{ECont} \rightarrow \text{Cont}
\]

\[
F' \text{ uz d ecnt} = \text{ uz d ecnt}
\]

The performance of the "unfold" action can be thought of as replacing "unfold" with some other action (previously specified using "unfolding"). In this example, calling the function \( \text{uz} \) represents this "replacing".

Now assume that we have the actions \( Z \) and \( E \) which can be modelled by \( Z' \), and \( E' \) respectively. The action \( Z \) requires a data input and produces a data output. The action \( F \) defined above is some sub-action of \( E \). \( Z \) is defined as,

\[
Z = \text{unfolding}
\]

\[
\quad E.
\]

Let us give the following types to \( Z' \) and \( E' \).

\[
E' : (\text{Data} \rightarrow \text{ECont} \rightarrow \text{Cont}) \rightarrow \text{Data} \rightarrow \text{ECont} \rightarrow \text{Cont}
\]

\[
Z' : \text{Data} \rightarrow \text{ECont} \rightarrow \text{Cont}
\]

We can therefore define \( Z' \) as follows.

\[
Z' = \text{fix}(E')
\]

The definition of \( E' \) would look something like the following
The argument \texttt{uz} is used to represent a recursive call to \texttt{E'}. This works because \texttt{E'} is "fixed" in the definition of \texttt{Z'}.

All of the above examples assume that we are only interested in the data flow, and that there are no other effects of actions. This is of course not normally the case - some actions might affect bindings, or send messages etc. However the principles of modelling these effects remain the same.

### 5.3.3 Primitive actions and yielders

The previous section explored how action combinators can be modelled using Denotational Semantics. This section looks at how we can model the primitive actions and yielders. Let us return to a previous example. Consider the following Action Semantics of the semantic function "evaluate".

\[
\text{evaluate \{ VarNam:Variable-Name \} = give the object stored in the cell bound to VarNam.}
\]

The primitive action here is "give". The effect of "give" is to give as a result the data item given as an argument. This semantic function can be modelled in Denotational Semantics as follows.

\[
\text{evaluate \{ VarNam:Variable-Name \} ecnt retcnt env st = ecnt (st (env VarNam)) st}
\]

The yielder "the object stored in the cell bound to VarNam" is modelled by "st (env VarNam)". This produces a data result that must be passed on as a result of the whole function in order to model "give". This is achieved by passing it as an argument to the given expression continuation.
5.4 Conclusions

This chapter has introduced some concepts commonly found in Denotational Semantics. Some of the common constructs found in Action Semantics have also been examined, and equivalent Denotational Semantics forms have been found.

Using the techniques discussed in this chapter we can (for some definitions of languages) start with an Action Semantics of that language and generate a Denotational Semantics. We should then be able to prove that these two semantics do in fact define the same language, using the techniques outlined in the following chapter. Once we have a Denotational Semantics of the language it is then much easier to make comparisons with other languages defined in Denotational Semantics, or even alternative definitions of the same language.

It is not possible from the work presented here to say in general that for all language definitions we can generate a Denotational Semantics based on the Action Semantics, or vice versa.

It is important to note that the Denotational Semantics generated using the methods discussed here will have a structure that closely follows the structure of the Action Semantics definition. This is important in the proof of equivalence that will be carried out in the next chapter.

The concepts that have been discussed here have been applied to the Action Semantics of Smalltalk that was given in Chapter 4. This has produced a Denotational Semantics of Smalltalk.
Chapter 6: On a Correspondence between Formal Semantics Definition Methods

6.1 Introduction

Hoare and Lauer in 1974 [HoL74], described a number of techniques for defining the semantics of languages, and then attempted to show for a particular language that a definition of the semantics written using one method could be shown to be equivalent to the same definition written using another method. This chapter follows Hoare and Lauer in attempting to show equivalence for a specific instance, across the modern notations of Denotational Semantics and Action Semantics. It does not attempt to generalise for all languages. The techniques used are then applied to one aspect of the two semantics definitions of Smalltalk presented earlier in this thesis.

The equivalence proofs that are presented in this chapter use the technique of structural induction [Bur69]. This requires that the two semantics definitions have a fundamentally compatible structure. Since the Denotational Semantics of Smalltalk was constructed from the Action Semantics version, they should hopefully be equivalent by construction - and have a compatible structure.

Hoare and Lauer also proposed that complementary semantic definitions of a language should be provided, i.e. that separate definitions using different frameworks should be given for the same language. These definitions can then be used for different purposes, but would be relatable due to the fact that it could be proved that the definitions were equivalent. Mosses [Mos92, pp. 5] states that one of the aims of
Chapter 6: On a Correspondence between Formal Semantics Definition Methods

Action Semantics is to avoid the need for such complementary descriptions. It is unclear at this stage whether this aim has been achieved successfully or not, although the experience of the author in producing this thesis is that the theoretician will have difficulty working with certain aspects of Action Semantics.

Only the relationship of equivalence is examined here. There are of course other relationships between the semantics that may be of interest. For example relative abstractness, complexity, length, readability etc.

A variation of the work shown in this chapter has also been presented in [Cas97].

6.2 A Simple Language

Action Semantics is a new framework compared with Denotational Semantics. Despite the fact that one of the stated objectives of Action Semantics is to avoid the need for complementary semantic descriptions, we must still be able to relate work carried out using the other more established frameworks to it. This will help us to evaluate how well it has done in meeting its objectives, and also allow us to apply new techniques used in other frameworks to it.

The first section of this chapter investigates the feasibility of proving the equivalence of two different definitions of a specific example language.

In order to illustrate the equivalence proofs a well known language that is already well defined could have been chosen. However, the proofs would be long and tedious, and would obscure the important point i.e. that it is possible to present such proofs at all. Therefore a very simple language will be used. This language is essentially the same as the one used by Hoare and Lauer [HoL74], and consists of just four different types of statement. The language exhibits no object oriented properties. The syntax and semantics of expressions are not given or discussed here. The language consists of an assignment; an iterative construct; a sequencer and a block instruction. The abstract syntax is as follows.
The semantic descriptions of this language make no use of a "store" variable, but instead everything is stored in an "environment". The result of a program is taken to be the final environment. Informally the semantics of this language are as follows.

1) $Id := E$, means that the value of the Expression, $E$, is evaluated in the current environment. As a result the statement produces a new environment which is the same as the old one except that the Identifier, $Id$, is now associated with the value of $E$.

2) $B * S$, means that the value of the Boolean, $B$, is evaluated in the current environment. If the value is false then this statement simply produces as a result the unchanged current environment. If the value is true then the statement $S ; B * S$ is executed, and the environment produced by these statements is given as a result.

3) $S_1 ; S_2$, means that the Statement, $S_1$, is executed in the current environment. Statement, $S_2$, is then executed in the environment produced by executing $S_1$. The result of this compound statement is the environment that was produced by the execution of $S_2$.

4) $( S )$, is simply a block instruction, i.e., the statement, $S$, is evaluated in the current environment. The result of executing $S$ is given as the result of the block.

The auxiliary function, the \texttt{value_of}, is required by the definitions. This is overloaded to evaluate both Boolean and Expression.
Chapter 6: On a Correspondence between Formal Semantics Definition Methods

**Definition 1**

\[
\text{the\_value\_of} :: \text{Expression} \times \text{Environment} \to \text{Value}
\]

\[
\text{the\_value\_of} :: \text{Boolean} \times \text{Environment} \to \text{Value}
\]

No formal description of how Expressions and Booleans are evaluated is given here. It is not necessary to give a formal definition since the function \text{the\_value\_of} is common to both of the semantic definitions. It would only serve to complicate matters if such a definition was given. It is enough to know that the operation to evaluate Expressions and Booleans remains the same.

The Denotational Semantics definition of the language is given below. This definition defines \text{d\_execute} which is used to evaluate statements. The domain of Value is left unspecified. The definition also requires the operator “fix” which constructs the least fixed point of its argument. A discussion of fixed points is given in section 5.2.5.

\[
\begin{align*}
\text{Environment} : & \quad \text{Identifier} \to \text{Value} \\
\text{d\_execute} : & \quad \text{Statement} \to \text{Environment} \to \text{Environment} \\
\text{X1}) \quad \text{d\_execute} [ Id := E ] env = & \quad \text{env}[Id \mapsto \text{the\_value\_of}(E, env)] \\
\text{X2}) \quad \text{d\_execute} [ B \ast S ] = & \quad \text{fix}(\Gamma) \\
& \text{where} \\
& \quad \Gamma (x) = \lambda env.\text{the\_value\_of}(B, env) = \text{true} \to \\
& \quad x (\text{d\_execute} S \ env), \\
& \quad env \\
\text{X3}) \quad \text{d\_execute} [ S_1 ; S_2 ] env = & \quad \text{d\_execute} S_2 (\text{d\_execute} S_1 env) \\
\text{X4}) \quad \text{d\_execute} [ (S) ] env = & \quad \text{d\_execute} [ S ] env
\end{align*}
\]

The Action Semantics description is given below. The definition of this language has been influenced by the design of the Denotational Semantics version for the purposes
Chapter 6: On a Correspondence between Formal Semantics Definition Methods

of making this example simpler to understand. This means that the description that is presented is somewhat unusual, in that it uses the declarative facet to describe the imperative aspects of the language. This is because the language uses an environment to store values rather than the traditional method of a store. It would not be difficult to present a language that additionally made use of a store. The presence of a store would not stop the production of equivalence proofs similar to those presented in this chapter, it would only make them longer and more complicated.

introduces: execute_

- execute_ :: Statement \rightarrow \text{action}

execute [ Id := E ] = 
    furthermore bind Id to the value of E.

execute [ B \ast S ] =
    unfolding
    
    check the value of B is true and then 
    execute S hence unfold
    or
    check the value of B is false and then rebind.

execute [ S_1 ; S_2 ] =
    execute S_1 hence execute S_2.

execute [ ( S ) ] =
    execute S.

Since Action Semantics requires no explicit variable to represent the environment, this aspect of the semantics is hidden. Since there is no explicit variable for the environment we have to use a slightly amended version of our function the_value_of called the\_value\_of. This simply takes as an argument the Boolean or Expression to evaluate. The environment is handed implicitly to it.
Chapter 6: On a Correspondence between Formal Semantics Definition Methods

6.3 Equivalence of Semantics Definition Methods

In order to discuss the equivalence of different definition methods, it is first necessary to define what is meant by equivalence. The important aspect of the semantics of a language is its externally observable behaviour. For example, it is unnecessary to know the details of how a variable is assigned a value in an assignment statement, it is merely enough to know how the state (or environment) changes. In our example language we are interested in changes in the environment. A statement is given an initial environment and gives a new environment as a result. The details of how the environment is calculated may change depending on the semantics definition method that is used. In order to define equivalence between the different semantics definition methods for a specific language it is important to identify those aspects of the semantics of the language that can be classed as externally observable. This externally observable behaviour must remain the same for them to be equivalent. In this example all externally observable behaviour can be seen in changes in the environment. The definition of equivalence within the context of this example is stated formally below.

Definition 2

\[ \text{Def}_1 \equiv \text{Def}_2 \quad \text{iff} \quad \forall S, \text{env}, \text{env}'. \]

\[ \text{Def}_1(S, \text{env}) = \text{env}' \Leftrightarrow \text{Def}_2(S, \text{env}) = \text{env}' \]

where \( \text{Def}_1 \) and \( \text{Def}_2 \) are of type \( \text{Statement} \times \text{Environment} \rightarrow \text{Environment} \), \( S \) is a \( \text{Statement} \) and \( \text{env}, \text{env}' \) are \( \text{Environments} \). \( \text{Def}_1 \) and \( \text{Def}_2 \) may be partial functions, and are undefined in the case of non-termination.

That is, given two different definitions of a language, \( \text{Def}_1 \) and \( \text{Def}_2 \), they are equivalent if and only if given any \( \text{Statement} \) and \( \text{Environment} \), they produce exactly the same \( \text{Environment} \) as a result. Similarly if a \( \text{Statement} \) diverges, then both \( \text{Def}_1 \) and \( \text{Def}_2 \) will be undefined. This is really just the same as extensional equality of functions.
Chapter 6: On a Correspondence between Formal Semantics Definition Methods

In general the externally observable behaviour for a language is the output that is produced given a specific program and its input. In the case of the above example the input is the start environment, and the output is the new environment. It would be possible to extend the definition of equivalence given above to other languages if we took the first element of the tuple accepted by Def\(_1\) and Def\(_2\) to be the program and the second element of the tuple to be the more general concept of input instead of an environment, and the result to be output.

6.3.1 Equivalence between the Denotational and Action Semantics of the Example Language

Action Semantics definitions are characterised by the fact that they use no explicit variable for representing stores and environments. However to enable us to compare an Action Semantics definition with a Denotational one, we must somehow make these stores and environments explicit. In order to achieve this we utilise the Operational Semantics description of Action Notation given by Mosses in [Mos92].

Mosses presents a complete specification of a class of algebras within which Action Notation works. The remainder of this section makes extensive use of the definitions as laid out in [Mos92], and for the sake of brevity we do not repeat them here. It is assumed that the Denotational Semantics given earlier in this chapter also operates within these algebras so that we have some common basis with which to compare them. This should not be difficult to achieve. Mosses [Mos92, Appendix C] presents a complete operational semantics of Action Notation using this class of algebras. We would have to extend the algebras with the rules for lambda calculus. Suitable rules to do this are presented in [Bak84].

The language which is being used here only uses an environment (i.e. no store). The representation of an environment in the Denotational definition and the Action definition are different but obviously are closely related. In order for us to compare the two it is assumed that a function \texttt{conv} has been defined such that:
Definition 3

\[ \text{conv}(env[Id \mapsto v]) = \text{overlay}( \text{map of } Id \text{ to } v, \text{conv}(env)) \]

and \[ \text{conv}(\lambda x.\bot) = \text{empty-map} \]

where \( env \) is an Environment, \( Id \) is an Identifier and \( v \) is a Value. \( \text{overlay} \) and \( \text{map \ of} \ _ \ \text{to} \ _ \) are as defined in [Mos92, Appendix E].

The Structural Operational Semantics of the Action Notation [Mos92, Appendix C], describes the semantics of a small kernel of Action Notation. The rest of the Action Notation is then described in terms of this kernel. To make the proof easier the same Action Semantic definition as given earlier in this chapter is shown below, but converted into kernel Action Notation. It is this new definition that is used in the proof. Appendix D presents a proof that the two semantic definitions are equivalent. The proof is quite straight forward and could in fact be included within the main proof of Theorem 1 below. However it is presented here as a separate stage to make the main proof simpler to understand.
Chapter 6: On a Correspondence between Formal Semantics Definition Methods

introduces: execute

* execute :: Statement → action

Y1) execute [ Id := E ] =
  produce current bindings
moreover
  bind Id to the value of E.

Y2) execute [ B * S ] =
  unfolding
  |  give true & ( the value of B is true ) then give ()
  and then
  |  execute S
  hence
  |  unfold
  of
  |  give true & ( the value of B is false ) then give ()
  and then
  |  produce current bindings.

Y3) execute [ S_1 ; S_2 ] =
  execute S_1 hence execute S_2.

Y4) execute [ ( S ) ]
  execute S.

The function name, Den, is used here to refer to the Denotational Semantics definition of this language, and the function name, Act, is used to refer to the Action Semantics definition of this language, i.e.,
Chapter 6: On a Correspondence between Formal Semantics Definition Methods

Definition 4

\[ \text{Den}( S, env ) = env' \iff d\_execute S\ env = env' \]

and

\[ \text{Act}( S, env ) = env' \]

\[ \iff \text{run}( \text{given}( \text{received}( [\text{execute}\ S ] , \text{conv}( env ) ) , d:\text{data} ), l:\text{local-info} ) \]

\[ \geq ( [ "\text{completed}\ d'\text{data}\ \text{conv}( env' ) ] , l':\text{local-info} , c:\text{commitment} ) \]

where \text{Den} and \text{Act} are of type \text{Statement} \times \text{Environment} \rightarrow \text{Environment}, \ S is a \text{Statement} and \ env, env' are \text{Environments}. \text{run}, \text{given}, \text{received}, \text{data}, \text{local-info} and \text{commitment} are as defined in [Mos92, Appendix C].

Informally \text{run} is a function defining performance of an action. \text{given} models passing transient data to an action, and \text{received} models passing scoped information to an action. It is not necessary to have an understanding of the semantics of \text{local-info} and \text{commitment} in order to be able to understand the following proofs. The symbol "\geq" indicates that the item following it is a sub-sort of the item preceding it.

Mosses [Mos92] defines \text{run} shown below. \text{stepped} is a function that performs a single step of an action.

```
introduces: run _, stepped _ .

• run _ :: state \rightarrow\ Terminated, local-info, commitment)

(1) stepped(A, l) \geq (A':Intermediate, l':local-info, c':commitment);
    run(A', l') \geq (A":Terminated, l":local-info, c":commitment) \Rightarrow
    run(A:\text{Acting}, l:\text{local-info}) \geq (A", l", \text{concatenation}(c',c")) .

(2) stepped(A, l) \geq (A':Terminated, l':local-info, c':commitment) \Rightarrow
    run(A:\text{Acting}, l:\text{local-info}) \geq (A', l', c') .

• stepped _ :: state \rightarrow (state, commitment) .

(3) stepped(A:\text{Terminated}, l:\text{local-info}) = \text{nothing} .
```

The theorem for the equivalence of Denotational Semantics and Action Semantics is then as follows,
Chapter 6: On a Correspondence between Formal Semantics Definition Methods

Theorem 1

Den = Act

Before considering this theorem further it is first necessary to show two basic properties of the operational semantics of action notation.

**Lemma 1 - Breaking down a run into a series of discrete steps**

\[
\text{run}_n(A:\text{Intermediate}, I:\text{local-info}) \geq \\
(A^{n+1}:\text{Terminated}, I^{n+1}:\text{local-info}, c^n:\text{commitment}) \\
\Leftrightarrow \text{stepped}(A, I) \geq (A':\text{Intermediate}, I':\text{local-info}, c:\text{commitment}) \land \\
\text{stepped}(A', I') \geq (A'':\text{Intermediate}, I'':\text{local-info}, c':\text{commitment}) \land \\
\ldots \\
\text{stepped}(A^{n-1}, I^{n-1}) \geq (A^n:\text{Intermediate}, I^n:\text{local-info}, c^{n-1}:\text{commitment}) \land \\
\text{stepped}(A^n, I^n) \geq (A^{n+1}, I^{n+1}, c^n)
\]

where "run\_n" is the relation "run" defined by n applications of axiom (1) in the definition of "run" given above.

This lemma basically states that a "run" of an action semantic description can be decomposed into a series of discrete steps. The proof of this lemma is deferred to Appendix E.

**Lemma 2 - Provision of transient and scoped information is order independent**

given( received(A:Action, b:bindings), d:data )
\[\Leftrightarrow\] received( given(A, d), b )

122
In the structural operational semantics of action notation "given" is used to provide an action with transient information, whilst "received" is used to provide it with scoped information. The lemma states that it does not matter in which order the transient and scoped information is presented to the action. Again the proof of this lemma is deferred to Appendix E.

Lemmas 3-6 constitute the main part of the proof of the theorem. Each of the four possible types of statement are considered in turn.

**Lemma 3 - Assignment**

\[
(d\_execute \ [ Id := E \ ] \ env = env') \\
\iff \ \text{run( } \text{given( } \text{received( } \text{execute } \ [ \text{Id} := \text{E} \ ] \ , \ \text{conv( env )} \) , \ d: data ) ,} \\
\text{ } l: \text{local-info } \} \\
\geq ( [ "completed" \ d': \text{data conv( env' )} , \ l': \text{local-info} , \ c: \text{commitment} ] )
\]

**Proof**

1) \ \text{run( given( } \text{received( } \text{execute } \ [ \text{Id} := \text{E} \ ] \ , \ \text{conv( env )} \) , \ d: data ) ,} \\
\text{ } l: \text{local-info )} \\
\iff \ \text{run( given( } \text{received( [ [ } \text{"produce" } \text{"current bindings" } ] \ ) } \text{"moreover"} \\
[ [ \text{"bind" } \text{Id} \text{"to" } \text{"the value of" } \text{E} ] ] \ , \ \text{conv( env )} , \ d ) , \ l)( \text{ from 6.3.1 Y1)}

2) \ \text{run( given( received( [ [ } \text{"produce" } \text{"current bindings" } ] \ ) } \text{"moreover"} \\
[ [ \text{"bind" } \text{Id} \text{"to" } \text{"the value of" } \text{E} ] ] \ , \ \text{conv( env )} , \ d: \text{data ) ,} \\
\text{ } l: \text{local-info )} \\
\geq ( [ "completed" ( \text{overlay( map Id to the_value_of( E, env ) , conv( env ) )} ] ) , \\
\text{ } l': \text{local-info} , \ c: \text{commitment} )

(See Appendix F)
3) \[
\text{run( given( received( } \exists \text{ execute } [ \text{Id} := E ] ] , \text{conv( env }} ) ) , d: \text{data } ), \\
\text{l:local-info})
\]
\[
\geq ( \exists \text{ "completed" } d': \text{data conv( env”) ] , l':local-info, c:commitment) }
\]
\[
\Leftrightarrow \text{conv( env”) } = \text{overlay( map Id to the_value_of( E, env }, \text{conv( env }) )}
\]
\[
\text{ ( from 1; 2 )}
\]
\[
\Leftrightarrow \text{env”} = \text{env}[\text{Id } \mapsto \text{the_value_of( E, env}] 
\]
\[
\text{( from definition 3 )}
\]
\[
( \text{d_execute} [ \text{Id} := E ] \text{env} = \text{env’})
\]
\[
\Leftrightarrow \text{env’} = \text{env}[\text{Id } \mapsto \text{the_value_of( E, env}] 
\]
\[
\text{ (from 6.2 X1)}
\]

So,
\[
( \text{d_execute} [ \text{Id} := E ] \text{env} = \text{env’})
\]
\[
\Leftrightarrow \text{run( given( received( } \exists \text{ execute } [ \text{Id} := E ] ] , \text{conv( env }} ) ) , d: \text{data } ), \\
\text{l:local-info})
\]
\[
\geq ( \exists \text{ "completed" } d': \text{data conv( env’) ] , l’:local-info, c:commitment })
\]

\[\square\]

**Lemma 4 - Block**

\[
( ( \text{d_execute} [ S ] \text{env}_1 = \text{env’}_1 )
\]
\[
\Leftrightarrow \text{run( given( received( } \exists \text{ execute } [ S ] ] , \\
\text{conv( env}_1 ) ), d_1: \text{data }, l_1: \text{local-info})
\]
\[
\geq ( \exists \text{ "completed" } d_1': \text{data conv( env’}_1 ) ] , l_1’:local-info, \\
c_1: \text{commitment })
\]
\[
\Rightarrow \quad ( ( \text{d_execute} [ ( S ) ] \text{env} = \text{env’})
\]
\[
\Leftrightarrow \text{run( given( received( } \exists \text{ execute } [ ( S ) ] ] , \\
\text{conv( env }) ), d: \text{data }, l: \text{local-info})
\]
\[
\geq ( \exists \text{ "completed" } d': \text{data conv( env’) ] , l’:local-info, c:commitment })
\]

124
Chapter 6: On a Correspondence between Formal Semantics Definition Methods

Proof

Follows directly from 6.2(X4) and 6.3.1(Y4).

□

Lemma 5 - Sequence

\[
\begin{align*}
&\left( (\text{d}_\text{execute} \llbracket S_1 \rrbracket \text{env}_1 = \text{env}_1') \right) \\
&\iff \text{run}(\text{given} (\text{received}(\llbracket \text{execute} \llbracket S_1 \rrbracket \rrbracket , \text{conv}(\text{env}_1)), \text{d}_1:\text{data}), \\
&\quad \text{l}_1:\text{local-info}) \\
&\quad \geq (\llbracket \text{"completed" } \text{d}_1':\text{data conv}(\text{env}_1') \rrbracket , \text{l}_1':\text{local-info}, \\
&\quad \text{c}_1:\text{commitment})
\end{align*}
\]

\land

\[
\begin{align*}
&\left( (\text{d}_\text{execute} \llbracket S_2 \rrbracket \text{env}_2 = \text{env}_2') \right) \\
&\iff \text{run}(\text{given} (\text{received}(\llbracket \text{execute} \llbracket S_2 \rrbracket \rrbracket , \text{conv}(\text{env}_2)), \text{d}_2:\text{data}), \\
&\quad \text{l}_2:\text{local-info}) \\
&\quad \geq (\llbracket \text{"completed" } \text{d}_2':\text{data conv}(\text{env}_2') \rrbracket , \text{l}_2':\text{local-info}, \\
&\quad \text{c}_2:\text{commitment})
\end{align*}
\]

\Rightarrow

\[
\begin{align*}
&\left( (\text{d}_\text{execute} \llbracket S_1 ; S_2 \rrbracket \text{env} = \text{env}') \right) \\
&\iff \text{run}(\text{given} (\text{received}(\llbracket \text{execute} \llbracket S_1 ; S_2 \rrbracket \rrbracket , \text{conv}(\text{env})), \text{d}:\text{data}), \\
&\quad \text{l}:\text{local-info}) \\
&\quad \geq (\llbracket \text{"completed" } \text{d}:\text{data conv}(\text{env}') \rrbracket , \text{l}:\text{local-info}, \\
&\quad \text{c:commitment})
\end{align*}
\]

Proof

\[
(\text{d}_\text{execute} \llbracket S_1 \rrbracket \text{env} = \text{env}''') \land (\text{d}_\text{execute} \llbracket S_2 \rrbracket \text{env}''' = \text{env}')
\]
\[
\iff \text{d}_\text{execute} \llbracket S_1 ; S_2 \rrbracket \text{env} = \text{env}' \quad \text{(from 6.2 X3 )}
\]
Also,

\[(\text{run( given( received( \text{execute } [S_1] ], \text{conv( env}) ), d: data ), l: local-info )} \]

\[\geq (\text{"completed" } d''': data \text{ conv( env''' ) }, l''' : local-info, c': commitment )\) \wedge

\[(\text{run( given( received( \text{execute } [S_2] ], \text{conv( env''' ) ), d ), l'' )} \]

\[\geq (\text{"completed" } d''': data \text{ conv( env''' ) }, l'' : local-info, c: commitment )\)

\(\Leftrightarrow (\text{run( given( received( \text{execute } [S_1] ], \text{conv( env}) ), d ), l )} \]

\[\geq (\text{"completed" } d'' \text{ conv( env'' ) }, l'', c' )\) \wedge

\[(\text{stepped( given( received( \text{execute } [S_2] ], \text{conv( env'' ) }, d ), l'' )} \]

\[\geq (A_2 : \text{Intermediate}, l_2 : local-info, c_2 : commitment )\) \wedge

\[(\text{stepped}(A_2, l_2) \geq (A_2 : \text{Intermediate}, l_2 : local-info, c_2 : commitment )\) \wedge

\[\ldots\]

\[(\text{stepped}(A_2^{n-2}, l_2^{n-2}) \geq (A_2^{n-1} : \text{Intermediate}, l_2^{n-1} : local-info, c_2^{n-1} : commitment )\) \wedge

\[(\text{stepped}(A_2^{n-1}, l_2^{n-1}) \geq (\text{"completed" } d''' \text{ conv( env''' ) }, l', c))\)

( lemma 1 )
\[\Leftrightarrow (\text{run( given( received( [\text{execute } S_1 ] ] , conv( env ))), d), l}) \]
\[\geq ( [\text{"completed" } d'' \text{ conv( env'')} ] ] , l'', c') \) \]
\[\text{stepped( given( received( [\text{execute } S_2 ] ] , conv( env''))) , d), l'')} \]
\[\geq ( A_2, l_2, c_2 ) ( \text{stepped}( A_2, l_2 ) \geq ( A_2', l_2', c_2' )) \]
\[\text{stepped}( A_2^{n-2}, l_2^{n-2} ) \geq ( A_2^{n-1}, l_2^{n-1}, c_2^{n-1} ) \] 
\[\text{stepped( ["completed" } d' \text{ empty-map "and" } A_2^{n-1} ] ] , l_2^{n-1} ) \]
\[\geq ( \text{simplified } ["completed" } d'' \text{ empty-map "and" }
\[\text{"completed" } d'\prime, \text{conv( env'')} ] ] , l', c ) \]
\[= ( [\text{"completed" } (d'', d'') ] ) \]
\[\text{(disjoint-union( empty-map, conv( env''))) ] ] , l', c ) \]
\[= ( [\text{"completed" } d' \text{ conv( env'')} ] ] , l', c ) \]
\[\Rightarrow (\text{run( given( received( [\text{execute } S_1 ] ] , conv( env ))), d), l}) \]
\[\geq ( [\text{"completed" } d'' \text{ conv( env'')} ] ] , l'', c') \) \]
\[\text{stepped( given( received( ["execute" S_2 ] ] , conv( env''))) ] ] , l'') \]
\[\geq ( [\text{"completed" } d' \text{ conv( env'')} ] ] , l', c ) \] (lemma 1)
\[\Rightarrow (\text{run( given( received( [\text{execute } S_1 ] ] , conv( env ))), d), l}) \]
\[\geq ( \text{"completed", } d'\prime, \text{conv( env'')} , l'', c') ) \] \]
\[\text{stepped( given( received( ["execute" S_2 ] ] , conv( env''))) ] ] , l'') \]
\[\geq ( [\text{"completed" } d' \text{ conv( env'')} ] ] , l', c ) \] (lemma 2)
Chapter 6: On a Correspondence between Formal Semantics Definition Methods

\[ \iff \text{stepped( given( received( \{ \text{execute} S_1 \} ) , conv( env ) ), d ), l }) \geq ( A_I:Intermediate, I_J:local-info, c_J:commitment ) \iff ( \text{stepped}( A_I, I_J ) \geq ( A_I':Intermediate, I_J':local-info, c_J':commitment ) ) \iff \]

\[ ( \text{stepped}( A_I^{n-2}, I^{n-2} ) \geq ( A_I^{n-1}:Intermediate, I^{n-1}:local-info, c^{n-1}:commitment ) ) \iff ( \text{stepped}( A_I^{n-1}, I^{n-1} ) \geq ( "completed", d", conv( env""), l", c'" ) ) \iff ( \text{run( } \{ \text{"completed" } d" empty-map "and" received(given( \{ \"execute" S_2 \} , d),conv( env"")) \} , l" ) \geq ( \{ \text{"completed" } d' conv( env"" ) \} , l', c ) ) \iff \text{lemma 1} \iff ( \text{stepped( given( received( \{ \text{execute} S_1 \} ) , conv( env ) ), d ), l }) \geq ( A_I, I_J, c_J ) \iff ( \text{stepped}( A_I, I_J ) \geq ( A_I', I_J', c_J' ) ) \iff \]

\[ ( \text{stepped}( A_I^{n-2}, I^{n-2} ) \geq ( A_I^{n-1}, I^{n-1}, c^{n-1} ) ) \iff ( \text{run( } \{ A_I^{n-1} "hence" given( \{ \"execute" S_2 \} , d ) \} , I^{n-1} ) \geq ( \{ \text{"completed" } d' conv( env"" ) \} , l', c ) ) \iff ( [\text{Mos92, C.3.3.2.1 (5); C.3.3.2.2 (13)}] ) \iff \text{run( } \{ \text{given( received( \{ \"execute" S_1 \} , conv( env ) ), d ) "hence" given( \{ \"execute" S_2 \} , d ) \} , I ) \geq ( \{ \text{"completed" } d' conv( env"" ) \} , l', c ) ) \iff \text{lemma 1} \iff \text{run( } \{ \text{given( received( \{ \"execute" S_1 \}, \{ S_2 \} , conv( env ) ), d ) \} , I ) \geq ( \{ \text{"completed" } d' conv( env"" ) \} , l', c ) ) \iff ( 6.3.1 Y3; [\text{Mos92, C.3.3.2.4 (4); C.3.3.2.5 (5)}] ) \]

\[ \text{hence,} \]

\[ env"""" = env"""" \]

( from assumption )
And so,

\[ env' = env'' \]  

(from assumption)

The lemma is therefore proved.

\( \square \)

**Lemma 6 - Iteration**

\[
((d_{execute}[S] env_I = env_I') \\
\Rightarrow run(given(received([execute[S] ] , conv(env_I)), d_I: data), l_I: local-info) \\
\geq (["completed" d_I: data conv(env_I')] , l_I: local-info, c_I: commitment) \\
\Rightarrow ((d_{execute}[B*S] env = env') \land \\
\Rightarrow run(given(received([execute[B*S] ] , conv(env')), d: data), l: local-info) \\
\geq (["completed" d': data conv(env')] , l': local-info, c': commitment))
\]

**Proof**

Let P(n) be the property,

\[
((d_{execute}[S] env_I = env_I') \\
\Rightarrow run(given(received([execute[S] ] , conv(env_I)), d_I: data), l_I: local-info) \\
\geq (["completed" d_I: data conv(env_I')] , l_I: local-info, c_I: commitment))
\]
Chapter 6: On a Correspondence between Formal Semantics Definition Methods

\[
\Rightarrow ((\text{d\_execute}_n [ B \ast S ] env = env')
\]

\[
\Leftrightarrow \text{run}_n( \text{given}( \text{received}([\text{execute} [ B \ast S ]], \text{conv}(env)), d:\text{data}),
\]

\[
I:\text{local\_info})
\]

\[
\geq ([ "\text{completed}" d:\text{data conv}(env')], I':\text{local\_info},
\]

\[
c':\text{commitment})
\]

Where \text{d\_execute}_n [ B \ast S ] env = env', implies that \text{d\_execute} iterates with the \text{value\_of}( B, env ) evaluating to true \( n \) times, and then the \text{value\_of}( B, env ) evaluates to false. Similarly with \text{run}_n(...).

1) \text{P}(0)

\[
\text{d\_execute}_0 [ B \ast S ] env = env'
\]

\[
\Leftrightarrow \Gamma (\text{d\_execute} [ B \ast S ] ) env = env'
\]

\[
\Leftrightarrow (\text{false} \rightarrow \text{d\_execute} [ B \ast S ] \text{d\_execute} S env, env') = env'
\]

\[
\Leftrightarrow env = env'
\]

\[
\text{run}_0( \text{given}( \text{received}([\text{execute} [ B \ast S ]], \text{conv}(env)), d:\text{data}), I:\text{local\_info})
\]

\[
\geq ([ "\text{completed}" d:\text{data conv}(env' )], I':\text{local\_info}, c':\text{commitment})
\]

\[
\Leftrightarrow \text{run}_0( \text{given}( \text{received}([\text{execute} [ B \ast S ]], \text{conv}(env)), d ), I)
\]

\[
\geq ([ "\text{completed}" () \text{conv}( env )], I, \text{uncommitted})
\]

\[
\Leftrightarrow \text{conv}( env ) = \text{conv}( env' )
\]

\[
\Leftrightarrow env = env''
\]

So,

\[
((\text{d\_execute} [ S ] env_1 = env_1')
\]

\[
\Leftrightarrow \text{run}( \text{given}( \text{received}([\text{execute} [ S ]], \text{conv}(env_1)), d_1:\text{data}),
\]

\[
I_1:\text{local\_info})
\]

\[
\geq ([ "\text{completed}" d_1:\text{data conv}(env_1')], I_1':\text{local\_info},
\]

\[
c_1:\text{commitment})
\]

130
Chapter 6: On a Correspondence between Formal Semantics Definition Methods

\[ ((\text{d\_execute}_0 [ B \ast S ]\ env = env') )
\]
\[ \Rightarrow \text{run}_0(\ \text{given}(\ \text{received}( [\text{execute} [ B \ast S ] ] ,\text{conv}( env ) ), d:\text{data} ), \]
\[ l:\text{local-info} )
\]
\[ \geq ( [ "\text{completed}" d':\text{data conv}( env' ) ], l':\text{local-info}, \]
\[ c':\text{commitment} ) )
\]

1) \( P(n) \Rightarrow P(n+1) \)

\[ \text{d\_execute}_{n+1} [ B \ast S ]\ env = env' \]
\[ \iff \Gamma( \text{d\_execute}_n [ B \ast S ]\ env = env' ) \quad \text{(from 6.2 X2)}
\]
\[ \iff ( \text{true} \rightarrow \text{d\_execute}_n [ B \ast S ]\ d\_execute S\ env, env') = env' \quad \text{(from 6.2 X2)}
\]
\[ \iff ( \text{d\_execute}_n [ B \ast S ]\ d\_execute S\ env ) = env' \]
\[ \iff ( \text{d\_execute}_n [ B \ast S ]\ env'' ) = env' \wedge \]
\[ ( \text{d\_execute} S\ env ) = env'' \]

\[ \text{run}_{n+1}(\ \text{given}(\ \text{received}( [\text{execute} [ B \ast S ] ] ,\text{conv}( env ) ), d:\text{data} ), \]
\[ l:\text{local-info} )
\]
\[ \geq ( [ "\text{completed}" d':\text{data conv}( env' ) ], l':\text{local-info}, c':\text{commitment} )
\]
\[ \iff \text{run}( [\text{given}(\ \text{received}( [\text{execute} S ] ,\text{conv}( env ) ), d ) ], l )
\]
\[ \geq ( [ "\text{completed}" d'':\text{data conv}( env''' ) ], l'':\text{local-info}, c'':\text{commitment} ) \land \]
\[ \text{run}_n(\ \text{received}(\ \text{given}( [\text{execute} [ B \ast S ] ] , d ),\text{conv}( env''' ) ), l'' )
\]
\[ \geq ( [ "\text{completed}" d'':\text{data conv}( env''' ) ], l'', c' )
\]
\[ ( \text{where } d'=(d'', d'''); \text{see Appendix F} )
\]

Hence,

\[ env''' = env'''' \]
\[ \text{(from assumption)} \]

So,

\[ ((\text{d\_execute} [ S ]\ env_1 = env_1') )
\]
\[ \iff \text{run}(\ \text{given}(\ \text{received}( [\text{execute} [ S ] ] ,\text{conv}( env_1 ) ), d_1:\text{data} ), \]
\[ l_1:\text{local-info} )
\]
\[ \geq ( [ "\text{completed}" d_1':\text{data conv}( env_1' ) ], l_1':\text{local-info}, \]
\[ c_1':\text{commitment} ) )
\]
Chapter 6: On a Correspondence between Formal Semantics Definition Methods

\[ \Rightarrow ( ( \text{d\_execute}_{n+1} \downarrow B * S \uparrow \text{env} = \text{env'}) \]
\[ \Leftrightarrow \text{run}_{n+1}( \text{given}( \text{received}( ( \text{execute} ( B * S ) ) , \text{conv}( \text{env} ) ) ,
\begin{align*}
&d:\text{data}, l:\text{local-info} \\
\geq ( & ( "\text{completed}\" d':\text{data} \text{conv}( \text{env}' ) ) , l':\text{local-info},
\begin{align*}
&c':\text{commitment} ) \\
&\text{(from assumption and P(n))}
\end{align*}
\end{align*}
\]

Therefore the lemma is proved by mathematical induction.

\[ \square \]

The theorem that the Denotational Semantics and Action Semantics of our example language are equivalent, therefore follows directly from structural induction and lemmas 3-6.

\[ \square \]

6.4 Equivalence of the Two Semantics Definitions of Smalltalk

Chapter 4 gave a definition of Smalltalk using Action Semantics. It would be useful to be able to relate the ideas and techniques that were used in that chapter to other techniques that have been used for other formal semantics of Smalltalk. Many of these previous attempts have been in the framework of Denotational Semantics. A Denotational Semantics of Smalltalk with the same structure as the Action Semantics given in Appendix B is presented in Appendix C. This Denotational Semantics should hopefully be equivalent to the Action Semantics of Smalltalk and could be used to relate it to other work on the semantics of Smalltalk. However for this to be useful we need to be able to prove that the two semantics really are equivalent.

It should be possible to demonstrate the equivalence of the two versions of the semantics, due to the fact that the Denotational Semantics was generated from the Action Semantics, i.e. they should be equivalent by construction. This means that the
structures should be compatible. This chapter attempts to show that they are equivalent for one particular aspect of the language, i.e. the sending of messages to objects. This will allow us to:

- Demonstrate that our message sending system developed in the Action Semantics is the same as that given in the Denotational Semantics.
- To provide a case study and example of the techniques outlined in the first part of this chapter.
- To provide a stepping stone that can be used to compare our Action Semantics of Smalltalk against other approaches.
- Further our knowledge of the theory of Action Semantics.

The message sending system has been chosen for this example as the author considers this to be the most interesting aspect of the language. The way that messages are sent and methods discovered utilises aspects of the language that are purely object-oriented. The rest of the semantics could also be proved equivalent. However this would serve little purpose since it would merely repeat the ideas and techniques that are used in this chapter.

### 6.4.1 Defining Equivalence

The definition of equivalence that is used here is broader than that given in the first part of this chapter. Here we say that a given semantics $\text{Def}_1$ will take some arguments and produce a result. Similarly a second semantics $\text{Def}_2$ will also take some arguments and produce a result. The two semantics $\text{Def}_1$ and $\text{Def}_2$ are equivalent if and only if, given the same arguments (for all possible values of arguments) they produce the same result.

**Definition 5. General Equivalence of Semantics Definitions**

$$\text{Def}_1 = \text{Def}_2 \iff \forall \text{args}, \text{result}. \text{Def}_1(\text{args}) = \text{result} \iff \text{Def}_2(\text{args}) = \text{result}$$
This definition of equivalence effectively ignores the structure of the actual semantics definitions themselves, and is only interested in the inputs and corresponding outputs. This is in fact the situation that we want; we are only interested in the observable behaviour of programs written in a specific language. Note that in general it may not be possible to prove equivalence using the techniques described here - the semantics definitions have to be of compatible structure so as to facilitate the use of structural induction.

6.4.2 Converting Environments and Stores

In order for us to be able to compare the Action Semantics and Denotational Semantics of Smalltalk, we must be able to examine the effects on an environment of executing Smalltalk code. The structure of an environment in the Action Semantics and Denotational Semantics of Smalltalk are different, but equivalent (isomorphic). This is not proved here, but instead we assume the existence of a function (isomorphism) "conv" that converts between the two. The Action Semantics equivalent of a Denotational Semantics environment "env" is written as "conv(env)".

For example one important rule for "conv" is its effect on the Action Notation function "overlay". The equivalent of this in our Denotational Semantics is the "[]" notation. Therefore for two environment \( e_1 \) and \( e_2 \), the following is true.

\[
\text{overlay}(\text{conv}(e_1), \text{conv}(e_2)) = \text{conv}(e_2[e_1])
\]

In a similar way "conv" is overloaded to convert between the different representations of a store in the Denotational Semantics and the Action Semantics.

6.4.3 Semantics of Method Calls

With the Action Semantics definition of Smalltalk, the definition of method calls is represented by the action "call the method _ with arguments _ at _". The definition of
this is given in Appendix B. To put this in terms of definition 5 we define a function \( \text{Act} \text{call-method} \) as follows.

**Definition 6. Action Semantics of "call-method"**

\[
\text{Act} \text{call-method}(\text{tok}, \text{args}, \text{ob}, \text{env}, \text{state}) = \text{"failed" } \Rightarrow \\
\text{run(} \\
\text{given(} \\
\text{received(} \text{call the method tok:token with arguments args:argument-list at} \\
\text{ob:object } \text{, conv(env):bindings),} \\
\text{d:data),} \\
\text{r:redirections, conv(state):storage, l:li-tail}) \\
\geq \text{("failed", l:local-info, c:commitment)} \\
\text{)} \\
\text{)} \\
\text{)} \\
\text{)}
\]

\[
\text{Act} \text{call-method}(\text{tok}, \text{args}, \text{ob}, \text{env}, \text{state}) = \text{("completed", res, state') } \Leftrightarrow \\
\text{run(} \\
\text{given(} \\
\text{received(} \text{call the method tok:token with arguments args:argument-list at} \\
\text{ob:object } \text{, conv(env):bindings),} \\
\text{d:data),} \\
\text{r:redirections, conv(state):storage, l:li-tail}) \\
\geq \text{("completed", res:object conv(env')}, \text{r':redirections, conv(state'), l':li-tail, c:commitment)} \\
\text{)} \\
\text{)} \\
\text{)} \\
\text{)}
\]

\[
\text{Act} \text{call-method}(\text{tok}, \text{args}, \text{ob}, \text{env}, \text{state}) = \text{("returning", res, sndr, state') } \Leftrightarrow \\
\text{run(} \\
\text{given(} \\
\text{received(} \text{call the method tok:token with arguments args:argument-list at} \\
\text{ob:object } \text{, conv(env):bindings),} \\
\text{d:data),} \\
\text{r:redirections, conv(state):storage, l:li-tail}) \\
\geq \text{("escaped" (res, sndr):data, r':redirections, conv(state'), l':li-tail, c:commitment)}
\]
where $\text{Act}_{\text{call-method}}$ is of type

$\text{token} \times \text{argument-list} \times \text{object} \times \text{Environment} \times \text{State} \rightarrow$

"failed" +

("completed" $\times$ object $\times$ State) +

("returning" $\times$ object $\times$ Integer $\times$ State)

run, given and received are as described in section 6.3.1. bindings represent the current scoped information. data represents the current transient information. storage represents the current stable information, i.e the state.

li-tail is defined such that

$\text{local-info} = (\text{redirections}, \text{storage}, \text{li-tail})$

An understanding of, redirections, commitment and local-info is not required for this section. All of these are defined formally in [Mos92, Appendix C].

In the Denotational semantics of Smalltalk there is a function "call-method", which should be equivalent to the Action Semantics version. We define a function $\text{Den}_{\text{call-method}}$ in terms of the function "call-method".

**Definition 7. Denotational Semantics of "call-method"**

$\text{Den}_{\text{call-method}}(\text{tok}, \text{args}, \text{ob}, \text{env}, \text{state}) = \text{"failed"}$

$\iff \text{call-method ecnt retcnt (tok, class, args) ob env state = } 1$

$\text{Den}_{\text{call-method}}(\text{tok}, \text{args}, \text{ob}, \text{env}, \text{state}) = \text{("completed", res, state')}$

$\iff \text{call-method ecnt retcnt (tok, class, args) ob env state = ecnt res state'}$
Chapter 6: On a Correspondence between Formal Semantics Definition Methods

\[ \text{Den}_{\text{call-method}}(\text{tok, args, ob, env, state}) = ("\text{returning}", \text{res, sndr, state}) \]

\[ \Rightarrow \text{call-method cnt retcnt (tok, class, args) ob env state} = \text{retcnt (res, sndr) state'} \]

where \( \text{Den}_{\text{call-method}} \) is of type

\[ \text{token} \times \text{argument-list} \times \text{object} \times \text{Environment} \times \text{State} \rightarrow \\
\text{"failed" +} \\
\text{("completed" \times \text{object} \times \text{State} ) +} \\
\text{("returning" \times \text{object} \times \text{Integer} \times \text{State})} \]

Given these two definitions, the theorem for equivalence is then:

**Theorem 2. Equivalence of Action and Denotational Semantics of "call-method"**

\[ \text{Act}_{\text{call-method}} = \text{Den}_{\text{call-method}} \]

In order to prove this we must examine each of the possible outcomes from performing a method call. There are four possible outcomes from performing a method call:

- The method is not defined within the object, or the abstraction fails
- The abstraction completes normally
- The abstraction returns to this object
- The abstraction returns but not to this object

The proofs for each of these four cases are shown below.

137
Proof

Case 1: The method is not defined within the object or the abstraction fails

call-method ecnt retcnt sel args ob env st
= (methods-of ob st sel) ecnt retcnt' (ivd, md, iv, m) (succ (env sender)) args st
    (from defn. of call-method)
= ⊥
    (from assumption)
⇒ Den_{call-method}(tok, args, ob, env, st) = "failed"
    (from defn. of Den_{call-method})

run(given( received([I call the method tok:token with arguments args: argument-list at ob:object [I , conv(env)), d:data), r:redirections, conv(st), l:li-tail)
⇒ Act_{call-method}(tok, args, ob, env, st) = "failed"
    (from Appendix F)

Hence the theorem is proved for this case.

Case 2: The abstraction completes normally

call-method ecnt retcnt sel args ob env st
= (methods-of ob st sel) ecnt retcnt' (ivd, md, iv, m) (succ (env sender)) args st
    (from defn. of call-method)
= ecnt res st'
    (from assumption - where res is the result of executing the method)
⇒ Den_{call-method}(tok, args, ob, env, st) = ("completed", res, st')
Chapter 6: On a Correspondence between Formal Semantics Definition Methods

run(given( received([ call the method tok:token with arguments args: argument-list at ob:object ] , conv(env)), d:data), r:redirections, conv(st), l:li-tail)

≥([ "completed" ret:object conv(env) ] , r:redirections, conv(st'), l':li-tail,
c:commitment)

(from Appendix F)

⇒ Act_{call-method}(tok, args, ob, env, st) = ("completed", res, st')

Hence the theorem is proved for this case.

Case 3: The abstraction returns to this object

call-method ecnt retcnt sel args ob env st
= (methods-of ob st sel) ecnt retcnt' (ivd, md, iv, m) (succ (env sender)) args st
(from defn. of call-method)

= retcnt' (ret, s) st'
(from assumption - where ret is the returned object and s an integer)

= (λ(res, send).send = (env sender)) → ecnt res, retcnt (res, send))(ret, s) st'
(from defn. of call-method)

= ecnt ret st'
(from assumption)

⇒ Den_{call-method}(tok, args, ob, env, st) = ("completed", ret, st')

run(given( received([ call the method tok:token with arguments args: argument-list at ob:object ] , conv(env)), d:data), r:redirections, conv(st), l:li-tail)

≥([ "completed" ret:object conv(env) ] , r:redirections, conv(st'), l':li-tail,
c:commitment)

(from Appendix F)

⇒ Act_{call-method}(tok, args, ob, env, state) = ("completed", ret, st')

Hence the theorem is proved for this case.
Case 4: The abstraction returns, but not to this object

call-method ecnt retcnt sel args ob env st
= (methods-of ob st sel) ecnt retcnt' (ivd, md, iv, m) (succ (env sender)) args st
   (from defn. of call-method)
= retcnt' (ret, s) st'
   (from assumption - where ret is the returned object and s an integer)
= (λ(res, send).(send = (env sender)) → ecnt res, retcnt (res, send))(ret, s) st'
   (from defn. of call-method)
= retcnt (ret, s) st'
   (from assumption)
⇔ Den_{call-method}(tok, args, ob, env, st) = ("returning", ret, s, st')

run(given( received( receive( call the method tok:token with arguments
   args: argument-list at ob:object , conv(env)), d: data), r: redirections,
   conv(st), l: li-tail)
≥("escaped" (ret, s): data, r': redirections, conv(st'), l': li-tail, c: commitment)
   (from Appendix F)
⇔ Act_{call-method}(tok, args, ob, env, state) = ("returning", ret, s, st')

Hence the theorem is proved for this case.

6.5 Conclusions

This first half of this chapter has shown that for a specific example language, a
definition of the semantics of that language can be presented in Denotational
Semantics and Action Semantics. Additionally the Denotational Semantics can be
shown to be equivalent to the Action Semantics. The techniques used to complete this
proof could be applied to some other languages, where the structures of the two
versions of the semantics are fundamentally compatible, so as to facilitate a proof by
structural induction.
It has been seen that when proofs involving Action semantics need to make explicit use of variables representing items such as the environment, reference to the Structural Operational Semantics of the Action Notation is required. This makes proofs long and difficult. In the author's opinion however this will not be a serious impediment to the development of Action Semantics, due to the other advantages of the framework.

The second part of this chapter has presented a proof of the equivalence of the message sending aspects of two different definitions of the semantics of Smalltalk. The proof was constructed by taking the semantic function that is used to send messages in each of the two semantics definitions, and considering each of the possible outcomes of performing the semantic function in turn. Each outcome for one of the semantics was then proved to be equivalent to the corresponding outcome in the other semantics, thus proving the equivalence of the overall function.

The proofs that have been presented here are in a sense manufactured in that the Denotational Semantics of Smalltalk was designed to have a structure similar to that of the Action Semantics, in order to make the equivalence proofs straightforward. In order to be able to use the techniques outlined here it is a requirement that the two semantics have a similar structure. There may be potential for future work on the construction of Denotational Semantics definitions from Action Semantics. It would be nice to be able to produce a Denotational Semantics from the Action Semantics which by construction is equivalent. This might be achieved for example by replacing the current definition of Action Notation given in [Mos92], by a Denotational Definition. This would give us an equivalent Denotational Definition of any Action Semantics "for free".

It would also be desirable to be able to take a given Denotational Semantics of a language and construct an equivalent Action Semantics, so as to produce a more easily readable version.
Chapter 7: Conclusions

Conclusions

The work presented in this thesis has examined the application of the Action Semantics framework to defining the semantics of object-oriented languages.

Chapter 3 introduced the concepts behind Action Semantics. It then examined how Action Semantics might be applied to the definition of the semantics of object-oriented languages.

Any method of defining the semantics of an object-oriented language would have to be able to successfully model the concepts of encapsulation and inheritance. Encapsulation is essentially the idea that data owned by an object are accessible only to that object. Any changes to that data can only be made via the services which are provided by the object. Inheritance is the concept that a new class can be defined in terms of the difference between itself and a super class. Methods defined by the super class are accessible within the new class. The new class may also define extra methods, and override inherited methods. An example language, EIL, was described which exhibits the properties of encapsulation and inheritance.

Four potential methods for defining object-orientation were given. Traditionally many authors have used the concept of a cell within a store to model an object. The language POOL was defined in Action Semantics using agents to model objects by Palma et al. [PMM95]. The traditional algorithm for searching for methods is the "method look-up" algorithm. The "copy-down" approach, copies all methods from classes into instances of those classes at their creation time. The two models for objects, and the two models for searching for methods can be combined together to
produce four different ways of modelling encapsulation, i.e. "cell based look-up", "cell based copy-down", "agent based look-up" and "agent based copy-down". The "copy-down" approach was inspired by the work of Wolczko [Wo188].

It was found that the similarity between objects and agents is largely superficial. Palma et al. were able to define a simple language using agents to model objects. However their language contained no inheritance mechanism. It becomes much more difficult to define in a simple and intuitive manner the semantics of a more fully-featured object-oriented language. This is due to the very strong encapsulation that is enforced by the use of agents. It is not possible for one agent based object to directly inspect the state of some other agent based object. A message needs to be sent. This is not a problem however in the cell based approach.

The main advantage in using the agent based approach is its inherent ability to model parallel features of languages. This is why this approach was particularly suitable for defining the semantics of POOL.

There was little difference found in terms of the relative length, complexity or intuition between the copy-down and look-up approaches. Both can be used effectively to define the semantics of object-oriented languages.

In Chapter 4 one of the four potential methods discussed above was chosen for a case study in defining the Action Semantics of an object-oriented language. The method chosen was the "cell based copy-down" approach. The language defined was a restricted version of Smalltalk-80.

Smalltalk has been used to demonstrate how to write the semantics of object-oriented languages using different frameworks (mainly Denotational Semantics) by other authors. The work presented in Chapter 4 enables us to compare the Action Semantics approach to other systems. It has also proved that it is possible to produce the semantics of "real" object-oriented languages using the Action Semantics framework.
Chapter 7: Conclusions

It can also be used as an example for producing the Action Semantics of other object-oriented languages.

One of the major innovations of our semantics of Smalltalk is the fact that classes are treated as normal objects. This is closer to most people's intuitive understanding of Smalltalk, and is unlike other approaches which have tended to make classes a special case semantic entity.

Chapter 5 introduced the concepts of Denotational semantics. We discussed how one might attempt to generate a Denotational semantics of a language based on an Action Semantics of that language. This generation was attempted for the Action Semantics of Smalltalk given in Chapter 4, and a Denotational Semantics was produced. This Denotational Semantics has a very similar structure to the Action Semantics version. This feature is important for the work that was presented in Chapter 6.

Chapter 6 discussed methods of proving equivalence of a definition of a language across the frameworks of Action Semantics and Denotational Semantics. An equivalence proof of this kind can only be achieved if the underlying structures of the semantics definitions are fundamentally compatible.

A specific example language was defined using each of the two semantics definition methods. This language was based on the one used by Hoare and Lauer in [Hol74]. These two definitions were then proved to be equivalent. During the process of constructing these proofs it was found that when working with proofs concerning the Action Semantics, specific reference was required to variables that represented the environment etc. This required the use of the definition of Action Notation that was given in [Mos92, Appendix C]. The definitions given in [Mos92, Appendix C], are very long and complicated, and this in turn made constructing the proofs also very long and tedious.

The techniques that were developed at the beginning of Chapter 6 for proving equivalence across the frameworks were applied to a specific portion of the two
semantics descriptions of Smalltalk. By proving equivalence between the Denotational Semantics form and the Action Semantics form, we can compare the techniques used with other attempts to define the semantics of Smalltalk. It is important to note that because the Denotational Semantics was generated from the Action Semantics, the two definitions had a very similar structure. It is this property that made the equivalence proof possible.

7.1 The Aims of This Thesis

The aims of this thesis were set out in Chapter 1. We now look back at those aims and consider whether they have been met.

i) To demonstrate that it is possible to describe object-oriented languages using Action Semantics.

It is certainly possible to describe object-oriented languages using Action Semantics, as can be seen from the examples given in this thesis. In particular Chapter 4 gives a case study of the Action Semantics of Smalltalk.

ii) To develop generic models that can be used within Action Semantics to define languages that exhibit object-oriented features.

Four models have been developed that are described in Chapter 3. These are the "cell based look-up", "cell based copy-down", "agent based look-up" and "agent based copy-down".

iii) To provide examples of how object-oriented languages can be described using Action Semantics.

Examples of how to produce the Action Semantics of object-oriented languages have been given throughout Chapters 3 and 4. Chapter 4 gives a complete semantics for a restricted version of Smalltalk.
iv) To suggest methods by which Action Semantics Definitions of object-oriented languages can be related and compared to the existing body of knowledge on the semantics of such languages.

Chapter 6 investigated how it is possible to prove the equivalence of two versions of the semantics of a given language, where the semantics have been written in two different frameworks. Chapter 5 discussed how a Denotational Semantics can be generated from an Action Semantics. Much of the existing work on the semantics of languages has been given in the Denotational Semantics framework. It is possible to compare an Action Semantics with existing Denotational Semantics indirectly via a generated semantics.

v) To form a basis for future research into Action Semantics and object-oriented languages.

The basic underlying concepts found in most object-oriented languages have been examined, and their Action Semantics produced. This research can be used as a starting point for producing the Action Semantics of other languages, or for investigating the semantics of novel language features.

vi) To investigate the usefulness and flexibility of Action Semantics.

Action Semantics has been found to be a useful tool in defining the semantics of languages. It is this author's opinion that the readability of such semantics definitions is better than in other frameworks. It is however difficult to theorise about the semantics in instances where reference is required to the operational semantics of Action Notation.

This author believes as a result of the work presented in this thesis that the advantages of Action Semantics are greater than the disadvantages.

7.2 Future Work

We have not considered multiple inheritance within this thesis. This concept seems to be not well understood with the Object-Oriented community. It would be interesting
to examine this topic, and explore how languages that exhibit this property might be defined using Action Semantics.

The only "real" language defined has been Smalltalk. It would be helpful to have more definitions of the semantics of other "real" languages. The languages (such as EIL) that are created by theorists to help explain the concepts are often very simple. Once one starts to define a larger language more problems are encountered. Action Semantics has been used to define the semantics of the language PASCAL [MoW93]. However no such attempts have been made for large object-oriented languages.

We have stated a number of times, the aims of Action Semantics is to improve the readability of semantics definitions. It would be interesting and informative to examine this further. An experiment could be designed to test the readability of Action Semantics definitions by exposing different groups of users to them and examining their reactions (e.g. by means of a questionnaire).
References


References


References


[MoW93] Mosses, P.D. and Watt, D.A. PASCAL Action Semantics (vsn 0.6), Dept. of Computer Science, University of Aarhus, Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark.


References


Appendix A: The Action Semantics of EIL

The Action Semantics of EIL

A.1 Framework Semantics of EIL

Abstract Syntax
Semantic Functions needs: Abstract Syntax, Semantic Entities.
Semantic Entities

A.1.1 Abstract Syntax

needs: [Mos92]/Data Notation/Characters/ASCII (letter, digit)
closed.

grammar:

A.1.1.1 Identifiers

- User-Object-Identifier = D.
- Object-Identifier = User-Object-Identifier | "object" (disjoint).
- Variable-Identifier = D.
- Method-Identifier = D.

A.1.1.2 Expressions

- Expression = "self" | Variable-Identifier | Object-Identifier |
  Variable-Identifier "is" Expression |
  Expression Method-Identifier Expression |
  "super" Method-Identifier Expression .

A.1.1.3 Classes

- Class = "class" Object-Identifier ClassBody .
- ClassBody = "inherits" Object-Identifier InstVar* Method+ .
- InstVar = "instance" Variable-Identifier .
- Method = "method" Method-Identifier "is" Expression+ .
Appendix A: The Action Semantics of EIL

A.1.2 Semantic Functions

A.1.2.1 Creating Classes

introduces: create_

- create__: Class^ \rightarrow \text{action}
  [binding] .
  [using current bindings] .

(1) create < C_1:Class C_2:Class^ > =

  | create C_1
  | before
  | create C_2

A.1.2.2 Elaborating Variables

introduces: elaborate_

- elaborate__: InstVar^ \rightarrow \text{action}
  [binding]
  [using current bindings | current storage] .

(1) elaborate < I_1:InstVar I_2:InstVar^ > =

  | elaborate I_1
  | and
  | elaborate I_2

(2) elaborate \{ "instance" id:Variable-Identifier \} =

  | allocate a cell
  | then
  | bind id to the given cell.

A.1.2.3 Expressions

introduces: evaluate_

- evaluate__: Expression \rightarrow \text{action}
  [giving an object | storing | diverging]
  [using current bindings | current storage] .

(1) evaluate < E_1:Expression E_2:Expression^ > =

  | evaluate E_1
  | then
  | evaluate E_2
Appendix A: The Action Semantics of EIL

(2) evaluate \(["self"]\) =
give the object bound to self.

(3) evaluate \([\text{VarNam}:\text{Variable-Identifier}]\) =
give the object stored in the cell bound to VarNam.

(4) evaluate \([\text{ObId}:\text{Object-Identifier}]\) =
give the object bound to ObId.

(5) evaluate \([\text{VarNam}:\text{Variable-Identifier} \ "is" \ E: \text{Expression}]\) =
  | evaluate E
  then
  | store the given object in the cell bound to VarNam
  and
  | regive.

(6) evaluate \([E_1: \text{Expression} \ Sel: \text{Method-Identifier} \ E_2: \text{Expression}]\) =
  | evaluate E_1
  and then
  | evaluate E_2
  then
  | call the method Sel with argument the given object#2 at the given object#1.

A.1.3 Semantics Entities

includes: [Mos92]/Action Notation.

introduces: object, call the method _ with argument _ at _.

- object = .
- call the method _ with argument _ at _ :: Method-Identifier, object, object \rightarrow\) action.
Appendix A: The Action Semantics of EIL

A.2 Cell Based Copy-down Semantics of EIL

Abstract Syntax
Semantic Functions needs: Abstract Syntax, Semantic Entities.
Semantic Entities

A.2.1 Abstract Syntax

includes: Framework Semantics of EIL/Abstract Syntax.

A.2.2 Semantic Functions

includes: Framework Semantics of EIL/Semantic Functions.

introduces: create standard object, construct _, the new method.

A.2.2.1 Creating User Defined Classes

(1) create 
"class" id:Object-Identifier
    ["inherits" iid:Object-Identifier ivs:InstVar* methods:Method+] =
    allocate an object
    and
    furthermore bind super-class-methods to
    the methods defined by the object bound to iid
    hence
    produce the bindings bound to super-class-methods
    moreover
    construct the methods
    hence
    give the current bindings
    then
    store the object-record of(
        empty-map,
        the new method,
        concatenation(
            the instance variables defined by the object bound to iid,
            list of ivs),
        the given methods#2) in the given object#1
    and
    bind id to the given object#1.

A.2.2.2 Constructing new methods

- construct _:: Method+ → action
  [binding]
  [using current bindings].
Appendix A: The Action Semantics of EIL

(1) construct $<M_1:Method \ M_2:Method^+>$ =
    | construct $M_1$
    and
    | construct $M_2$.

(2) construct $["method" \ id:Method-Identifier "is" \ E:Expression^+] =$
    bind id to closure of abstraction of
    furthermore
    | bind self to the given object#1
    and
    | produce the instance variables of the given object#1
    and
    | bind "arg":Object-Identifier to the given object#2
    hence
    | evaluate E.

A.2.2.3 Expressions

(1) evaluate $["super" \ id:Method-Identifier \ E:Expression]$ =
    give the object bound to self and evaluate E
    then
    enact application of the abstraction yielded by
    (the methods bound to super-class-methods) at id to
    (the given object#1, the given object#2).

A.2.3 Semantic Entities


A.2.3.1 Sorts

introduces: super-class-methods, self, reserved-tokens, object, method-abstraction, methods, instance-variables, instance-variable-tokens, object-record, object-record of _, methods of _, methods defined by _, instance variables of _, instance variables defined by _,

• datum = methods | instance-variables | object | abstraction | token.
• bindable = object | abstraction.
• storable = object.
• component = token | object | instance-variables | methods | instance-variable-tokens.
• item = token | object.
• reserved-tokens = super-class-methods | self (individual).
• token = string of ( letter, ( letter | digit )* ) | reserved-tokens (disjoint).
• object ≤ cell.
• method-abstraction = abstraction
  [given an object | storing | diverging]
  [using the given (object, object) | current storage].
• methods = map[token to method-abstraction].

156
Appendix A: The Action Semantics of ElL

- instance-variables = map[token to cell].
- instance-variable-tokens = list of token*.
- object-record = (instance-variables, methods) | (instance-variables, methods, instance-variable-tokens, methods).
- object-record of _ :: component* → object-record.
- instance variables of _ :: object → instance-variables.
- methods of _ :: object → methods.
- instance variables defined by _ :: object → instance-variables-tokens.
- methods defined by _ :: object → methods.

(1) object-record of component* = object-record & component*.

(2) instance variables of ob:object =
   the instance-variables yielded by component#1 of the object-record stored in ob.

(3) methods of ob:object =
   the methods yielded by component#2 of the object-record stored in ob.

(4) instance variables defined by ob:object =
   the instance-variable-tokens yielded by component#3 of the object-record stored in ob.

(5) methods defined by ob:object =
   the methods yielded by component#4 of the object-record stored in ob.

A.2.3.2 Creating the Standard Object

- create standard object :: action
  [binding] .

(1) create standard object =
   allocate an object
   then
   store object-record of (empty-map, the new method, empty-list, empty-map) in the given object
   and
   bind "object" to the given object.
A.2.3.3 The method "new"

- the new method: methods.

(1) the new method =  
  map of "new" to closure of abstraction of  
  allocate an object and regive  
  then  
    elaborate the instance variables defined by the given object#1  
    and  
    bind self to the given object#1  
  hence  
  store object-record of (  
    the current bindings,  
    the methods defined by the given object#2)  
  and  
  give the given object#1.

A.2.3.4 Invoking Methods

- call the method _ with argument _ at _:: token, object, object → action  
  [giving an object | storing | diverging]  
  [using current storage] .

(1) call the method meth:token with argument arg:object at ob:object =  
  enact application of  
  the method-abstraction yielded by ((the methods of ob) at meth)  
  to (ob, arg).
A.3 Cell Based Look-up Semantics of EIL

Abstract Syntax
Semantic Functions needs: Abstract Syntax, Semantic Entities.
Semantic Entities

A.3.1 Abstract Syntax

includes: Framework Semantics of EIL/Abstract Syntax.

A.3.2 Semantic Functions

includes: Framework Semantics of EIL/Semantic Functions.

introduces: create standard object, construct _, the new method,
look up the abstraction for _ in _.

A.3.2.1 Creating User Defined Classes

(1) create [ "class" id:Object-Identifier
                [ "inherits" iid:Object-Identifier ivs:InstVar* methods:Method+ ] ]
        =
         | allocate an object
         then
         | furthermore bind super-class to the super class of the given object
         hence
         | construct the methods
         hence
         | regive and give the current bindings
         then
         | store the object-record of( 
             the object bound to meta-class,
             empty-map,
             the object bound to iid,
             concatention( 
                 the instance variables defined by the object bound to iid,
                 list of ivs),
             the given bindings#2) in the given object#1
         and
         | bind id to the given object#1.

A.3.2.2 Constructing new methods

• construct _:: Method+ → action
        [binding]
        [using current bindings].
Appendix A: The Action Semantics of EIL

(1) construct \(< M_1 : \text{Method} M_2 : \text{Method}^+ > =
\begin{align*}
\text{construct } M_1 \\
\text{and} \\
\text{construct } M_2.
\end{align*}

(2) construct \([ "\text{method}\ id : \text{Method-Identifier}\ "\text{is}\ E : \text{Expression}^+ ] =
\begin{align*}
\text{bind } \text{id} \text{ to closure of abstraction of} \\
\text{furthermore bind } \text{self} \text{ to the given object#1} \\
\text{and} \\
\text{produce the instance variables of the given object#1} \\
\text{and} \\
\text{bind } "\text{arg}": \text{Object-Identifier} \text{ to the given object#2} \\
\text{hence} \\
\text{evaluate } E.
\end{align*}

A.3.2.3 Expressions

(1) evaluate \([ "\text{super}\ id : \text{Method-Identifier}\ E : \text{Expression} ] =
\begin{align*}
\text{look up the abstraction for } \text{id} \text{ in the object bound to super-class} \\
\text{and} \\
\text{evaluate } E \\
\text{then} \\
\text{enact application of the given abstraction#1 to} \\
\text{(the object bound to self, the given object#2).}
\end{align*}

A.3.3 Semantic Entities


A.3.3.1 Sorts

introduces: super-class-methods, self, reserved-tokens, meta-class, object, method-abstraction, methods, instance-variables, instance-variable-tokens, object-record, object-record of _, class object of _, instance variables of _, super class of _, instance variables defined by _, methods defined by _.

- datum = methods | instance-variables | object | abstraction | token.
- bindable = object | abstraction.
- storable = object.
- component = token | object | instance-variables | methods | instance-variable-tokens.
- item = token | object.
- reserved-tokens = super-class-methods | self | meta-class (individual).
- token = string of ( letter, ( letter | digit )* ) | reserved-tokens (disjoint).
- object \leq \text{cell}.
- method-abstraction = abstraction
  \begin{align*}
  &\text{[giving an object | storing | diverging]} \\
  &\text{[using the given (object, object) | current storage].}
\end{align*}
- methods = map[token to method-abstraction].
Appendix A: The Action Semantics of EIL

- instance-variables = map[token to cell].
- instance-variable-tokens = list of token*.
- object-record = (object, instance-variables) | (object, instance-variables, object, instance-variable-tokens, methods).
- object-record of _ :: component* → object-record.
- class object of _ :: object → object.
- instance variables of _ :: object → instance-variables.
- super class of _ :: object → object.
- instance variables defined by _ :: object → instance-variables-tokens.
- methods defined by _ :: object → methods.

(1) object-record of component* = object-record & component*.

(2) class object of ob:object =
   the object yielded by component#1 of the object-record stored in ob.

(3) instance variables of ob:object =
   the instance-variables yielded by componen#2 of the object-record stored in ob.

(4) super class of ob:object =
   the object yielded by component#3 of the object-record stored in ob.

(5) instance variables defined by ob:object =
   the instance-variable-tokens yielded by component#4 of the object-record stored in ob.

(6) methods defined by ob:object =
   the methods yielded by component#5 of the object-record stored in ob.

A.3.3.2 Creating the Standard Object

- create standard object :: action
  [binding].
Appendix A: The Action Semantics of EIL

(1) create standard object
    allocate an object and allocate an object
    then
    store object-record of
        the given object#1,
        empty-map,
        the given object#2,
        empty-list,
        the new method) in the given object
    and
    bind meta-class to the given object#1
    and
    store object-record of
        the given object#1,
        empty-map,
        the given object#2,
        empty-list,
        empty-map) in the given object#2
    and
    bind "object" to the given object#2

A.3.3.3 The method "new"

- the new method:: methods.

(1) the new method =
    map of "new" to closure of abstraction of
    allocate an object and regive
    then
    elaborate the instance variables defined by the given object#2
    and
    bind self to the given object#1
    hence
    store object-record of
        the given object#2,
        the current bindings) in the given object#1
    and
    give the given object#1.

A.3.3.4 Invoking Methods

- call the method _ with argument _ at _:: Method-Identifier, object, object → action
  [giving an object | storing | diverging]
  [using current storage].

- look up the abstraction for _ in _:: token, object → action
  [giving an abstraction].
Appendix A: The Action Semantics of EIL

(1) call the method meth with argument arg at ob:object =
    look up the abstraction for meth in the class object of ob
    then
    enact application of the given method-abstraction to (ob, arg).

(2) look up the abstraction for meth:Method-Identifier in ob:object =
    give (the methods defined in ob) at meth
    or
    check not meth is in the mapped-set of the methods defined by ob
    and then
    look up the abstraction for meth in the super class of ob.

A.4 Agent Based Copy-down Semantics of EIL

Abstract Syntax
Semantic Functions needs: Abstract Syntax, Semantic Entities.
Semantic Entities

A.4.1 Abstract Syntax

includes: Framework Semantics of EIL/Abstract Syntax.

A.4.2 Semantic Functions

includes: Framework Semantics of EIL/Semantic Functions.

introduces: contract _.
A.4.2.1 Creating User Defined Classes

(1) create \[ "class" id:Object-Identifier
\[ "inherits" iid:Object-Identifier ivs:InstVar* methods:Method+ \] \] =
|| send a message[to the object bound to iid][containing behaviour-request]
then
|| receive a message[from the object bound to iid
\| [containing (instance-variable-tokens,methods)]
then
|| furthermore bind super-class-methods to
\| component#2 of the contents of the given message
\| produce the bindings bound to super-class-methods
\| moreover
\| construct the methods
\| hence
\| bind class-methods to the current bindings
\| and
\| bind class-instances to concatenation(component#1 of the given message,
\| list of ivs)
\| hence
\| make a new class
\| hence
\| subordinate an object
\| and
\| give closure of abstraction of activate message loop
then
|| send a message[to the given object#1][containing the given abstraction#2]
\| and
\| bind id to the given object#1.

A.4.2.2 Constructing new methods

- construct :: Method+ → action
  [binding]
  [using current bindings].

(1) construct < M_1:Method M_2:Method+ > =
  \| construct M_1
  and
  \| construct M_2.
Appendix A: The Action Semantics of EIL

(2) construct \[
\text{"method" id:Method-Identifier "is" E:Expression}^+ \] =
bind id to closure of abstraction of
| bind self to the given object\#1
and
| furthermore produce the given instance-variables\#2
and
| bind instance-vars to the given instance-variables\#2
and
| bind "arg":Object-Identifier to the given object\#3
hence
| evaluate E.

A.4.2.3 Expressions

(1) evaluate \[
\text{"super" id:Method-Identifier E:Expression} \] =
evaluate E
then
enact application of the abstraction yielded by
| (the methods bound to super-class-methods) at id to
| (the performing agent, the instance-variables bound to instance-vars,
| the given object\#1).

A.4.3 Semantic Entities


A.4.3.1 Sorts

introduces: object, super-class-methods, self, class-methods, class-instances, instance-variables, reserved-tokens, behaviour-request, method-abstraction, instance-variables, methods, instance-variable-tokens.

- object = agent.
- datum = bindings | object | abstraction | token.
- bindable = object | abstraction.
- storable = object.
- component = token | object.
- item = token.
- method-abstraction = abstraction
  [giving an object | storing | diverging]
  [using the given (object, instance-variables, object) | current storage].
- methods = map[token to method-abstraction].
- instance-variables = map[token to cell].
- instance-variable-tokens = list of token*.
- reserved-tokens = super-class-methods | self | class-methods | class-instances |
  instance-vars | behaviour-request (individual).
- token = string of ( letter, ( letter | digit )* ) | reserved-tokens (disjoint).
A.4.3.2 Creating the Standard Object

introduces: create standard object, make a new class.

- create standard object :: action
  [binding | communicating]
  [using current bindings].

- make a new class :: action
  [binding]
  [using current bindings].

1. create standard object
   bind class-methods to empty-map
   and
   bind class-instances to empty-list
   hence
   make a new class
   hence
   give closure of abstraction of activate message loop
   and
   subordinate an object
   then
   send a message[to the given object#2][containing the given abstraction#1]
   and
   bind "object" to the given object#2.

2. make a new class =
   bind "new" to closure of abstraction of
   subordinate an object
   and
   elaborate the instance-variable-tokens bound to class-instances
   hence
   bind instance-variables to the current bindings
   and
   produce the methods bound to class-methods
   hence
   give closure of abstraction of activate message loop
   then
   submit the given abstraction#2 to the given object#1 and give the given object#1
   and
   bind instance-variables to empty-map.
A.4.3.3 The Message Loop

introduces: activate message loop.

- activate message loop :: action
  [storing | communicating | diverging]
  [using current bindings | current storage | current buffer].

(1) activate message loop =

    unfolding
    | receive a message[from an object][containing (token, object)]
    then
    | enact application of the abstraction bound to
    | component#1 of the contents of the given message
    | to (the performing agent, the instance-variables bound to instance-vars,
    | component#2 of the contents of the given message)
    and
    | give the sender of the given message
    then
    | send a message[to the given object#2][containing the given object#1]
    and
    | receive a message[from an object][containing a behaviour-request]
    then
    | send a message[to the sender of the given message][containing
    |   the instance-variable-tokens bound to class-instances,
    |   the methods bound to class-methods]
    and
    | unfold.

A.4.3.4 Invoking Methods

(1) call the method meth:token with argument arg:object at ob:object =

    send a message[to ob][containing (meth, arg)]
    and then
    | receive a message[from ob][containing an object]
    then
    | give the contents of the given message.
Appendix A: The Action Semantics of EIL

A.5 Agent Based Look-up Semantics of EIL

Abstract Syntax
Semantic Functions needs: Abstract Syntax, Semantic Entities.
Semantic Entities

A.5.1 Abstract Syntax

includes: Framework Semantics of EIL/Abstract Syntax.

A.5.2 Semantic Functions

includes: Framework Semantics of EIL/Semantic Functions.

introduces: make all the methods and variables of _, construct _.

A.5.2.1 Creating User Defined Classes

• make all the methods and variables of _ :: ClassBody \rightarrow action
  [binding | communicating]
  [using current buffer].

(1) create [[ "class" id:Object-Identifier body:ClassBody ]] =
    subordinate an object
    and
    make all the methods and variables of body
    hence
    give closure of abstraction of activate message loop
    then
    submit the given abstraction#2 to the given object#1
    and
    bind id to the given object.#1
(2) make all the methods and variables of

```
[ "inherits" id:Object-Identifier ivs:InstVar* methods:Method+ ]
```

- send a message[to the object bound to id][containing behaviour-request]
- then
- receive a message[from the object bound to id]
  [containing instance-variable-tokens]
  then
- give the contents of the given message
  then
- bind class-instances to concatenation(list of InstVar*,
  the given instance-variable-tokens)
  and
- bind super-class to the object bound to id
  hence
- construct the methods.
  hence
- bind class-methods to the current bindings
  and
- bind super-class to the object bound to id
  and
- bind this-class to the object bound to meta-class.

A.5.2.2 Constructing new methods

- construct _:: Method+ → action
  [binding]
  [using current bindings].

(1) construct < M1:Method M2:Method+ >=

```
| construct M1
  and
  construct M2 .
```

(2) construct [ "method" id:Method-Identifier "is" E:Expression+ ] =

```
bind id to closure of abstraction of
| furthermore bind self to the given object#1
  and
  produce the given bindings#2
  and
  bind "arg":Object-Identifier to the given object#3
  hence
  evaluate E.
```

A.5.2.3 Invoking Methods

- call the method _ with argument _ at _:: Method-Identifier, object, object → action
  [giving an object | storing | diverging]
  [using current storage].
Appendix A: The Action Semantics of EIL

(1) call the method meth:Method-Identifier with argument arg:object at ob:object =
   | send a message[to ob][containing (meth, arg)]
   and then
   | receive a message[from ob][containing an object]
   then
   | give the contents of the given message.

A.S.2.4 Expressions

(1) evaluate [ "super" id:Method-Identifier E:Expression ] =
   | evaluate E
   and
   | look up the abstraction for id in the object bound to super-class
   then
   enact application of
   | the closure of the given abstraction#2
   to the given object#1.

A.S.3 Semantic Entities


A.S.3.1 Sorts

introduces:  object, super-class-methods, self, class-methods, class-instances, instance-
variables, this-class, super-class, reserved-tokens, method-abstraction,
instance-variables, methods, instance-variable-tokens.

- object = agent.
- datum = bindings | object | abstraction | token.
- bindable = object | abstraction.
- storable = object.
- component = token | object.
- item = token.
- method-abstraction = abstraction
  [giving an object | storing | diverging]
  [using the given (object, instance-variables, object) | current storage].
- methods = map[token to method-abstraction].
- instance-variables = map[token to cell].
- instance-variable-tokens = list of token*.
- reserved-tokens = behaviour-request |super-class-methods | self | class-methods |
  class-instances | instance-variables | this-class | super-class (individual).
- token = string of ( letter, ( letter | digit )* ) | reserved-tokens (disjoint).
A.5.3.2 Creating the Standard Object

introduces: create standard object, make an abstraction for the standard object, make the meta class, the new method.

- create standard object :: action
  [binding | communicating].

- make an abstraction for the standard object :: abstraction.

- make the meta class :: action
  [binding | communicating].

- the new method :: method-abstraction.

(1) create standard object
   make the meta class
   hence
   subordinate an object and make an abstraction for the standard object
   then
   send a message[to the given object#1][containing the given abstraction#2]
   and
   furthermore bind "object" to the given object#1.

(2) make an abstraction for the standard object
   give closure of abstraction of
   bind class-instances to empty-list
   moreover
   bind class-methods to empty-map
   moreover
   bind this-class to the object bound to meta-class
   hence
   activate message loop.
Appendix A: The Action Semantics of EIL

(3) make the meta class =
| subordinate an object
then
| bind this-class to the given object
hence
| regive
and
| give closure of abstraction of
| furthermore bind class-methods to map of "new" to the new method
moreover
| bind class-instances to empty-list
hence
| activate message loop
then
| send a message[to the given object#1][containing the given abstraction#2]
and
| bind meta-class to the given object.#1.

(4) the new method =
closure of abstraction of
| subordinate an object
and
| give the given object#1
and
| produce the given bindings#2
hence
| initialise the items of the flat-list bound to class-instances
and
| bind this-class to the given object#2
hence
| give closure of abstraction of activate message loop
then
| submit the given abstraction to the given object#1 and give the given object#1.

A.5.3.3 The Message Loop

introduces: activate message loop.

- activate message loop :: action
  [storing | communicating | diverging | failing]
  [using current bindings | current storage | current buffer].
Appendix A: The Action Semantics of EIL

(1) activate message loop =

  unfolding
  | receive a message[from an object][containing (Method-Identifier, object)]
  | then
  |   look up the abstraction for component#1 of the contents of the given message
  |   in the object bound to this-class
  | then
  |   enact the application of the given method-abstraction to
  |   (the performing-agent, the current bindings,
  |   component#2 of the contents of the given message)
  | and
  |   give the sender of the given message
  | then
  |   send a message[to the given object#2][containing the given object#1]
  | and
  | receive a message[from an object][containing a behaviour-request]
  | then
  |   send a message[to the sender of the given message]
  |   [containing the flat-list bound to class-instances]
  | and
  | receive a message[from an object][containing a Method-Identifier]
  | then
  |   give the method-abstraction yielded by
  |   (the method-bindings bound to class-methods) at
  |   the contents of the given message
  | or
  |   check not the contents of the given message is in
  |   the mapped-set of the method-bindings bound to class-methods
  | and then
  |   look up the abstraction for the contents of the given message
  |   in the object bound to super-class
  | and give the sender of the given message
  | then
  |   send a message[to the given object#2][containing the given method-
  |   abstraction#1]
  | and
  | unfold .

A.5.3.4 Looking up a method

introduces: look up the abstraction for _ in _.

- look up the abstraction for _ in _ :: token, object → act<ion
  [giving a method-abstraction | communicating]
  [using current buffer].
Appendix A: The Action Semantics of Eil.

(1) look up the abstraction for meth:token in ob:object=
    | send a message[to ob][containing meth]
    then
    | receive a message[from ob][containing a method-abstraction]
    then
    | give the contents of the given message.
Appendix B: The Action Semantics of Smalltalk

B.1 Abstract Syntax

grammar:

B.1.1 Lexical elements

- Selector = "new" |
- Integer-Constant = .
- Non-Root-Class = "Array" | "BlockContext" | "Class" | "Integer" |
- Class-Name = Non-Root-Class | "Object". (disjoint)
- Instance-Variable = .
- Arg-Variable = .
- Temporary-Variable = .

B.1.2 Variables and Literals

- Variable-Name = [ Instance-Variable ] | [ Arg-Variable ] | [ Temporary-Variable ] (disjoint).
- Literal = [ Integer-Constant ] | [ "array" Literal* ] .

B.1.3 Expressions

- Expression = [ Variable-Name ] | [ Class-Name ] | [ Literal ] | [ Expression Expression+ ] | [ "^" Expression ] | [ "[" Arg-Variable* Expression "]" ] ] | [ Variable-Name "←" Expression ] | [ Expression Selector Expression* ] | [ "super" Selector Expression* ] | [ "self" ] .

B.1.4 Classes

- Method = [ Sel:Selector Method-Def ] .
- Method-Def = [ Arg-Variable* Temporary-Variable* Expression ] | [ Arg-Variable* Temporary-Variable* "primitive" Integer-Constant Expression ] .
Appendix B: The Action Semantics of Smalltalk

- Class-Def = [ Class-Name "inheriting" 
  Class-Name Instance-Variable* Method* ]

B.2 Semantic Functions

needs: Abstract Syntax, Semantic Entities.

B.2.1 Classes

introduces: create _, construct _, make method _

B.2.1.1 User Classes

- create _ : Class-Def* → action
  [binding | storing]
  [using current bindings | current storage].

1. create < C1 : Class-Def C2 : Class-Def* > =
   create C1
   before
   create C2

2. create [ ClsNam : Class-Name "inheriting" iid : Class-Name
   ivs : Instance-Variable* mthds : Method* ] =
   allocate an object
   and
   furthermore bind super-class-methods to
   the methods defined by the object bound to iid
   hence
   produce the methods bound to super-class-methods
   moreover
   construct the mthds
   hence
   give the current bindings
   then
   store the object-record of(
   empty-map,
   the new method,
   concatenation(
   the instance variables defined by the object bound to iid,
   list of ivs),
   the given methods#2) in the given object#1
   and
   bind ClsNam to the given object#1.
B.2.1.2 Constructing new methods

- `construct _ :: Method* \rightarrow action`
  
  [binding].

1. `construct < > =`
   
   complete.

2. `construct < M_1 : Method M_2 : Method > =`
   
   `construct M_1`
   
   and
   
   `construct M_2`.

3. `construct [ Sel : Selector MethDef : Method-Def ]`
   
   bind Sel to make method MethDef.

B.2.1.3 User-defined Method Abstraction

- `make method _ :: Method-Def \rightarrow method-abstraction`.

1. `make method [ Arg : Arg-Variable* Tmp : Temporary-Variable* E : Expression ] =`
   
   closure of abstraction of
   
   furthermore
   
   `bind self to the given object_1`
   
   and
   
   `bind sender to the given integer_2`
   
   and
   
   `produce the instance variables of the given object_1`
   
   and
   
   `initialise Arg to the given argument-list_3`
   
   and
   
   `initialise Tmp`
   
   hence
   
   evaluate E.

2. `make method [ Arg : Arg-Variable* Tmp : Temporary-Variable*"
   
   "primitive" Int : Integer-Constant E : Expression ] =`
   
   primitive method number decimal Int receiving arguments Arg with non-primitive
   
   (make method [ Arg Tmp E ] ).
B.2.2 Expressions

introduces: evaluate _, evaluate the list _.

B.2.2.1 Simple Expressions

- evaluate _ :: Expression → action
  [giving an object | storing | escaping with (object, integer) | diverging | failing]
  [using current bindings | current storage].

(1) evaluate [ VarNam:Variable-Name ] =
give the object stored in the cell bound to VarNam.

(2) evaluate [ ClsNam:Class-Name ] =
give the object bound to ClsNam.

(3) evaluate [ Int:Integer-Constant ] =
call the method "new" with arguments empty-list at the object bound to "Integer"
then
  store the decimal of Int in the cell yielded by
  (the instance variables of the given object) at object-value
  and
  regive.

(4) evaluate [ "array" Arr:Literal* ] =
call the method "new" with arguments empty-list at the object bound to "Array"
and then
  evaluate the list Arr
then
  store the given flat-list#2 in the cell yielded by
  (the instance variables of the given object#1) at object-value
  and
  give the given object#1.

(5) evaluate [ E:Expression EL:Expression+ ] =
evaluate E
then
  evaluate EL .

(6) evaluate [ "↑" E:Expression ] =
evaluate E
then
  escape with (the given object, the integer bound to sender).
Appendix B: The Action Semantics of Smalltalk

(7) evaluate \[[" \text{Arg:Arg-Variable}^* \text{E:Expression "]]} =
\begin{align*}
\text{call the method} \ "\text{new}\" \text{ with arguments empty-list at} \\
\text{the object bound to} \ "\text{BlockContext}\" \\
\text{then} \\
\text{store closure of abstraction of} \\
furthermore initialise \text{Arg} \text{ to the given argument-list} \\
\text{hence} \\
evaluate \text{E} \\
in the cell yielded by (the instance variables of the given object) at block-contents \\
\text{and} \\
ext\text{egive}.
\end{align*}

(8) evaluate \[[ \text{VarNam:Variable-Name} \ "\text{←}" \text{E:Expression } ] =
\begin{align*}
evaluate \text{E} \\
\text{then} \\
\text{store the given object in the cell bound to VarNam} \\
\text{and} \\
ext\text{egive}.
\end{align*}

(9) evaluate \[\text{E:Expression Sel:Selector EL:Expression}^*\] =
\begin{align*}
evaluate \text{E} \\
\text{and then} \\
evaluate \text{the list EL} \\
\text{then} \\
call the method \text{Sel} \text{ with arguments the given argument-list}^2 \text{ at the given object}^1.
\end{align*}

(10) evaluate \[\"\text{self}\"\] =
give the object bound to self.

(11) evaluate \[\"\text{super Sel:Selector EL:Expression}\"^*\] =
\begin{align*}
evaluate \text{the list EL} \\
\text{then} \\
\text{enact application of the abstraction yielded by} \\
((\text{the methods bound to super-class-methods) at Sel}) \text{ to} \\
(\text{the object bound to self, the successor of the integer bound to sender,} \\
\text{the given argument-list}^3). \\
\text{trap} \\
\text{check the given integer}^2 \text{ is the successor of the integer bound to sender and then} \\
give the given object}^1 \\
or \\
\text{check not the given integer}^2 \text{ is the successor of the integer bound to sender} \\
\text{and then} \\
\text{escape with the given tuple}.
\end{align*}
B.2.2.2 Expression Lists

- evaluate the list _ :: Expression* → action
  [giving a list of object* | storing | escaping with (object, integer) | diverging | failing]
  [using current bindings | current storage].

(1) evaluate the list <> =
   give empty-list.

(2) evaluate the list < E:Expression EL:Expression*> =
    evaluate E
    and then
    evaluate the list EL
    then
    give concatenation of (list of the given object#1, the given flat-list#2).

B.3 Semantic Entities

includes: [Mos92]/Action Notation .
includes: [Mos92]/Data Notation .

B.3.1 Sorts

introduces: object, super-class-methods, sender, block-contents, object-value, self, reserved-tokens .

introduces: method-abstraction, argument-list, methods, instance-variables, instance-variable-tokens, object-record, object-record of _, methods of _, methods defined by _, instance variables of _, instance variables defined by _.

- datum = methods | instance-variables | object | abstraction | token.
- bindable = object | abstraction.
- storable = object.
- component = token | object | instance-variables | methods | instance-variable-tokens.
- item = token | object.
- reserved-tokens = super-class-methods | sender | block-contents | object-value | self (individual).
- token = string of ( letter, ( letter | digit )* ) | reserved-tokens (disjoint).
- object ≤ cell.
- method-abstraction = abstraction
  [given an object | storing | escaping with (object, integer) | diverging | failing]
  [using the given (object, integer, argument-list) | current storage].
- argument-list = list of object*.
- methods = map[token to method-abstraction].
- instance-variables = map[token to cell].
- instance-variable-tokens = list of token*.
Appendix B: The Action Semantics of Smalltalk

- object-record = object-record of (instance-variables, methods) | object-record of (instance-variables, methods, instance-variable-tokens, methods).
- object-record of _ :: component' → object-record.
- instance variables of _ :: object → instance-variables.
- methods of _ :: object → methods.
- instance variables defined by _ :: object → instance-variables-tokens.
- methods defined by _ :: object → methods.

(1) object-record of component' = object-record & component'.

(2) instance variables of ob:object =
   the instance-variables yielded by component#1 of the object-record stored in ob.

(3) methods of ob:object =
   the methods yielded by component#2 of the object-record stored in ob.

(4) instance variables defined by ob:object =
   the instance-variable-tokens yielded by component#3 of the object-record stored in ob.

(5) methods defined by ob:object =
   the methods yielded by component#4 of the object-record stored in ob.

B.3.1.1 Standard Class, Class

introduces: create object called class.

- create object called class:action
  [binding | storing]
  [using current storage].

(1) create object called class =
   allocate an object
   then
   store the object-record of(
     empty-map,
     the new method,
     empty-list,
     the new method) in the given object
   and
   bind "Class" to the given object.
B.3.1.2 Standard Class, Object

introduces: create object called object.

- create object called object: action
  [binding | storing]
  [using current storage].

(1) create object called object =
  allocate an object
  then
  store object-record of (empty-map,
  the new method,
  empty-list,
  empty-map) in the given object
  and
  bind "Object" to the given object.

B.3.1.3 The new method

introduces: the new method.

- the new method: methods.

(1) the new method =
  map of "new" to closure of abstraction of
  give the given object#1 and allocate an object
  then
  initialise the instance variables defined by the given object#1
  hence
  store object-record of (the current bindings,
  the methods defined by the given object#1) in the given object#2
  and
  give the given object#2.

B.3.1.4 Variable Initialisation

introduces: initialise _, initialise _ to _.

- initialise _ :: token * → action
  [storing | binding]
  [using current storage].

- initialise _ to _ :: token *, argument-list → action
  [storing | binding]
  [using current storage].
Appendix B: The Action Semantics of Smalltalk

(1) initialise <> =
      complete.

(2) initialise <Var:token Var_list:token*> =
      allocate a cell
      then
      bind Var to the given cell
      and
      initialise Var_list.

(3) initialise <> to <> =
      complete.

(4) initialise <Arg:token Arg_list:token*> to Ist:argument-list =
      allocate a cell
      then
      store the head of Ist in the given cell
      and
      bind Arg to the given cell
      and
      initialise Arg_list to tail of Ist.

B.3.1.5 Calling a method

introduces: call the method _ with arguments _ at _.

• call the method _ with arguments _ at _ :: token, argument-list, object → action
  [giving an object | binding | escaping with (object, object) | diverging | failing]
  [using current bindings | current storage].

(1) call the method Sel:token with arguments args:argument-list at ob:object =
      enact application of
      the method-abstraction yielded by ((the methods of ob) at Sel)
      to (ob, the successor of the integer bound to sender, args)
      trap
      check the given integer#2 is the successor of the integer bound to sender and then
      give the given object#1
      or
      check not the given integer#2 is
      the successor of the integer bound to sender and then
      escape with the given tuple.
B.3.1.6 Primitive Methods

introduces: primitive method number _ receiving arguments _.

• primitive method number _ receiving arguments _ with non primitive _ ::
  Integer, token", method-abstraction-> abstraction.

/* The BlockContext "value" method equals primitive method number 81 in [GoR83] */
(1) primitive method number 81 receiving arguments Arg:token"
  with non primitive NonPrim:method-abstraction=
  abstraction of
  | enact application of
  | the abstraction stored in the cell yielded by
  | (the instance variables of the given object#1) at block-contents
  to the given argument-list#2.
Appendix C: The Denotational Semantics of Smalltalk

The Denotational Semantics of Smalltalk

C.1 Abstract Syntax

C.1.1 Lexical elements

- Selector
- Integer-Constant
- Non-Root-Class
- Class-Name
- Instance-Variable
- Arg-Variable
- Temporary-Variable

C.1.2 Variables and Literals

- Variable-Name
- Literal

C.1.3 Expressions

- Expression

C.1.4 Classes

- Method
- Method-Def
- Class-Def
C.2 Semantic Functions

needs: Abstract Syntax, Semantic Entities.

C.2.1 Classes

C.2.1.1 User Classes

unfixed-create : (Class-Def → Environment → State → Environment × State) → Class-Def → Environment → State → Environment × State
create : Class-Def → Environment → State → Environment × State

(1) unfixed-create cr < C1:Class-Def C2:Class-Def > env st =
let
  (env', st') = cr C1 env st'
and
  (env'', st'') = cr C2 env' st''
in
  (env'[env''], st'')

(2) unfixed-create cr [ClsNam:Class-Name "inheriting" iid:Class-Name ivs:Instance-Variable* methods:Methods·] env st =
let
  cell = allocate-cell st
and
  env' = env[super-class-methods → (methods-def-by env[iid] st)]
and
  env'' = env'[env' super-class-methods][construct methods env']
and
  st' = st[cell → (empty-env, new-method, (instance-variables-def-by env[iid] st) || ivs),
  
  (empty-env[ClsNam → cell], st')

(3) create = fix(unfixed-create)

C.2.1.2 Constructing new methods

unfixed-construct (Method* → Environment → Environment) → Method* → Environment → Environment
construct : Method* → Environment → Environment

(1) unfixed-construct cn <> env =
  empty-env
Appendix C: The Denotational Semantics of Smalltalk

(2) unfixed-construct cn < M₁:Method M₂:Method+ > env =
    (cn M₁ tok)[cn M₂ tok]

(3) unfixed-construct cn [[ Sel:Selector MethDef:Method-Def]] env =
    empty-env[Sel → (make-method MethDef env)]

(4) construct = fix(unfixed-construct)

C.2.1.3 User-defined Method Abstraction

unfixed-make-method (Method-Def → Environment → method-abstraction) →
    Method-Def → Environment → method-abstraction

make-method : Method-Def → Environment → method-abstraction

(1) unfixed-make-method mm [[ Arg:Arg-Variable* Tmp:Temporary-Variable*]
    E:Expression ]] env =
    λecn.t.λr.etcn.λob.λsnd.λargs.λstate.
    let
    | env' = instance-variables of ob state
    and
    | (env", st") = init-args Arg args state
    and
    | (env"", st"") = init-vars Tmp st'
    in
    evaluate E ecnt retcnt env[env'][env"][env""][self → ob][sender → snd] st"

(2) unfixed-make-method mm [[ Arg:Arg-Variable* Tmp:Temporary-Variable*]
    "primitive" Int:Integer-Constant E:Expression ]] env =
    prim (decimal Int) Arg (mm [[ Arg Tmp E ]] env)

(3) make-method = fix(unfixed-make-method)

C.2.2 Expressions

C.2.2.1 Simple Expressions

evaluate: Expression → ECont → ECont → Environment → Cont

(1) evaluate [[ VarNam:Variable-Name ]] ecnt retcnt env st =
    ecnt (st (env VarNam)) st

(2) evaluate [[ ClsNam:Class-Name ]] ecnt retcnt env st =
    ecnt (env VarNam) st
(3) evaluate \[ \text{Int:Integer-Constant} \] ecnt retcnt env =
\[
\text{let } \\
\text{ecnt' = } \lambda(ivd, md, iv, m).\lambda\text{state}. \\
\text{ecnt (ivd, md, iv, m) state}[(\text{iv object-value} \rightarrow (\text{decimal Int})] \\
\text{in } \\
\text{call-method ecnt' retcnt "new" <> (env "Integer") env}
\]

(4) evaluate \[ \text{"array" Arr:Literal*} \] ecnt retcnt env =
\[
\text{let } \\
\text{ecnt' = } \lambda(ivd, md, iv, m). \\
\text{let elcnt = } \lambda\text{oblist. } \lambda\text{state} \\
\text{ecnt (ivd, md, iv, m) state(\text{iv object-value} \rightarrow \text{oblist})} \\
\text{in } \\
\text{evaluate-list Arr elcnt retcnt} \\
\text{in } \\
\text{call-method ecnt' retcnt "new" <> (env "Array") env}
\]

(5) evaluate \[ \text{E:Expression EL:Expression+} \] =
\[
\text{evaluate } E \lambda \text{ob.(evaluate EL)}
\]

(6) evaluate \[ \text{"↑" E:Expression} \] ecnt retcnt env =
\[
\text{evaluate } E (\lambda \text{ob.(retcnt (ob, env sender))) retcnt env}
\]

(7) evaluate \[ \text{[" Arg:Arg-Variable* E:Expression "]} \] ecnt retcnt env =
\[
\text{let } \\
\text{ecnt' = } \lambda(ivd, md, iv, m).\lambda\text{state.} \\
\text{let blk = } \lambda\text{st. } \lambda\text{rc. } \lambda\text{args.} \\
\text{evaluate } E \text{ ec rc env[initialise Arg args] st} \\
\text{in } \\
\text{ecnt (ivd, md, iv, m) state(\text{iv block-contents} \rightarrow blk)} \\
\text{in } \\
\text{call-method ecnt' retcnt "new" <> (env "BlockContext") env}
\]

(8) evaluate \[ \text{VarNam:Variable-Name "←" E:Expression} \] ecnt retcnt env =
\[
\text{evaluate } E (\lambda \text{st.ecnt ob st((env VarNam) \rightarrow ob)) retcnt env}
\]

(9) evaluate \[ \text{E:Expression Sel:Selector EL:Expression*} \] ecnt retcnt env =
\[
\text{evaluate } E (\lambda \text{st.evaluate-list EL (ob-list.call-method ecnt retcnt Sel ob-list ob) retcnt env st}) retcnt env}
\]

(10) evaluate \[ \text{"self"} \] ecnt retcnt env =
\[
\text{ecnt (env self)}
\]

188
(11) evaluate \([ "super" \text{ Sel:Selector EL:Expression}^* ]\) ecnt retcnt env=

\[
\text{let}
\begin{align*}
\text{retcnt'} &= \lambda (\text{res, send}). \\
& \quad (\text{send} = (\text{succ (env sender)})) \rightarrow \text{ecnt res, retcnt (res, send)} \\
\text{in}
\end{align*}
\]

\[
evaluate E (\lambda \text{ob.e evaluate-list EL (\lambda ob-list.(env super-class-methods) Sel) ob (succ (env sender)) oblist) retcnt env}) \text{ retcnt env}
\]

C.2.2.2 Expression Lists

evaluate-list: Expression* \rightarrow \text{ELCont} \rightarrow \text{ECont} \rightarrow \text{Environment} \rightarrow \text{Cont}

(1) evaluate-list <= elcnt retcnt =

\[
elcnt <=
\]

(2) evaluate-list <= E:Expression EL:Expression* > elcnt retcnt env=

\[
evaluate E (\lambda \text{ob.e evaluate-list EL (\lambda ob-list.elcnt (<ob> || ob-list)) retcnt env}) \text{ retcnt env}
\]

C.3 Semantic Entities

C.3.1 Sorts

**Answer**

Bindable = Object + MethodAbstraction

Cont : State \rightarrow Answer

ECont : Object \rightarrow Cont

ELCont : Object' \rightarrow Cont

Environment : Token \rightarrow Bindable

Integer = \{ ..., -3, -2, -1, 0, 1, 2, 3, ... \}

Object = Identifier \times \text{Environment} \times \text{Environment} \times \text{Environment}

Cell

State: Cell \rightarrow Object

Identifier

Token = Identifier + ReservedTokens

ReservedTokens = \{ super-class-methods, sender, block-contents, object-value, sender, self \}

MethodAbstraction : ECont \rightarrow \text{ECont} \rightarrow \text{Object} \rightarrow \text{Integer} \rightarrow \text{Object'} \rightarrow \text{Cont}

methods-of : Object \rightarrow State \rightarrow \text{Environment}

methods-def-by : Object \rightarrow State \rightarrow \text{Environment}

instance-variables-of : Object \rightarrow State \rightarrow \text{Environment}

instance-variables-def-by : Object \rightarrow State \rightarrow Instance-Variable*
Appendix C: The Denotational Semantics of Smalltalk

(1) instance-variables-of =
\[
\lambda \text{ob.st.} \{
\text{let} \\
(\text{ivs}, \text{meths}) = \text{st ob} \\
\text{in} \\
\text{ivs}
\}
\]

(2) instance-variables-of =
\[
\lambda \text{ob.st.} \{
\text{let} \\
(\text{ivs}, \text{meths, ivdef, methdef}) = \text{st ob} \\
\text{in} \\
\text{ivs}
\}
\]

(3) methods-of =
\[
\lambda \text{ob.st.} \{
\text{let} \\
(\text{ivs}, \text{meths}) = \text{st ob} \\
\text{in} \\
\text{meths}
\}
\]

(4) methods-of =
\[
\lambda \text{ob.st.} \{
\text{let} \\
(\text{ivs}, \text{meths, ivdef, methdef}) = \text{st ob} \\
\text{in} \\
\text{meths}
\}
\]

(5) instance-variables-def-by =
\[
\lambda \text{ob.st.} \{
\text{let} \\
(\text{ivs}, \text{meths, ivdef, methdef}) = \text{st ob} \\
\text{in} \\
\text{ivdef}
\}
\]

(6) methods-def-by =
\[
\lambda \text{ob.st.} \{
\text{let} \\
(\text{ivs}, \text{meths, ivdef, methdef}) = \text{st ob} \\
\text{in} \\
\text{methdef}
\}
\]
C.3.1.1 Standard Class, Class

create-object-called-class: State → Environment × State

(1) create-object-called-class st =
    let
    | cell = allocate-cell st
    and
    | st' = st[cell → (empty-env, new-method, empty-list, new-method)]
    in
    (empty-env["Class" → cell], st')

C.3.1.2 Standard Class, Object

create-object-called-object: State → Environment × State

(1) create-object-called-object env st =
    let
    | cell = allocate-cell st
    and
    | st' = st[cell → (empty-env, new-method, empty-list, empty-map)]
    in
    (empty-env["Object" → cell], st')

C.3.1.3 The new method

new-method : Environment
new-method-abstraction : MethodAbstraction

(1) new-method = empty-env["new" → new-method-abstraction]

(2) new-method-abstraction =
    λecnt.λretcnt.λob.λsnd.λargs.λstate.
    let
    | (env, st') = init-vars (instance-variables-def-by ob state) state
    and
    | ob' = allocate-cell st'
    and
    | env' = env[self → ob']
    and
    | st" = st'[ob' → (env, methods-def-by ob state)]
    in
    ecnt ob' st'

C.3.1.4 Variable Initialisation

InitArgs = Argument-Variable* → Object* → State → Environment × State
unfixed-init-args : InitArgs → InitArgs
Appendix C: The Denotational Semantics of Smalltalk

\textbf{init-args : InitArgs}
\textbf{InitVars = Variable-Name} \to \textbf{State} \to \textbf{Environment} \times \textbf{State}
\textbf{unfixed-init-vars : InitVars \to InitVars}
\textbf{init-vars : InitVars}

(1) \textbf{unfixed-init-args in } \langle \rangle \langle \rangle \textbf{ st} =
\hspace{1cm} \langle \text{empty-env, st} \rangle

(2) \textbf{unfixed-init-args in } \langle \text{Arg:Argument-Variable Arglist:Argument-Variable*} \rangle \langle \text{ob:Object oblist:Object*} \rangle \textbf{ st} =
\hspace{1cm} \langle \text{let cell = allocate-cell st}
\hspace{2cm} \text{and (env, st') = in Arglist oblist st[cell } \to \text{ ob]}
\hspace{2cm} \text{in (env[Arg } \to \text{ cell], st')} \rangle

(3) \textbf{init-args} = \textbf{fix(unfixed-init-args)}

(4) \textbf{unfixed-init-vars in } \langle \rangle \langle \rangle \textbf{ env} =
\hspace{1cm} \langle \text{env} \rangle

(5) \textbf{unfixed-init-vars in } \langle \text{Var:Variable-Name Varlist:Variable-Name*} \rangle \textbf{ st} =
\hspace{1cm} \langle \text{let cell = allocate-cell st}
\hspace{2cm} \text{and (env, st') = in Varlist st[cell } \to \text{ ]}
\hspace{2cm} \text{in (env[Arg } \to \text{ cell], st')} \rangle

(6) \textbf{init-vars} = \textbf{fix(unfixed-init-vars)}

\textbf{C.3.1.5 Calling a method}

\textbf{call-method: ECont} \to \textbf{ECont} \to \textbf{Selector} \to \textbf{Object'} \to \textbf{Object} \to \textbf{Environment} \to \textbf{Cont}

(1) \textbf{call-method ecnt retcnt sel args ob env st} =
\hspace{1cm} \langle \text{let retcnt' = } \lambda (\text{res, send}).(\text{send } = (\text{env sender})) \to \text{ ecnt res, retcnt (res, send)}
\hspace{2cm} \text{in (methods-of ob st sel) ecnt retcnt' (ivd, md, iv, m) (succ (env sender)) args st} \rangle

192
C.3.1.6 Primitive Methods

prim : Integer → Arg-Variable* → Cont → MethodAbstraction

/* BlockContext value */
(1) prim 81 Arg cnt =
   λecnt.λretcnt.λob.λsnd.λargs.λstate.
   state ((instance-variables-of ob state) block-contents) state ecnt retcnt args

C.3.2 Environments

empty-env: Environment

(1) empty-env = λt.undefined

(2) env[env'] = λt.((env' t) = undefined) → env t, env' t

(3) env[t → b] = λt'.(t = t') → b, env t'

C.3.3 Integers

succ : Integer → Integer

the successor function on Integers - left unspecified here

C.3.4 Allocating Cells

allocate-cell : State → Cell

returns an unused Cell in the Store - left unspecified here
Appendix

D

Proof of Equivalence of Two Action Semantics Definitions

This appendix presents a proof of the equivalence of the Action Semantics presented in section 6.2 and the Action Semantics presented in section 6.3.1. The proofs here make use of the rules presented in appendix B of [Mos92]. The codes given to the right of each step of the proofs refer to the appropriate section in [Mos92] which contains the rule that has been used. The number in brackets refers to the rule number. For example "B.2.1 (7)", refers to rule 7 in section B.2.1. Proofs are only presented for two of the rules, since the other two rules remain the same.

Rule A1:

| furthermore bind Id to the value of E
| ⇔ | rebind
| moreover
| bind Id to the value of E
| ⇔ | produce the current bindings
| moreover
| bind Id to the value of E

B.3.1 (2)  B.3.1 (1)
Rule A2:

unfolding
   check the value of B is true and then  
   execute S hence unfold
or
   check the value of B is false and then rebind

\[ \iff \]
unfolding
   give the true yielded by (the value of B is true) then give ()
   and then
   execute S hence unfold
or
   give the true yielded by (the value of B is false) then give ()
   and then
   rebind  \( B.2.1 (7) \)

\[ \iff \]
unfolding
   give true & (the value of B is true) then give ()
   and then
   execute S hence unfold
or
   give true & (the value of B is false) then give ()
   and then
   rebind  \( B.1.2 (3) \)

\[ \iff \]
unfolding
   give true & (the value of B is true) then give ()
   and then
   execute S hence unfold
or
   give true & (the value of B is false) then give ()
   and then
   produce the current bindings  \( B.3.1 (1) \)
Appendix E: Action Semantics Proofs

Note 1: References to rules of the form, C.x.y....z (a), refer to axiom (a), in section C.x.y....z of [Mos92].

Note 2: There are a number of errors in Appendix C of [Mos92]. In particular the following points, [Las95], should be noted before attempting to read the proofs presented in this appendix.

Axiom (3) section C.3.3.2.2 should read:

(3) \[ A_1' O A_2' \] : \[ Intermediate Action-Infix Intermediate \]
    | \[ Completed Interleaving Intermediate \]
    | \[ Intermediate Interleaving Completed \] \Rightarrow
    simplified \[ A_1' O A_2' \] = \[ A_1' O A_2' \] .

Axiom (4) of C.3.3.2.4 and axiom (4) of C.3.3.2.5 should have, "or", inserted after, \( O : \).

The following axioms should be added to section C.3.3.2.4.

(6) \[ \text{given(} A_1:\text{Acting "before" } A_2:\text{Acting } \text{b:bindings} \text{)} = \]
    \[ \text{given(} A_1, \text{d) "before" given(} A_2, \text{d } \text{b} \text{)} \text{].} \]

(7) \[ \text{given(} A_1:\text{Acting "then before" } A_2:\text{Acting } \text{b:bindings} \text{)} , \text{d: data}) = \]
    \[ \text{given(} A_1, \text{d) "then before" } A_2 \text{ b} \text{].} \]

E.1 Proof of Lemma 1

Lemma 1 - Breaking down a run into a series of discrete steps

\[ \text{run}_n( A:\text{Acting, } \text{l:local-info) \geq} \]
\[ (\text{A}^{n+1}:\text{Terminated, } \text{l}^{n+1}:\text{local-info, } \text{c}^{n}:\text{commitment}) \]
\[ \Leftrightarrow \text{stepped}(A, l) \geq (A':\text{Intermediate, } l':\text{local-info, c':commitment}) \land \]
\[ \text{stepped}(A', l') \geq (A'':\text{Intermediate, } l'':\text{local-info, c'':commitment}) \land \]
\[ \text{stepped}(A''', l''') \geq (A''':\text{Intermediate, } l''':\text{local-info, c'''':commitment}) \land \]
\[ \text{stepped}(A^{n-1}, l^{n-1}) \geq (A^n:\text{Intermediate, } l^n:\text{local-info, c}^{n-1}:\text{commitment}) \land \]
\[ \text{stepped}(A^n, l^n) \geq (A^{n+1}, l^{n+1}, c^n) \]

where run\(_n\) is the relation run defined by n applications of axiom (1) in section C.3.3 [Mos92].
Appendix E: Action Semantics Proofs

Proof

Let $P(n)$ be the property that the above lemma applies to $\text{run}_n$. Therefore to show that the property holds for all natural numbers $n$:

1) $P(0)$

$$\text{run}_0( A:\text{Acting}, l:\text{local-info} ) \geq ( A', l', c ) \iff \text{stepped}( A, l ) \geq ( A':\text{Terminated}, l':\text{local-info}, c:\text{commitment} )$$

Follows directly from C.3.3 (2), and the fact that there are no applications of axiom C.3.3 (1).

2) $P(n) \Rightarrow P(n+1)$

$$\text{run}_{n+1}( A:\text{Acting}, l:\text{local-info} ) \geq ( A^{n+1}, m^{n+1}, c^n ) \iff \text{stepped}( A, l ) \geq ( A', l', c ) \land \text{run}_n( A', l' ) \geq ( A^{n+2}, m^{n+2}, c^{n+1} ) \quad (\text{From C.3.3 (1)})$$

$$\text{stepped}( A', l' ) \geq ( A'':\text{Intermediate}, l'':\text{local-info}, c':\text{commitment} ) \land \text{stepped}( A'', l'' ) \geq ( A'':\text{Intermediate}, l'':\text{local-info}, c':\text{commitment} ) \land \text{stepped}( A^n, l^n ) \geq ( A^{n+1}, m^{n+1}, c^n ) \land \text{stepped}( A^{n+1}, m^{n+1} ) \geq ( A^{n+2}, m^{n+2}, c^{n+1} ) \land \text{stepped}( A^{n+2}, m^{n+2} ) \geq ( A^{n+3}, m^{n+3}, c^{n+2} ) \quad (\text{Follows from } P(n))$$

Therefore the lemma follows from mathematical induction [Hen90].

\[ \square \]

E.2 Proof of Lemma 2

Lemma 2 - Provision of transient and scoped information is order independent

$$\text{given}( \text{received}( A:\text{Action}, b:\text{bindings} ), d:\text{data} ) \iff \text{received}( \text{given}( A, d ), b )$$

To prove this we must first prove 8 other lemmas.

Lemma 2.1

$$\text{given}( \text{received}( A, b:\text{bindings} ), d:\text{data} ) \iff \text{received}( \text{given}( A, d ), b )$$

where $A$: Simple-Action | [[] "unfolding" Action [[]]
Appendix E: Action Semantics Proofs

Proof

LHS
⇔ given( A, b, d )
⇔ ( A, d, b )

RHS
⇔ received( A, d, b )
⇔ ( A, d, b )

Hence LHS ⇔ RHS

Lemma 2.2
( given( received( A:Action, b:j:bindings), d:j: data ) ⇔ received( given( A, d ), b ) )
⇒ ( given( received( [ O A ] , b:bindings ), d: data ) ⇔ received( given( [ O A ] , d ), b ) )

where O: "indivisibly" | "patiently"

Proof

LHS
⇔ given( [ O (received( A, b )) ] , d )
⇔ [ O (given( received( A, b ), d )) ]
⇔ [ O (received( given( A, d ), b )) ]

RHS
⇔ received( [ O (given( A, d )) ] , b )
⇔ [ O (received( given( A, d ), b )) ]

Hence LHS ⇔ RHS

Lemma 2.3
( given( received( A_1:Action, b_1:bindings ), d_1: data ) ⇔ received( given( A_1, d_1 ), b_1 ) )
\land
( given( received( A_2:Action, b_2:bindings ), d_2: data ) ⇔ received( given( A_2, d_2 ), b_2 ) )
⇒ ( given( received( [ A_1 O A_2 ] , b:bindings ), d: data ) ⇔ received( given( [ A_1 O A_2 ] , d ), b ) )

where O: "or" | "and" | "and then" | "moreover" | "and then moreover"
Appendix E: Action Semantics Proofs

Proof

LHS
\[ \iff \text{given(} \{ \text{received}(A_1, b)\} \text{O (received}(A_2, b)) \} , d) \] (C.3.3.2.5 (4))
\[ \iff \{ \text{given( received}(A_1, b), d)\} \text{O (given( received}(A_2, b), d)) \} \] (C.3.3.2.4 (4))
\[ \iff \{ \text{received( given(A_1, d), b)\} O (received( given(A_2, d), b)) \} \] (assumption)

RHS
\[ \iff \text{received(} \{ \text{given(A_1, d)}\} \text{O (given}(A_2, d)) \} , b) \] (C.3.3.2.4 (4))
\[ \iff \{ \text{received( given(A_1, d), b)\} O (received( given(A_2, d), b)) \} \] (C.3.3.2.5 (4))

Hence LHS \iff RHS

Lemma 2.4

( given( received(A_1:Action, b_1:bindings), d_1:data ) \iff received( given(A_1, d_1), b_1 ))
( given( received(A_2:Action, b_2:bindings), d_2:data ) \iff received( given(A_2, d_2), b_2 ))
\[ \Rightarrow ( \text{given( received(} \{ A_1 "hence" A_2 \} , b:bindings), d: data) \iff received( \text{given(} \{ A_1 "hence" A_2 \} , d), b )) \]

Proof

LHS
\[ \iff \text{given(} \{ \text{received}(A_1, b)\} "hence" A_2 ) \} ) \] (C.3.3.2.5 (5))
\[ \iff \{ \text{given( received}(A_1, b, d)\} "hence" (given(A_2, d)) \} \] (C.3.3.2.4 (4))
\[ \iff \{ \text{received( given}(A_1, d), b)\} "hence" (given(A_2, d)) \} \] (assumption)

RHS
\[ \iff \text{received(} \{ \text{given}(A_1, d)\} "hence" (given(A_2, d)) \} ) \] (C.3.3.2.4 (4))
\[ \iff \{ \text{received( given}(A_1, d), b)\} "hence" (given(A_2, d)) \} \] (C.3.3.2.5 (5))

Hence LHS \iff RHS

Lemma 2.5

( given( received(A_1:Action, b_1:bindings), d_1:data ) \iff received( given(A_1, d_1), b_1 ))
( given( received(A_2:Action, b_2:bindings), d_2:data ) \iff received( given(A_2, d_2), b_2 ))
\[ \Rightarrow ( \text{given( received(} \{ A_1 "before" A_2 \} , b:bindings), d: data) \iff received( \text{given(} \{ A_1 "before" A_2 \} , d), b )) \]

199
Appendix E: Action Semantics Proofs

Proof

LHS
\[\Rightarrow \text{given}(\ [\text{received}(A_1, b) \ "before" \ A_2 b \ ] , d) \]
\[\Rightarrow \ [\text{given}(\ \text{received}(A_1, b), d) \ "before" \ (\text{given}(A_2, d)) b \ ] \]
\[\Rightarrow \ [\text{received}(\ \text{given}(A_1, d), b) \ "before" \ (\text{given}(A_2, d)) b \ ] \]

RHS
\[\Rightarrow \ \text{received}(\ [\text{given}(A_1, d) \ "before" \ (\text{given}(A_2, d)) \ ] \]
\[\Rightarrow \ [\text{received}(\ \text{given}(A_1, d), b) \ "before" \ (\text{given}(A_2, d)) b \ ] \]

Hence LHS \(\Rightarrow\) RHS

Lemma 2.6
\[
(\ \text{given}(\ \text{received}(A_1:Action, b_1:bindings), d_1:\text{data}) \Rightarrow
\text{received}(\ \text{given}(A_1, d_1), b_1) ) \land
(\ \text{given}(\ \text{received}(A_2:Action, b_2:bindings), d_2:\text{data}) \Rightarrow
\text{received}(\ \text{given}(A_2, d_2), b_2) )
\]
\[\Rightarrow (\ \text{given}(\ [A_1 O A_2 \ ] , b:\text{bindings}), d:\text{data}) \Rightarrow
\text{received}(\ \text{given}(\ [A_1 O A_2 \ ] , d), b) )
\]

where O: "then" | "trap" | "then moreover"

Proof

LHS
\[\Rightarrow \text{given}(\ [\text{received}(A_1, b) \ O (\text{received}(A_2, b)) \ ] , d) \]
\[\Rightarrow \ [\text{given}(\ \text{received}(A_1, b), d) \ O (\text{received}(A_2, b)) \ ] \]
\[\Rightarrow \ [\text{given}(\ \text{received}(A_1, d), b) \ O (\text{received}(A_2, b)) \ ] \]

RHS
\[\Rightarrow \ \text{received}(\ [\text{given}(A_1, d) \ O (\text{given}(A_2, d)) \ ] , b) \]
\[\Rightarrow \ [\text{given}(\ \text{received}(A_1, d), b) \ O (\text{received}(A_2, b)) \ ] \]

Hence LHS \(\Rightarrow\) RHS

Lemma 2.7
\[
(\ \text{given}(\ \text{received}(A_1:Action, b_1:bindings), d_1:\text{data}) \Rightarrow
\text{received}(\ \text{given}(A_1, d_1), b_1)) \land
(\ \text{given}(\ \text{received}(A_2:Action, b_2:bindings), d_2:\text{data}) \Rightarrow
\text{received}(\ \text{given}(A_2, d_2), b_2))
\]
\[\Rightarrow (\ \text{given}(\ [A_1 "\text{thenence}\" A_2 \ ] , b:\text{bindings}), d:\text{data}) \Rightarrow
\text{received}(\ \text{given}(\ [A_1 "\text{thenence}\" A_2 \ ] , d), b) )
\]
Appendix E: Action Semantics Proofs

Proof

LHS
⇒ given( [ (received(A1, b)) "thence" A2 ] , d ) (C.3.3.2.5 (5))
⇒ [ (given( received( A1, d ) ) "thence" A2 ] (C.3.3.2.4 (5))
⇒ [ (received( given( A1, d ) , b ) ) "thence" A2 ] (assumption)

RHS
⇒ received( [ (given( A1, d ) ) "thence" A2 ] ) (C.3.3.2.4 (5))
⇒ [ (received( given( A1, d ) , b ) ) "thence" A2 ] (C.3.3.2.5 (5))

Hence LHS ⇒ RHS

Lemma 2.8

( given( received( A1:Action, b1:bindings ), d1:data ) ⇔
  received( given( A1, d1 ) , b1 ) ) ∧
( given( received( A2:Action, b2:bindings), d2:data ) ⇔
  received( given( A2, d2 ) , b2 ) )
⇒ ( given( received( [ A1 "then before" A2 ] , b:bindings ), d:data ) ⇔
  received( given( [ A1 "then before" A2 ] , d ), b ) )

Proof

LHS
⇒ given( [ (received(A1, b)) "then before" A2 b ] , d ) (C.3.3.2.5 (6))
⇒ [ (given( received( A1, b ) , d )) "then before" A2 b ] (C.3.3.2.4 (7))
⇒ [ (received( given( A1, d ) , b ) ) "then before" A2 b ] (assumption)

RHS
⇒ received( [ (given( A1, d ) ) "then before" A2 ] , b ) (C.3.3.2.4 (5))
⇒ [ ( received( given( A1, d ), b ) ) "then before" A2 b ] (C.3.3.2.5 (6))

Hence LHS ⇒ RHS

Lemma 2 is therefore proved from structural induction and lemmas 2.1-2.8.

E.3 Evaluation of [ ld := E ]

1) evaluated( "current bindings", d:data, b:bindings, l:local-info )
   = b (C.3.2 (7))

2) stepped( [ "produce" "current bindings" ] , d:data, b:bindings, l:local-info )
   = ( "completed", (), b, l, uncommitted ) (C.3.3.1.3 (2); 1)
Appendix E: Action Semantics

Proofs

3) \( \text{evaluated}( \text{ld}, d:\text{data}, b:\text{bindings}, l:\text{local-info} ) \)
\[ = \text{ld}:\text{token} \quad (C.3.2 \ (1)) \]

4) \( \text{evaluated}( \ [\ \text{"the value of" } E:\text{Expression}\ ] , d:\text{data}, b:\text{bindings}, l:\text{local-info}) \)
\[ = \text{the_value_of}( E, \text{conv}^{-1}(b) ) \quad (C.3.2 \ (2); \text{definition of "the value of"}) \]

5) \( \text{stepped}( \ [\ \text{"bind" } \text{ld}:\text{token} \ \text{"to"} \ [\ \text{"the value of" } E:\text{Expression}\ ] \ ] , d:\text{data}, b:\text{bindings}, l:\text{local-info} ) \)
\[ = (\ \text{"completed"}, () \ \text{map ld to the_value_of}( E, \text{conv}^{-1}(b) ), l, \text{uncommitted} ) \quad (C.3.3.1.3 \ (1); 3; 4) \]

6) \( \ [\ \text{"produce" } \text{"current bindings"} \ ] d:\text{data} b:\text{bindings } \text{"moreover"} \)
\[ \quad [\ \text{"bind" } \text{ld}:\text{token} \ \text{"to"} [\ \text{"the value of" } E:\text{Expression}\ ] ] d b ] \]
\[ : [\ \text{Intermediate Interleaving Intermediate} ] \quad (C.2.1) \]

7) \( \ [\ \text{"produce" } \text{"current bindings"} \ ] d:\text{data} b:\text{bindings } \text{"moreover"} \)
\[ \quad \text{"completed"} () \ \text{map ld:token to the_value_of}( E:\text{Expression}, \text{conv}^{-1}(b) ) ] \]
\[ : [\ \text{Intermediate Interleaving Completed} ] \quad (C.2.1) \]

8) \( \text{stepped}( [\ [\ \text{"produce" } \text{"current bindings"} \ ] d:\text{data} b:\text{bindings } \text{"moreover"} \)
\[ \quad [\ \text{"bind" } \text{ld}:\text{token} \ \text{"to"} [\ \text{"the value of" } E:\text{Expression}\ ] ] d b ] , l:\text{local-info} ) \]
\[ \geq (\ \text{simplified } [\ [\ \text{"produce" } \text{"current bindings"} \ ] d b \ \text{"moreover"} \)
\[ \quad \text{"completed"} () \ \text{map ld to the_value_of}( E, \text{conv}^{-1}(b) ) ] , l, \text{uncommitted} ) \quad (C.3.3.2.1 (6); 6; 5) \]
\[ = ( \ [\ [\ \text{"produce" } \text{"current bindings"} ] d b \ \text{"moreover"} \)
\[ \quad \text{"completed"} () \ \text{map ld to the_value_of}( E, \text{conv}^{-1}(b) ) ] , l, \text{uncommitted} ) \quad (C.3.3.2.2 (3); 7) \]

9) \( \text{stepped}( [\ [\ \text{"produce" } \text{"current bindings"} \ ] d:\text{data} b:\text{bindings} \text{"moreover"} \text{"completed"} () \)
\[ \quad \text{map ld:token to the_value_of}( E:\text{Expression}, \text{conv}^{-1}(b) ) ] , l:\text{local-info}, \text{uncommitted} ) \]
\[ \geq ( \ \text{simplified } \ [\ [\ \text{"complete"} ] b \ \text{"moreover"} \text{"completed"} () \)
\[ \quad \text{map ld to the_value_of}( E, \text{conv}^{-1}(b) ) ] , l, \text{uncommitted} ) \quad (C.3.3.2.1 (5); 7; 2) \]
\[ = (\ \text{"completed"}, (), \text{overlay}( \text{map ld to the_value_of}( E, \text{conv}^{-1}(b) ), b ) , l, \text{uncommitted} ) \quad (C.3.3.2.2 (10)) \]

10) \( \text{run}(\ \text{given( received(} [\ [\ \text{"produce" } \text{"current bindings"} ] \text{"moreover"} \)
\[ \quad [\ \text{"bind" } \text{ld}:\text{token} \ \text{"to"} [\ \text{"the value of" } E:\text{Expression}\ ] ] d b ] \]
\[ , b:\text{bindings}, d:\text{data}, l:\text{local-info} ) \]
\[ = \text{run}( [\ [\ \text{"produce" } \text{"current bindings"} ] d b \ \text{"moreover"} \)
\[ \quad [\ \text{"bind" } \text{ld}:\text{token} \ \text{"to"} [\ \text{"the value of" } E ] ] d b ] , l ] \quad (C.3.3.2.4; C.3.3.2.5) \]
\[ \geq (\ \text{"completed"}, (), \text{overlay}( \text{map ld to the_value_of}( E, \text{conv}^{-1}(b) ), b ) , l, \text{uncommitted} ) \quad (C.3.3 (1); C.3.3 (2); 8; 9) \)
E.4 Evaluation of \([ B^* S ]\), value of \(B\) is false

\[ Y_1 = \begin{array}{c} \text{"true" "," ("the value of" } B:\text{Boolean "is" "true" ) } \end{array} \]
\[ Y_5 = \begin{array}{c} \text{"true" "," ("the value of" } B:\text{Boolean "is" "false" ) } \end{array} \]

\[ X_2 = \begin{array}{c} \text{"give" } Y_1 \text{ d:data } b:\text{bindings } \text{"then" } \begin{array}{c} \text{"give" } () \text{ } b \end{array} \end{array} \]
\[ X_3 = \begin{array}{c} \text{given( received( } \begin{array}{c} \text{execute } S \end{array} , b:\text{bindings }, d:\text{data} ) \text{"hence" } \begin{array}{c} \text{"unfolding" } \begin{array}{c} X_4 \text{"or" } X_7 \end{array} \text{ } d \end{array} \end{array} \]
\[ X_4 = \begin{array}{c} \text{"and then" } X_3 \end{array} \]
\[ X_6 = \begin{array}{c} \text{"give" } Y_5 \text{ d:data } b:\text{bindings } \text{"then" } \begin{array}{c} \text{"give" } () \text{ } b \end{array} \end{array} \]
\[ X_7 = \begin{array}{c} \text{"produce" } \text{"current bindings" } d \text{ } b \end{array} \]

\[ A_2 = \begin{array}{c} \text{"give" } Y_1 \text{ "then" } \begin{array}{c} \text{"give" } () \end{array} \end{array} \]
\[ A_3 = \begin{array}{c} \text{execute } S \text{ "hence" } \text{"unfold" } \end{array} \]
\[ A_4 = \begin{array}{c} \text{"and then" } A_3 \end{array} \]
\[ A_6 = \begin{array}{c} \text{"give" } Y_5 \text{ "then" } \begin{array}{c} \text{"give" } () \end{array} \end{array} \]
\[ A_7 = \begin{array}{c} \text{"produce" } \text{"current bindings" } \end{array} \]

1) \[ \text{evaluated( } Y_1 , d:\text{data} , b:\text{bindings} , l:\text{local-info} \) \]
\[ = \text{nothing} \quad \text{(C.3.2 (2))} \]

2) \[ \text{stepped( } [ \text{"give" } Y_1 ] , d:\text{data} , b:\text{bindings} , l:\text{local-info} \) \]
\[ = \text{("failed", } l, \text{uncommitted)} \quad \text{(C.3.3.1 (1); 1)} \]

3) \[ X_2 : [ \text{Intermediate Sequencing Intermediate} ] \quad \text{(C.2.1)} \]

4) \[ [ \text{"failed" } \text{"then" } \begin{array}{c} \text{"give" } () \end{array} \text{ } b:\text{bindings} ] : [ \text{Failed Sequencing Intermediate} ] \quad \text{(C.2.1)} \]

5) \[ \text{stepped( } X_2 , l:\text{local-info} \) \]
\[ \geq \text{(simplified } [ \text{"failed" } \text{"then" } \begin{array}{c} \text{"give" } () \end{array} \text{ } b ] , l, \text{uncommitted}) \]
\[ = \text{("failed", } l, \text{uncommitted)} \quad \text{(C.3.3.2.1 (5); 2; 3)} \]
\[ \quad \text{(C.3.3.2.2 (1); 7)} \]

6) \[ X_4 : [ \text{Intermediate Sequencing Intermediate} ] \quad \text{(C.2.1)} \]

7) \[ [ \text{"failed" } \text{"and then" } X_3 ] : [ \text{Failed Sequencing Intermediate} ] \quad \text{(C.2.1)} \]

8) \[ \text{stepped( } X_4 , l:\text{local-info} \) \]
\[ \geq \text{(simplified } [ \text{"failed" } \text{"and then" } X_3 ] , l, \text{uncommitted}) \]
\[ = \text{("failed", } l, \text{uncommitted)} \quad \text{(C.3.3.2.1 (5); 6; 5)} \]
\[ \quad \text{(C.3.3.2.2 (1); 7)} \]

9) \[ \text{evaluated( } Y_5 , d:\text{data} , b:\text{bindings} , l:\text{local-info} \) \]
\[ = \text{true} \quad \text{(C.3.2 (2))} \]
Appendix E: Action Semantics Proofs

10) stepped ( [ "give" Y₅ ] , d:data, b:bindings, l:local-info )
   = ( "completed", true, empty-map, l, uncommitted )
   ( C.3.3.1.2 (1); 9 )

11) X₆ : [ Intermediate Sequencing Intermediate ]
    ( C.2.1 )

12) [ "give" "()" ] : Simple-Action
    ( C.2.1 )

13) stepped( X₆, l:local-info )
   ≥ ( simplified [ "completed" "true" empty-map "then" [ "give" "()" ] b ] ,
      l, uncommitted )
   ( C.3.3.2.1.2 (5); 10; 11 )
   = ( [ "completed" () empty-map "and" ( given([ "give" "()" ] , b, true ) ) ] ,
      l, uncommitted )
   ( C.3.3.2.2 (8) )
   = ( [ "completed" () empty-map "and" [ "give" "()" ] true b ] ,
      l, uncommitted )
   ( C.3.3.2.4 (2); 12 )

14) evaluated( "()", d:data, b:bindings, l:local-info )
   = ()
   ( C.3.2(1) )

15) stepped( [ "give" "()" ] , d:data, b:bindings, l:local-info )
   = ( "completed", (), empty-map, l, uncommitted )
   ( C.3.3.1.2 (1); 14 )

16) [ "completed" () empty-map "and" [ "give" "()" ] true b:bindings ] :
    [ Completed Interleaving Intermediate ]
    ( C.2.1 )

17) stepped( [ "completed" () empty-map "and" [ "give" "()" ] true b:bindings ] ,
      l:local-info )
   ≥ ( simplified [ "completed" () empty-map "and" "completed" () empty-map ] ,
      l, uncommitted )
   ( C.3.3.2.1 (6); 15; 16 )
   = ( "completed", (), empty-map, uncommitted )
   ( C.3.3.2.2 (6) )

18) evaluated( "current bindings", d:data, b:bindings, l:local-info )
   = b
   ( C.3.2 (7) )

19) stepped( [ "produce" "current bindings" ] , d:data, b:bindings, l:local-info )
   = ( "completed", (), b, l, uncommitted )
   ( C.3.3.1.3 (2); 18 )

20) X₇ : [ Intermediate Sequencing Intermediate ]
    ( C.2.1 )

21) [ [ "completed" () empty-map "and" [ "give" "()" ] true b:bindings ]
   "and then" [ "produce" "current bindings" ] d:data b ] :
    [ Intermediate Action-Infix Intermediate ]
    ( C.2.1 )

22) stepped( X₇, l:local-info )
   ≥ simplified( [ [ "completed" () empty-map "and" [ "give" "()" ] true b ]
      "and then" [ "produce" "current bindings" ] d:b ] ,
      l, uncommitted )
   ( C.3.3.2.1 (5); 20; 13 )
   = ( [ [ "completed" () empty-map "and" [ "give" "()" ] true b ]
      "and then" [ "produce" "current bindings" ] d:b ] ,
      l, uncommitted )
   ( C.3.3.2.4 (2); 12 )

204
Appendix E: Action Semantics Proofs

\[ \text{Intermediate Sequencing Intermediate} \]

23) \[ \text{completed} \] () empty-map "and" [ "give" "()" ] true \( b \):bindings

and then [ "produce" "current bindings" ] \( d \):data \( b \) :

Intermediate Sequencing Intermediate \[ \text{Intermediate Sequencing Intermediate} \] (C.2.1)

24) stepped( \[ \text{completed} \] () empty-map "and" [ "give" "()" ] true \( b \):bindings

and then [ "produce" "current bindings" ] \( d \):data \( b \)

\[ l \), uncommitted \]

\[ \text{Intermediate Sequencing Intermediate} \]

\[ \text{Intermediate Sequencing Intermediate} \] (C.3.3.2.1 (5); 23; 17)

25) \[ \text{completed} () empty-map "and"

[ "produce" "current bindings" ] \( d \):data \( b \):bindings

Intermediate Sequencing Intermediate \[ \text{Intermediate Sequencing Intermediate} \] (C.2.1)

26) stepped( \[ \text{completed} () empty-map "and"

[ "produce" "current bindings" ] \( d \):data \( b \):bindings

\[ l \), local-info \]

\[ \text{Intermediate Sequencing Intermediate} \]

\[ \text{Intermediate Sequencing Intermediate} \] (C.3.3.2.1 (6); 25; 19)

27) stepped( \[ \text{X4} "or" \text{X7} \] , \( l \):local-info

\[ \text{Intermediate Sequencing Intermediate} \]

\[ \text{Intermediate Sequencing Intermediate} \] (C.3.3.2.1 (7); 8)

28) stepped( \[ \text{unfolding} [ \text{A4} "or" \text{A7} \] , \( d \):data, \( b \):bindings, \( l \):local-info

\[ \text{Intermediate Sequencing Intermediate} \]

\[ \text{Intermediate Sequencing Intermediate} \] (C.3.3.2.1 (1))

29) run( given( received( unfolded( \[ \text{A4} "or" \text{A7} \] , \[ "unfolding" \[ \text{A4} "or" \text{A7} \] )

\[ \text{Intermediate Sequencing Intermediate} \]

\[ \text{Intermediate Sequencing Intermediate} \] (C.3.3.2.4; C.3.3.2.5)

E.5 Evaluation of \[ B \ast S \] , value of \( B \) is true

\[ Y_1 = [ "true" "\&" ( "the value of" \( B \):boolean "is" "true" ) ] \]

\[ Y_5 = [ "true" "\&" ( "the value of" \( B \):boolean "is" "false" ) ] \]

\[ X_2 = [ "give" \ Y_1 \ d \):data \( b \):bindings "then" [ "give" "()" ] \( b \) ] \]
Appendix E: Action Semantics Proofs

X₃ = [given( received( [execute S], b:bindings ), d:data ) ”hence”
  [”unfolding” [A₄ ”or” A₇ ]] d ]

X₄ = [X₂ ”and then” X₃ ]

X₆ = [ [ ”give” Y₅ ] d:data b:bindings ”then” [ ”give” ”0” ] b ]

X₇ = [X₆ ”and then” [ ”produce” ”current bindings” ] d b ]

A₂ = [ [ ”give” Y₁ ”then” [ ”give” ”0” ] ]
A₃ = [ [ [execute S ] ”hence” ”unfold” ]
A₄ = [ A₂ ”and then” A₃ ]
A₆ = [ [ ”give” Y₅ ”then” [ ”give” ”0” ] ]
A₇ = [ A₆ ”and then” [ ”produce” ”current bindings” ] ]

1) evaluated( Y₁, d:data, b:bindings, l:local-info )

   = true (C.3.2(2))

2) stepped ( [ ”give” Y₁ ] , d:data, b:bindings, l:local-info )

   = ( ”completed”, true, empty-map, l, uncommitted )

(C.3.3.1.2 (1); 1 )

3) X₂ : [ Intermediate Sequencing Intermediate ]

   (C.2.1 )

4) stepped( X₂, l:local-info )

   ≥ ( simplified [ ”completed” ”true” empty-map ”then” [ ”give” ”0” ] b ] ,
   l, uncommitted )

   = ( [ ”completed” () empty-map ”and” ( given( [ ”give” ”0” ] , b, true ) ) ] ,
   l, uncommitted )

   = ( [ ”completed” () empty-map ”and” [ ”give” ”0” ] true b ] ,
   l, uncommitted )

(C.3.3.2.4 (2))

5) evaluated( ”0”, d:data, b:bindings, l:local-info )

   = ()

   (C.3.2(1))

6) stepped( [ ”give” ”0” ] , d:data, b:bindings, l:local-info )

   = ( ”completed”, (), empty-map, l, uncommitted )

(C.3.3.1.2 (1); 5 )

7) [ ”completed” () empty-map ”and” [ ”give” ”0” ] true b:bindings ] :

   [ Completed Interleaving Intermediate ]

   (C.2.1 )

8) stepped( [ ”completed” () empty-map ”and” [ ”give” ”0” ] true b:bindings ] ,
   l:local-info )

   ≥ ( simplified [ ”completed” () empty-map ”and” ”completed” () empty-map ] ,
   l, uncommitted )

   = ( ”completed”, (), empty-map, uncommitted )

(C.3.3.2.2 (6) )

9) evaluated( Y₅, d:data, b:bindings, l:local-info )

   = nothing (C.3.2(2))
Appendix E: Action Semantics Proofs

10) stepped([[ "give" Y5 ]] , d: data, b: bindings, l: local-info )
    = ( "failed", l, uncommitted )
    ( C.3.3.1 (1); 9 )

11) X6 : [[ Intermediate Sequencing Intermediate ]] 
    ( C.2.1 )

12) [[ "failed" "then" [[ "give" "()" ]] b ]] : [[ Failed Sequencing Intermediate ]] 
    ( C.2.1 )

13) stepped( X6, l:local-info )
    ≥ ( simplified [[ "failed" "then" [[ "give" "()" ]] b ]] , l, uncommitted )
    = ( "failed", l, uncommitted )
    ( C.3.3.2.1 (5); 10; 11 )
    ( C.3.3.2.2 (1); 12 )

14) X7 : [[ Intermediate Sequencing Intermediate ]] 
    ( C.2.1 )

15) [[ "failed" "and then" [[ "produce" "current bindings" ]] d b ]] :
    [[ Failed Sequencing Intermediate ]] 
    ( C.2.1 )

16) stepped( X7, l:local-info )
    ≥ ( simplified [[ "failed" "and then" [[ "produce" "current bindings" ]] d b ]] ,
    l, uncommitted )
    ( C.3.3.2.1 (5); 13; 14 )
    = ( "failed", l, uncommitted )
    ( C.3.3.2.2 (1); 15 )

17) X4 : [[ Intermediate Sequencing Intermediate ]] 
    ( C.2.1 )

18) [[ [[ "completed" () empty-map "and" [[ "give" "()" ]] true b ]] "and then" X3 ]] :
    [[ Intermediate Action-Infix Intermediate ]] 
    ( C.2.1 )

19) stepped( X4, l:local-info )
    ≥ ( simplified [[ [[ "completed" () empty-map "and" [[ "give" "()" ]] true b ]] 
    "and then" X3 ]] , l, uncommitted )
    ( C.3.3.2.1 (5); 4; 17 )
    = ( [[ [[ "completed" () empty-map "and" [[ "give" "()" ]] true b ]] 
    "and then" X3 ]] , l, uncommitted )
    ( C.3.3.2.2 (3); 18 )

20) [[ [[ "completed" () empty-map "and" [[ "give" "()" ]] true b ]] "and then" X3 ]] :
    [[ Intermediate Sequencing Intermediate ]] 
    ( C.2.1 )

21) stepped( [[ [[ "completed" () empty-map "and" [[ "give" "()" ]] true b ]] 
    "and then" X3 ]] , l:local-info )
    ≥ ( simplified [[ "completed" () empty-map "and then" X3 ]] , l, uncommitted )
    ( C.3.3.2.1 (5); 8; 20 )
    = ( [[ "completed" () empty-map "and" X3 ]] , l, uncommitted )
    ( C.3.3.2.2 (7) )
Given that \( X_3 \), when run, completes,

\[ \text{run}(X_3, l:local-info) \geq (\text{"completed" } d':data \ b':bindings, l':local-info, c':commitment) \]


\[ \Leftrightarrow \text{stepped}(X_3, l) \geq (X_3':\text{Intermediate}, l''':local-info, c':commitment) \land \text{stepped}(X_3', l'') \geq (X_3'':\text{Intermediate}, l'''':local-info, c'':commitment) \land \]

\[ \text{stepped}(X_3^{n-1}:\text{Intermediate}, l^{n}:local-info) \geq (X_3^{n}:\text{Intermediate}, l^{n+1}:local-info, c^{n}:commitment) \land \text{stepped}(X_3^{n}, l^{n+1}) \geq (\text{"completed } d' b'\}], l', c) \]

(lemma 1)

23) \[ \text{"completed" } () \text{ empty-map } \"and\" X_3 \] :
\[ \text{Completed Interleaving Intermediate} \land \]
\[ \text{"completed" } () \text{ empty-map } \"and\" X_3^n ] :
\[ \text{Completed Interleaving Intermediate} \]

(C.2.1)

24) \[ \text{stepped}(\text{"completed" } () \text{ empty-map } \"and\" X_3 \] , l:local-info )
\[ \geq (\text{simplified } \text{"completed" } () \text{ empty-map } \"and\" X_3':\text{Intermediate} \] , l'''':local-info, c'''':commitment) \]
\[ = (\text{"completed" } () \text{ empty-map } \"and\" X_3' \] , l'', c'') \land \]
\[ \text{stepped}(\text{"completed" } () \text{ empty-map } \"and\" X_3^n:\text{Intermediate} \] , l^{n+1}:local-info)
\[ \geq (\text{simplified } \text{"completed" } () \text{ empty-map } \"and\" \text{"completed" } d':data \ b':bindings \] , l':local-info, c':commitment) \]
\[ = (\text{"completed" } d' b'\}], l', c') \]

(C.3.3.2.1 (6); 22; 23)

25) \[ X_4 \"or\" X_7 ] : [ \text{Intermediate } \"or\" \text{Intermediate} ] \]

(C.2.1)

26) \[ \text{stepped}(X_4 \"or\" X_7 \] , l )
\[ \geq (\text{simplified } X_4 \"or\" \"failed\" \] , l:local-info, uncommitted ) \]
\[ = (X_4, l, \text{uncommitted}) \]

(C.3.3.2.1 (8); 16; 25)

27) \[ \text{stepped}(\text{"unfolding" } [ A_4 \"or\" A_7 \] ] , d: data, b: bindings, l: local-info) \]
\[ = (\text{given( received( unfolded( } [ A_4 \"or\" A_7 \] ] , [ \"unfolding\" } [ A_4 \"or\" A_7 \] ] ] , b), d), l, \text{uncommitted}) \]
\[ = (X_4 \"or\" X_7 \] , l, uncommitted) \]

(C.3.3.2.3; C.3.3.2.4; C.3.3.2.5)
Appendix E: Action Semantics Proofs

28) \[\text{run( given( received( \{ "unfolding" \{ A4 "or" A7 \}, b:bindings), d:data), l:local-info )} \]
\[= ( \{ "completed" (d', d'') b" \}, m+m+2, c'n+m+1 ) \]
\[\Leftrightarrow \text{run( given( received( X3, b), d ), l) = ( \{ "completed" (d', d'') b" \}, m+m+2, c'n+m+1 )} \]
\[= ( \{ "completed" (d', d'') b" \}, m+m+2, c'n+m+1 ) \]
\[\Leftrightarrow \text{run( given( [ execute S ] "hence" [ execute [ B * S ] ] b, d, l) = ( \{ "completed" (d', d'') b" \}, m+m+2, c'n+m+1 ) } \]
\[\quad \text{(from definition of X3 and rule A2)} \]
\[\Leftrightarrow \text{stepped([I given( received( [ execute S ] b, d ) "hence" given( [ execute [ B * S ] ] d ) ] l) } \]
\[\quad \geq ( \{ I "hence" given( [ execute [ B * S ] ] d ) ] , l' ; c ) \land \]
\[\quad \text{stepped( [ I"hence" given( [ execute [ B * S ] ] d ) ] , l', c' ) } \land \]
\[\quad . \]
\[\quad \text{stepped( [ m-l "hence" given( [ execute [ B * S ] ] d ) ] , l^m ) } \]
\[\quad \geq ( \text{simplified [ "completed" d'b' "hence" given( [ execute [ B * S ] ] d ) ] , l^m+1, c^n ) } \land \]
\[\quad \text{stepped( [ "completed" d' empty-map "and" ( received( given( [ execute [ B * S ] ] d ) , b' ) ) ] , m+l+1, c^n ) } \land \]
\[\quad \geq ( \{ "completed" d' empty-map "and" J \}, m+2, c'n+l ) \]
\[. \]
\[\quad \text{stepped( [ "completed" d' empty-map "and" m-l ] , m+m+1 ) } \]
\[\geq ( \text{simplified [ "completed" d' empty-map "and" [ "completed" d" b" ] , m+m+2, c'n+m+1 ] } \land \]
\[\quad = ( \{ "completed" (d', d'') b" \}, m+m+2, c'n+m+1 ) \]
\[\quad \text{(lemma 1; C.3.3.2.1 (5); C.3.3.2.2 (13); C.3.3.2.1 (6); C.3.3.2.2 (6))} \]
\[ \Leftrightarrow \text{stepped}(\llbracket \text{given( received( } \llbracket \text{execute } S \rrbracket, b), d \rrbracket, l) \rrbracket, I) \]
\[ \geq (I, l', c) \land \]
\[ \text{stepped}(I, l') \]
\[ \geq (I', l'', c') \land \]

\[ \text{stepped( } m-l, m ) \]
\[ \geq (\llbracket \text{"completed" } d' b' \rrbracket, m+l, e^n) \land \]

\[ \text{run( given( received( } \llbracket \text{execute } B \ast S \rrbracket, d), b'), I) \]
\[ \geq (\llbracket \text{"completed" } d' b' \rrbracket, m+l, e^n) \land \]
\[ \text{run( received( given( } \llbracket \text{execute } B \ast S \rrbracket, d), b'), m+l) \]
\[ \geq (\llbracket \text{"completed" } d' b' \rrbracket, m+m', e^n+m+1) \]

(least relation)

(lemma 1)
Appendix F: Properties of the Action Semantics of Smalltalk

This Appendix proves certain properties of the Action Semantics of Smalltalk presented in Appendix B. Specifically one particular aspect (method calls) is examined. These properties are used in the proofs presented in Chapter 6.

The properties that are proved relate to the following clause taken from Appendix B. This clause has been translated into a format similar to that shown for the Action Semantics given in Appendix D.

(1) call the method Sel:token with arguments args:argument-list at ob:object =

| enact application of |
| the method-abstraction yielded by ((the methods of ob) at Sel) |
| to (ob, the successor of the integer bound to sender, args) |

| trap |
| give the true yielded by the given integer#2 is |
| the successor of the integer bound to sender then give () |
| and then |
| give the given object#1 |

| or |
| give the true yielded by not the given integer#2 is |
| the successor of the integer bound to sender then give () |
| and then |
| give the given tuple then escape. |

F.1.1 Definitions

\[ Y_1 = \llbracket\text{application of the method-abstraction yielded by ((the methods of ob) at Sel) to (ob, the successor of the integer bound to sender, args)}\rrbracket \]

\[ Y_2 = \llbracket\text{the true yielded by the given integer#2 is the successor of the integer bound to sender}\rrbracket \]

\[ Y_3 = \llbracket\text{the given object#1}\rrbracket \]

\[ Y_4 = \llbracket\text{the true yielded by not the given integer#2 is the successor of the integer bound to sender}\rrbracket \]

\[ Y_5 = \llbracket\text{the given tuple}\rrbracket \]

\[ A_1 = \llbracket\text{enact } Y_1\rrbracket \]

\[ A_2 = \llbracket\llbracket\text{give } Y_2\rrbracket\text{ then } \llbracket\text{give } ()\rrbracket\rrbracket \]
Appendix F: Properties of the Action Semantics of Smalltalk

Let "abstraction of X" represent the definition of some method bound to "Sel" in the object "ob".

Additionally we shall assume that we can assert:
run(X, d: data, b, l: local-info)
≥ (T: Terminated, l': local-info, c: commitment)

F.1.2 Case: enacting the abstraction fails

1) evaluated(Y, d, b: bindings, l: local-info) = 
"abstraction of" X (from assumption)

2) stepped([ "enact" Y, d, b, l: local-info) ≥ (given(received(X, empty-map), ()), l) (C.3.3.1.5 (1); 1)

3) stepped([ A, "trap" A₈], l) ≥ (given(received(X, empty-map), ()) "trap" A₈], l', c')
^ stepped([ given(received(X, empty-map), ()) "trap" A₈], l) ≥ (A': Intermediate "trap" A₈], l", c")
... ^ stepped([ Aⁿ: Intermediate "trap" A₈], lⁿ⁺², cⁿ⁺¹) ≥ (simplified [ "failed" "trap" A₈], l', uncommitted) = ("failed", l', uncommitted) (C.3.3.1. (5); C.3.3.1.1 (6); 2; assumption)

F.1.3 Case: enacting the abstraction completes

1) evaluated(Y, d, b: bindings, l: local-info) = 
"abstraction of" X (from assumption)

2) stepped([ "enact" Y, d, b, l: local-info) ≥ (given(received(X, empty-map), ()), l) (C.3.3.1.5 (1); 1)
Appendix F: Properties of the Action Semantics of Smalltalk

3) \( \text{stepped}(\llbracket A_i \text{ "trap" } A_s \rrbracket, l) \geq (\llbracket \text{given}(\text{received}(X, \text{empty-map}), () \text{ "trap" } A_s \rrbracket, l', c') \)
\&
\( \text{stepped}(\llbracket \text{given}(\text{received}(X, \text{empty-map}), () \text{ "trap" } A_s \rrbracket, l') \)
\geq (\llbracket A':\text{Intermediate "trap" } A_s \rrbracket, l'', c'')

\(= \text{("completed", d', b', l', uncommitted)} \)

(C.3.3.2.1 (5); C.3.3.2.1 (6); 2; assumption)

F.1.4 Case: enacting the abstraction escapes, \( Y_2 \) is nothing, \( Y_4 \) is true

1) \( \text{evaluated}(Y_2, d, b:\text{bindings}, l:\text{local-info}) = \llbracket \text{"abstraction of" } X \rrbracket \)
\(= \text{(from assumption)} \)

2) \( \text{stepped}(\llbracket \text{"enact" } Y_1 \rrbracket, d, b, l:\text{local-info}) \geq (\text{given}(\text{received}(X, \text{empty-map}), ()), l) \)
\(= \) (C.3.3.1.5 (1); 1)

3) \( \text{stepped}(\llbracket A_i \text{ "trap" } A_s \rrbracket, l) \geq (\llbracket \text{given}(\text{received}(X, \text{empty-map}), () \text{ "trap" } A_s \rrbracket, l', c') \)
\&
\( \text{stepped}(\llbracket \text{given}(\text{received}(X, \text{empty-map}), () \text{ "trap" } A_s \rrbracket, l') \)
\geq (\llbracket A':\text{Intermediate "trap" } A_s \rrbracket, l'', c'')

\(= \text{("completed", d', b', l', uncommitted)} \)

(C.3.3.2.1 (5); C.3.3.2.1 (6); 2; assumption)

4) \( \text{evaluated}(Y_2, d, b:\text{bindings}, l:\text{local-info}) = \) nothing
\(= \) (from assumption)

5) \( \text{stepped}(\llbracket \text{"give" } Y_2 \rrbracket, d, b:\text{bindings}, l:\text{local-info}) = \) ("failed", l, uncommitted)
\(= \) (C.3.3.1 (1); 4)
Appendix F: Properties of the Action Semantics of Smalltalk

6) stepped([[ "give" Y ]] d b:bindings "then" [[ "give" "0" ]] b ] , l:local-info) ≥ (simplified [[ "failed" "then" [[ "give" "0" ]] b ]] , l, uncommitted) = ("failed", l, uncommitted) (C.3.3.2.1 (5); 5)

7) stepped([[ A_2 "and then" A_3 ]] , l:local-info) ≥ (simplified [[ "failed" "and then" A_3 ]] , l, uncommitted) = ("failed", l, uncommitted) (C.3.3.2.2 (1))

8) stepped([[ [ A_2 "and then" A_3 ]] "or" [[ A_5 "and then" A_6 ]] ] , l:local-info) ≥ (simplified [[ "failed" "or" [[ A_5 "and then" A_6 ]] ]] , l, uncommitted) = ([ A_5 "and then" A_6 ] , l, uncommitted) (C.3.3.2.1 (6); 6)

9) evaluated(Y, d, b:bindings, l:local-info) = true (from assumption)

10) stepped([[ give Y ]] , d, b:bindings, l:local-info) = ("completed", true, empty-map, l, uncommitted) (C.3.3.1.2 (1); 9)

11) evaluated( "()", d, b:bindings, l:local-info) = ()

12) stepped([[ "give" "()" ]] , d, b:bindings, l:local-info) = ("completed", (), empty-map, l, uncommitted) (C.3.3.1.2 (1); 11)

13) stepped([A_5], l:local-info) ≥ (simplified [[ "completed" "true" empty-map "then" [[ "give" "()" ]] b ]] , l, uncommitted) = ([ "completed" () empty-map "and" [[ "give" "()" ]] true b ]] , l, uncommitted) (C.3.3.2.1 (5); 10)

14) stepped([[ "completed" () empty-map "and" [[ "give" "()" ]] true b:bindings ]] , l:local-info) ≥ (simplified [[ "completed" () empty-map "and" "completed" () empty-map ]] , l, uncommitted) = ("completed", (), empty-map, l, uncommitted) (C.3.3.2.2 (6))
Appendix F: Properties of the Action Semantics of Smalltalk

15) \[ \text{evaluated}(Y_s, d: \text{tuple}, b:\text{bindings}, I:\text{local-info}) = d \]

16) \[ \text{stepped}(\text{give } Y_s, d b, I:\text{local-info}) \geq (\text{"completed"}, d, \text{empty-map}, I, \text{uncommitted}) \] (C.3.3.1.2 (1); 15)

17) \[ \text{stepped}(\text{"escape" } d b, I:\text{local-info}) \geq (\text{"escaped"}, d, I, \text{uncommitted}) \] (C.3.3.1.1 (2))

18) \[ \text{stepped}(\text{give } Y_s, d b, \text{"then" } \text{"escape" } b) I:\text{local-info}) \geq \text{\{simplified } \text{"completed" } d \text{empty-map } \text{"then" } \text{"escape" } b I, \text{uncommitted}\]

19) \[ \text{stepped}(\text{"completed" } () \text{empty-map } \text{"and" } \text{\{given(} \text{"escape" } b I, d)\}, I, \text{uncommitted}) \geq (\text{\{simplified } \text{"completed" } () \text{empty-map } \text{"and" } \text{"escaped" } d I, I, \text{uncommitted}\]

20) \[ \text{stepped}(\text{A}, \text{then } A_6 I, I:\text{local-info}) \geq \text{\{simplified } \text{"completed" } () \text{empty-map } \text{"and" } \text{\{"give" } ()\}, I, \text{uncommitted}\]

21) \[ \text{stepped}(\text{\{"completed" } () \text{empty-map } \text{"and" } \text{\{"give" } ()\}, I, \text{uncommitted}) \geq (\text{\{simplified } \text{"completed" } () \text{empty-map } \text{"and" } A_6 I, I, \text{uncommitted}\]

22) \[ \text{stepped}(\text{\{"completed" } () \text{empty-map } \text{"and" } A_6 I, I:\text{local-info}) \geq (\text{\{simplified } \text{"completed" } () \text{empty-map } \text{"and" } \text{\{"completed" } () \text{empty-map } \text{"and" } \text{\{given(} \text{"escape" } b I, d)\}, I, \text{uncommitted}\]

23) \[ \text{stepped}(\text{\{"completed" } () \text{empty-map } \text{"and" } \text{\{"completed" } () \text{empty-map } \text{"and" } \text{\{given(} \text{"escape" } b I, d)\}, I, \text{local-info}) \geq (\text{\{simplified } \text{"completed" } () \text{empty-map } \text{"and" } \text{"escaped" } d I, I, \text{uncommitted}\]

215
Appendix F: Properties of the Action Semantics of Smalltalk

24) run([ A, "trap" A, ] , l:local-info)
   ≥ ("escaped", d, l, c:commitment)
   (C.3.3 (1); C.3.3 (2); 3; 8; 20; 21; 22; 23)

F.1.5 Case: enacting the abstraction escapes, Y₂ is true, Y₄ is nothing

1) evaluated(Y₁, d, b:bindings, l:local-info)
   = [ "abstraction of" X ]
   (from assumption)

2) stepped([ "enact" Y, ] , d, b, l:local-info)
   ≥ (given(received(X, empty-map), ()), l)
   (C.3.3.1.5 (1); 1)

3) stepped([ A, "trap" A, ] , l)
   ≥ ([ given(received(X, empty-map), ()) "trap" A, ] , l', c')
   ∧ stepped([ given(received(X, empty-map), ()) "trap" A, ] , l)
   ≥ ([ A.:intermediate "trap" A, ] , l', c')
   ...
   ∧ stepped([ A^n:intermediate "trap" A, ] , l^n+2, c^n+1)
   ≥ (simplified [ "escaped" d "trap" A, ] , l', uncommitted)
   = (given(A, d'), l', uncommitted)
   = ([ A or A, ] , l', uncommitted)
   (C.3.3.2.1 (5); C.3.3.2.1 (6); 2; assumption)

4) evaluated(Y₄, d, b:bindings, l:local-info)
   = nothing
   (from assumption)

5) stepped([ "give" Y, ] , d, b:bindings, l:local-info)
   = ("failed", l, uncommitted)
   (C.3.3.1 (1); 4)

6) stepped([ [ "give" Y, ] d b:bindings "then" [ "give" "(" ] b ] , l:local-info)
   ≥ (simplified [ "failed" "then" [ "give" "(" ] b ] , l, uncommitted)
   = ("failed", l, uncommitted)
   (C.3.3.2.2 (1))

7) stepped([ A, "and then" A, ] , l:local-info)
   ≥ (simplified [ "failed" "and then" A, ] , l, uncommitted)
   = ("failed", l, uncommitted)
   (C.3.3.2.2 (1))
Appendix F: Properties of the Action Semantics of Smalltalk

8) stepped([ [ A_2 "and then" A_3 ] "or" [ A_5 "and then" A_6 ] ] , l:local-info)
   ≥ (simplified [ [ A_2 "and then" A_3 ] "or" "failed" ] , l, uncommitted)  
   = ([ A_2 "and then" A_3 ] , l, uncommitted)  
   (C.3.3.2.1 (6); 7)

9) evaluated(Y,, d, b:bindings, l:local-info)
   = true
   (from assumption)

10) stepped([ give Y,, ] , d, b:bindings, l:local-info)
    = ("completed", true, empty-map, l, uncommitted)  
    (C.3.3.1.2 (1); 9)

11) evaluated( "()", d, b:bindings, l:local-info)
    = ()

12) stepped([ "give" "()" ] , d, b:bindings, l:local-info)
    = ("completed", (), empty-map, l, uncommitted)  
    (C.3.3.1.2 (1); 11)

13) stepped(A_2, l:local-info)
    ≥ (simplified [ "completed" "true" empty-map "then" [ "give" "()" ] b ] , l, uncommitted)  
    = ([ "completed" () empty-map "and" [ "give" "()" ] true b ] , l, uncommitted)  
    (C.3.3.2.1 (5); 10)

14) stepped([ "completed" () empty-map "and" [ "give" "()" ] true b:bindings ] , l:local-info)
    ≥ (simplified [ "completed" () empty-map "and" "completed" () empty-map ] , l, uncommitted)
    = ("completed", (), empty-map, l, uncommitted)  
    (C.3.3.2.2 (8))

15) evaluated(Y,, (ob, d), b:bindings, l:local-info)
    = ob

16) stepped([ give Y,, (ob, d) b ] , l:local-info)
    ≥ ("completed", ob, empty-map, l, uncommitted)  
    (C.3.3.1.2 (1); 15)
17) stepped([ A; and then A ]; l:local-info)
≥ (simplified [ [ "completed" () empty-map "and" [ "give" () ] true b ] "and then" A ]; l, uncommitted)
= ([ [ "completed" () empty-map "and" [ "give" () ] true b ] "and then" A ]; l, uncommitted)

(C.3.3.2.1 (5); 13; C.3.3.2.2 (3))

18) stepped([ [ "completed" () empty-map "and" [ "give" () ] true b ] "and then" A ]; l)
≥ (simplified [ [ "completed" () empty-map "and then" A ]; l, uncommitted)
= ([ [ "completed" () empty-map "and" A ]; l, uncommitted)

(C.3.3.2.1 (5); 14; C.3.3.2.2 (7))

19) stepped([ "completed" () empty-map "and" A ]; l:local-info)
≥ (simplified [ "completed" () empty-map "and" "completed" ob empty-map ]; l, uncommitted)
= ("completed", ob, empty-map, l, uncommitted)

(C.3.3.2.1 (6); 16; C.3.3.2.2 (6))

20) run([ A; "trap" A ]; l:local-info)
≥ ("completed", ob, l, c:commitment)

(C.3.3 (1); C.3.3 (2); 3; 8; 17; 18; 19)