Implementation of Action Semantics in Coq

Author:
Sandra Ward

Supervisor:
Dr. James Power

Submitted to
The School of Computer Applications
Dublin City University

for the degree of
Master of Science

October, 1996

This thesis is based on the candidate's own work

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Science degree is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: SANDRA KARI Date: 6/2/97
Candidate

Date: _____

Acknowledgements

I would like to thank James for all his help and his outstanding patience. I would also like to acknowledge the help of Peter Mosses during my research.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Abstract

Formal Semantics is a topic of major importance in the study of programming language design. Action semantics is a recently developed framework for the specification of formal semantics which allows understandable, modular and reusable semantic descriptions of programming languages. Action laws are algebraic properties of primitive actions and action combinators which can be used to prove the existence of semantic equivalence between pairs of constructs, expressions etc. of programming language.

This thesis endeavours to show how action semantics can be formalised computationally by reporting on the representation of the kernel of action notation in CAML. CAML is a functional language whose type systems allow the user to define his/her own data structures. It allows the definitions of functions manipulating these data structures with the security provided by strict type verification. The representation of the kernel in the specification language of the Coq development system is also outlined. The Coq system is an implementation of the Calculus of Inductive Constructions and provides goal-directed tactic-driven proof search. The proof engine of the Coq system is then used to prove various action laws.

CHAPTER 1

Introduction

1.1 Introduction

Designers, implementors and serious users of languages need a complete and accurate understanding of the semantics (meaning) and the syntax (form) of every construct of the language they are working with [Tennent, 1991]. There is a well developed and widely known mathematical theory of formal languages supporting accurate description of the syntax of languages. A rigorous mathematical theory of the semantics of programming languages is then needed to support correct description and implementation of their meanings, systematic development and verification of programs, analysis of existing programming languages and design of new languages. Formal descriptions are mathematical theories used to model and analyse the essential properties of programming languages and programs [Meyer, 1991]. In this thesis, we propose to adopt action semantics as the specification type we use to formalise programming languages semantically. Action semantics blends formality with good pragmatic features and is one of the most comprehensible and accessible types of semantic specification. Action semantics uses semantic entities called "actions" where actions can either be primitive or composite. A composite action is formed by action combinators which combine two or more primitive actions. These primitive actions and combinators satisfy a series of algebraic properties i.e. action laws which can lead to the proof of the existence of semantic equivalences between pairs of constructs in a programming language. Our overall objective was to prove the truth of these algebraic properties. We proposed to do this by looking at the formal semantics of the action notation and, somehow, represent it in a form which allowed us to prove these properties. We chose the Coq development system as it is a proof assistant and it possesses its own specification language and therefore, was an ideal choice. However, we decided to, firstly, make the translation to CAML to increase our

familiarisation with action notation, the structural operational semantics of action notation and for debugging purposes. Figure 1.1 illustrates the different modules involved in this thesis. In this chapter, we give the advantages and disadvantages of using formal descriptions. We then go on to give an overview of the research underlying this thesis. Finally, the contents of the chapters are outlined.

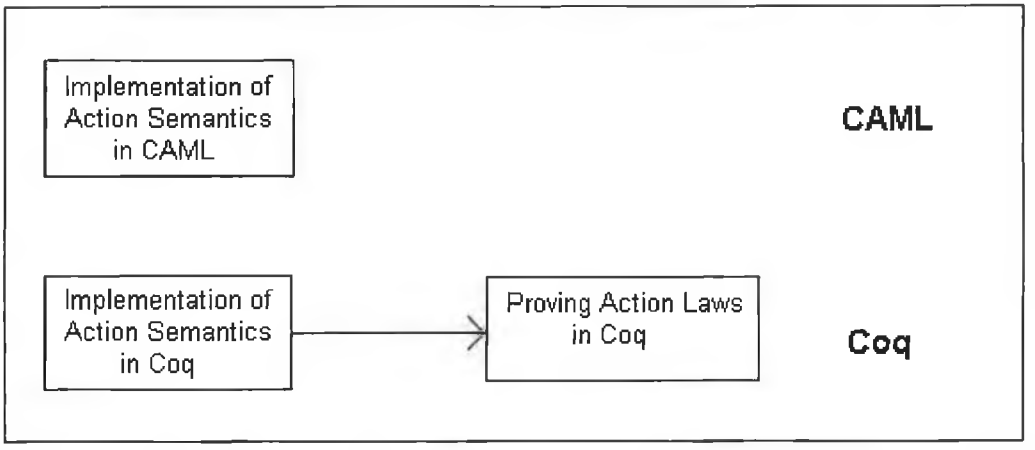

Figure 1.1 Research Modules

1.2 Why Formal Descriptions ?

We now look at the reasons behind the use of formal descriptions for the specification of programming languages.

Formal descriptions are useful for several reasons:

- help in the understanding of languages
- support language standardisation
- provide guidelines for the design of languages
- aid in the writing of compilers and language systems
- support program verification and software reliability
- act as a model for software specification

1.2.1 Help in Language Understanding

A formal specification provides insight that an informal approach would not. It is possible that an informal specification may leave many questions unanswered whereas formal specifications of various programming language issues e.g. data types, block structuring, recursion etc. provide powerful insights. Therefore, programmers that are familiar with formal specification techniques may have a deeper understanding of programming languages.

1.2.2 Support Language Standardisation

A problem faced by programmers is the one of portability i.e. programs are needed which will adapt with minimum difficulty to different environments. For portability, standardisation is needed. Standards are needed for hardware interfaces, programming languages etc. However, some languages (such as Fortran, C, Pascal, Ada) are not free from portability problems. This is due to the fact that a programming language involves a large amount of fine points which are difficult to cover satisfactorily in a document written in a natural language. Formal specifications can help to solve this problem i.e. mathematical techniques are particularly effective whenever circumstances dictate that precision and absence of ambiguity are required.

1.2.3 Guidelines for the Design of Languages

Proper design of programming languages is an important issue. The quality of the result of language design is mostly determined by the designer's talent and experience. As with any design discipline, certain general principles apply. Simplicity of specification is an important guideline i.e. concepts that are difficult to specify often turn out, once they are transposed to language features

to be hard to implement. Although it is balanced with other requirements and there is no absolute criterion in language design, mathematical simplicity usually pays off.

1.2.4 Help in Writing Compilers and Language Systems

Formal descriptions provide a solid basis on which to design compilers, language systems (language systems refer to compilers with the tools that support the use of high level languages e.g. debuggers). The results of some of the formal language description methods e.g. denotational semantics may be understood as high level descriptions of abstract compilers for the languages studied.

1.2.5 Support for Program Verification and Software Reliability

Among the fundamental issues in software engineering are the issues of correctness and robustness of programs. Much time has been devoted by researchers to the development of techniques for proving programs' correctness. The idea is to associate with the program a mathematical transform and to use proof techniques to ensure that this transform achieves the program's desired purpose.

Problems involved with proving the correctness of programs are:

- the purpose of each program or program element must be stated precisely
- the right type of mathematical theories must be developed to reason about programs and prove properties of their behaviour
- efficient tools are needed to support the detailed proof of a system

Formal descriptions of programming languages are essential to achieve the second goal i.e. they provide the mathematical basis for reasoning for programs.

1.2.6 Models for Software Specification

Formal specifications of programming languages have an indirect but important bearing on software specification. The problem plays a fundamental role in program correctness - how can we describe the purpose of a software product precisely and unambiguously without overspecification. Formal descriptions are more concerned, in this context, with the specification of programming languages and not software systems. It transpires, however, that the methods used for the first of these goals gives powerful insights into the second goal. Many of the basic issues and techniques are the same.

1.3 Why Action Semantics ?

Having looked at the advantages of formal descriptions, we now look at the particular type of formal semantic specification which we propose to adopt during this thesis. An important question when considering this research was - why use action semantics for the specification of programming languages? As we know, action semantics is not the only framework available for giving the formal semantic description of programming languages. Why choose action semantics in preference to other types of formal semantics when considering the description of full-scale realistic languages? We should be aware that potential users of action semantics are likely to be reluctant to leave their current frameworks. We will consider the advantages of using action semantics in chapter 2 and will see this type of semantic description combines the strengths of informal descriptions i.e. readibility, understandability with formality.

The development of action semantics originated from denotational semantics. Therefore, it is not surprising to observe many similarities between the two approaches along with many differences. For example, both approaches map abstract syntactic entities compositionally to semantic entities and semantic equations are used to defined semantic functions. The essential difference is concerned with the nature of semantic entities and how they are expressed. Denotational semantics uses higher order functions or so-called Scott domains and uses a rich, typed λ-notation to express particular functions together with the values on which the functions are defined [Mosses, 1992]. When specifying the usual constructs of programming languages, the functions required tend to be rather complex. This is due to the fact that the basic operations on functions provided by λ-notation e.g. application, abstraction do not correspond directly to the basic concepts of programming languages. Purely functional representation can make it difficult to read semantic equations and hard to understand the operational implications. The serious pragmatic problems lie in poor modifiability and extensibility. It is possible to reduce some of the pragmatic problems of denotational semantics by using auxiliary functions representing action primitives and combinators hence defining the interpretation of action notation as higher order functions.

With structural operational semantics, which was developed for use in describing programming languages by Plotkin [Plotkin, 1981] [Plotkin, 1983], transitions are specified in a structural way keeping track of control implicitly. The transitions for a compound phrase depends on the transitions appropriate for its subphrases giving a compositional flavour to structural operational semantics specification. As we know, in [Mosses, 1992] (Appendix C), the definition of action notation is based on a structural operational semantics (SOS). The pragmatic properties of SOS are acceptable but the modifiability

and extensibility of SOS descriptions are better than for denotational semantics but not as good as action semantics.

Looking at some of the applications of action semantics and various tools, we see that a complete formal description of the dynamic semantics of Standard Pascal [Mosses & Watt, 1986] exists. Also, the ANDF-FS [Nielsen & Toft, 1994] is the first example of the use of action semantics in industry. The ANDF-FS is being actively used for reference in the construction of an interpreter for ANDF-FS. The various parties working on this project have very little background in formal semantics, yet they found the document to be quite accessible. The ASD tools can be used for parsing, syntax-directed editing, checking and the interpretation of action semantics specifications. These tools are implemented using the ASF and SDF system [Klint, 1991] which is based on the Centaur system. David A. Watt's group at the University of Glasgow has developed prototype tools for interpreting action notation for compiling it into 'C'. Also, some work has been carried out by David A. Watt in [Watt, 1986] where he defines a method for executing action semantic descriptions. In his approach, actions are defined as higher order functions in Standard ML. Also, in [Moura, 1992], Hermano Moura describes an action notation interpreter (ANI) giving the meaning of program specifications in action semantics. This interpreter was also implemented in SML. The interpretation using ANI gives an output consisting of a triple representing transients, bindings and storage produced by an action.

1.4 Implementation of Action Semantics

Having decided on the type of formal semantic specification, our next step was to find an environment which allowed us to prove the truth of the action laws as discussed in section 1.1. As outlined before, we chose the Coq development

system for these proofs but decided to firstly make the translation to CAML. The first question we should probably ask is why make the translation to CAML i.e. why make an intermediate translation at all? The reasons are rather basic and concern the achievement of a better understanding of action notation before looking at Coq, the implementation of action semantics executable specifications. Also, it is fundamental to understand the structural operational semantics of action notation as we found that, on first reading, that the document was rather difficult to follow initially and not suitable for direct implementation. Through the use of the functional language, CAML, it was possible to trace various action semantics specifications through the structural operational semantics i.e. observe the application of the various semantic functions and obtain interpretations for these specifications. This was not possible with Coq. For example, the use of lists to hold values for bindings and storage greatly aided the tracing procedures. These were subsequently replaced by functions in Coq. Therefore, it was possible to take the formal description to a different level where implementation could take place hence leading to executable semantic descriptions. The initial effort in attempting to completely understand the SOS through the use of CAML was required for correct representation in Coq. Coq is the environment which provides both a specification language and a proof assistant which would enable us to go about proving the various algebraic properties of actions. We should note that, in [Mosses, 1992] (Appendix C), the definition of observational and testing equivalence on actions relates the SOS of action notation to the algebraic properties as laid down in [Mosses, 1992] (Appendix B). We were also particularly interested in the program extraction facilities provided by Coq with respect to further work (which will be discussed in a later section).

1.5 Limitations of Formal Descriptions

As well as the advantages of formal descriptions, there are also disadvantages. The argument against the use of formal specifications is their difficulty - it is said that they are hard to learn, write, read. Significant advances have been made over the past years making formal descriptions more understandable and usable. However, it is still the case that in order to create formal descriptions, a certain level of mathematical ability with substantial effort is required. A second argument that existed against formal descriptions was that they were only used for toy examples and not for full size realistic languages. This argument has since disappeared since the appearance of complete descriptions of languages such as Algol 60, Pascal, PL/1, Ada.

1.6 Previous Work

Much work has been carried out in the area of action semantics. An example of this work is the type inference system for action semantics implemented in object-oriented Pascal by Tony Jakobsen. Christian Lynbech (University of Aarhus) has implemented action semantics in Scheme. However, the system is only capable of evaluating simple expressions from simple specifications. The Actress [Brown et al., 1990] subset of action notation has already been implemented in Sictus-Prolog by Stephan Diehl (University of Saarbrücken). This implementation consists of two parts - one part using semantic equations to translate abstract syntax of a source program into an action and the second part interpreting the action. The "Actress" system is an action semantics directed compiler generator developed by Deryck F. Brown, Hermano Moura and David A. Watt [Brown et al., 1990]. It consists of a number of modules in SML that can be composed to construct either an action notation compiler or a simple compiler generator. An action interpreter (ANI) has been implemented in SML

by Hermano Moura (University of Glasgow) [Moura, 1992]. The initial inspiration to guide the implementation came from the structural operational semantics in [Mosses, 1992] (Appendix C). ANI gives a clear picture of the behaviour of actions. It can be used in conjunction with the "actioneer generator" (developed at the University of Glasgow) to obtain an interpreter for a language from its action semantic description. Refer to [Moura, 1992] for a complete action semantic description of a small language and how this language can be used with ANI and the actioneer generator to generate an interpreter. Looking at Coq, Jill Seaman and Amy Felty have used the Coq development system to prove properties of the operational semantics of a lazy functional language [Seaman & Felty, 1993].

1.7 Outline of Chapters

In this thesis, we focus on the research which was carried out which consisted of three parts. The first part dealt with the implementation of the structural operational semantics (SOS) of action notation in CAML. The second part concerned the implementation of the SOS in the specification language of the Coq development system. Finally, we looked at the proof of some action laws using the proof assistant in Coq. This work is divided into appropriate chapters as described below. In chapter 2, we focus on the different types of formal semantics and in particular, action semantics. Chapter 3 takes a look at the formal description of the kernel of action notation while chapter 4 illustrates the translation of this formal description to CAML. In chapter 5, we look at the Coq development system and observe the conversion into the specification language of the Coq system. We also see, in this chapter, the proofs of the various action laws using the Coq proof assistant. Chapter 6 is responsible for illustrating our conclusions and further work.

CHAPTER 2

Semantics

2.1 Introduction

We previously saw that to specify a programming language totally, the syntax and semantics of that language needed to be described. To describe syntax, all symbols that can appear in programs are enumerated and grouped to indicate how phrases are formed. To describe semantics, the behaviour of the language must be specified. There are several different ways of doing this each with a distinctive type of semantic specification. This chapter deals with formal semantics and introduces three different types of approach to the semantic formalisation of programming languages. It also introduces a fourth type which is the kind of semantic description with which we are particularly interested. Illustrations of the different types of semantics can be found in [Watt, 1991] and [Pagan, 1981].

2.2 Operational/Natural Semantics

The meaning of a construct of a language is specified by the computation it induces when executed on a machine, refer to [Hennessy, 1990]. It is of interest how the effect of a computation is produced.

An operational explanation of the meaning of a construct will tell us how to execute it:

- To execute a sequence of statements separated by ";" , the individual statements are executed one after the other.
- To execute a statement consisting of a variable followed by a ":=" and another variable, determine the value of the second variable and assign it to the first variable.

Looking at Figure 2.1, the execution of the three assignment statements can be recorded starting in a state where x has the value 5, y has the value 7 and z has the value 0 by following the derivation sequence illustrated in Figure 2.1.

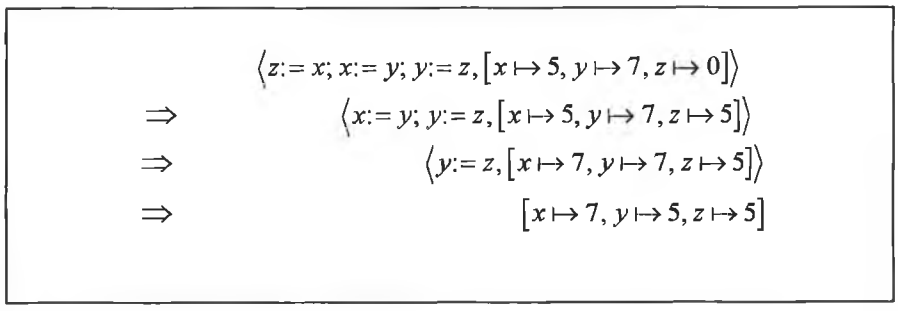

Figure 2.1 Example of operational semantics

The above figure is an explanation giving an abstraction of how the program is executed. Details of registers and machine addresses are ignored. The above figure illustrates the different states before and after execution of the assignment statements where the states are enclosed in square brackets. For example, after execution of the first assignment statement i.e where z is assigned the value of x, the state arrived at consists of x having the value 5, y having the value 7 and z having the value 5.

An alternative operational semantics is natural semantics which hides even more execution details. Using the example of the three assignment statements in Figure 2.1, the execution using the same initial state is described in Figure 2.2.

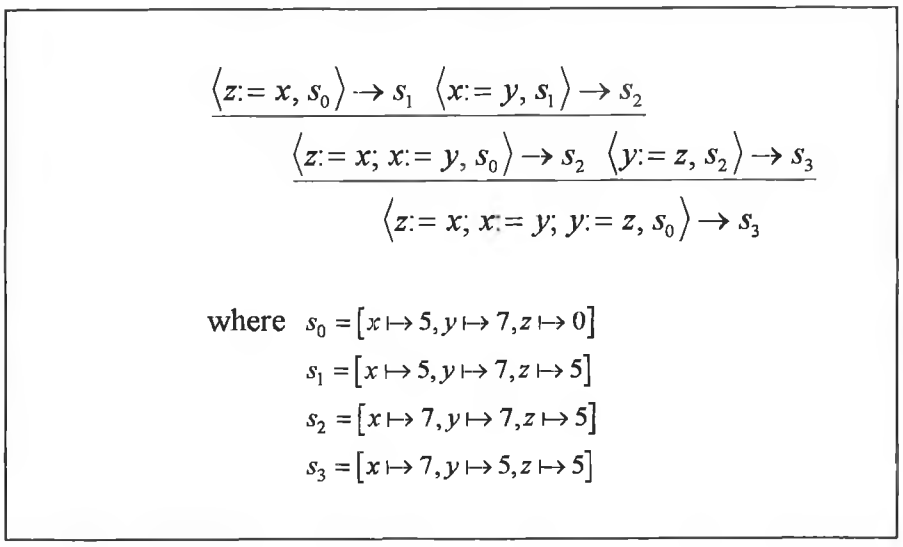

Figure 2.2 Example of natural semantics

Figure 2.2 can be read as follows:

if the execution of z:=x in state s_0 will result in state s_1, the execution of x:=y in state s_1 will result in state s_2, then the execution of y:=z in state s_2 will result in s_3. We see that the four states as illustrated in Figure 2.1 are numbered and we can also observe that it is possible to say that after execution of the initial two assignment statements starting in state s_0, we arrive at state s_2. Also, after execution of the three assignment statements starting in state s_0, state s_3 is reached.

2.3 Axiomatic Semantics

The axiomatic approach is by far the most abstract of all the semantic definition methods considered so far, refer to [Meyer, 1991]. The principles behind it indicate that the semantics of a programming language may be considered to be sufficiently defined if the specifications allow true statements to be proven about the effect of executing a program or program section. The specifications are similar to the axioms and rules of inference of a logical calculus. They prescribe a minimal set of constraints that any implementation of the subject language

must satisfy. The most useful application of axiomatic semantics is in the construction of proofs that programs possess various properties.

There is no standard meta-notation for axiomatic semantics but notational conventions have been adopted by different authors e.g. using logical operators $(\wedge, \vee, \neg, \Rightarrow, \equiv, =)$, quantifiers $(\exists, \forall)$ and logical constants (true, false). The purpose of forming a logical expression in this context is usually to make an assertion about the values of one/more program variables or relationships between values.

Consider a program is partially correct if with respect to a pre- and post-condition, whenever the initial state satisfies the precondition and the program terminates, then the final state is guaranteed to fulfil the postcondition. Now, consider the example in Figure 2.3.

$\{x = n \wedge y = m\}$ $z:=x;$ $x:=y$ $y:=z$ $\{y = n \wedge x = m\}$
where $\{x = n \wedge y = m\}$ is the pre-condition
and $\{y = n \wedge x = m\}$ is the post-condition.

Figure 2.3 Example of axiomatic semantics

The state $[x \mapsto 5, y \mapsto 7, z \mapsto 0]$ satisfies the pre-condition by taking $n = 5$, $m = 7$ and when the partial correctness property has been proven, it can be deduced that if the program terminates, it will do so in a state where $y = 5$, $x = 7$.

2.4 Denotational Semantics

Here the effect of executing programs is concentrated on and this can be modelled by mathematical functions, refer to [Schmidt, 1986]. Denotational

semantics was developed in the early 70's by Strachey and Scott. In denotational semantics, a meaning is assigned not only to a complete program but also to every phrase in a programming language i.e. every command, expression, declaration etc. [Watt, 1991]. The meaning of every phrase is also defined in terms of the meaning of its subphrases so a structure is imposed on the semantics. The meaning of a phrase is referred to as its denotation. Programming language semantics are specified by functions that map phrases to their denotations.

Consider a simple example with denotational semantics used to specify the assignment statement in an imperative language, refer to [Schmidt, 1986]. The semantic algebras are illustrated in Figure 2.4.

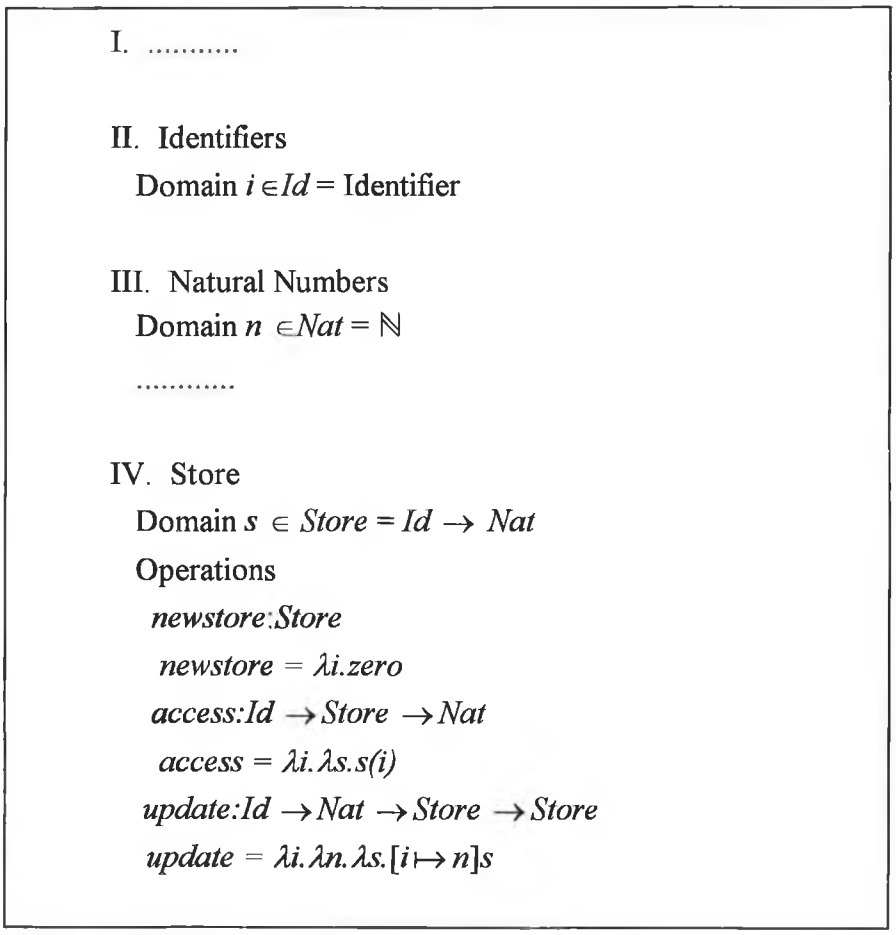

Figure 2.4 Semantic Algebras

Figure 2.4 presents the semantic algebras necessary for the assignment statement in the imperative language. The 'Store' domain denotes a mapping from the language's identifiers to their values. The operations on the store include an operation for accessing the store and also an operation for updating the store. The abstract syntax and the appropriate valuation functions are given in Figures 2.5 and 2.6 respectively.

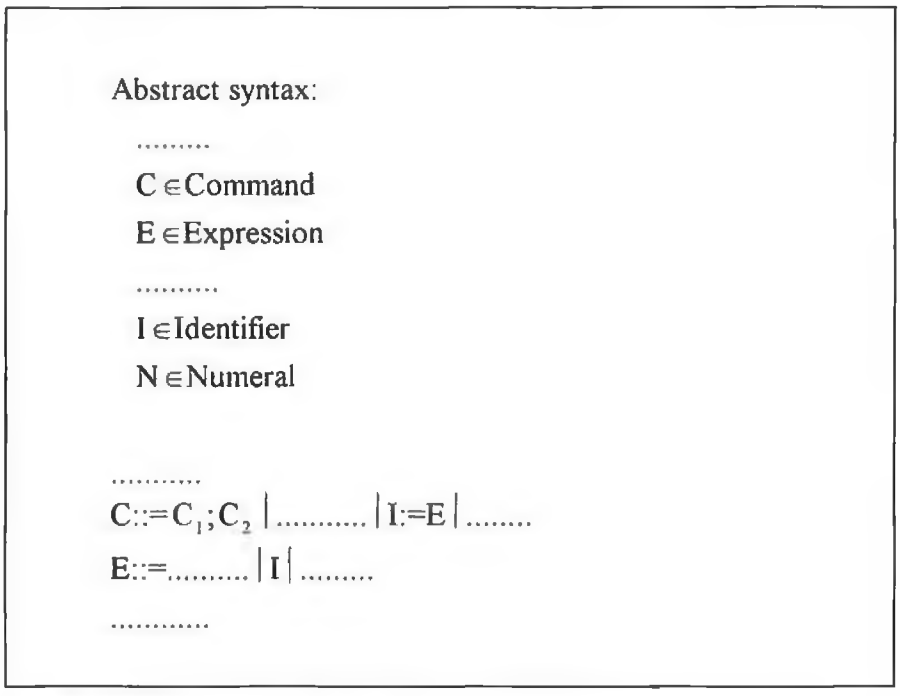

Figure 2.5 Abstract Syntax

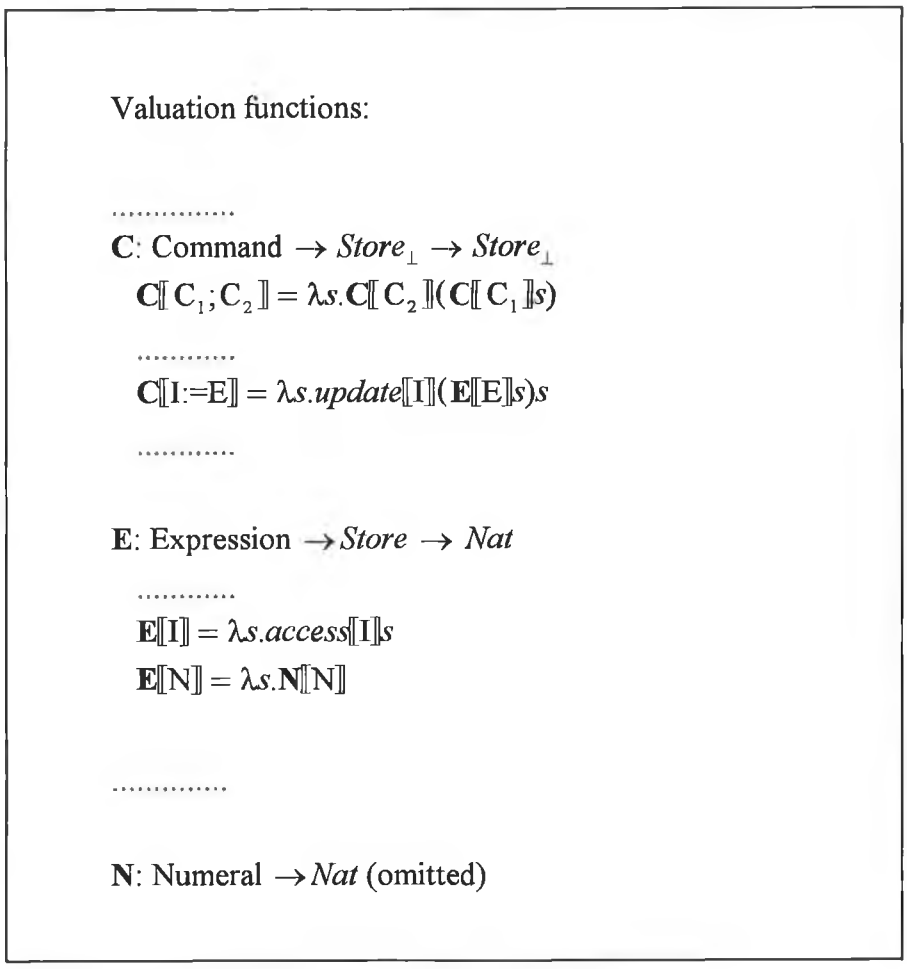

Figure 2.6 Valuation Functions

The purpose of a command is to produce a new store from an old store provided as an argument. However, the command may not terminate its actions on the store i.e. it may loop (not shown in Figure 2.6). Therefore, nontermination is a possible outcome. So, in the valuation function '**C**', the store is lifted. On the other hand, the '**E**' valuation function needs a store argument but does not alter the store in any way. Looking at the clause in the '**C**' function dealing with assignment, the identifier I in the current store is mapped to the evaluation of the expression E, hence producing the new store.

We now consider the assignment statement - Z:=1. The denotation of this command is given in Figure 2.7.

$$
\begin{aligned}
&\mathbf{C}[\![\mathrm{Z}:=1]\!]\mathit{newstore}\\
=\quad &(\lambda s.\mathit{update}[\![\mathrm{Z}]\!](\mathbf{E}[\![1]\!]s)s)\mathit{newstore}\\
=\quad &\mathit{update}[\![\mathrm{Z}]\!](\mathbf{E}[\![1]\!]\mathit{newstore})\mathit{newstore}\\
=\quad &\mathit{update}[\![\mathrm{Z}]\!](\mathbf{N}[\![1]\!])\mathit{newstore}\\
=\quad &\mathit{update}[\![\mathrm{Z}]\!]\ \mathit{one}\ \mathit{newstore}\\
=\quad &[[\![\mathrm{Z}]\!]\mapsto \mathit{one}]\mathit{newstore}
\end{aligned}
$$

Figure 2.7 Denotation of Z:=1

2.5 Action Semantics

Action semantics, [Mosses, 1992][Watt, 1991], was developed in an attempt to make semantic specifications more intelligible. They are written in an English-like notation that can be easily understood. In action semantics, each program is viewed as an action. Action semantics provides a particular notation for expressing actions. The symbols of action notation are suggestive words which makes it possible to get a broad impression of an action semantic description, on first reading. Because other formal semantic specifications specify concepts like control flow, storage and bindings indirectly, specifications tend to be hard to understood and the larger the specification, the more incomprehensible it becomes. This makes the use of action semantic specifications more attractive. Also, action semantic specifications are modular allowing easy modification and they can be reused in specifications of other related languages. Applications include the specification of a variety of imperative and functional languages including the semantics of the programming languages Pascal and ML .

The action combinators e.g. or, and, then, are a notable feature of action notation and obey algebraic laws that can be used for reasoning about semantic equivalence.

2.5.1 Basic Concepts

Usually, programmers are accustomed to thinking of a program in terms of the steps (or actions) that will be performed when the program is executed on a computer. For example, if we consider a command made up of two consecutive commands - *C1* ; *C2*, it is executed by first executing *C1* and then executing *C2*. If we impose a structure on this using the emphatic brackets from denotational semantics, the clause can be formalised as follows:

execute⟦*C1* ; *C2*⟧ = *execute C1* and then *execute C2*

The action *execute C1* is the action of executing the command *C1*. The action combinator 'and then' tells us that the commands should be performed in sequence. The above is a simple example of action notation. It is clear to see that the notation is designed to be convenient and easy to understand. It should be noted that because the action notation has been formally specified, a programming language specification using action notation is entirely formal.

An **action** is an entity that can be performed using data passed to it from other actions [Watt, 1991]. An action can either complete (terminate normally), escape (terminate abnormally), fail or diverge (not terminate). An action's outcome can also depend on the data that is passed to it. An action can use transient data passed to it by other actions and it can supply data if it completes. The performance of an action can use or produce bindings which are identifier-datum associations. An action can also manipulate storage. Actions can also be associated with different facets which will be described in a later section. In action notation, a number of action primitives, action combinators and data

operations are provided. An action primitive represents a single step in the computation. Action combinators combine one or more subactions into a composite actions. It also dictates the flow of control and flow of data between subactions.

2.5.2 Facets

When performed, actions process information gradually. The different types of information give rise to a set of facets where each facet deals with a particular type of information. The main facets are :

- basic: processing independently of information
- functional: processing transient information
- declarative: processing scoped information
- imperative: processing stable information

We deal with the above facets only since they provide an adequate basis for the specification of programming languages. However, there are many other facets e.g. reflective, communicative.

There are three kinds of semantic entity used in action semantics

- actions
- yielders
- data

The main kind, of course, is actions while yielders and data are considered as subsidiary semantic entities. The notation in action semantics for specifying actions and the subsidiary entities is referred to as action notation. In action notation, there are a number of actions, yielders and data associated with each facet. The standard action notation can be reduced to a kernel as in [Mosses,

1992] (the reduction is purely for technical reasons i.e. it reduces the number of constructs to be considered in the formal specification of action notation).

2.5.3 Semantic Entities

Actions are essentially computational entities. Performance of an action represents information processing behaviour and reflects the stepwise nature of computation. Actions represent the semantics of programs i.e. they represent possible program behaviour. An action can be nondeterministic with different possible performances for the same initial information i.e. transient information, scoped information and stable information. Transient information is used by an action immediately. Scoped information can usually be referred to throughout an entire action although it may be hidden temporarily. Stable information can be changed but not hidden in the action, it persists until destroyed. When an action is performed, transient information is given only on complete or escape. Scoped information is produced only on completion. Changes to stable information made during the performance of an action are unaffected by subsequent divergence, failure or escape.

Yielders are unevaluated items of data whose value depends on the current information i.e. the currently available data, bindings and storage. Yielders can be evaluated during action performance. Compound yielders can be formed by the application of data operations to yielders. An example is when the sum of two yielders is formed.

Data items are mathematical entities representing pieces of information. Data includes familiar mathematical entities such as truth values, numbers, maps, lists etc. Data can also include cells and tokens.

The action notation showing the actions and yielders appropriate to each facet with an indication of whether that action/yielder is part of the kernel of action notation is illustrated in Tables 2.1 - 2.8

Action	Kernel	Informal Meaning
complete	**Y**	Terminates normally.
escape	**Y**	Terminates abnormally.
fail	**Y**	Fails immediately.
diverge	**Y**	Nontermination.
unfold	**Y**	Dummy action used with **unfolding.**
unfolding A	**Y**	Performs A iteratively. The action **unfold** is replaced by A whenever it is encountered.
A_1 **or** A_2	**Y**	Performs either A_1 or A_2. If the chosen subaction i.e. A_1, A_2 fails, the other subaction is chosen.
A_1 **and** A_2	**Y**	Performs A_1 and A_2 collaterally. Bindings given by A_1, A_2 are merged.
A_1 **and then** A_2	**Y**	A_1 and A_2 are performed sequentially. Otherwise, it behaves like 'A_1 **and** A_2'.
A_1 **trap** A_2	**Y**	A_1 is performed. If A_1 escapes, perform A_2 .

Table 2.1 Basic Facet - actions

Yielder	Kernel	Informal Meaning
the d **yielded by** y	**Y**	If d is a sort of data and y is a yielder, when y yields an individual, it yields that individual provided it's in the sort, otherwise it yields nothing
data-operation $(y_1 \ldots\ldots y_n)$		After evaluation of yielders $y_1 \ldots\ldots y_n$, the data operation is applied to the yielded data.

Table 2.2 Basic Facet - yielders

Action	Kernel	Informal Meaning
give *y*	**Y**	Gives the data yielded by *y*
escape with *y*		Escapes with the data yielded by *y*
regive		Gives any given data
choose *y*	**Y**	Gives one datum of the sort yielded by *y*
check *y*		Where *y* is a truth-value yielder, this represents a guard checking that the truth value yielded by *y* is true.
A_1 **then** A_2	**Y**	A_1 and A_2 are performed sequentially. Transients from A_1 are passed onto A_2

Table 2.3 Functional Facet - actions

Yielder	Kernel	Informal Meaning
given *d*		A data yielder which yields the transient data given to its evaluation provided the data is of sort *d*
given *d#p*		Where *d* is a sort of datum and *p* is a positive integer, it yields the *p*th component of the transient data given to its evaluation provided the datum is of sort *d*
it		A datum yielder that yields a single datum given to its evaluation
them	**Y**	A data yielder which yields all the data given to its evaluation as transients

Table 2.4 Functional Facet - yielders

Action	Kernel	Informal Meaning
bind T **to** Y	**Y**	Where T is a token and Y is a yielder of bindable data, it produces the bindings of the token T to the bindable data
unbind T	**Y**	Where T is a token, it produces the bindings of the token T to the datum 'unknown'
rebind		Produces all the received bindings
produce Y	**Y**	Produces the bindings yielded by Y
furthermore A	**Y**	Represents propagating the received bindings but letting bindings produced by A take precedence when there is conflict
A_1 **moreover** A_2	**Y**	Like 'A_1 **and** A_2' but gives priority to bindings produced by A_2
A_1 **hence** A_2	**Y**	A_1 and A_2 are performed sequentially. Bindings produced by A_1 are passed to A_2.
A_1 **before** A_2	**Y**	A_1 and A_2 are performed sequentially. A_2 receives initial bindings overlaid by bindings produced by A_1.

Table 2.5 Declarative Facet - actions

Yielder	Kernel	Informal Meaning
current bindings	**Y**	Yields the collection of bindings received by the evaluation
the d **bound to** T		Yields the data of sort d to which T is bound by the received bindings
Y_1 **receiving** Y_2	**Y**	Where Y_2 is a yielder of bindings maps, it represents evaluation of Y_1 using bindings yielded by Y_2

Table 2.6 Declarative Facet - yielders

Action	Kernel	Informal Meaning
store Y_1 **in** Y_2	**Y**	Stores the storable yielded by Y_1 in the cell yielded by Y_2
unstore Y	**Y**	Represents destroying a piece of stable information where Y is a yielder of a cell.
reserve Y	**Y**	Extends stable information with an extra uninitialised piece where Y is a yielder of a cell.
unreserve Y	**Y**	Represents the destruction of stable information where Y is a yielder of a cell.

Table 2.7 Imperative Facet - actions

Yielder	Kernel	Informal Meaning
current storage	Y	Yields the current state of storage
the d **stored in** Y		Yields the data of sort d stored in the cell yielded by Y according to current storage

Table 2.8 Imperative Facet - yielders

2.5.4 Action Semantic Descriptions

A semantic description comprises three main parts

- abstract syntax
- semantic functions/equations
- semantic entities

2.5.4.1 Abstract Syntax

Generally, formal context-free grammars augmented with some form of regular expressions are used to specify concrete syntax. A formal grammar is made up of a set of production rules, which are made up of terminal and non-terminal symbols. The terminal symbols can be characters or strings. We can adapt

these formal grammars to represent abstract syntax as in [Mosses, 1992]. If we take an example, we can illustrate this formal notation. Consider a simple language which includes an assignment statement. The abstract syntax is shown in Figure 2.8.

Abstract Syntax:

grammar:

(1)	Statement =	⟦Identifier ":=" Expression⟧
	|	⟦Statement ";" Statement⟧.
(2)	Identifier =	⟦letter (letter | digit)* ⟧.
(3)	Expression =	Numeral .
(4)	Numeral =	⟦digit + ⟧.

Figure 2.8 Abstract Syntax

The module in Figure 2.8 is made up of a set of numbered equations. Terminal symbols are written as strings of characters in quotes. Nonterminal symbols are not enclosed in quotes e.g. Expression. There are usually a number of alternatives for each nonterminal which are separated by ' | '. Some equations involve a type of regular expression e.g. an optional repeatable part R* and an obligatory repeatable part R^+. One other point to note is that we can say that the nonterminal symbol 'Statement' is recursive both to the left and right.

2.5.4.2 Semantic Functions

In action semantics, semantic functions are specified with semantic equations. Each equation defines the semantics of a particular type of phrase in terms of the semantics of its components. The equation may use constants and

operations for constructing semantic entities. A set of semantic equations can be considered as an inductive definition mapping syntactic entities to semantic entities. So, it is basically a translation from programming language syntax to notation for semantic entities. Programmers may regard semantic equations as a definition of mutually recursive functions by cases. Let us consider the semantic functions/equations of our simple language in Figure 2.9. We can see that the semantic function takes a single, syntactic argument and gives a semantic entity. The placeholder in each semantic function indicates where the argument is placed. In Figure 2.9, the functionality of each semantic function is given e.g. the semantic function 'evaluate' indicates that when performed, it may give a value. The right hand sides of the semantic equations are expressed in the standard notation for actions and data given by action semantics. It should be noted that the notation is completely formal despite the fact that it possible to read it informally. Each semantic equation defines how a particular semantic function is applied to any abstract syntax tree with a root node whose form is one of the syntactic constructs. It does this by applying semantic functions to the branches of the node. For example, if we consider equation 1 in the semantic equations, it defines the application of the semantic function 'execute' to nodes with three branches, where the first branch is the identifier *I*, the second branch is ":=" and the third is an expression *E*.

introduces: execute _ , evaluate _ , the value of _ .

- execute _ :: Statement → action
  [completing | storing]

(1) execute⟦ *I*:Identifier ":=" *E*:Expression ⟧=
(give the cell bound to *I* and evaluate *E*)
then store the given number#2 in the given cell#1.

(2) execute⟦ S_1 :Statement ";" S_2 :Statement⟧=
execute S_1 and then execute S_2 .

- evaluate _ :: Expression → action
  [giving a value]

(3) evaluate⟦*N*:Numeral⟧=
give the value of *N*.

- the value of _ :: Numeral → value

(4) the value of⟦*d*:digit+⟧ =
d.

Figure 2.9 Semantic Functions/Equations

2.5.4.3 Semantic Entities

To complete the semantic description, the notation used in the semantic equations for specifying semantic entities has to be specified. The standard action notation already includes all the notation required for specifying actions. It includes notation for action primitives and combinators. Each action primitive is associated with one facet i.e. one kind of information flow whereas each combinator deals with different types of information flow. The notation

possesses enough action primitives and combinators to express most common patterns of information processing in a straightforward manner. It also has a basic notation for data e.g. truth-values, lists etc. The semantic entities required for our simple example are illustrated in Figure 2.10. We must bear in mind, however, that our example is rather unlikely since it assumes that identifiers are already bound to cells in storage. Obviously, we cannot assign a value to an identifier that does not exist.

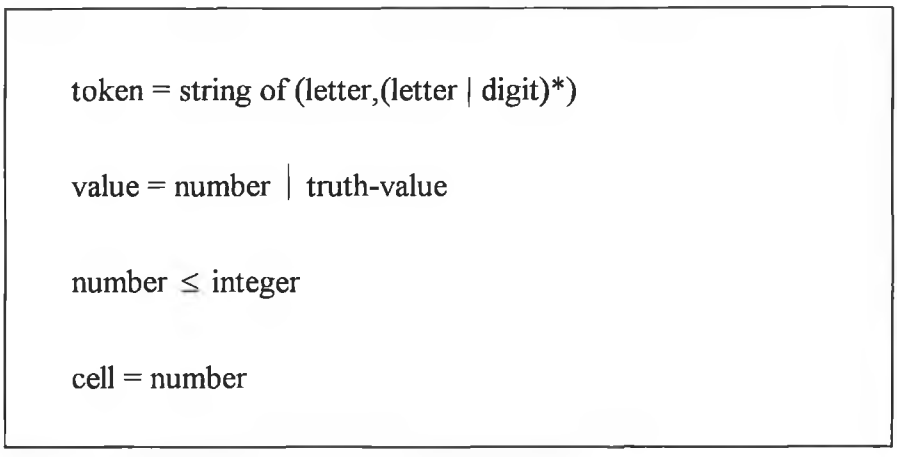

Figure 2.10 Semantic Entities

We must bear in mind, however, that our example is rather unlikely since it assumes that identifiers are already bound to cells in storage. Obviously, we cannot assign a value to an identifier that does not exist.

2.5.5 Action Laws

The primitive actions and action combinators satisfy a series of algebraic laws. It may then be possible to prove semantic equivalences exist between programs, commands, expressions etc. using these laws. This is done by showing their denotations i.e. their resulting actions are equivalent. A sample of these laws is given in Figure 2.11

(1) **check true = complete**

(2) **check false = fail**

(3) **complete and then** A = A **and then complete** = A

(4) **escape and then** A = **escape**

(5) **escape trap** A = A **trap escape** = A

(6) **complete trap** A = **complete**

(7) **fail trap** A = **fail**

(8) **fail or** A = A **or fail** = A

(9) A_1 **or** A_2 = A_2 **or** A_1

Figure 2.11 Action Laws

2.6 Summary

In this chapter, we looked at the different types of formal semantics. We related the reasons for choosing action semantics over the other types of semantics e.g. its English-like notation, its comprehensibility. As we were particularly interested in action semantics, we took a closer look at the standard action notation. We then showed the component parts of an action semantic specification. Finally, a subset of the algebraic properties of the primitive actions and combinators was observed.

CHAPTER 3

Formal Description of Action Notation

3.1 Introduction

This chapter looks at the translation from standard action notation to the kernel of action notation. It then observes the formal description of the kernel and finally illustrates, using an example, that the formal description is indeed a true representation of the kernel.

3.2 Translation to the Kernel

We know that it is possible to reduce the standard action notation to a kernel. Examples of kernel actions and yielders have been given in Tables 2.1 - 2.8 in section 2.5.3. The algebraic properties of action notation given in [Mosses, 1992] (Appendix B) are sufficient to make that translation to the kernel. If we consider our previous example in section 2.5.4, it is possible using the algebraic properties outlined in [Mosses, 1992] (Appendix B) to translate the action notation of the assignment statement to the kernel. Note, however, that we assume the identifier I is already bound to a cell in storage. To illustrate this translation, refer to Figure 3.1. We should note that we have made appropriate substitutions for identifier I and Expression E.

(i) execute⟦"*I*" :Identifier ":=" "2":Expression⟧ =
(give the cell bound to "*I*" and evaluate "2")
then store the given number#2 in the given cell#1.

(ii) execute⟦"*I*":Identifier ":=" "2":Expression⟧ =
(give the cell bound to "*I*" and give 2)
then store the given number#2 in the given cell#1.
(by equations (3),(4))

(iii) execute⟦"*I*":Identifier ":=" "2":Expression⟧ =
(give the cell yielded by
current bindings at the token yielded by "*I*"
and give 2)
then store the number yielded by
component#2 of them in the cell yielded by
component#1 of them
(by algebraic properties laid down in [Mosses, Appendix B]

Figure 3.1 Translation to the kernel - assignment statement

3.3 Formal Description of the Kernel

If we consider that the standard action notation and the kernel are languages in their own right, it must be possible to provide a semantic description of these languages too. This has been illustrated by Peter Mosses in [Mosses, 1992] (Appendix C) which gives the complete formal description of the kernel using structural operational semantics. As we know, it is possible to translate the standard action notation to the kernel and therefore, the formal description written down is sufficient to describe standard action notation. The structural operational semantics (SOS) in [Mosses, 1992] (Appendix C) is written using an algebraic specification framework. The idea behind the specification is to use

a transition function to map individual configurations to arbitrary sorts[1] of configurations hence coping with nondeterminism of actions. Also, the result can be a single configuration when the transition from a particular configuration is deterministic. As well as this, the configuration may be blocked and this may be represented by the vacuous sort "nothing". Note that the kernel is syntactically of moderate size which is illustrated by its grammar.

3.3.1 Abstract Syntax

The grammar specifies the abstract syntax of the kernel while the algebraic laws in [Mosses, 1992] (Appendix B) give the remaining notation for actions and yielders in terms of the kernel notation. The abstract syntax for data has been left open to allow the user of action notation to add extra notation. A section of the abstract syntax is shown in Figure 3.2.

grammar:

- Action = Simple-Action | .....| ⟦Action Action-Infix Action⟧ .
- Simple-Action = Constant-Action | ⟦Simple-Prefix Yielder⟧ |
  ⟦To-Prefix Yielder "to" Yielder⟧ .....
- Simple-Prefix = "give" | .....
- To-Prefix = "bind" | .....
- Action-Infix = "or" | "and" | "and then" | "then" | .....

Figure 3.2 Abstract Syntax of the kernel

3.3.2 Semantic Entities

The semantic entities in the operational semantics are made up of some syntactic components which indicate what remains to be performed. This is quite distinct

[1] A sort classifies the individual values of a universe according to some common attributes

from action semantic descriptions. The specification of the semantic entities use standard data notation for maps etc. They also use data sorts like data and bindings. Note that data and bindings have been specified in [Mosses, 1992] (Appendix B). There are two main kinds of semantic entity - actings and states, described below.

3.3.2.1 Actings

An Acting is a generalisation of an action. The acting supports the representation of the state of an action i.e. an acting can either be terminated or intermediate. An acting which is intermediate contains information about the remaining actions to be executed. An acting which is terminated holds details about the type of termination i.e. escaped, failed or completed. An acting can exist as an action with associated data, bindings or both. The action combinators (or action infixes as they are called here) are classified into three categories - sequencing, interleaving and normal. This is purely for convenience. The specification for actings are given in Figure 3.3.

grammar:

- Acting = Terminated | Intermediate.
- Terminated = Completed | Escaped | Failed.
- Completed = ⟨ "completed" data bindings ⟩.

.....

- Intermediate = Simple-Action | ..... | ⟦Acting Action-Infix Acting⟧ ⟨ Action data ⟩ | ⟨ Action bindings ⟩ | ⟨ Action data bindings ⟩ | .....

Figure 3.3 Actings

3.3.2.2 States

A state represents a point in performance of an action. It consists of an acting and local information. Local information corresponds to the current stable information (storage). The transient data and bindings are incorporated in the acting component of the state. Therefore, a state can also be represented as an action, transient data and bindings and local information. The type of acting involved in the state dictates the type of state. The definitions of the semantic entity "state" and the subsidiary entities - local info and info - are given in Figure 3.4.

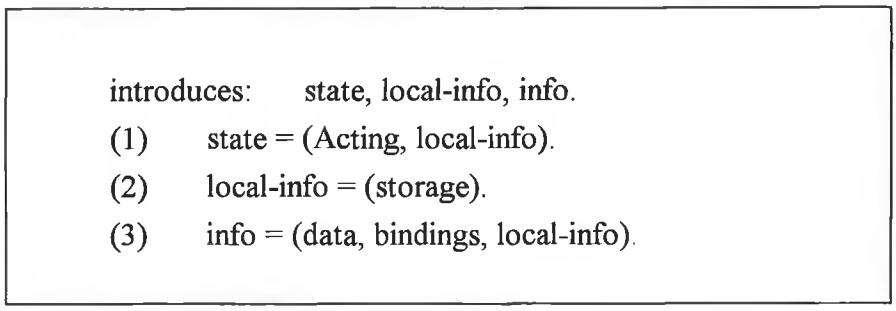

Figure 3.4 States

3.3.3 Semantic Functions

The semantic functions are categorised by actions, yielders and data.

3.3.3.1 Actions

The main semantic functions are called "run" and "stepped". The function "run" is responsible for taking an intermediate state and advancing it to a terminated state using successive applications of the function "stepped" as seen later. The semantic function is give by:

- run _ :: state → (Terminated, local-info)

where "Terminated" relates to the type of acting involved in the state arrived at after "run" is applied.

The semantic function "stepped", when applied, gives the sort of states obtained from performing the first transition from an intermediate state. Both the above functions are not defined on terminated states. The function type is illustrated as:

- stepped _ :: state → state

The definition of the function "run" is given in Figure 3.5.

(1) stepped $(A,I) \geq (A'$:Intermediate, I':local-info);
run $(A',I') \geq (A''$:Terminated, I'' :local-info) $\Rightarrow$
run (A:Acting, I:local-info) $\geq (A'', I'')$.

(2) stepped $(A,I) \geq (A'$:Terminated, I':local-info) $\Rightarrow$
run (A:Acting, I:local-info) $\geq (A', I')$.

Figure 3.5 Definition of semantic function "run"

In figure 3.5, we can explain (1) by saying that if we advance the state (A, I) by one transition to a state (A',I') and apply "run" to advance this state to the terminated state (A'', I''), it is the same as applying "run" to the state (A, I) to give the terminated state (A'', I'').
(2) says that if it only takes one transition using "stepped" to reach a terminated state, then it is the same as applying "run" to the original state to reach that terminated state.

Take an example of a state containing a composite acting - A_1 "then" A_2 "then" A_3 where A_1, A_2, A_3 are primitive. If we apply "run" to this state, it will

essentially involve applying "stepped" three times to advance this state to a terminated state. Therefore, the number of primitive actions denotes the number of times "stepped" will be applied to advance to a terminated state.

The function "simplified" , see [Mosses, 1992] (Appendix C, C.3.3.2.2), is only applied to an intermediate compound acting in the form - $[\![A_1$ Action-Infix $A_2]\!]$ where the intermediate component of $[\![A_1$ Action-Infix $A_2]\!]$ is the acting part of the result of applying "stepped". The function is responsible for simplifying a composite acting. For example, take a composite acting - (A_1:Completed "and" A_2:Completed). An application of "simplified" would convert this composite acting to a simple acting - (A:Completed) where the tupled data from A_1, A_2 and the disjoint union of bindings from A_1 and A_2 are associated with the acting A. The function's type is given by:

- simplified _ :: Acting → Acting

The functions "given" and "received", see [Mosses, 1992] (Appendix C, C.3.3.2.4, C.3.3.2.5), are responsible for the flow of data and bindings into actions. The behaviour of these functions is governed by the various action combinators. If we take for example the application of "given" to (A,d) where A is an acting and d is data, the function would freeze the initial transient data given to A. Similarly for the application of "received" to (A,b) where A is an acting and b is bindings, the function would freeze the initial bindings given to A. The functions' types are given as:

- given _ :: (Acting, data)→ Acting
- received _ :: (Acting, bindings) → Acting

The application of the "unfolded" function , see [Mosses, 1992] (Appendix C, C.3.3.2.3), to (A, [\["unfolding" A]\]) is used to replace occurrences of "unfold" in A with [\["unfolding" A]\] before performing A. So, we can say that performing

⟦"unfolding" "unfold"⟧ takes infinitely many steps. Note that the prefix "unfolding" would only be associated with the definition of loops in programming languages. The function "unfolded" is given by:

- unfolded _ :: (Action, Action) → Action

3.3.3.2 Yielders

The semantic function used with yielders is called "evaluated". It is given by

- evaluated _ :: (Yielder, Info) → data

As we have already seen, a yielder is an unevaluated item of data and therefore, "evaluated" is responsible for converting that yielder to data. The evaluation may depend on the current information available i.e. data, bindings and storage.

3.3.3.3 Data

The semantic function "entity" is merely an identity function. It is given as

- entity _ :: Data → data

So, for any data term *d* with abstract syntax *D*, (entity *D*) = *d*.

3.4 Illustration using the SOS

To demonstrate how the operational semantics is a true formal representation of action notation, we take out previous example of an assignment statement through the necessary steps. As the operational semantics represents only the kernel of action notation, we start with the translated version of the declaration as shown in Figure 3.6.

Figure 3.6 Kernel action notation of an assignment statement

```
execute ⟦"I":Identifier ":=" "2":Expression⟧ =
  (give the cell yielded by
          current bindings at the token yielded by
                "I" and give 2)
  then store the number yielded by
          component#2 of them in the cell yielded by
                component#1 of them.
```

Note the substitution of "I" for the identifier *I* and the numeral "2" for the expression *E*. The transformation will be illustrated in detail in Appendix A. A introduction to these steps will be given below.

1. The initial intermediate compound state looks as follows:
   (((action 1, *b*) "then" (action 2, *b*)), *l*)
   where action 1 is
   (give the cell yielded by
   current bindings at the token yielded by "I"
   and give 2)
   and action 2 is
   (store the number yielded by component #2 of them
   in the cell yielded by component #1 of them)
   and *b* consists of the binding of "I" to cell 1 (see Appendix A)
   and *l* is empty.

2. We then apply the semantic function "run" to the above state which tells us to apply "stepped" to the same state, see [Mosses, 1992] (Appendix C, C.3.3) and check if the resulting state is terminated or intermediate. As the state in step 1 is compound, we apply "stepped" for compound states, see [Mosses, 1992] (Appendix C, 3.3.2.1. (5)).

This instance of "stepped" tells us to firstly apply "stepped" to ((action 1, *b*),*l*).

3. As the state ((action 1, *b*),*l*) is also a compound state, we must apply "stepped" for compound states, as before. If we break action 1 into its constituent actions and label these actions - action 1a and action 1b, they would look as follows:
action 1a - (give the cell yielded by
current bindings at the token yielded by "I")
and action 1b - (give 2).

The function "stepped" tells us to then apply "stepped" to action 1a. As we can see, compound actions are breaking down into their constituent primitive actions. When we apply "stepped" to a state consisting of a primitive action, the result is a terminated state. For the above example, three applications of "run" is required to reach a terminated state for our initial compound state i.e. one application of "run" for each primitive action. Note that after two applications of "run", we still arrive at an intermediate state. All the steps involved in advancing our initial compound state to a terminated state are detailed in Appendix A.

3.5 Action Laws with the SOS

We have introduced in chapter 2 the notion of action laws. Here, we will look at one of these algebraic properties in conjunction with the SOS. We should note that the primitive actions and action combinators were designed to satisfy these algebraic properties. We can use these laws to apply equational reasoning to actions and we already know, this can lead to proofs of semantic equivalence between equivalent actions. If we take the basic action law,

unfolding fail = fail

we can prove that both actions are equivalent by taking them through the necessary steps as dictated by the SOS.

Firstly, we must convert the above action law to the representation required by the SOS, therefore we must convert the action "**unfolding fail**" to an acting. The appropriate acting is the acting - "("unfolding" "fail" *d b,l*)" where *d* refers to transient data, *b* refers to bindings and *l* is local info, all collectively known as info. We then translate "fail" to the acting - "("fail" *d b,l*)". So, the theorem to be proved looks as follows:

("unfolding" "fail" *d b ,l*) = ("fail" *d b,l*)

If we apply the function "run" to the left hand side, we see that we then apply "stepped" as dictated by "run" which gives

(given (received (unfolded ("fail", ⟦"unfolding" "fail"⟧ ,*b*),*d*),*l*))

Applying the function "unfolded", firstly, gives the following result:

"fail"

The overall result after applying "received" and "given" is as follows:

("fail" *d b,l*)

We can now see that after applying one instance of "stepped" to the left hand side, the two sides of the equation are identical. If we look again at "run", it tells us that if we have applied "stepped" once and this has resulted in an intermediate state, then reapply "run". Since this is the appropriate case, it means that we must reapply "run" to

("fail" *d b,l*)

We know that we must also apply "run" to the right hand side and therefore, this proves the action law "unfolding""fail" = "fail" by the SOS.

3.6 Summary

In this chapter, we have taken a look at the structural operational semantics of the kernel of action notation. We have also observed the translation from standard action notation to the kernel. A verification of the correctness of the SOS was given by taking an example i.e. a constant declaration and bringing it through the operational semantics to check its correctness. Finally, we looked again at action laws in conjunction with the SOS and illustrated a proof of an action law using the SOS.

CHAPTER 4

Implementation of Action Notation in CAML

4.1 Introduction

We have seen in the previous chapter that it is possible to formally represent action notation in the SOS [Mosses, 1992] (Appendix C). We then looked at formalising action notation computationally. It was sensible to consider using a functional language so the semantic functions could be represented appropriately. Also, it is a fact that in a functional language, functions and values are treated as mathematical items obeying well-established mathematical rules and are therefore, suited to formal reasoning [Myers et al., 1993]. It was decided that the functional language CAML would be ideal for these purposes. The following chapter gives a description of functional languages with CAML and the conversion process with its associated difficulties.

4.2 Functional Languages and CAML

The following sections discuss functional languages in general and then CAML as a functional language.

4.2.1 Functional Languages

Programming languages are said to be functional if their basic component is the notion of the "function" and their essential control structure is the "function application". The Lisp language can be referred to as a functional language as it possesses these two properties. However, we want the programming notion of function to be as close as possible to the mathematical notion of function.

In mathematics, we would present the successor function as:

successor: $\mathbb{N} \rightarrow \mathbb{N}$

$n \mapsto n + 1$

We also note the importance of:

- The notion of a "type". A mathematical function always has a domain and co-domain. They correspond to the notion of "type".
- Lexical binding. When we wrote "successor", we assume that the addition function "+" has been previously defined.
- The notion of "function abstraction". The name "successor" represents the functional value mapping any natural number n to n+1.

ML dialects adhere to the above notions. However, they do allow non-functional styles and so, are not purely functional. ML dialects, see [Myers et al., 1993] [Mauny, 1991], are based on a sugared version of lambda calculus. The evaluation regime is call-by-value i.e. the argument is evaluated before it is passed to the function and they use Milner's[2] type system. Since 1984, the CAML language has been under design between INRIA and LIENS[3]. The first release appeared in 1987 and the main implementors were Svárez, Weis and Mauny.

4.2.2 CAML

CAML is a powerful programming language that is easy to learn, easy to use and yet amazingly powerful, see [Mauny, 1991]. The features of CAML are as follows:

- Types: It is statically type checked but there is no need to give type information in programs (as in Ada, Pascal and C).

[2]Milner proposed the language ML in 1978.
[3]Laboratoire d'Informatique de l'Ecole Normale Supérieure.

- Functions: There are no restrictions in the definition and usage of functions. They can be passed as arguments or returned as values.
- Automatic memory management and incremental garbage collection: CAML features automatic memory management i.e. allocation and deallocation of data structures is kept implicit and is handled by the run-time system. This means that programs are much safer and spurious memory corruptions can never occur. The memory manager works in parallel with the application so there is no noticeable stop of the CAML program when the garbage collector is running.
- Imperative: Full implementation capabilities including updatable arrays, imperative variables etc.
- Modules: Batch compilation or separate compilation via a module system. The CAML Light compiler generates object programs that are small and portable.
- Interactivity: Interactive top-level 'read-eval-print' loop which is good for debugging and learning i.e. there is no need for files or printouts to get results.
- Error recovery: There is a general exception mechanism to handle or recover from errors or exceptional situations.
- Polymorphism: CAML features polymorphic typing. Functions and procedures can be applied to any kind of data regardless of type.
- Evaluation regime: CAML is a strict language but first order functions allow the manipulation of delayed expressions.
- Powerful libraries: There are lots of libraries available including portable graphics and various interfaces with well-known technology.
- Applications: CAML is used for complex systems e.g. theorem provers and compilers e.g. CAML Light compiler, Coq theorem prover.

Also, just to note that CAML allows the user to define his/her own data structures and also allows the manipulation of these data structures with the security provided by strict type verification.

4.3 CAML Representation of Action Notation

We firstly note again that only syntactic entities, semantic entities and semantic functions associated with the basic, functional, declarative and imperative facets were represented in CAML. We also aimed to preserve the structure as in the operational semantics [Mosses, 1992] (Appendix C). Refer to the Appendix for the mappings from the SOS into CAML.

The structure defined was as follows:

- Most syntactic and semantic entities were defined as new types.
- The following semantic functions were defined

  run, stepped, given, received, unfolded, evaluated, entity.
- Auxiliary functions were defined e.g.

  overlay (to overlay bindings), access (to access bindings / access storage), alter (to alter storage) for use in the semantic equations.
- The various collections of action combinators were defined i.e. Normal, Sequencing and Interleaving.

The structure of the syntax, semantic entities and semantic functions will be observed below.

4.3.1 Syntax

Some examples of the CAML definition of the syntactic entities with the SOS explanation, see [Mosses, 1992] (Appendix C, section C.1.2), are given in Figures 4.1, 4.2.

```
type Yielder = Data_Con of Data
                    (* Data-Constant *)
             |      Unary_op of (un_op * Yielder)
                    (* Data-Unary *)
             |      Binary_op of (bin_op * Yielder * Yielder)
                    (* Data-Binary *)
             |      Selected of (Yielder * Yielder * Yielder)
                    (*"if" Yielder "then" Yielder "else" Yielder *)
             |      Yield_by of (Data * Yielder)
                    (* "the" Data "yielded by" Yielder)
             |      Received of (Yielder * Yielder)
                    (* Yielder "receiving" Yielder *)
             |      them
                    (* "them" *)
             |      current_bindings
                    (* "current bindings" *)
             |      current_storage
                    (* "current storage" *)
             |      At of (Yielder * Yielder)
             |      Comp of (int * Yielder)
             |      Next_cell ;;
```

Figure 4.1 CAML definition of Yielders

Figure 4.1 gives the definition of yielders. We should note that the types "un_op" and "bin_op" have been previously defined. If we compare this definition to the SOS, it can be seen that three additional yielders have been

defined. This will be discussed in a section 4.4.2. The yielder is referred to as a syntactic entity. However, Data as defined in Figure 4.2 is referred to as a syntactic and semantic entity.

```
and   Data = incell of cell          (* cell *)
      |      inval of val            (* value *)
      |      intoken of token        (* token / identifier *)
      |      inbool of bool          (* boolean *)
      |      inBind of Bindings      (* bindings *)
      |      inStore of Store        (* storage *)
      |      Err                     (* "nothing" (SOS) *)
      |      Data_list of Data list  (* list of data *)
      |      bbool                   (* domain of boolean *)
      |      vval                    (* domain of value *)
      |      ccell                   (* domain of cell *)
      |      ttoken ;;               (* domain of token *)
```

Figure 4.2 CAML definition of Data

Note in Figure 4.2, the existence of "Err" in the definition. This corresponds to the "nothing" data type in the structural operational semantics. Note also that we have specified a number of data components as a list. This corresponds to the tupling of data in the operational semantics e.g. in the "simplified" function.

4.3.2 Semantic Entities

The definition of the semantic entity "Acting" is given in Figure 4.3. Also, the subsidiary entities are given in Figure 4.4.

```
type Acting =
                Stopped of Terminated
                (* Terminated *)
        |       Inter of Intermediate
                (* Intermediate *)
and Terminated =
                Compl of (Data * Bindings)
                (* ⟨"completed" data bindings ⟩ *)
        |       Escape of Data
                (* ⟨"escaped" data⟩ *)
        |       failed
                (* "failed" *)
and Intermediate =
                APA of (Action_Prefix * Acting)
                (* [[ Action-Prefix Acting ]] *)
        |       AIA of (Acting * Action_Infix * Acting)
                (* [[ Acting Action-Infix Acting ]] *)
        |       Adb of (Action * opdb)
                (* ⟨ Action data bindings ⟩ *)
        |       AbA of (Acting * Action _Infix * Acting *
                Bindings)
                (* [[ Acting ("before" | "then before") Acting bindings]] *)
and opdb =
                no_db
        |       is_d of Data
        |       is_b of Bindings
        |       is _db of (Data * Bindings) ;;
```

Figure 4.3 Definition of Acting

In Figure 4.3, we can see that some of the hierarchy for the definition of "Acting" has been eliminated. Also, we have reduced the size of the definition

of "Intermediate" by introducing the new data type "opdb". This allows us to define once only the following instances of "Intermediate":

- ⟨ Action data ⟩
- ⟨ Action bindings ⟩
- ⟨ Action data bindings ⟩ (SOS [1, Appendix C])

```
type cell = = int
and val = = int
and token = = string ;;
type .....
and Bindings = = (token * Data) list
and Store = = (cell * Data) list ;;
```

Figure 4.4 Definition of subsidiary entities

In Figure 4.4, we represent cells as integers and values are restricted to integer values. Also, tokens are strings of characters. Bindings are represented as a list of token, Data tuples and Storage is represented as a list of cell, Data tuples. Lists were used to facilitate debugging.

4.3.3 Semantic Functions

The semantic functions that have been defined in the CAML representation are illustrated in Figure 4.5. We now illustrate the differences between the definition of semantic functions in the SOS and our definition in CAML with the examples in Figure 4.6, 4.7, 4.8. In Figure 4.6, it can be seen that the CAML version does not deal with subsorts i.e. $x \geq y$ shows that the value of the term y is a subsort of that of the term x. We should emphasis, at this point, that the CAML implementation is deterministic and does not deal with the

nondeterminism of actions. If we consider Figure 4.6, the idea behind the definition of the function "run" is that an individual configuration *s* (or state) is mapped to another configuration. However, besides the determinism of actions illustrated in the representation, the CAML representation remains faithful to the SOS.

```
entity:  Data → Data
evaluated:      Yielder → opdb → Local_info → Data
unfolded:       Action → Action → Action
given:   Acting → Data → Acting
received:       Acting → Bindings → Acting
simplified:     Acting → Acting
run:     State → State
stepped:        State → State
```

Figure 4.5 Semantic functions defined in CAML

operational semantics: (1) stepped $(A,l) \geq (A'$:Intermediate, l':local-info);
run$(A',l') \geq (A''$:Terminated, l'':local-info) $\Rightarrow$
run(A:Acting, l:local-info) $\geq (A'',l'')$.

(2) stepped $(A,l) \geq (A'$:Terminated, l':local-info) $\Rightarrow$
run(A:Acting, l:local-info) $\geq (A',l')$.

```
CAML:           let rec run ((A:Acting),(l:Local_info)) =
                (match stepped(A,l) with
                        (Inter (i), l') → run(Inter (i), l')
                |       (Stopped (t), l') → (Stopped (t), l')) ;;
```

Figure 4.6 SOS and CAML versions of the "run" function

Figure 4.7 illustrates how "stepped" is applied to a primitive action. In the CAML version, we check to see what kind of Acting *A* is. If the action involved is a constant action, the function "step_basic" is invoked. We then test the kind of the constant action and apply the appropriate instance. Note that in Figure 4.7, in the CAML version of the function "stepped", *ca* is of type "Constant_Action" and *inf* is of type "opdb". Also, the function "get_data" is responsible for extracting data from *inf*, if data exists. Note that "Err" corresponds to the error value for data or the "nothing" data type in the SOS. Finally, observe the exception handler provided by CAML for dealing with errors. In Figure 4.8, the CAML version of "stepped" checks the kind of Acting in question. If it is a compound Acting, the function "step_infix" is called. The kind of Actings involved are checked. If the composite Acting is of kind "Intermediate Sequencing Intermediate", then we apply the function "simplified" with the newly stepped *A1*.

Operational semantics:

(1) stepped ("complete", *d*:data, *b*:bindings, *l*:local-info) =
("completed", ( ), empty-map, *l*).

(2) stepped ("escape", *d*:data, *b*:bindings, *l*:local-info) =
("escaped", *d*, *l*).

(3) stepped ("fail", *d*:data, *b*:bindings, *l*:local-info) =
("failed", *l*).

(4) stepped ("unfold", *d*:data, *b*:bindings, *l*:local-info) =
nothing.

CAML:

```
.....
and stepped(A,l) =
match A with
    .....
|   Inter (Adb (Body (Cons_Action (ca)), inf)) → (step_basic ca inf l)
|   .....

and step_basic (ca:Constant_Action) (inf:opdb) (l:Local_info) =
(match ca with
    complete → (Stopped (Compl (Err, [ ] )), l)
|   escape → (Stopped (Escape (get_data inf)), l)
|   fail → (Stopped (failed), l)
|   unfold → raise failure "Cannot step 'unfold'")
.....
```

Figure 4.7 SOS and CAML versions of "stepped" applied to primitive actions

Operational Semantics:
stepped(A_1, l) ≥ (A_1':Acting, l':local-info);
⟦A_1 O A_2⟧: [[Intermediate Sequencing Intermediate]] ⇒
stepped (⟦A_1 O A_2 ⟧,l:local-info) ≥ (simplified ⟦A_1' O A_2⟧, l').

CAML:

.....
and stepped(*A*,*l*) =
match *A* with
.....
| Inter (AIA (*A1, O, A2*)) → (step_infix *A1 O A2 l*)
.....
and step_infix (*A1*:Acting) (*O*:Action_Infix) (*A2*:Acting) (*l*:Local_info) =
(match *A1,A2* with
(Inter (*i*), (Inter (*i'*)) →
let (*A1', l'*) = stepped (*A1,l*) in
if (Sequencing *O*) then
(simplified (Inter (AIA (*A1', O, A2*))), *l'*)
.....

Figure 4.8 Operational Semantics and CAML version of "stepped" (2)

4.4 Issues with Implementation

During the conversion to CAML, some difficulties and problems were encountered. Various compromises had to be made to give a complete representation. These problems and compromises are shown below.

4.4.1 Yielders

Looking at the yielders available, we found that yielders could not be represented properly without the definition of an "At" function corresponding to the "At" data operation which takes a map, depending on the current

information, an element and returns the range. The "At" data operation is used as a look-up for bindings and storage. Consider the following compound yielder (assuming it is a component part of an overall program)

the cell yielded by

current bindings at

the token yielded by "I"

To evaluate this yielder, we could introduce an "At" auxiliary function. However, each time the function is invoked, we must provide the current information i.e. the current transient data, bindings and storage. As it is not possible to provide this information dynamically (as the program executes) and it is only possible to provide the current information initially, the interpretation of the whole program would be incorrect. Therefore, we found it was necessary to introduce an extra yielder referred to as the "At" yielder as the evaluation of a yielder always has access to the current information. The yielder in CAML looks as follows:

which is translated into CAML as

```
Yield_by (ccell,
At (current_bindings, Yield_by (ttoken, Data_Con (intoken "I"))))
```

Also, a "Component" yielder was required which would extract individual items from a Data list.

4.4.2 Error values and Exceptions

A value was required corresponding to the "nothing" data type, see [Mosses, 1992] (Appendix C, C.3.3.1) which allows the enclosing action to fail. For debugging purposes, exception handling was also needed. This was due to the fact that most semantic functions were partial. These exceptions would be

expected to provide suitable error messages. So, the value "Err" was added to the definition of Data. CAML also provided a facility for exception handling.

4.4.3 Inconsistency

An inconsistency was encountered in the SOS in the application of stepped to the state

(⟦"unfolding" *A*:Action ⟧, *d*:data, *b*:bindings, *l*:local-info).

After application, the following state was returned

(given (received (unfolded (*A*, ⟦"unfolding" *A* ⟧), *b*), *d*), *l*)

We must note that the type of the semantic functions for "unfolded" and "received" are

- unfolded _ :: (Action, Action) → Action
- received _ :: (Acting, bindings) → Acting

We see that the application of "unfolded" to the actions - *A*,⟦"unfolding" *A*⟧ returns an action, say *ac*. So, the application of "received" to (*ac*,*b*) cannot take place.

In our CAML representation, the function "stepped" calls a function "step_unfold" for this particular kind of state. This function, in turn, invokes the function "act_out" which converts the action after application of "unfolded" to an appropriate Acting. Appropriate calls to "given" and "received" take place. The function "stepped", "step_unfold" and "act_out" are illustrated in Figure 4.9

```
and stepped (A,l) =
match A with
.....
|       (Inter (Adb (Pre_Action (unfolding, a) inf)) →
        (step_unfold a inf l)
.....
and step_unfold (a:Action) (inf:opdb) (l:Local_info) =
let (ac:Action) = (unfolded (a, (Pre_Action (unfolding, a)))) in
(match inf with
        no_db → ((act_out ac), l)
|       is_d(d) → (given (act_out ac, d), l)
|       is_b(b) → (received (act_out ac, b), l)
|       is_db (d,b) → (given (received (act_out ac, b), d), l))
.....
and act_out (A:Action) =
(match A with
        Body (sa) → Inter (Adb (Body (sa), no_db))
|       Pre_Action (ap,a) → Inter (Adb (Pre_Action (ap,a), no_db))
|       In_Action(a1,ai,a2)→ Inter(AIA((act_out a1),ai,(act_out a2))))
```

Figure 4.9 The CAML functions "stepped", "step_unfold" and "act_out"

4.4.4 Additions to "simplified"

There was no instance of "simplified" to deal with:

$[\![A_1 \ O \ A_2]\!]$: [[Completed Interleaving Intermediate]]

So, "simplified" was updated with an instance dealing with

$[\![A_1 \ O \ A_2]\!]$: [[Completed Interleaving Intermediate]]

to avoid a failure in pattern matching. This instance returned the Acting provided to "simplified".

4.4.5 Storage Yielder

There was no yielder to return the next available cell in storage. If we consider the "choose a cell" simple action, a "Next cell" yielder was defined to facilitate this. We must note again that our implementation is deterministic and therefore, the above yielder will have an individual outcome.

4.4.6 Additions to "stepped"

There was no instance of "stepped" dealing with the Intermediate Acting

⟦Acting ("before"|"then before") Acting bindings⟧

A function "step_dec" was created to deal with this scenario illustrated in Figure 4.10.

```
and stepped (A,l) =
match A with
.....
|       Inter (AbA (A1, O, A2, b)) → (step_dec A1 O A2 b l)
.....

and step_dec (A1:Acting) (O:Action_Infix) (A2:Acting) (b:Bindings)
                (l:Local_info) =
(match (A1,A2) with
        (Inter (i), Inter (i')) →
        let (A1', l') = stepped (received (A1,b), l) in
        (simplified (Inter (AbA (A1', O, A2, b))), l')
|       _ → raise failure "Not appropriate" )
```

Figure 4.10 Definition of the functions" stepped" and "step_dec"

4.5 Verifying the Correctness of the CAML Representation

Some examples of action notation have been used to test the correctness of the CAML representation. These are as follows:

- Variable declaration
- Variable declaration followed by an assignment statement
- Two consecutive declarations i.e. a variable declaration followed by a constant declaration
- Two consecutive declarations (as above) followed by an assignment statement
- An "if" statement
- A "while" loop

However, we must emphasise that these tests fall short of thoroughly testing the CAML implementation. More extensive testing could not be carried out due to time constraints. The above examples were translated into kernel notation. This was then converted to an appropriate CAML representation which was executed alongside the existing functions to give appropriate CAML output states. The CAML representations of these examples are rather lengthy so a very simple example will be given in Figure 4.11. Note that "choose" is a simple action which chooses the next available cell in storage. "Next_cell" is the yielder used with the "choose" simple action. "no_db" is of type "opdb" and corresponds to empty data and bindings.

```
kernel notation:        choose a cell
                then reserve the cell yielded by them

CAML version:
        run (Inter (AIA
                (Inter (Adb
                        (Body (Simp_Pre_Action(choose,Next_cell)),no_db)),
                inthen,
                Inter (Adb
                        (Body(Simp_Pre_Action(reserve,
                                Yield_by(ccell, them))),no_db))))), [ ]) ;;

State output by CAML:
        ((Stopped (Compl (Err, [ ]))), [(1,Err)]):State
```

Figure 4.11 Example to reserve the next cell in storage

4.6 Summary

In this chapter, we have taken a look at functional languages and in particular, the functional language "CAML". We then illustrated the translation from the SOS to CAML along with the difficulties encountered during the translation. The various solutions to these difficulties were also described. Finally, we looked at the various examples created to test our CAML representation.

CHAPTER 5

Implementation of Action Notation in Coq

5.1 Introduction

Since we are satisfied that the CAML version adequately represents certain facets of action notation, we propose to translate this to a version written in the specification language of the Coq development system. Since the primitive actions and action combinators of action notation satisfy a variety of algebraic laws, this can lead to the proof that semantic equivalences exist between pairs of constructs, expressions etc. of a programming language. Therefore, since Coq is a proof assistant, it should be possible to prove the existence of the algebraic laws and possibly, at a later date, look at the notion of semantic equivalence using these proofs. In this chapter, we give a description of Coq, the translation to Coq compared with CAML and also, various proofs of the action laws. We should emphasise, at this point, that Coq is suitable for the implementation of the nondeterminism of action notation whereas the CAML implementation is purely deterministic.

5.2 Description of Coq

Coq is a proof assistant for higher order logic which is constructive allowing powerful axiomisations and inductive definitions, refer to [Cornes et al., 1995] [Dowek et al., 1993]. Coq is an implementation of the Calculus of Inductive Constructions (CIC) which is a variety of type theory where theorems to be proved are represented as types. Coq allows the interactive construction of formal proofs. It is the result of about ten years of research of the Formel project and has three main attributes - the logical language in which axiomatisations and specifications are written referred to as "Gallina", a proof assistant which allows the development of mathematical proofs and a program extractor which can create a program matching its formal specification.

There are two basic sorts in Coq - a Set allowing the definition of objects and a Proposition which allows the definition of predicates and relations about these objects as well as the definition of propositions to be proved.

5.2.1 Sets

Objects can be defined and axioms can be declared. The type of natural numbers with its constructors O and S can be introduced as shown in Figure 5.1.

```
Parameter nat:Set.
Parameter O:nat.
Parameter S:nat → nat.
```

Figure 5.1 Definition of natural numbers

So, according to Figure 5.1, "nat" is introduced as a type with its constructors O and S introduced as types "nat" and "nat → nat" respectively. The main constructions allowed have the form:

x, (M N), [*x*:T]M, (*x*:T)P

"*x*" denotes variables or constants. (M N) denotes the applications of a functional object M to object N e.g. (S O). [*x*:T]M denotes λ-abstraction with *x* as the bound variable of type T and M is the body e.g. [*x*:nat](S (S *x*)) is a function mapping *x* to the successor of its successor. (*x*:T)P denotes a product type. A definition allows terms to be related to names e.g.

Definition plus_two = [*x*:nat](S (S *x*)):nat → nat.

In Coq, a name can be replaced by its definition e.g.

(plus_two (S O))

(([*x*:nat](S (S *x*))) (S O))

(S (S (S O))) (obtained by β-reduction)

Inductive sets can also be defined. The new type is added with its constructors as is an induction principle for propositions, a recursion principle for sets and a destructor operator "Match" for defining recursive functions over the type.

The Set of natural numberscould be re-defined as an inductive Set as follows:

Inductive Set nat = O:nat | S:nat → nat

The following principles are added by the system:

nat_ind: (*P*:nat → Prop)(*P* O)→((*x*:nat)(*P x*)→(*P* (S *x*)))→(*n*:nat)(*P n*)

nat_rec: (*P*:nat → Set)(*P* O)→((*x*:nat)(*P x*)→(*P* (S *x*)))→(*n*:nat)(*P n*)

An inductive set can be defined with parameters:

Inductive Set list[*A*:Set] = nil:(list *A*) | cons:(*A*→(list *A*)→(list *A*).

Pattern matching on inductive types is done using "Match" e.g. Match *t* with *e1 e2*.....*en*. An example follows:

Definition plus: nat → nat → nat =

[n,m:nat] (⟨nat⟩Match n with

(* O *) m

(* S p *) [p:nat] [pluspm:nat] (S pluspm)).

In this example, (plus m O) is convertivle with m (the first argument of the match operation) and (plus (S p) m) is convertible with (S (plus p m)) (the second argument [p:nat] [pluspm:nat] (S pluspm) applied to p and to the recursive call (plus p m)). There is one clause corresponding for each constructor of natural numbers. In the clause for "S" (successor), there is one argument corresponding to the argument of "S" .

5.2.2 Propositions

The type "Prop" is used for defining propositions, statements that may be proved. Predicates and relations are defined as functional terms from sets to propositions. A predicate over the natural numbers could have type "nat → Prop". In Coq, various logical connectives are primitive while others are defined. These definitions and inference rules are loaded when the system is started. Some examples are given in Figure 5.2.

- True — Tautological Proposition
- False — Absurd Proposition
- P→Q — P implies Q
- (*x*:T)P — If P is proposition where a free variable of type T may occur then (*x*:T)P is the proposition ("for all *x* in T, P")
- ~P — not P
- P∧Q — P and Q
- P∨Q — P or Q
- <T>Ex([*x*:T]P) — If P is a proposition where a free variable of type T may occur, then <T>Ex([*x*:T]P) is the proposition ("there exists an *x* in T such that P")

Figure 5.2 Examples of Propositions

As well as composing propositions from the above connectives, predicates and relations can be inductively defined with types as described above e.g. consider the following predicate that defines when a natural number is even:

Inductive Definition Even:nat → Prop =

Even_O:(Even O)

| Even_SSn:(*n*:nat)(Even *n*)→(Even (S (S *n*))).

The new predicate "Even" is defined as type "nat → Prop". The new constructors (or labels) "Even_O" and "Even_SSn" can be used in the proof of propositions. The clause labelled "Even_O" defines that the natural number O is even and the clause labelled "Even_SSn" defines that if a given natural number is even, then the successor of the successor of that number is also even.

5.2.3 The Proof Engine

This is the goal-directed theorem prover. To prove a proposition, it is entered at the Coq prompt after the command "Goal". Then, tactics can be entered which apply backward proof steps to the goal in an attempt to prove the proposition. The tactics can be outlined as follows as in [Seaman & Felty, 1993]:

1. The introduction tactics discharge universally quantified variables and hypotheses into the local context[4] e.g. if A → B is a goal, "Intro" introduces A into the local context and the goal changes to B. A name can be assigned to the term using "Intro n". "Intros" repeats "Intro" until the goal is no longer a product.

2. "Exact H" proves the goal if the goal is a hypothesis in the local context and is labelled "H". In the case of "Assumption", this tactic looks for a proof by

[4] current set of hypotheses for the current goal

an assumption in the local context. If there is no hypothesis in the local context which proves the goal, it fails.

3. "Apply H" applies a theorem or hypothesis H to the goal. If the goal is B, and hypothesis H is $A \rightarrow B$, "Apply H" eliminates B as the goal. However, A then becomes the goal.

4. "Elim H" when H is a hypothesis, axiom or proved theorem. For example, if H is "$A \wedge B$", the goal "C" is transformed into "$A \rightarrow B \rightarrow C$". The tactic "Induction n" is equivalent to performing "Intro" until n is reached following by "Elim n" if n is a quantified variable in the goal.

5. Tactics dealing with connectives like $\vee, \wedge, =$ are for example, "Left" which applies or-introduction-left if the goal is $A \vee B$ changing the goal to A and similarly for "Right".

There are three tactics working with equality - if the goal is a reflexive equation, it is solved with the tactic "Reflexivity". "Symmetry" changes a goal a=b to b=a, "Transitivity c" gives two subgoals a=c and c=b. Also, assume H:a=b is a hypothesis or theorem, "Rewrite ->H" replaces occurrences of a in the goal with b and "Rewrite <-H" replaces occurrences of b with a. "Replace a with b" replaces a in the goal with b and adds a=b as a new subgoal unless it is one of the hypotheses in the local context.

6. "Absurd H" allows proofs by contradiction. The current goal is proved by elimination of "False" and "False" comes from proofs of both H and ~H. Therefore, the tactic generates two subgoals H and ~H.

7. "Unfold f" replaces occurrences of f in the goal with its definition. "Change A" replaces the goal with A as long as the goal is convertible (following the β rule, δ rule and elimination rules for inductive terms) with A. "Red" replaces only the head constant of the conclusion with its definition e.g. ~A becomes A→False. "Simpl" simplifies the goal by unfolding constants with their definitions and performing β-reduction.

5.3 Coq Representation of Action Notation

We should note that only the semantic/syntactic entities and semantic functions for the functional, basic, declarative and imperative facets have been defined. We should emphasise that the translation to the Coq specification language was made using the SOS as the source document and not the CAML implementation.

The structure defined in the Coq representation was as follows:

- Most syntactic and semantic entities were inductively defined with appropriate constructor names and types. The definitions were very similar to the CAML definitions.
- The semantic functions that were represented in the CAML version were defined i.e. run, stepped, given, received, unfolded, evaluated, entity. However, they were defined relationally to facilitate theorem proving at a later date. This was made possible using inductive definitions. We should note that since it was possible to define the semantic functions relationally, it was possible to implement the nondeterminism of actions. So, we endeavour to show in a later section that the Coq implementation is indeed faithful to the SOS , see [Mosses, 1992] (Appendix C) using proofs of the various action laws.

- The auxiliary functions were defined as functions and relations where appropriate. It was sometimes the case that types and commands which were built into CAML needed to be defined in Coq e.g. the "bool" type, the "if" operator. This shall be discussed in a later section.
- The action combinator collections were defined i.e. Normal, Sequencing and Interleaving.

The structure of the syntax, semantic entities, auxiliary functions and semantic functions in our Coq representation will be illustrated below. Note that the structure of the sections in chapter 4 will remain in this chapter to allow the comparison between the Coq and CAML versions. Refer to the Appendix for the mappings from the SOS into Coq.

5.3.1 Syntax

Some examples of the Coq definitions of syntactic entities with the associated SOS explanations are given below in Figures 5.3, 5.4. Note the similarities with the CAML definitions. Note in Figure 5.3 that the sets "bin_op" and "un_op" were used before defined. When using a basic inductive definition, this is not permitted. However, this was overridden by the use of a mutual inductive definition.

```
Mutual Inductive Yielder:Set: = Data_Con: Data → Yielder
                (* Data-Constant *)
                |       Unary_Op: un_op → Yielder → Yielder
                (* Data-Unary *)
                |       Binary_Op: bin_op → Yielder → Yielder → Yielder
                (* Data-Binary *)
                |       Selected: Yielder → Yielder → Yielder → Yielder
                (* "if" Yielder "then" Yielder "else" Yielder *)
                |       Yield_by: Datum → Yielder → Yielder
                (* "the" Data "yielded by" Yielder *)
                |       Received: Yielder → Yielder → Yielder
                (* Yielder "receiving" Yielder *)
                |       them: Yielder
                (* "them" *)
                |       current_bindings: Yielder
                (* "current bindings " *)
                |       current_storage:Yielder
                (* "current storage" *)
                |       At: Yielder → Yielder → Yielder
                |       Comp: nat → Yielder →Yielder
                |       Next_cell: Yielder
with                    bin_op:Set:=
                        oplus:bin_op
                |       ominus:bin_op
                |       oeq:bin_op
with                    un_op:Set:=
                        bool_not:un_op.
```

Figure 5.3 Coq definition of Yielders

```
Mutual Inductive Datum:Set:=
                    incell:cell → Datum          (* cell *)
                |   inval:val → Datum            (* value *)
                |   intoken:token → Datum        (* token *)
                |   inbool:bool → Datum          (* boolean *)
                |   inBind:Bindings →Datum       (* bindings *)
                |   inStore:Store → Datum        (* storage *)
                |   Data_list:(list Datum) → Datum (* list of data *)
                |   vval:Datum                   (* domain of value *)
                |   ccell:Datum                  (* domain of cell *)
                |   ttoken:Datum                 (* domain of token *)
                |   boolean:Datum                (* domain of boolean *)
                |   Err:Datum                    (* "nothing" *)
with
                    Bindings:Set:=
                    Bind: ((token → Datum) → Bindings)
with
                    Store:Set:=
                    Storage: (((cell → Datum) * current ) →Store).
```

Figure 5.4 Definition of Data

Note that in Figure 5.4, that bindings and storage are represented using functions. Therefore, because we are not representing storage using lists, this can cause difficulties when trying to calculate the next cell in storage. This problem was overcome by introducing the type "current" which retains information about the current state of storage. This was tupled with the storage map. Note also the existence of Err for Data corresponding to the "nothing" data type in the operational semantics.

5.3.2 Semantic Entities

The definition of "Acting" is given in Figure 5.5 with the subsidiary entities given in Figure 5.6. Also, the definition for "Stat" (State) is given in Figure 5.7.

```
Mutual Inductive Acting:Set:=
                    Stopped:Terminated → Acting
                |   Inter:Intermediate → Acting
with
                    Intermediate:Set:=
                    APA: Action_Prefix → Acting → Intermediate
                |   Adb: Action → opdb → Intermediate
                |   AIA: Acting → Action_Infix → Acting → Intermediate
                |   AbA: Acting → Action_Infix → Acting → Bindings →
                    Intermediate
with
                    Terminated:Set:=
                    Compl: Datum → Bindings → Terminated
                |   Escape: Datum → Terminated
                |   failed: Terminated
with
                    opdb:Set:=
                    no_db:opdb
                |   is_d: Datum → opdb
                |   is_b: Bindings → opdb
                |   is_db: Datum → Bindings → opdb
```

Figure 5.5 Definition of Acting

As explained in section 4.3.2, the size of the definition of "Intermediate" has been reduced and some of the hierarchy for the definition of "Acting" has been eliminated.

```
Definition token := nat.
Definition cell := nat.
Definition val := nat.
Definition current := cell.
```

Figure 5.6 Definition of subsidiary entities

We can see from Figure 5.6, that we chose to represent tokens as natural numbers. This is due to the fact that there are no character" or "string" types available in Coq. Figure 5.7 gives the definition of a state. The error state was included to facilitate theorem proving. This is somewhat different from the CAML version where an error state is not required. This will be discussed in more detail in section 5.3.4.

```
Inductive Stat:Set:=
        Stat_ok: Acting → Local_Info →Stat
|       Stat_err:Stat.
```

Figure 5.7 Definition of Stat (state)

5.3.3 Auxiliary Functions

When performing the translation from CAML to Coq, the absence of an "if" operator (for Sets) in Coq caused some inconvenience. So, it was appropriate to define the "if" operator as it is used frequently in our CAML version. We needed to introduce the type "bool" (boolean) to complete the definition. This allowed case by case analysis allowing the definition of the "if" operator. The "bool" type is defined as follows:

Inductive bool:Set:= true:bool | false:bool.

It was then appropriate to define our version of the "if" operator as illustrated in Figure 5.8.

Recursive Definition ifset[X:Set]: bool $\rightarrow X \rightarrow X \rightarrow X$:=
true $x\ y \Rightarrow x$
| false $x\ y \Rightarrow y$.

Figure 5.8 Definition of "if" operator (strict)

In Figure 5.8, a boolean expression is provided as an argument which evaluates to either true or false. If true, x is returned, else y is returned. The function is polymorphic and therefore, can be defined on any Set. Other auxiliary functions had to be defined on the type "bool" to allow expressions to be formed e.g. "equal_nat", which compares two natural numbers. The parameterised type "list" was also defined with appropriate operations.

As we chose to define bindings and storage as functions instead of lists in the Coq version, the operations on these had to be redefined. Also, note that storage not only consists of the storage map but also of the current cell. The differences between the manipulation of bindings and storage in CAML and Coq

are given in Figure 5.9. We use the function "access" and the relation "access_env" as examples. Note that we have adopted a relational approach in Coq. This was required to facilitate theorem proving. Also, note when we are using the relation "access_env" on storage, we only provide the storage map as an argument.

```
CAML:
(* access:      'a → ('a * 'b)list → 'b *)
let rec access t =
function        [] → raise failure "access error"
        |       x::L → if (fst (x) = t) then
                                snd(x)
                        else
                                access t L ;;

Coq version:
(* access_env:          nat → (nat → Datum) → Datum → Prop *)
Inductive access_env: nat → (nat → Datum) → Datum → Prop:=
acc_env: (t:nat) (mp:nat → Datum) (access_env t mp (mp t)).
```

Figure 5.9 Definition of "access" in both CAML and Coq.

5.3.4 Semantic Functions

The types of the semantic functions defined are illustrated in Figure 5.10. As we can see, all semantic functions have been defined relationally with the exception of "entity".

entity:	Datum → Datum
evaluated:	Yielder → opdb → Local_Info → Datum → Prop
unfolded:	Action → Action → Action → Prop
given:	Acting → Datum → Acting → Prop
received:	Acting → Bindings → Acting → Prop
simplified:	Acting → Acting → Prop
run:	Stat → nat → Stat → Prop
stepped:	Stat → Stat → Prop

Figure 5.10 Semantic functions defined in Coq

We now illustrate the differences between the semantic functions as defined in section 4.3.3 (CAML) and in Coq. Consider Figures 5.11 -Figure 5.13. In Figure 5.11, we can see that there is many differences between the two versions. In the Coq version, we have introduced a natural number *n* which indicates the number of steps it takes to advance an intermediate state to termination. This shall be discussed in more detail in section 5.5.1 in relation to theorem proving. The first instance of "run" (labelled RStop) deals with the situation where it only takes one step for a state to arrive at a terminated state. The second instance (labelled RInter) deals with the circumstances where it takes more than one step to terminate.

```
CAML:   let rec run ((A:Acting),(l:Local_info))=
        (match stepped (A,l) with
                (Inter (i), l') → run(Inter (i), l')
        |       (Stopped (t), l') → (Stopped (t), l')) ;;

Coq:    Inductive run: Stat → nat → Stat → Prop:=
        RStop:(A:Acting)(l:Local_Info)(t:Terminated)(l':Local_Info)
        (stepped (Stat_ok A l) (Stat_ok (Stopped t) l')) →
         (run (Stat_ok A l) O (Stat_ok (Stopped t) l' ))

      |  RInter:(A:Acting)(l:Local_Info)(St:Stat)
                (i:Intermediate)(l':Local_Info)(n:nat)
        (gt n O)→
         (stepped (Stat_ok A l) (Stat_ok (Inter i) l')) →
          (run (Stat_ok (Inter i) l') (pred n) St) →
           (run (Stat_ok A l) n St)

      |  RErr: (A:Acting)(l,l':Local_Info)(i:Intermediate)
         (stepped (Stat_ok A l) (Stat_ok (Inter i) l')) →
          (run (Stat_ok A l) O Stat_err).
```

Figure 5.11 CAML and Coq versions of the "run" function

The third (labelled RErr) explains that if it the state is not terminated after taking one step, then it has arrived at an error state. As we can see, only two instances of "run" are required in the CAML version as we were not concerned with theorem proving at that point.

In the Coq section of Figure 5.13, the function "is_inter" is responsible for checking whether an Acting is "Intermediate".

```
CAML:
.....
and stepped (A, l) =
match A with
        .....
|       (Inter (Adb (Body (Cons_Action (ca)), inf)) → (step_basic ca inf l)
|       .....

and step_basic (ca:Constant_Action) (inf:opdb) (l:Local_info) =
(match ca with
        complete → (Stopped (Compl (Err, [ ])), l)
|       escape → (Stopped (Escape (get_data inf)), l)
|       fail →  (Stopped (failed), l)
|       unfold → raise failure "Cannot step 'unfold'")
.....

Coq:
Mutual Inductive stepped :Stat → Stat → Prop:=
        .....
|       St_AdbCA:(ca:Constant_Action)(inf:opdb)(l:Local_Info)(st:Stat)
         (step_basic ca inf l st) →
          (stepped (Stat_ok (Inter (Adb (Body (Cons_Action ca)) inf)) l) st)
.....
with
step_basic:Constant_Action → opdb → Local_Info → Stat →Prop:=
        Sb_comp:(l:Local_Info)(inf:opdb)
        (step_basic complete inf l (Stat_ok (Stopped (Compl Err (Bind
        empty_bindings))) l))
|       Sb_esc:(l:Local_Info)(inf:opdb)(d:Datum)
        (get_data inf d) →
         (step_basic escape inf l (Stat_ok (Stopped (Escape d)) l))
|       Sb_fail:(l:Local_Info)(inf:opdb)
        (step_basic fail inf l (Stat_ok (Stopped failed) l))
.....
```

Figure 5.12 CAML and Coq version of "stepped" applied to primitive actions

```
CAML:

.....
and stepped (A, l) =
match A with
        .....
|       Inter (AIA (A1, O, A2)) → (step_infix A1 O A2 l)
.....
and step_infix (A1:Acting)(O:Action_Infix)(A2:Acting)(l:Local_Info) =
(match A1, A2 with
        (Inter (i), (Inter (i')) →
        let (A1', l') = stepped (A, l) in
                if (Sequencing O) then
                        (simplified (Inter (AIA (A1', O, A2))), l')
.....

Coq:

Mutual Inductive stepped: Stat → Stat →Prop:=
        .....
|       St_AIA: (A1,A2:Acting)(O:Action_Infix)(l:Local_Info)(st:Stat)
         (step_infix A1 O A2 l st)→
          (stepped (Stat_ok (Inter (AIA A1 O A2)) l) st)
        .....
with
step_infix:Acting → Action_Infix → Acting → Local_Info → Stat → Prop:=
        Si_InIn:(A1,A2,A1',A:Acting)(O:Action_Infix)(l,l':Local_Info)
         ((is_inter A1)/\((Sequencing O)\/(Interleaving O)\/(O=inor))/\
         (is_inter A2))→
          (stepped (Stat_ok A1 l) (Stat_ok A1' l'))→
           (simplified (Inter (AIA A1' O A2)) A)→
            (step_infix A1 O A2 l (Stat_ok A l'))
.....
```

Figure 5.13 CAML and Coq version of "stepped" (2)

5.4 Issues with Implementation

With reference to the problems encountered when translating the SOS to CAML, we look at how Coq has dealt with them.

5.4.1 Yielders

The additional three yielders were added to the definition of yielders as shown in Figure 5.3.

5.4.2 Error values and Exceptions

Error values were added in the definitions of "Datum" and "Stat". A value corresponding to the “nothing” data type was added in the definition of “Datum”. Also, the definition of “Stat” (State) includes a value for an error State. Exceptions were overridden through the use of inductive definitions where only possible situations were defined.

5.4.3 Inconsistency

With regard to the inconsistency discovered in the application of "stepped" to the following state:

(⟦"unfolding" *A*:Action⟧, *d*:data, *b*:bindings, *l*:local-info),

the conversion function introduced in the CAML version was also defined in Coq i.e. "act_out".

5.4.4 Addition to "simplified"

An instance of "simplified" to deal with

⟦A_1 O A_2⟧: ⟦Completed Interleaving Intermediate⟧

was defined.

5.4.5 Addition to "stepped"

The "stepped" relation was also expanded to deal with the instance:

⟦Acting ("before" | "then before") Acting⟧ .

5.5 Proving Action Laws in Coq

As we have seen previously, there is a collection of laws that characterise actions. These laws can then be used in semantic equivalence proofs between pairs of constructs of a programming language. In this section, we outline the possibilities that Coq provides for the proof of equivalence between actions along with the verification that equivalences exist between actions. Note that we are not discussing equality between actions but equivalence which shall be discussed in the following section. The action laws that we endeavoured to prove are illustrated in Figure 5.14. These action laws were chosen as they encompass all the action combinators.

(1) **unfolding fail = fail**

(2) **fail hence** *A* **= fail**

(3) **fail and then** *A* **= fail**

(4) *A2* **=** *A2* **and then complete**

(5) *A1* **= complete and then** *A1*

(6) **escape and then** *A* **= escape**

(7) *A1* **or** *A2* **=** *A2* **or** *A1*

(8) *A2* **or** *A1* **=** *A1* **or** *A2*

(9) **escape trap** *A* **=** *A*

(10) **fail trap** *A* **= fail**

(11) **complete trap** *A* **= complete**

Figure 5.14 Action laws to be proved

5.5.1 Representing Equivalence between Actions

Firstly, we should define "equivalence between actions". Consider two states S1, S2 containing the two actions A1, A2 repectively and the same local information. We say that equivalence between actions exists if given two states S1, S2 (as above), if it takes *n* steps for S1 to terminate in state S1', then there exists some natural number *m* such that it takes *m* steps for S2 to terminate in state S1'. Note that if it takes *n* steps for S1 to terminate and *m* steps for S2 to terminate in equivalent states, if we step S1 and S2 and then step the resulting

states (n-1) and (m-1) times respectively, we should obtain the same result as above. However, if we step S1 and S2 (n-1) and (m-1) times respectively, the resulting states should be undefined as S1 and S2 have not been run to termination. We have also defined the "undefined state" which denotes the state that is not terminated. The state reached after n - 1 steps is always undefined if it takes n steps to run to termination. We saw in section 5.3.4 that the above notions are represented in the definition of the semantic function "run". The definition of equivalence between actions is defined in the "equiv" relation in Figure 5.15. The definition of "state" is given in Figure 5.7.

```
Inductive equiv: Stat → Stat → Prop:=
        St_Err:(St2:Stat)(equiv Stat_err St2)
|       Err_St:(St1:Stat)(equiv St1 Stat_err)
|       St_St:  (St1,St2:Stat)(St1 = St2) → (equiv St1 St2).
```

Figure 5.15 Definition of the "equiv" relation

5.5.2 Proving Equivalence between Actions

The main aspect of Coq that we are interested in here is the goal directed theorem prover. To prove a lemma, it is entered at the Coq prompt and tactics are then entered in an attempt to complete the proof. Consider the scenario where the specification of the operational semantics of action notation has already been loaded into Coq. Our next step is to start proving the properties of action notation i.e. the action laws. We have decided to prove a subset of these action laws, as illustrated in Figure 5.14. To perform the proofs, the laws were translated into a style conforming to our specification i.e. we used the semantic function "run" to show that after executing the left-hand side and the right-hand

side, they arrived at the same state. As an example, consider action law (1) in Figure 5.14. The lemma appropriate to this property is shown in Figure 5.16. We should note that the "equiv" relation is responsible for determining whether two states are equivalent. This relation is much weaker than the equality relation. The definition tells us that if either or both of the two states we are comparing is an error state, then the two states are automatically equivalent. In the case where neither of the two states are error states, we check for equality. In Figure 5.16, we construct a state consisting of an Acting appropriate to the action "**unfolding fail**" i.e. (Inter (Adb (Pre_Action unfolding .....) and also a state which involves an Acting representing the primitive action "**fail**". The two states are evaluated after application of "run" and are then compared by applying "equiv". Obviously, they should evaluate to the same state. Note that in Figure 5.16, we run the state containing the action "**unfolding fail**" for an additional step. This is due to the presence of "**unfolding**". Note that this is not a necessary condition for the law to be valid.

```
Lemma unfolding_fail:
        (n:nat)
         (l:Local_Info)
          (St1,St2:Stat)
           (run
            (Stat_ok
             (Inter
              (Adb (Pre_Action unfolding (Body (Cons_Action fail)))
             no_db)) l)
            (S n) St1)
           →(run
               (Stat_ok (Inter (Adb (Body (Cons_Action fail)) no_db)) l)
               n St2)
             → (equiv St1 St2).
```

Figure 5.16 Action law "unfolding fail = fail" represented in Coq

5.5.2 Proving Theorems in Coq

The lemma in 5.16 was entered at the Coq prompt. Appropriate tactics were then entered. The lemmas appropriate to the action laws in Figure 5.14 were all proved by induction on the natural numbers. If we take action law (1) and consider the effects after entering an initial tactic i.e. Intro , refer to Figure 5.17. We can see that n has been introduced into the local context. We then eliminate n as elimination tactics are useful to prove statements by induction i.e they make use of the induction principles generated with induction definitions. The effect of eliminating **n** is given in Figure 5.18.

```
n:nat
============================
(l:Local_Info)
 (St1:Stat)
  (St2:Stat)
   (run
    (Stat_ok
     (Inter
      (Adb (Pre_Action unfolding (Body (Cons_Action fail))) no_db))
     l)
    (S n) St1)
  →(run (Stat_ok (Inter (Adb (Body (Cons_Action fail)) no_db)) l)
       n St2)
      →(equiv St1 St2)
```

Figure 5.17 The goal after applying "Intro."

We can then use the various instances of "run" to prove the "O" case. Usually, we would use the induction hypothesis to prove the "n+1" case, but in this example, the "n+1" case is absurd. The reason for this is that it will never take "(S n)" steps to run the primitive action "**fail**".

Now, we should take a look at the action combinator "**or**", If we consider the action "*A1* **or** *A2*", it chooses either *A1* or *A2* to be performed. If one sub-action fails, the other sub-action is chosen. However, if neither subaction fails, the choice is non-deterministic. We had to deal with this situation when proving the actions laws (7) and (8) in Figure 5.14. The solution was to

```
n:nat
============================
(l:Local_Info)
 (St1:Stat)
  (St2:Stat)
   (run
    (Stat_ok
     (Inter
      (Adb (Pre_Action unfolding (Body (Cons_Action fail))) no_db))
     l)
    (S O) St1)
   →(run (Stat_ok (Inter (Adb (Body (Cons_Action fail)) no_db)) l) n St2)
         →(equiv St1 St2)

subgoal 2 is:
(n:nat)
 ((l:Local_Info)
   (St1:Stat)
    (St2:Stat)
     (run
      (Stat_ok
       (Inter
        (Adb (Pre_Action unfolding (Body (Cons_Action fail))) no_db)) l) (S n)
       St1)
     →(run (Stat_ok (Inter (Adb (Body (Cons_Action fail)) no_db)) l) n St2)
       →(equiv St1 St2))
→(l:Local_Info)
    (St1:Stat)
     (St2:Stat)
      (run
       (Stat_ok
        (Inter
         (Adb (Pre_Action unfolding (Body (Cons_Action fail))) no_db)) l)
       (S (S n)) St1)
      →(run
         (Stat_ok (Inter (Adb (Body (Cons_Action fail)) no_db)) l) (S n) St2)
        →(equiv St1 St2)
```

Figure 5.18 Goal after eliminating n (Elim n)

```
Lemma A1 or A2:
        (n:nat)(A1,A2:Acting)(l:Local_Info)(St1:Stat)
        (run (Stat_ok (Inter (AIA A1 inor A2)) l) n St1) →
        (Ex [St2:Stat]
         (run (Stat_ok (Inter (AIA A2 inor A1)) l) n St2) /\
          (equiv St1 St2)).
```

Figure 5.19 Action law "*A1* **or** *A2* = *A2* **or** *A1*"

to introduce existential quantification on one side of the equations. The lemma appropriate to (7) is given in Figure 5.19. By introducing an existential quantifier over St2, it was possible to choose the evaluation path of St2 in accordance with St1. The semantic function "stepped" gives two choices for the evaluation of the composite Acting "*A1* inor *A2*" i.e. either step *A1* first or *A2*.

To prove some of the action laws, two additional lemmas were required. These are outlined in Figure 5.20. An axiom "deter" was also defined, see Figure 5.20. The axiom "deter" defined the equality of states which was required in the invocation of "equiv". Also, the equality between actions contained in states is proven in the lemmas "injstat","injstat2".

```
Lemma injstat:(A:Acting)(A':Acting)(l:Local_Info)(l':Local_Info)
              (Stat_ok A l) = (Stat_ok A' l') → A = A'.

Lemma injstat2:(A:Acting)(A':Acting)(l:Local_Info)(l':Local_Info)
              (Stat_ok A l) = (Stat_ok A' l') → l = l'.

Axiom deter: (A1:Acting)(A2:Acting)(A3:Acting)
              (l1:Local_Info)(l2:Local_Info)(l3:Local_Info)
                    (stepped (Stat_ok A1 l) (Stat_ok A3 l3))
                    →     (stepped (Stat_ok A1 l1) (Stat_ok A2 l2))
                          →     (Stat_ok A3 l3) = (Stat_ok A2 al2).
```

Figure 5.20 Lemmas "injstat, injstat2" and axiom "deter"

The lemmas in Figure 5.20 were proved using the "Injection" tactic.

5.6 Summary

We have given an introduction to the Coq development system. Some aspects of the specification language and the proof assistant have been explained. We then illustrated the translation of the SOS to the specification language of the Coq system. The compromises made in this translation along with some additional features were given. We then dealt with the proofs of the various action laws using the Coq proof assistant. The action laws which were proved were outlined as in Figure 5.14. We found one proof in particular to be much more difficult than other proofs. The proof of the action law *A1* **or** *A2* = *A2* **or** *A1* could be described as a long and complicated proof. The nondeterminism of the "or" action combinator meant all instances had to be proved hence making this proof long and complicated.

CHAPTER 6

Conclusions and Further Work

6.1 Research Considerations & Conclusions

During the research underlying this thesis, the following results were achieved:

- Representation of action notation in the functional language CAML
- Representation of action notation in the Coq development system
- Proof of various action laws using the Coq theorem prover

Before deciding on action semantics as the chosen semantic specification type, many other types of semantics were investigated. It was concluded from these investigations that action semantics was one of the more accessible types of specification. For example, we found that denotational semantics, in particular, was sometimes confusing and difficult to understand due to its purely functional representation. Action semantics specifications, on first reading, proved to be clear, concise, easy to read and hence more comfortable and understandable. It had the added advantage that it combined formality with good pragmatic features. After looking at denotational semantics in great detail, it was good to discover that there was another type of specification based on denotational semantics that was English-like (and hence readable) and comprehensible. Therefore,it was natural to become interested in this "new" type of semantic specification. We found that much effort was involved in understanding denotational semantics without possessing a detailed mathematical background and so our alternative became much more attractive.

We then started considering the theory underlying action notation and found that the structural operational semantics (SOS) had already been written down in [Mosses, 1992] (Appendix C). However, much effort was required to gain a thorough understanding of the SOS. Unfortunately, after gaining that knowledge, we found that some aspects of the representation were not

consistent and would require some alterations in the implementation. It was worth considering, at this point, whether it would be more advantageous to have an SOS which was more suitable for direct implementation. As this was not the case, these inconsistencies had to be identified and appropriate solutions given.

We decided to use the functional language CAML to implement the SOS. We found that CAML was indeed an excellent choice and proved to be a good introduction to the area of functional languages which then gave a good basis for the implementation in the Coq specification language. This implementation required some extra effort in comparison to the translation into CAML. Unfortunately, Coq does not have the possibilities required for the tracing of data, storage etc. when executing an action semantics specification. This is achieved with our CAML implementation. We are able to use the CAML implementation to trace transients, bindings and storage during the execution of an action semantics specification. After executing a program, statement etc., a state is given consisting of the type of state reached and the transient data, bindings and storage. We have, after much testing, concluded that the CAML representation is a true representation of the basic, functional, declarative and imperative facets of action notation. Our subsequent translation into Coq was based somewhat on the CAML representation but our indication that it was a true representation was not based on testing but on proving the truth of the action laws, as detailed below.

We then started looking again at the laws associated with action notation. We were now in a position to start writing proofs for these laws. The action notation was represented in the specification language of the Coq system and provided the foundation for these proofs. We began looking at the Coq theorem prover. This was probably the most difficult part of the research.

After considerable effort, the action laws outlined in Chapter 5 were eventually proved. Some proofs were more difficult and complicated than others, in particular the commutativity of the "or" action combinator and the proof that the combinator "and then" has the action primitive "complete" as a unit.

We should emphasise, at this point, that the learning curve required to complete this thesis was quite steep. Considerable effort was required to firstly, understand particular types of semantic specification. Secondly, as the implementation in CAML and Coq required that the structural operational semantics be fully understood, the effort needed for implementation was great. We know, at this point, that the structural operational semantics is a rather complex document. We also found that the Coq development system is an environment which is initially difficult to come to terms with.

6.2 Further Work

6.2.1 Additional Facets for Implementation

There are various paths to follow based on this research. Although, firstly we should note that some of the facets in the structural operational semantics have not been implemented i.e. the reflective and communicative facets see sections 4.3, 5.3. Therefore, some of the semantic entities were also not required i.e. commitments, processing. Obviously it would be very good if the above features were added in the future.

6.2.2 Parser/Translator for Action Notation

Also, using CAML and the Coq specification language, we have defined the abstract syntax of the kernel of action notation. Unfortunately, to interpret

actions using either representation, the action semantic specification must be translated into the appropriate abstract syntax depending on the representation used. This means that the action semantic specification must be translated to the kernel and then represented in the appropriate abstract syntax. This is rather a confusing and awkward process and therefore, it would be better to define a concrete syntax with rules for the transformation to the abstract syntax. Obviously, this would involve defining a translator for the conversion process from standard action notation to the kernel. In CAML, this could be achieved using its grammar facility. A similar facility is also available in Coq.

6.2.3 Proof of Semantic Equivalences

Based on our proofs of some of the action laws, it may be possible taking the action semantic specifications of a programming language to prove that semantic equivalence exists between constructs e.g. the semantic equivalence of a 'repeat' and 'while' loop. It may be possible, in the Coq environment, to prove the existence of semantic equivalences using the Coq proofs of the action laws. Therefore, the full set of constructs of a programming language could be reduced down to a core set. For example, to prove that in a toy language IMP, see [Watt, 1991]

C; **skip** $\equiv$ C

we must prove that both commands have the same denotation. A subset of the semantic equations of the language IMP appears in Figure 6.1.

(1) execute $[\![C_1 : \text{Command} \ ";" \ C_2 : \text{Command}]\!]$
= execute C_1 and then execute C_2

(2) execute $[\![$"skip"$]\!]$ = complete

Figure 6.1 Semantic equations of the language IMP

The proof is given in Figure 6.2. Note that within the proof we have made use of the Action law - *A* and then complete = *A*.

execute ⟦*C* ";" "skip"⟧
= execute *C* and then execute ⟦"skip"⟧ (by (1)
= execute *C* and then complete (2)
= execute *C* (action law)

Figure 6.2 Proof of "*C* ; **skip** ≡ *C*"

Another example looks as follows

while *E* **do** *C* ≡ **if** *E* **then begin** *C* ; **while** *E* **do** *C* **end else skip**

The relevant semantic equations are given in Figure 6.3. The denotations for both sides of the equations are given in Figure 6.4, 6.5. Compare the denotations in both to see that they are semantically equivalent. Based on the proofs of the various action laws already completed, we can see that it should be possible to prove, using Coq, that the above semantic equivalences exist e.g. in Figure 6.2, we should be able apply the proven theorem corresponding to - *A* and then complete = *A*.

(1) execute⟦"while" E:Expression "do" C:Command⟧ =
unfolding
evaluate E
then
check (the given value is true) and then
execute C and then unfold
or
check (the given value is false) and then complete

(2) execute⟦"if" E:Expression "then" C_1:Command "else" C_2:Command⟧ =
evaluate E
then
check (the given value is true) and then execute C_1
or
check (the given value is false) and then execute C_2

(3) execute⟦"begin" C:Command "end"⟧ = execute C

Figure 6.3 Some semantic equations of IMP

6.2.4 SOS suitable for Implementation

We have seen that the SOS required much adaption before it was suitable for implementation in CAML. It is possible that implementations of the SOS in different languages may have different associated problems. It could therefore be imagined that, as more implementations become available, feedback from the implementors could result in a restructuring of the SOS.

6.2.6 The End Product

It might be worth considering what remains to be achieved with reference to the work carried out to date to make the interpreting of action semantic

specifications as user-friendly as possible. We know that a parser/translator needs to be written. We then need to consider how the interface with the user should look. So, it would be nice if a thoroughly user friendly interface was constructed to help the user in the creation of action semantic specifications which could be interpreted giving the appropriate results.

```
execute⟦"while" E:Expression "do" C:Command⟧ =
        unfolding                                                    (by (1))
                evaluate E
          then
                  check (the given value is true) and then
                  execute C and then unfold
                or
                  check (the given value is false) and then complete

=               evaluate E                                           (action law)
          then
                  check (the given value is true) and then
                  execute C and then
                  unfolding
                        evaluate E
                  then
                        ..... or .....
                or
                  check (the given value is false) and then complete
=
                evaluate E                                           (by (1))
          then
                  check (the given value is true) and then
                  execute C and then
                  execute ‖ "while" E "do" C ‖
                or
                  check (the given value is false) and then complete
```

Figure 6.4 Denotation of left-hand command

execute⟦"if" E:Expression "then" "begin" C:Command ";"
"while" E:Expression "do" C:Command "end" "else" "skip"⟧

= evaluate E (by (2))
then
check (the given value is true) and then
execute ⟦"begin" C ";" "while" E "do" C "end"⟧
or
check (the given value is false) and then execute⟦"skip"⟧
=
evaluate E (by (3))
then
check (the given value is true) and then
execute ⟦C ";" "while" E "do" C⟧
or
check (the given value is false) and then complete
=
evaluate E (by Fig 6.1 (1))
then
check (the given value is true) and then
execute C and then execute ⟦"while" E "do" C⟧
or
check (the given value is false) and then complete

Figure 6.5 Denotation of right-hand command

BIBLIOGRAPHY

[Astesiano, 1991] E. Astesiano. Inductive and Operational Semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Report, pages 51-136. Springer-Verlag, 1991.

[Bertot, 1994] Yves Bertot. *User Guide to the Coq and Centaur Environment.* INRIA- Sophia-Antipolis, 1994.

[Bjørner & Jones, 1982] D. Bjørner and C. B. Jones, editors. *Formal Specification & Software Development.* Prentice-Hall, 1982.

[Brown et al., 1990] D. F. Brown, H. Moura and D. A. Watt. *Actress: an action semantics directed compiler generator*. In *CC'92, Proc. 4th Int. Conf. on Compiler Construction, Paderborn*, volume 641 of *Lecture Notes in Computer Science*, pages 95-109. Springer-Verlag, 1980.

[Casteréran, 1993] Pierre Castéran. Pro[gramm,v]ing (sic) with continuations: A development in Coq (Draft). URA CNRS 1304, Département d'Informatique, Université Bordeaux I, 33405 Talence CEDEX.

[Cornes et al., 1995] Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Gérard Huet, Pascal Manoury, Christine Paulin-Mohring, César Muñoz, Chetan Murthy, Catherine Parent, Amokrane Saï bi, Benjamin Werner. *The Coq Proof Assistant Reference Manual, Version 5.10.* INRIA-Rocquencourt - CNRS - ENS Lyon, 1995.

[Cousineau & Huet, 1989/1990] Guy Cousineau, Gérard Huet. The CAML Primer. INRIA-ENS, 1989/1990.

[Doh, 1993] Kyung-Goo Doh. *Action Semantics: A Tool for Developing Programming Languages.* Technical Report 93-1-005, the University of Aizu, 1993.

[Doh & Schmidt, 1993] Kyung-Goo Doh and David Schmidt. Action semantics-directed prototyping. *Comput. Lang.*, 19(4):213-233, 1993.

[Dowek et al., 1993] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, Benjamin Werner. *The Coq Proof Assistant User's Guide, Version 5.8.* INRIA-Rocquencourt - CNRS - ENS Lyon, 1991/1992/1993.

[Girard et al., 1989] Jean Yves Girard, Yves Lafont, Paul Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[Gunter, 1992] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, 1992.

[Hennessy, 1990] Matthew Hennessy. *The Semantics of Programming Languages - An Elementary Introduction using Structural Operational Semantics.* Wiley, 1990.

[Huet, 1994] Gérard Huet. Residual Theory in λ-calculus: A Complete Gallina Development. INRIA Rocquencourt, B.P. 105-78153 Le Chesnay CEDEX, France.

[Klint, 1991] P. Klint. A meta-environment for generating programming environments. In *Algebraic Methods II: Theory, Tools and Applications,* volume 490 of *Lecture Notes in Computer Science*, pages 105-124. Springer-Verlag, 1991.

[Kahn, 1991] G. Kahn. Natural Semantics. In *STAGS'87, Proc. Symp. on Theoretical Aspects of Computer Science,* number 247 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[Lassen, 1994] S. B. Lassen. Design and Semantics of Action Notation. In *Proceedings of the 1st International Workshop on Action Semantics,* BRICS Notes Series, NS-94-1, 1994.

[Lassen, 1995] Søren B. Lassen. Basic Action Theory. BRICS RS-95-25, May 1995.

[Lee, 1989] P. Lee. *Realistic Compiler Generation.* MIT Press, Cambridge, Massachusetts, 1989.

[Mauny, 1991] Michel Mauny. *Functional Programming using CAML.* INRIA Technical Report 129, May 1991.

[Meyer, 1991] Bertrand Meyer. *Introduction to the Theory of Programming Languages.* Prentice Hall, 1991.

[Michaelson, 1989] Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus.* Addison-Wesley, 1989.

[Mosses, 1980] Peter D. Mosses. A constructive approach to compiler correctness. In *Proc. Int. ICALP '80, Noordwijkerhout.* Springer LNCS 85, 1980.

[Mosses, 1983] Peter D. Mosses. Abstract semantic algebras! In Dines Bjørner, editor, *Formal Description of Programming Concepts II, Proc. IFIP TC2 Working Conference, Garmisch-Partenkirchen, 1982.* IFIP, North- Holland, 1983.

[Mosses, 1984] Peter Mosses. A Basic Abstract Semantic Algebra. In *Semantics of Data Types.* LNCS 173, Springer-Verlag, 1984.

[Mosses, 1990] P. D. Mosses. Denotational Semantics. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson and D. Perrin, editors, *Handbook of Theoretical Computer Science*, volume B, chapter 11. Elsevier Science Publishers, Amsterdam and MIT Press, 1990.

[Mosses, 1991] P. D. Mosses. A practical introduction to denotational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts,* IFIP State-of-the-Art Report, pages 1-49. Springer-Verlag, 1991.

[Mosses, 1992] Peter D. Mosses. *Action Semantics.* Cambridge University Press, 1992.

[Mosses, 1994] Peter D. Mosses. A Tutorial on Action Semantics. *Notes for FME '94.*

[Mosses & Watt, 1986] P. D. Mosses and D. A. Watt. Pascal action semantics. Available by anonymous FTP at ftp.daimi.aau.dk.

[Mosses & Watt, 1987] P. D. Mosses and D. A. Watt. The use of action semantics. In M. Wirsing, editor, *Formal Description of Programming Concepts III,* pages 135-163. North Holland, Amsterdam, Netherlands, 1987.

[Moura, 1992] H. Moura. An implementation of action semantics. Research report, Department of Computer Science, University of Glasgow, Scotland, 1992.

[Moura & Watt, 1994] H. Moura and D. Watt. Action transformations in the ACTRESS compiler generator. In *CC'94,* volume 786 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.

[Myers et al., 1993] Colin Myers, Chris Clack and Ellen Poon. *Programming with Standard ML.* Prentice Hall, 1993.

[Nielson & Nielson, 1992] Hanne Riis Nielson, Fleming Nielson. *Semantics with Applications - A Formal Introduction.* Wiley, 1992.

[Nielson & Toft, 1994] J. P. Nielsen and J. U. Toft. Formal specification of ANDF, existing subset. Technical Report 202104/RPT/19, issue 2, DDC International A/S, Lundtoftevej 1C, DK-2800, Lyngby, Denmark, 1994.

[Pagan, 1981] Pagan. *Formal Specifications of Programming Languages*. Prentice-Hall, 1981.

[Plotkin, 1981] G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN-19, Department of Computer Science, University of Aarhus, 1981. Currently available from the University of Edinburgh.

[Plotkin, 1983] G. D. Plotkin. A operational semantics for CSP. In D. Bjørner, editor, *Formal Description of Programming Concepts II, Proc. IFIP TC2 Working Conference, Garmisch-Partenkirchen, 1982.* IFIP, North- Holland, 1983.

[Schmidt, 1986] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

[Seaman & Felty, 1993] Jill Seaman, Amy Felty. *Proving Properties about a Lazy Functional Language with the Coq Development System.* Available by anonymous FTP at research.att.com.

[Tennent, 1991] R. D. Tennent. *Semantics of Programming Languages.* Prentice Hall, 1991.

[Turner, 1991]Raymond Turner. *Constructive Foundations for Functional Languages*. McGraw-Hill, 1991.

[Watt, 1986] D. A. Watt. Executable Semantic Descriptions. *Software - Practice and Experience,* 16(1):13-43, 1986.

[Watt, 1990] David A. Watt. *Programming Languages Concepts and Paradigms*. Prentice Hall, 1990.

[Watt, 1991] David A. Watt. *Programming Language Syntax and Semantics*. Prentice Hall, 1991.

Appendix A

In this Appendix, the steps involved in bringing the example of the assignment statement through the SOS are given. An overview of these steps is given in section 3.4. The steps are detailed as follows:

Step 1:

If we breakdown the translated version of the assignment statement into its constituent actions, we refer to action 1 as -

(give the cell yielded by
current bindings at the token yielded by "I"
and give 2)

and action 2 as -

(store the number yielded by component #2 of them
in the cell yielded by component #1 of them).

We identify the action infix as "then". We must assume, at this point, that the identifier "I" is bound to the cell 1. Therefore, the initial intermediate compound state looks as follows (according to the declarative behaviour of action infixes as dictated by action semantics):

(((action 1, *b*) "then" (action 2, *b*)), *l*)

where *b* consists of the binding of identifier "I" to cell 1. Also, *l* denoting storage is empty. Refer to [Mosses,1992] (Appendix C, section C.2.2) for an explanation of how states are formed.

Step 2:

Now that we have identified our inital intermediate state, our next step consists of applying the semantic function "run" to this state; refer to [Mosses, 1992] (Appendix C, C.3.3) . The "run" function tells us to apply the semantic function "stepped" and check if the resulting state is intermediate or terminated. Since, the acting component of the initial state is a compound acting, we apply "stepped" for compound actings, see [Mosses, 1992] (Appendix C, C.3.3.2.1 (5)).

Substituting appropriately, we arrive at the following:

stepped ((action 1, b),l) $\geq$ (A_1' :Acting, l' :local-info);

⟦(action 1, b) "then" (action 2, b)⟧ : ⟦Intermediate Sequencing Intermediate⟧⇒

stepped (⟦(action 1,b) "then"(action 2,b)⟧, l) $\geq$

(simplified ⟦ A_1' "then" (action 2,b)⟧, l').

Note that although the application of "stepped" can result in a sort of states, we shall only deal with a particular state. We can see, from above, that we must apply "stepped" to the state ((action 1, b),l).

Step 3:

We now apply "stepped" to ((action 1, b),l) i.e.

stepped (((give the cell yielded by

current bindings at the token yielded by "I"

and give 2), b) ,l)

We can see from above that this acting is also an intermediate compound acting which should look as follows:

stepped (((give the cell yielded by

current bindings at the token yielded by "I"),b)

"and"

((give 2), b), l)

Let us label the above two actions, action 1a and action 1b. Substituting accordingly into [Mosses, 1992] (Appendix C, C.3.3.2.1 (5)]:

stepped (((action 1a, b),l) $\geq$ (A_1'' , l'');

⟦(action 1a, b) and (action 1b, b)⟧:⟦Intermediate Interleaving Intermediate⟧⇒

stepped (⟦(action 1a, b) "and" (action 1b, b)⟧, l) $\geq$

(simplified ⟦ A_1'' "and" (action 1b, b)⟧, l'').

Step 4:

We now apply "stepped" to ((action 1a, b), l) i.e.

stepped ((⟦give the cell yielded by

current bindings at the token yielded by "I"⟧ , *b*), *l*).

Since the action above is indeed a primitive action, we apply [Mosses, 1992] (Appendix C, C.3.3.1.2 (1)). After evaluation of the yielders, we arrive at the following state:

("completed", 1, empty-map, *l*) where "completed" indicates the kind of the terminated state we have arrive at, 1 is the transient data, empty-map indicates empty bindings and *l* is unchanged.

Step 5:

We then substitute the result from step 4 into step 3 for the application of "simplified" i.e.

(simplified ⟦("completed", 1, empty-map) "and" (action 1b, *b*)⟧, *l*).

We then apply the function "simplified", see [Mosses, 1992] (Appendix C, C.3.3.2.2). Since no appropriate instance exists, we return the same acting, hence returning the same state.

Step 6:

We substitute the above result into step 2 i.e.

(simplified ⟦⟦"completed", 1, empty-map) "and" (action 1b, *b*)⟧

"then" (action 2, *b*)⟧, *l*).

We apply "simplified" which returns the same acting and hence the same state. We can see that the existing state is still intermediate and therefore, we must reapply the function "run", see [Mosses, 1992] (Appendix C, C.3.3 (1))

Step 7:

We reapply "run" which tells us to apply "stepped". We apply "stepped" for compound actings, see [Mosses, 1992] (Appendix C, C.3.3.2.1 (5)).

stepped ((⟦("completed",1,empty-map) "and" (action 1b,*b*)⟧), *l*) ≥

(A_1' :Acting, *l*' :local-info);

⟦⟦"completed", 1, empty-map) "and" (action 1b, *b*)⟧⟧) "then"

(action 2,b)]]:[[Intermediate Sequencing Intermediate]] $\Rightarrow$

stepped ([[("completed", 1, empty-map) "and" (action 1b, b)]]

"then" (action 2, b)]], I) $\geq$

simplified [[A_1' "then" (action 2, b)]], I').

Step 8:

We now "stepped" as follows:

stepped (([[("completed", "1", empty-map) "and" (action 1b, b)]]), I).

We see, from above, that the acting involved is an intermediate compound acting i.e. the acting (action 1b, b) is a completed acting. Therefore, we must apply "stepped" for compound actings, refer to [Mosses, 1992] (Appendix C, C.3.3.2.1 (6)). Substituting accordingly, we arrive at the following:

stepped ((action 1b, b), I) $\geq$ (A_2':Acting, I' :local-info);

[[("completed", 1, empty-map) "and" (action 1b, b)]] :

[[Completed Interleaving Intermediate]] $\Rightarrow$

stepped ([[("completed", 1, empty-map) "and" (action 1b, b)]], I) $\geq$

(simplified [[("completed", 1, empty-map) "and" A_2']], I').

Step 9:

So, we next apply "stepped" to

(give 2, b) i.e.

stepped (([[give 2]], b), I).

Since the above action is primitive, we apply [Mosses, 1992] (Appendix C, C.3.3.1.2 (1)). This results in the following state:

("completed", 2, empty-map, I).

Step 10:

We next substitute the above result into step 8 i.e. into the application of "simplified". This gives the following:

(simplified [[("completed", 1, empty-map) "and"

("completed", 2, empty-map) ⟧, *l*).

After applying "simplified" as in [Mosses, 1992] (Appendix C, C.3.3.2.2 (6)), we arrive at the following result:

(("completed", (1,2), empty-map), *l*).

Step 11:

We then substitute the result from step 10 into step 7 i.e.

(simplified ⟦("completed", (1,2), empty-map) "then" (action 2, b)⟧, *l*).

We apply "simplified" i.e. [Mosses, 1992] (Appendix C, C.3.3.2.2 (8)) giving the following result:

(⟦("completed", ( ), empty-map) "and" (action 2, (1,2), *b*)⟧, *l*).

The above result incorporates a call to the function "given" which passes the transient data i.e. (1,2) to (action 2, *b*).

Step 12:

We can see in step 11 that the result includes an intermediate acting and therefore, we must yet again apply the function "run". We reapply "run" which tells us to reapply "stepped" and check the result, see [Mosses, 1992] (Appendix C, C.3.3). Since we are looking at a compound acting, we apply "stepped" as in [Mosses, 1992] (Appendix C, C.3.3.2.1 (6)) substituting as follows:

stepped((action 2, (1,2), *b*), *l*) ≥ (A_2', *l*');

⟦("completed", ( ), empty-map) "and" (action 2, (1,2), *b*)⟧:

⟦Completed Interleaving Intermediate⟧ ⇒

stepped (⟦("completed", ( ), empty-map) "and" (action 2, (1,2), *b*)⟧, *l*) ≥

(simplified ⟦("completed", ( ), empty-map) "and" A_2'⟧, l').

Step 13:

We then apply "stepped" as dictated by step 12 as follows:

stepped(⟦store the number yielded by component #2 of them

in the cell yielded by component #1 of them, (1,2), b⟧, l).

Since the above action is primitive, we apply [Mosses, 1992] (Appendix C, C.3.3.1.4 (1)). We should note that yielder 1 -

the number yielded by component #2 of them

evaluates to 2 (number) and yielder 2 -

the cell yielded by component #1 of them

evaluates to 1 (cell) . We arrive at the following state:

("completed", ( ), empty-map, map 1 to 2).

The result above illustrates that we have altered the store i.e. we have mapped cell 1 to number 2.

Step 14:

We substitute the result in step 13 into step 12 i.e.

(simplified ⟦("completed", ( ), empty-map) "and"

("completed" , ( ), empty-map)⟧, map 1 to 2).

We apply "simplified", see [Mosses, 1992] (Appendix C, C.3.3.2.2 (6)) and arrive at the following state:

(⟨"completed", ( ), empty-map⟩, map 1 to 2).

We see that the resulting state is a completed state i.e. it is not intermediate, therefore, the above state is the final state as dictated by [Mosses, 1992] (Appendix C, C.3.3 (2)).

Appendix B

In this appendix, we illustrate the different structures of the semantic entities in the structural operational semantics, the implementations in CAML and Coq. We have chosen to show possible states and action infixes. The following tables endeavour to improve the comprehensibility when looking at the implementations in CAML and Coq. Note that the constructors' names in both implementations are identical e.g. "Inter", "Adb", "Cons_Action". Note also that we use a constructor "Stat_ok" in the Coq version to show a state consists of an Acting and Local Info.

Possible States:

SOS	CAML	Coq
("complete", *d, b, l*)	(Inter (Adb (Body (Cons_Action (complete)), inf)), l)	(Stat_ok (Inter (Adb (Body (Cons_Action complete)) inf)) l)
("escape", *d, b, l*)	(Inter (Adb (Body (Cons_Action (escape)), inf)), l)	(Stat_ok (Inter (Adb (Body (Cons_Action escape)) inf) l)
("fail", *d, b, l*)	(Inter (Adb (Body (Cons_Action (fail)), inf)), l)	(Stat_ok (Inter (Adb (Body (Cons_Action fail)) inf) l)
("unfold", *d, b, l*)	(Inter (Adb (Body (Cons_Action (unfold)), inf)), l)	(Stat_ok (Inter (Adb (Body (Cons_Action unfold)) inf) l)
([["give" *Y*]], *d, b,l*)	(Inter (Adb (Body (Simp_Pre_Action (give, Y)), inf)), l)	(Stat_ok (Inter (Adb (Body (Simp_Pre_Action (give Y))) inf)) l)
([["choose" *Y*]], *d, b, l*)	(Inter (Adb (Body (Simp_Pre_Action (choose, Y)), inf)), l)	(Stat_ok (Inter (Adb (Body (Simp_Pre_Action (choose Y))) inf)) l)
([["bind" *y1* "to" *y2*]],*d, b, l*)	(Inter (Adb (Body (To_Pre_Action (bind, y1, y2)), inf)), l)	(Stat_ok (Inter (Adb (Body (To_Pre_Action (bind y1 y2))) inf)) l)
([["unbind" *Y*]], *d, b, l*)	(Inter (Adb (Body (Simp_Pre_Action (unbind, Y)), inf)), l)	(Stat_ok (Inter (Adb (Body (Simp_Pre_Action (unbind Y))) inf)) l)
([["store" *y1* "in" *y2*]], *d, b, l*)	(Inter (Adb (Body (Store_Action (y1, y2)), inf)), l)	(Stat_ok (Inter (Adb (Body (Store_Action y1 y2))) inf)) l)
([["unstore" *Y*]], *d, b, l*)	(Inter (Adb (Body (Simp_Pre_Action (unstore, Y)), inf)), l)	(Stat_ok (Inter (Adb (Body (Simp_Pre_Action (unstore Y))) inf)) l)
([["reserve" *Y*]], *d, b, l*)	(Inter (Adb (Body (Simp_Pre_Action (reserve, Y)), inf)), l)	(Stat_ok (Inter (Adb (Body (Simp_Pre_Action (reserve Y))) inf)) l)
([["unreserve" *Y*]], *d, b, l*)	(Inter (Adb (Body (Simp_Pre_Action (unreserve, Y)), inf)), l)	(Stat_ok (Inter (Adb (Body (Simp_Pre_Action (unreserve Y))) inf)) l)
("completed", *d, b, l*)	(Stopped (Compl (d, b)), l)	(Stat_ok (Stopped (Compl d b)) l)
("failed", *l*)	(Stopped (failed), l)	(Stat_ok(Stopped failed) l)
("escaped", *d, l*)	(Stopped (Escape(d)), l)	(Stat_ok (Stopped (Escape d)) l)
([[*A1* "or" *A2*]], *l*)	(Inter (AIA (A1, inor, A2)), l)	(Stat_ok (Inter (AIA A1 inor A2)) l)
([["unfolding" *A*]], *d, b, l*)	(Inter (Adb (Pre_Action (unfolding, A), inf)), l)	(Stat_ok (Inter (Adb (Pre_Action unfolding A) inf)) l)

Action Infixes:

SOS	CAML	Coq
"and then"	and_then	and_then
"then"	inthen	inthen
"trap"	trap	trap
"and then moreover"	and_then_moreover	and_then_moreover
"then moreover"	then_moreover	then_moreover
"hence"	hence	hence
"thence"	thence	thence
"before"	before	before
"then before"	then_before	then_before
"and"	innand	innand
"moreover"	moreover	moreover
"or"	inor	inor