Equivalence of Formal Semantics Definition Methods*

M. J. A. Caswell

Dept. of Computer Studies, Loughborough University, Loughborough, Leicestershire, UK

Keywords: Programming language semantics; Denotational semantics; Structural operational semantics; Action semantics; Equivalence

Abstract. There are numerous methods of formally defining the semantics of computer languages. Each method has been designed to fulfil a different purpose. For example, some have been designed to make reasoning about languages as easy as possible; others have been designed to be accessible to a large audience and some have been designed to ease implementation of languages. Given two semantics definitions of a language written using two separate semantics definition methods, we must be able to show that the two are in fact equivalent. If we cannot do this then we either have an error in one of the semantics definitions, or more seriously we have a problem with the semantics definition methods themselves.

Three methods of defining the semantics of computer languages have been considered, i.e. Denotational Semantics, Structural Operational Semantics and Action Semantics. An equivalence between these three is shown for a specific example language by first defining its semantics using each of the three definition methods. The proof of the equivalence is then constructed by selecting pairs of the semantics definitions and showing that they define the same language.

1. Introduction

Formal semantics definition methods attempt to describe in a precise way the semantics (meaning) of computer languages. Informal definitions of the semantics of languages are inadequate because they are prone to ambiguity and omission.

Correspondence and offprint requests to: M. J. A. Caswell, Dept. of Computer Studies, Loughborough University, Loughborough, Leicestershire, UK. E-mail: M.J.A.Caswell@lboro.ac.uk

* A full version of this paper can be accessed via our web page <http://www.cs.man.ac.uk/fmethods/facj.html>

Formal semantics can be used in a number of ways. Language designers can use formal semantics as a tool in the design process. Implementors can use the semantics definition to ensure that their compiler/interpreter is standard and will produce code that will execute in the same manner as code produced on another manufacturer's compiler/interpreter. Users can use the semantics definition as an ultimate reference manual to a language, if they wish to find out how specific parts of a language will work in any given situation. Formal semantics also has the advantage that it can allow reasoning about programs written in the formally defined language.

There have been many methods suggested for formally defining semantics. Recently a new method has been suggested which is known as Action Semantics [Mos92]. For an introductory text to Action Semantics see [Wat91]. Two other methods that are used extensively are Denotational Semantics [Mis76] and Structural Operational Semantics [Plø81]. For introductory texts to these two methods see [Gor79] for Denotational Semantics, and [Hen90] for Structural Operational Semantics.

In 1974 Hoare and Lauer [HoL74], described a number of techniques for defining the semantics of languages, and then attempted to show for a particular language that a definition of the semantics written using one method could be shown to be equivalent to the same definition written using another method. This paper follows Hoare and Lauer in attempting to show equivalence across the modern notations of a definition of a particular language. Hoare and Lauer also proposed that complementary semantics definitions of a language should be provided, i.e. that separate definitions using different frameworks should be given for the same language. These definitions can then be used for different purposes, but would be relatable due to the fact that it could be proved that the definitions were equivalent. Mosses [Mos92, p. 5] states that one of the aims of Action Semantics is to avoid the need for such complementary descriptions. It is unclear at this stage whether this aim has been achieved successfully or not, although the experience of the author in producing this paper is that the theoretician will have difficulty working with Action Semantics.

This paper only examines the relationship of equivalence. There are of course other relationships between the semantics that may be of interest. For example relative abstractness, complexity, length, readability etc.

2. A Simple Language

The three semantic definition methods mentioned above have been designed with different purposes in mind. It is of course hoped that definitions written using one method will be equivalent to definitions written using a different method. However it would not be unreasonable to suspect that there may be differences given that the semantic definition methods were created for different purposes. It would, therefore, be reassuring if it could be proved that a description of a language written using one method is equivalent to a description of the same language written using a different method. If we cannot show equivalence then this would either indicate an error in one of the definitions, or more seriously a fault with a definition method.

Action Semantics is a new framework compared with Denotational or Structural Operational Semantics. Despite the fact that one of the stated objectives of Action Semantics is to avoid the need for complementary semantic descriptions,

we must still be able to relate work carried out using the other more established frameworks to it. This will help us to evaluate how well it has done in meeting its objectives, and also allow us to apply new techniques used in other frameworks to it.

This paper concerns itself with proving the equivalence of three different definitions of a specific language. This effectively shows us that for at least one language we can prove equivalence across the frameworks. Mosses, [Mos92], gives a complete Operational Semantics of Action Notation. It would also be possible to produce a complete Denotational Semantics of Action Notation. If this was done we could then generate a semantics in any of the other two frameworks given a semantics written in Action Notation. It can be seen that if we did this then we could say that for the subset of all languages that are expressible in Action Notation we can prove equivalence across the frameworks. Unfortunately, these generated semantics would not be in any way “natural” and would have a very artificial feel to them. In order to be able to prove equivalence for specific Action Semantic definitions and specific “natural” Denotational or Structural Operational Semantics definitions one could approach the problem in one of two ways. Firstly, one could prove the equivalence directly. Secondly one could generate an equivalent definition of the Action Semantics in each of the other frameworks, and then prove that the “natural” definitions are equivalent to the generated ones. This paper takes the former approach, due to the fact that until such time as a Denotational definition of Action Notation has been produced this is by far the simplest approach.

In order to illustrate the equivalence proofs a well known language that is already well defined could have been chosen. However, the proofs would be long and tedious, and would obscure the important point, i.e. that it is possible to present such proofs at all. Therefore a very simple language will be used. This language is essentially the same as the one used by Hoare and Lauer [HoL74], and consists of just four different types of statement. The syntax and semantics of expressions are not given or discussed here. The language consists of an assignment; an iterative construct; a sequencer and a block instruction. The abstract syntax is as follows.

$S, S_1, S_2 \in \text{Statement}$

$E \in \text{Expression}$

$B \in \text{Boolean}$

$Id \in \text{Identifier}$

**Statement = Identifier := Expression | Boolean * Statement
| Statement; Statement | (Statement)**

The semantic definitions of this language make no use of a “store” variable, but instead everything is stored in an “environment”. The result of a program is taken to be the final environment. Informally the semantics of this language are as follows.

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

2. **$B * S$** , means that the value of the Boolean, **B** , is evaluated in the current environment. If the value is false then this statement simply produces as a result

the unchanged current environment. If the value is true then the statement $S; B * S$ is executed, and the environment produced by these statements is given as a result.

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

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

Two auxiliary functions are required by the definitions. One, **write**, is used to make changes to the environment. The other, **the_value_of**, is overloaded to evaluate both **Boolean** and **Expression**.

Definition 1

$\text{write} :: \text{Environment} \times \text{Identifier} \times \text{Value} \rightarrow \text{Environment}$

$\text{write}(\text{env}, \text{Id}, v) =$

$\lambda x. x = \text{Id} \rightarrow v, \text{env}(\text{Id})$

where **env** is an **Environment** (see below); **Id** represents an **Identifier** and **v** represents a **Value**.

The **write** function takes three arguments. The first is the “old” environment; the second is an identifier name and the third is a value with which to associate the identifier. The result of the function is the new environment.

Definition 2

$\text{the_value_of} :: \text{Expression} \times \text{Environment} \rightarrow \text{Value}$

$\text{the_value_of} :: \text{Boolean} \times \text{Environment} \rightarrow \text{Value}$

No formal description of how **Expressions** and **Booleans** are evaluated is given here. The function **the_value_of** is common to all three of the semantic definitions, and so it is not required to know its exact operation. It is merely enough to know that the operation to evaluate **Expressions** and **Booleans** remains the same.

The Denotational Semantics definition of the language is given below. This definition defines **d_execute** which is used to evaluate statements. The domain of **Value** is left unspecified. The definition also requires the operator “*fix*” which constructs the *least fixed point* of its argument. A discussion of fixed points is beyond the scope of this paper. See [Win93] or [Gor79] for an introduction to fixed points.

$\text{Environment} : \text{Identifier} \rightarrow \text{Value}$

$\text{d_execute} : \text{Statement} \rightarrow \text{Environment} \rightarrow \text{Environment}$

D1) $\text{d_execute} \llbracket \text{Id} := E \rrbracket \text{env} =$
 $\text{write}(\text{env}, \text{Id}, \text{the_value_of}(E, \text{env}))$

D2) $\text{d_execute} \llbracket B * S \rrbracket =$
 $\text{fix}(\Gamma)$
 where
 $\Gamma(x) = \lambda \text{env}. \text{the_value_of}(B, \text{env}) = \text{true} \rightarrow$
 $x(\text{d_execute } S \text{ env}),$
 env

D3) $\text{d_execute} \llbracket S_1; S_2 \rrbracket \text{env} =$
 $\text{d_execute } S_2 (\text{d_execute } S_1 \text{ env})$

D4) $\text{d.execute } \llbracket (S) \rrbracket \text{env} =$
 $\text{d.execute } \llbracket S \rrbracket \text{env}$

The Structural Operational Semantics description of the language is given below. The rules given define a relation, $(S, \text{env}) \text{ gives } \text{env}'$. The intuitive idea behind this relation is that the statement S evaluated in the environment env gives the environment env' . Note that rules **S1** and **S4** have two expressions above the line. This should be read as both equating to true before the expression below the line can be inferred and vice versa. Rule **S3** has an expression by the side of the rule. This means that the rule is only valid in situations where this extra expression evaluates to true.

$$\begin{array}{lcl}
 \text{S1)} & \frac{\text{write}(\text{env}, Id, E_{\text{env}}) = \text{env}' \quad \text{the_value_of}(E, \text{env}) = E_{\text{env}}}{(Id := E, \text{env}) \text{ gives } \text{env}'} & \\
 \text{S2)} & \frac{\text{the_value_of}(B, \text{env}) = \text{false}}{(B * S, \text{env}) \text{ gives } \text{env}} & \\
 \text{S3)} & \frac{(B * S, \text{env}) \text{ gives } \text{env}'}{(S; B * S, \text{env}) \text{ gives } \text{env}'} & \text{the_value_of}(B, \text{env}) = \text{true} \\
 \text{S4)} & \frac{\frac{(S_1, \text{env}) \text{ gives } \text{env}' \quad (S_2, \text{env}') \text{ gives } \text{env}''}{(S_1; S_2, \text{env}) \text{ gives } \text{env}''}}{(S, \text{env}) \text{ gives } \text{env}'} & \\
 \text{S5)} & \frac{(S, \text{env}) \text{ gives } \text{env}'}{((S), \text{env}) \text{ gives } \text{env}'} &
 \end{array}$$

The Action Semantics definition is given below. The definition is in a sense incomplete in that the module specifying the abstract syntax is omitted. Also omitted is a description of the inter-module dependencies. The description that is presented is somewhat unusual, and not really in the spirit of Action Semantics, in that it uses the declarative facet to describe the imperative aspects of the language. This is because the language uses an environment to store values rather than the traditional method of a store. It would not be difficult to present a language that additionally made use of a store. The reason that this has not been done here is so that the language is as simple as possible. The presence of a store would not stop the production of equivalence proofs similar to those presented in this paper, it would only make them longer and more complicated.

introduces: `execute_`

- `execute_ :: Statement → action`

`execute` $\llbracket Id := E \rrbracket =$
 furthermore bind Id to the value of E .

`execute` $\llbracket B * S \rrbracket =$
 unfolding
 || check the value of B is true and then
 ||| execute S hence unfold

```

| or
|| check the value of  $B$  is false and then rebind.

execute  $\llbracket S_1; S_2 \rrbracket =$ 
  execute  $S_1$  hence execute  $S_2$ .

execute  $\llbracket (S) \rrbracket =$ 
  execute  $S$ .

```

Since Action Semantics requires no explicit variable to represent the environment, this aspect of the semantics is hidden. Action Semantics provides its own built-in operators for changing the environment, which means that the use of the function **write** is not required in this description. Additionally because there is no explicit variable for the environment we cannot use our function **the_value_of**. Instead we use a slightly amended version, **the value of**, which simply takes as an argument the Boolean or Expression to evaluate. The environment is handed implicitly to it.

3. Equivalence of the Definition Methods

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

Definition 3

$$\text{Def}_1 \equiv \text{Def}_2 \quad \text{iff} \quad \text{Def}_1(S, env) = env' \Leftrightarrow \text{Def}_2(S, env) = env'$$

where **Def₁** and **Def₂** are of type **Statement** $\times$ **Environment** $\rightarrow$ **Environment**, **S** is a **Statement** and **env, env'** are **Environments**. **Def₁** and **Def₂** may be partial functions, and are undefined in the case of non-termination.

That is, given two different definitions of a language, **Def₁** and **Def₂**, they are equivalent if and only if given a **Statement** and an **Environment**, they produce exactly the same **Environment** as a result. Similarly if a **Statement** diverges, then both **Def₁** and **Def₂** will be undefined. This is really just the same as extensional equality of functions.

In general the externally observable behaviour for a language is the output that is produced given a specific program and its input. In the case of the above example the input is the start environment, and the output is the new environment.

It would be possible to extend the definition of equivalence given above to other languages if we took the first element of the tuple accepted by **Def₁** and **Def₂** to be the program and the second element of the tuple to be the more general concept of input instead of an environment, and the result to be output.

3.1. Equivalence of Denotational and Structural Operational Semantics

A proof is now presented of the equivalence of the Denotational Semantics and the Structural Operational Semantics definitions of the language given in section 2. Similar proofs to this have been presented before, for example in [Win93]. The function name, **Den**, is used here to refer to the Denotational definition of the language, and the function name, **Op**, is used to refer to the structural operational definition of the language, i.e.,

Definition 4

$\text{Den}(S, \text{env}) = \text{env}' \Leftrightarrow \text{d_execute } S \text{ env} = \text{env}'$
and $\text{Op}(S, \text{env}) = \text{env}' \Leftrightarrow (S, \text{env}) \text{ gives } \text{env}'$

where **Den** and **Op** are of type **Statement** $\times$ **Environment** $\rightarrow$ **Environment**, **S** is a **Statement** and **env, env'** are **Environments**.

The theorem is as follows.

Theorem 1

$$\text{Den} \equiv \text{Op}$$

Proof. In order to complete the proof of this theorem we must consider all cases for the value of programs. Since we are using a very restricted language we only have four different types of statements to worry about. If we were to try to increase the size of this language the length of the proof given below would increase in proportion. Therefore the method of proof is to prove that the theorem is correct for all four possible values of statement, which gives rise to the four lemmas below.

Lemma 1 - Assignment

$$\text{Den}(\llbracket Id := E \rrbracket, \text{env}) = \text{env}' \Leftrightarrow \text{Op}(\llbracket Id := E \rrbracket, \text{env}) = \text{env}'$$

Lemma 2 - Block

$$\begin{aligned} & \text{Den}(\llbracket S \rrbracket, \text{env}_1) = \text{env}'_1 \Leftrightarrow \text{Op}(\llbracket S \rrbracket, \text{env}_1) = \text{env}'_1 \\ \Rightarrow & \text{Den}(\llbracket (S) \rrbracket, \text{env}) = \text{env}' \Leftrightarrow \text{Op}(\llbracket (S) \rrbracket, \text{env}) = \text{env}' \end{aligned}$$

Lemma 3 - Sequence

$$\begin{aligned} & \text{Den}(S_1, \text{env}_1) = \text{env}'_1 \Leftrightarrow \text{Op}(S_1, \text{env}_1) = \text{env}'_1 \\ \wedge & \text{Den}(S_2, \text{env}_2) = \text{env}'_2 \Leftrightarrow \text{Op}(S_2, \text{env}_2) = \text{env}'_2 \\ \Rightarrow & \text{Den}(\llbracket S_1 ; S_2 \rrbracket, \text{env}) = \text{env}' \Leftrightarrow \text{Op}(\llbracket S_1 ; S_2 \rrbracket, \text{env}) = \text{env}' \end{aligned}$$

Lemma 4 - Iteration

$$\begin{aligned} & \text{Den}(S, \text{env}_1) = \text{env}'_1 \Leftrightarrow \text{Op}(S, \text{env}_1) = \text{env}'_1 \\ \Rightarrow & \text{Den}(\llbracket B * S \rrbracket, \text{env}) = \text{env}' \Leftrightarrow \text{Op}(\llbracket B * S \rrbracket, \text{env}) = \text{env}' \end{aligned}$$

The theorem can then be proved directly from structural induction [Bur69] and lemmas 1-4. The proofs of these four lemmas are omitted here, since they are fairly straightforward. The main interesting case is lemma 4, which can be proved using mathematical induction on the number of times that a program loops. $\square$

3.2. Equivalence of Denotational and Action Semantics

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

Mosses presents a complete specification of a class of algebras within which Action Notation works. The remainder of this section makes extensive use of the definitions as laid out in [Mos92], and for the sake of brevity does not repeat them here. It is assumed here that the Denotational and Structural Operational Semantics given earlier in this paper also operate within these algebras so that we have some common basis with which to compare them. This should not be difficult to achieve. Mosses [Mos92, Appendix C] presents a complete operational semantics of Action Notation using this class of algebras. This effectively shows how operational definitions can be presented within this class of algebras. It is not quite so easy to write denotational definitions. In order to do this we must extend the algebras with the rules for lambda calculus. Suitable rules to do this are presented in [Bak84].

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

Definition 4

$\text{conv}(\text{write}(\text{env}, Id, v)) = \text{overlay}(\text{map of } Id \text{ to } v, \text{conv}(\text{env}))$
 and $\text{conv}(\lambda x. \perp) = \text{empty-map}$

where **env** is an **Environment**, **Id** is an **Identifier** and **v** is a **Value**. **overlay** and **map of to** are as defined in [Mos92, Appendix E].

As before we give the Action Semantics definition a name, **Act**. **Den** has the same meaning as in the previous section.

Definition 5

$\text{Act}(S, \text{env}) = \text{env}'$
 $\Leftrightarrow \text{run}(\text{given}(\text{received}(\llbracket \text{execute } S \rrbracket, \text{conv}(\text{env})), d : \text{data}), l : \text{local-info})$
 $\geq (\llbracket \text{"completed"} d' : \text{data conv}(\text{env}') \rrbracket, l' : \text{local-info}, c : \text{commitment})$

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

The theorem for the equivalence of Denotational Semantics and Action Semantics is then as follows,

Theorem 2

$\text{Den} \equiv \text{Act}$

Proof. Lemmas 5–8, as before, consider each of the four possible types of statement in turn.

Lemma 5 - Assignment

$$\text{Den}(\llbracket Id := E \rrbracket, env) = env' \Leftrightarrow \text{Act}(\llbracket Id := E \rrbracket, env) = env'$$

Lemma 6 - Block

$$\begin{aligned} \text{Den}(\llbracket S \rrbracket, env_1) = env'_1 &\Leftrightarrow \text{Act}(\llbracket S \rrbracket, env_1) = env'_1 \\ \Rightarrow \text{Den}(\llbracket (S) \rrbracket, env) = env' &\Leftrightarrow \text{Act}(\llbracket (S) \rrbracket, env) = env' \end{aligned}$$

Lemma 7 - Sequence

$$\begin{aligned} \text{Den}(S_1, env_1) = env'_1 &\Leftrightarrow \text{Act}(S_1, env_1) = env'_1 \\ \wedge \text{Den}(S_2, env_2) = env'_2 &\Leftrightarrow \text{Act}(S_2, env_2) = env'_2 \\ \Rightarrow \text{Den}(\llbracket S_1; S_2 \rrbracket, env) = env' &\Leftrightarrow \text{Act}(\llbracket S_1; S_2 \rrbracket, env) = env' \end{aligned}$$

Lemma 8 - Iteration

$$\begin{aligned} \text{Den}(S, env_1) = env'_1 &\Leftrightarrow \text{Act}(S, env_1) = env'_1 \\ \Rightarrow \text{Den}(\llbracket B * S \rrbracket, env) = env' &\Leftrightarrow \text{Act}(\llbracket B * S \rrbracket, env) = env' \end{aligned}$$

The theorem therefore follows directly from structural induction and lemmas 5–8. The proofs of these lemmas are omitted here. The main interesting case is lemma 8, which can be proved in a manner similar to lemma 4. $\square$

3.3. Equivalence of Structural Operational and Action Semantics

The third equivalence proof that is presented here is that between the Structural Operational Semantics and the Action Semantics. This is really just a trivial consequence of theorems 1 and 2, and uses the properties of transitivity and symmetry of equivalence, i.e., we wish to show,

Corollary to Theorems 1 and 2

$$\text{Op} \equiv \text{Act}$$

Proof. We know from theorem 1, $\text{Den} \equiv \text{Op}$, and we know from theorem 2, $\text{Den} \equiv \text{Act}$.

Hence from symmetry of $\equiv$ and theorem 1, $\text{Op} \equiv \text{Den}$. So from transitivity of $\equiv$, $\text{Op} \equiv \text{Act}$. $\square$

4. Conclusion

In this paper three methods of defining the semantics of computer languages have been examined, i.e., Denotational Semantics; Structural Operational Semantics and Action Semantics. It has been shown that for a specific example language, a description of the semantics of that language can be presented in each of the three description methods, such that the Denotational description of the languages can be shown to be equivalent to the Structural Operational Semantics. Additionally the Denotational Semantics can be shown to be equivalent to the Action Semantics, and the Structural Operational Semantics can be shown to be equivalent to the Action Semantics.

It has been seen that when proofs involving Action Semantics need to make explicit use of variables representing items such as the environment, reference to the Structural Operational Semantics of the Action Notation is required. This

makes proofs long and difficult. In the author's opinion however this will not be a serious impediment to the development of Action Semantics, due to the other advantages of the framework.

Acknowledgements

I would like to thank John Cooke for very helpful criticism of an earlier draft of this paper. I would also like to thank Roger Stone for many interesting discussions and comments.

References

- [Bak84] Baker-Finch, C. A.: Algebraic, Operational and Denotational Semantics of the Lambda Calculus. *The Australian Computer Journal*, 16(3), 96–101, 1984.
- [Bur69] Burstall, R. M.: Proving properties of programs by structural induction. *Computer Journal*, 12, 41–48, 1969.
- [Gor79] Gordon, M. J. C.: *The Denotational Description of Programming Languages: An Introduction*. Springer-Verlag, 1979.
- [Hen90] Hennessy, M.: *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics*. John Wiley & Sons, 1990.
- [HoL74] Hoare, C. A. R. and Lauer, P. E.: Consistent and Complementary Formal Theories of the Semantics of Programming Languages. *Acta Informatica*, 3, 135–153, 1974.
- [MiS76] Milne, R. and Strachey, C.: *A Theory of Programming Language Semantics*. Chapman and Hall, 1976.
- [Mos92] Mosses, P. D.: *Action Semantics*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1992.
- [Plo81] Plotkin, G. D.: *A Structural Approach to Operational Semantics*. Report DAIMI FN-19, Computer Science Department, Aarhus University, Ny Munkegade, Building 540, DK-8000, Aarhus C, Denmark, 1981.
- [Wat91] Watt, D. A.: *Programming Language Syntax and Semantics*. Prentice Hall, 1991.
- [Win93] Winskel, G.: *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, 1993.

Received September 1995

Accepted in revised form November 1996 by D. Simpson