Operational Semantics of an Imperative Language in Definite Clauses

Villadsen, Jørgen

Published in:
Proceedings of AGP-2003

Publication date:
2003

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Villadsen, J. (2003). Operational Semantics of an Imperative Language in Definite Clauses. In F. Buccafurri (Ed.), *Proceedings of AGP-2003* (pp. 337-349). DIMIT.

AGP-2003 (APPIA-GULP-PRODE Joint Conference on Declarative Programming, Reggio Calabria, Italy, 3-5 September 2003)

Operational Semantics of an Imperative Language in Definite Clauses

Jørgen Villadsen

Computer Science, Roskilde University

Building 42.1, DK-4000 Roskilde, Denmark

jv@ruc.dk

Abstract. We present the "big-step" operational semantics of a small programming language NIL (Natural Imperative Language) in definite clauses, thus building on the fixpoint semantics of logic programs. NIL operates on a state which is just a sequence of counters. As basic statements NIL has incrementation, decrementation and test for null. NIL allows for sequential composition and non-deterministic choice of statements as well as mutually recursive definitions of procedures, which we find support our long-term aim of formalizing and reasoning about specific actions and planning tasks for rational agents. A novelty is the use of the de Bruijn notation instead of names. To our knowledge the operational semantics of an imperative language like NIL have not been given in definite clauses, although it is well-known that it is possible.

The ISO Prolog source is available at `http://www.ruc.dk/~jv/nil.html`
Solutions obtained using GNU Prolog, see `http://gnu-prolog.inria.fr`

1 Introduction and Motivation

By an imperative language we have in mind a formal language in which changes to some kind of a state can be expressed. Our aim is to present the operational semantics [11] for NIL (Natural Imperative Language) directly in definite clauses, thus building on the fixpoint semantics of logic programs [2]. A first sketch of NIL was presented in the extended abstract [15].

In imperative languages the expressions that change the state are usually called statements, but in the present paper we call them actions since our long-term aim is to formalize and reason about specific actions and planning tasks for rational agents, cf. [15]. In computer science an agent is anything from a few lines of code automatically executed on certain primitive conditions to complex AI systems like robots. A rational agent bases its actions on a model of the world including both declarative and procedural knowledge, also called *know-that* and *know-how*, respectively. The know-how must describe how actions of agents change the world and we think that with respect to the planning task for rational agents both the operational semantics and the denotational semantics of NIL are worth investigating.

In the rest of this section we provide the motivation for NIL, which have a number of new features like the use of de Bruijn notation instead of names in action definitions [7]. In the following sections we describe the operational semantics of NIL using logic programs. In the final section we conclude and discuss related work, in particular the denotational semantics of NIL.

1.1 Idea

We have the following characteristics of NIL:

- A simple notion of a state.
- A simple formal semantics of the basic actions.
- A simple formal semantics of the compound actions.

These are in contrast both to the usual imperative programming languages (often based on the "while" statement for which we seem to need at least boolean expressions, e.g. comparisons) and also to the recursive functions, the untyped λ-calculus and other theoretical computers like the Turing machine.

We take as basic actions only incrementation, decrementation, and test for emptiness on counters. These actions are of course only meant as building blocks for more complicated actions.

We take as action combinations the usual sequential composition and also the non-deterministic choice construct. Since these correspond to relational composition and relational union in the denotational semantics we simply call these composition and union, respectively (note that skip and fail are the identity relation and the empty relation, respectively).

1.2 Examples

We can view the following axioms and rules as a logic program, which is pure Prolog without an ordering on subgoals and clauses. We discuss several small but non-trivial examples in details using an implementation in Prolog to compute the solutions (the relation between the input and output states).

For example, consider the action `skip` that does nothing at all. The axiom for the `skip` action is simply (in our notation which we shall define in details later):

```
act(skip,X-X).
```

Here `X-X` means that the state is unchanged, since the first `X` is the input state and the second `X` is the output state. We use italic for the predicate we are about to describe via axioms and rules, like *act*, and the actions themselves are in lowercase roman letters as opposed to the variables in upper-case roman letters. Of course the dash `-` in the axiom above is not the subtraction function, but merely a (pair) term constructor written infix.

Actions interact with the state, but not all actions change the state. One example of an action that does not change the state is the `skip` action already

mentioned. Another example is the `fail` action that for no input state has an output state. There are no axioms for the `fail` action since it never relates an input state to an output state.

As an opposite to the action `fail` we consider the somewhat strange action `miracle` that relates every input state to every output state. As we explain later it can be used as a kind of exception and the axiom for the `miracle` action is:

```
act(miracle,X-Y).
```

The actions `miracle`, `skip` and `fail` could be taken as basic, but we view them as special cases of action definitions and combinations as explained later.

1.3 The de Bruijn Notation

Instead of named definitions we use the so-called de Bruijn notation which is a coding of λ-terms where each occurrence of a bound variable is replaced by a natural number, indicating the structural distance from the occurrence to the abstraction that introduced the variable.

A few examples from [7] illustrate the notation:

1. The identity function $\lambda x.x$ becomes $\lambda 1$ in the de Bruijn notation. That is, the x of λx is removed, and the x of the body x is replaced by 1 to indicate the λ it refers to.
2. Similarly, $\lambda x.\lambda y.xy$ becomes $\lambda\lambda 21$. That is, the x and y of λx and λy are removed, and the x and y of the body xy are replaced by 2 and 1 respectively to indicate the λ's they refer to.
3. Finally, a few computations of distances show that $\lambda z.(\lambda y.y(\lambda x.x))(\lambda x.xz)$ becomes $\lambda(\lambda 1(\lambda 1))(\lambda 12))$.

However, in the following we compute the distance starting from 0 rather than as in the examples above from 1. We call the distance the de Bruijn index.

2 Logic Programs

In this section we explain the notation for the logic program [2] that we use to describe the operational semantics of NIL in the following sections.

2.1 Standard Notation

A term is either a variable X (in uppercase) or a compound term $f(t_1, \ldots, t_k)$ (in lowercase), where f is a functor and the t_i's are terms ($k \geq 0$). If $k = 0$ the compound term is just a constant and the parentheses are omitted.

A logic program is a finite set of definite clauses of the form:

$$p \leftarrow q_1, \ldots, q_n.$$

Here the p and the q_i's are non-variable terms ($n \geq 0$). If $n = 0$ the clause is an axiom (often called a fact in logic programming, in particular if it contains no variables), otherwise it is a rule. In a clause a functor of a term that is not part of another term is called a predicate.

A query is a clause of the form:

$$\leftarrow q_1, \ldots, q_m.$$

As for the logic programs the q_i's are non-variable terms ($m > 0$).

2.2 Pure Prolog

There are only two differences between logic programs and pure Prolog. The first is the trivial one that in Prolog `:-` is used instead of $\leftarrow$ in a rule (in a query `?-` is used). The second is that in Prolog the search for solutions is done applying the clauses *top-to-bottom left-to-right* corresponding to depth first traversal with backtracking while in logic programs all possible traversals are taken into account. The implications for NIL are illustrated in the examples below. We emphasize that the reference semantics is with logic programs — Prolog is just used in the following definite clauses and queries for illustrative purposes (hence in case of a non-deterministic choice there is no bias).

We use the usual list constructions with `[` and `]` (elements are separated by commas). The stroke `|` separates the head and the tail of a list. Lists are eliminable using right-nested terms with a binary functor `.` (the head to the left and the tail to the right) and the constant `[]` as the empty list.

Please observe the following convention. We use the infix operators `+` and `-` as binary functors with the convention that `-` is used when the two arguments are of the same type (e.g. both are states) and `+` if not. In both cases it is simply a pair construction. When arguments are e.g. just passed from predicate to predicate it is an advantage to use pairs instead of separate arguments.

We use the usual convention where any variable with only a single occurrence in an axiom or rule is written as `_` to indicate a so-called anonymous variable.

3 Overview of NIL

In this section we provide a brief overview of NIL.

3.1 States

Before we describe the actions expressible in NIL we need to elaborate on the notion of states:

- A state consists of counters referred to as $0, 1, 2, 3, \ldots$
- A counter has a value in $\{0, 1, 2, 3, \ldots\}$.

3.2 Basic Actions

We have the following basic actions:

- *Incrementation* `inc(counter)`
  Add 1 to a counter.
- *Decrementation* `dec(counter)`
  Subtract 1 from a counter or fail if counter = 0.
- *Test for emptiness* `emp(counter)`
  Fail if counter ≠ 0 (otherwise do nothing).

By using action definitions and combinations as explained below we can construct any other relevant action.

3.3 Action Combinations and Definitions

We have the following action combinations:

- *Composition* `com(action-list)`
  A sequence of actions.
- *Union* `uni(action-list)`
  Alternative actions.

These combinations can be seen as general problem-solving techniques (sequential composition and non-deterministic choice).

In e.g. [11] only binary action combinations are allowed, but then we are forced to settle on one of the following two readings for actions α, β and γ:

$$\alpha \ ; \ \beta \ ; \ \gamma \overset{?}{\rightsquigarrow} (\alpha \ ; \ \beta) \ ; \ \gamma \qquad \alpha \ ; \ \beta \ ; \ \gamma \overset{?}{\rightsquigarrow} \alpha \ ; \ (\beta \ ; \ \gamma)$$

We think that this is a rather artificial way of handling action combinations and therefore we allow simply a list of actions in action combinations.

We propose the following abbreviations:

$$\texttt{skip} \equiv \texttt{com([])} \qquad \texttt{fail} \equiv \texttt{uni([])}$$

In a composition all actions must be carried out; if there are none then nothing needs to be done. In a union one action must be carried out; if there are none then nothing can be done at all.

We also need action definitions:

- *Introduction of definitions* action-list
  Mutually recursive actions; perform first action.
- *Use of definition* `use(index)`
  Call the action referred to by the de Bruijn index.

Notice that we do not use a keyword for the introduction of definitions. Like for action combinations we also allow a list of actions in action definitions, and we propose the following abbreviation:

$$\texttt{miracle} \equiv \texttt{[]}$$

Hence if in an introduction of definitions no definitions are made then it is like hoping for a miracle.

4 Representation of NIL

In this section we explain how we represent states and actions in logic programs.

4.1 States and Actions

We represent a state as a list of counter values. The end of the list is always an anonymous variable; hence additional counters can always be accessed by extending the list and still leaving an anonymous variable at the end.

We represent a counter value as a list such that its length is the value of the counter. The list has arbitrary elements. For example, the list `[[],_,[_,_,_]|_]` represents a state where counter 0 has value 0, counter 1 has an arbitrary value, counter 2 has value 3 and the remaining counters all have arbitrary values. Notice the anonymous variable at the end.

Actions are represented using the constructions given in the previous section. We also represent a counter (not to be confused with the counter value just described) and the de Bruijn index as a list (again with arbitrary elements). For example, the list `[_,_,_]` represents counter 3 (or the de Bruijn index 3).

Using lists for the counters and the de Bruijn indices is a bit more complicated than using zero and successor constructions, but the benefit is that auxiliary predicates working on action-lists can be reused.

4.2 Example

We now consider the semantics of a "random number generator" that increments counter 0 an arbitrary number of times; i.e. either it does nothing or it increments counter 0 and recurses (using the de Bruijn index 0). The action is as follows (the action is an action-list, hence the introduction of definitions):

```
[uni( [skip, com( [inc([]),use([])] )] )]
```

The logic program and query below are not just of theoretical interest; for many actions the solutions can be computed automatically, for instance viewing the logic program and query as pure Prolog.

```
?- act( [uni( [com( [] ),
               com( [inc([]),use([])] )
              ] )
        ],W ).

W = A-A
W = [A|B]-[[_|A]|B]
W = [A|B]-[[_,_|A]|B]
W = [A|B]-[[_,_,_|A]|B]     ...
```

Here and in the following examples we recall that `skip` is an abbreviation for `com([])`. The variables `A` and `B` are automatically generated. We only show the first few solutions to the query (solutions are obtained as long as the user enters a semicolon at the prompt). We see that we have the following solutions:

- 0 increments — The input and output states are the same (both `A`).
- 1 increment of counter 0 — The value of counter 0 is `A` in the input state and is incremented in the output state (the remaining counters are not changed since `B` is used in both the input and output states).
- 2 increments of counter 0.
- 3 increments of counter 0. ...

As explained earlier the ordering of the solutions corresponds to the *top-to-bottom left-to-right* strategy for Prolog. No solutions will be produced in Prolog (loops until stack overflow) if the recursion is before the `skip` action:

```
?- act( [uni( [com( [inc([]),use([])] ),
                com( [] )
              ] )
        ],W ).
```

If in the main example above the recursion with the `use` action is replaced with the action `miracle` $\equiv$ `[]` we obtain exactly the following two solutions:

```
?- act( [uni( [com( [] ),
                com( [inc([]),[]] )
              ] )
        ],W ).

W = A-A
W = [_|_]-_
```

We immediately recognize the second solution as a "miracle" since the input and output states share absolutely no information. Furthermore, while we know nothing about the output state we do know that counter 0 has been accessed before the `miracle` action since the input state is `[_|_]` rather than just `_`.

It is also possible to perform actions after the "miracle" as the following minor modification of the previous query shows:

```
?- act( [uni( [com( [] ),
                com( [[],inc([])] )
              ] )
        ],W ).

W = A-A
W = _-[[_|_]|_]
```

Again we immediately recognize the second solution as a "miracle" since the input and output states share absolutely no information. Furthermore, while we know nothing about the input state we do know that counter 0 has been incremented after the `miracle` action since the output state is `[[_|_]|_]` rather than just `_`.

5 Operational Semantics of NIL

In this section we give the axioms and rules of the operational semantics of NIL using the above representation.

5.1 Act

The main issue is how to handle the action definitions using axioms and rules only. We use an auxiliary predicate *sub* that substitutes the actions definitions for all relevant indices in the first action and then we just use the predicate *act* on the result.

The *act* predicate takes an action and a pair of states, the input and output states respectively. If "empty" definition then "miracle": input and output states are not related. Otherwise we take the first action `H` in the definition and make a substitution in it; the resulting action `I` is then used. The auxiliary predicate *sub* must substitute the whole definition `[H|T]` for all indices that refer to the definitions just introduced. In order to do the substitution we need to take into account nested definitions and the second part of the pair `[H|T]+[]` accumulates the number of nested definitions using a list with a corresponding length (initially `[]` corresponding to no nested definitions).

```
act([],_-_).
act([H|T],W) :- sub(H,I,[H|T]+[]), act(I,W).
```

If "empty" composition then just skip: input and output states are the same. Otherwise we *act* on the first action `H` in the composition giving the output state `Y` and with `Y` as the input state we *act* on the remaining actions `T`.

```
act(com([]),X-X).
act(com([H|T]),X-Z) :- act(H,X-Y), act(com(T),Y-Z).
```

If "empty" union then fail (no clause at all). We have a clause where we *act* on the first action `H`, and a clause where we *act* on some of the other actions `T`.

```
act(uni([H|_]),W) :- act(H,W).
act(uni([_|T]),W) :- act(uni(T),W).
```

The first clause for incrementation considers just counter 0. Its input value is the first element `V` in the list representing the state and the length is increased in the output state with an arbitrary element (we use an anonymous variable

for this purpose). In the first clause for decrementation given below the roles of the input and output states are swapped. The remaining counters are not changed since `R` is used in both the input and output states. The second clause for incrementation (and similarly for decrementation and test for emptiness) considers all counters $\ell + 1$ except counter 0 ($\ell \geq 0$) and hence the value for counter 0 is the same for both the input and the output states, namely `V`. By considering input and output states `R-S` without counter 0 we then *act* with respect to incrementation (similarly for decrementation and test for emptiness) of the counter ℓ (kept in `L`).

```
act(inc([]),[V|R]-[[_|V]|R]).
act(inc([_|L]),[V|R]-[V|S]) :- act(inc(L),R-S).
```

As explained decrementation is similar to incrementation (the failure in case of trying to decrement a counter with value 0 is incorporated).

```
act(dec([]),[[_|V]|R]-[V|R]).
act(dec([_|L]),[V|R]-[V|S]) :- act(dec(L),R-S).
```

Also test for emptiness is similar and here the value for the counter in question is 0 (represented as `[]`) in both the input and output states.

```
act(emp([]),[[]|R]-[[]|R]).
act(emp([_|L]),[V|R]-[V|S]) :- act(emp(L),R-S).
```

We do not need to *act* on the `use` action since the action must be closed initially (the concepts of closed and open actions are taken over from λ-terms) and the substitution of action definitions ensures that the action stays closed.

5.2 Substitute

The auxiliary predicate *sub* maps actions to actions while carrying out the substitution given a pair consisting of the action definitions and the distance.

Nothing needs to be done for the "miracle" action (similarly for the skip, fail, incrementation, decrementation and test for emptiness actions below).

In case of a new definition the distance `N` has to be adjusted — the number of new definitions must be added to the distance. The auxiliary predicate *sep* takes 3 arguments and uses the length (but not the elements) of the list given as the first argument to separate the second and third arguments, using anonymous variables. The axioms and rules for the *sep* predicate are given later. The predicate is used several times below with either the second or third argument as the result argument.

```
sub([],[],_).
sub([H|T],[I|U],M+N) :- sep([H|T],N,J),
  sub(H,I,M+J), sub(T,U,M+J).
```

Substitution in compositions and unions is just substitution in the head and in the tail of the list of actions.

```
sub(com([]),com([]),_).
sub(com([H|T]),com([I|U]),E) :- sub(H,I,E),
  sub(com(T),com(U),E).

sub(uni([]),uni([]),_).
sub(uni([H|T]),uni([I|U]),E) :- sub(H,I,E),
  sub(uni(T),uni(U),E).

sub(inc(L),inc(L),_).
sub(dec(L),dec(L),_).
sub(emp(L),emp(L),_).
```

The real work is done for the `use` action. However, the first clause does nothing if the index is less than the distance. The comparison uses the *sep* predicate to check if a non-empty list `[_|_]` separates the index and the distance.

```
sub(use(F),use(F),_+N) :- sep(F,[_|_],N).
sub(use(F),C,[H|T]+N) :- sep(N,G,F),
  rep(G,H,[H|T],[_|U],I), sep([H|T],[],J),
  ren([I|U],A,[]-J), ren(A,B,G-[]), ren(B,C,J-G).
```

The second clause subtracts the distance from the index (using the *sep* predicate) to get the position `G` in the action definitions `[H|T]`. Then the action `I` at position `G` is replaced with the action `H` at position 0 (i.e. `[]`) using the predicate *rep* given later, so instead of `[H|T]` we now have `[I|U]`. Finally we must swap the indices for the actions at the two positions; we simply do a triple renaming by the predicate *ren* using a fresh index `J` obtained as the number of action definitions `[H|T]`. The addition is done using the *sep* predicate. Temporary results are kept in `A` and `B`, and the final result is `C`.

5.3 Rename

The auxiliary predicate *ren* maps actions to actions while carrying out the renaming of indices given as a pair. In case of a definition we must shift both indices by the length of the list of actions, hence `F` becomes `P` and `G` becomes `Q`, and the renaming of the first action `H` is straightforward. For the remaining actions `T` we use a small trick: we add 1 to the indices and use the predicate *ren* again (hence with `[_|F]` instead of `F` and `[_|G]` instead of `G`).

```
ren([],[],_).
ren([H|T],[I|U],F-G) :- sep([H|T],F,P),
  sep([H|T],G,Q), ren(H,I,P-Q),
  ren(T,U,[_|F]-[_|G]).
```

Renaming in compositions and unions is just renaming in the head and in the tail of the list of actions. Nothing needs to be done for incrementation, decrementation and test for emptiness actions.

```
ren(com([]),com([]),_).
ren(com([H|T]),com([I|U]),K) :- ren(H,I,K),
  ren(com(T),com(U),K).

ren(uni([]),uni([]),_).
ren(uni([H|T]),uni([I|U]),K) :- ren(H,I,K),
  ren(uni(T),uni(U),K).

ren(inc(L),inc(L),_).
ren(dec(L),dec(L),_).
ren(emp(L),emp(L),_).
```

For the matching indices we just do the renaming. We do nothing if the indices do not match.

```
ren(use(F),use(G),F-G).
ren(use(F),use(F),G-_) :- dif(F,G).
```

The check for different indices is done by the predicate *dif* given later.

5.4 Separate

The auxiliary predicate *sep* takes 3 arguments and uses the length (but not the elements) of the list given as the first argument to separate the second and third arguments, using anonymous variables (it is like the usual list append, except that all elements are anonymous variables).

```
sep([],N,N).
sep([_|T],N,[_|J]) :- sep(T,N,J).
```

5.5 Replace

The auxiliary predicate *rep* takes 5 arguments and uses the length of the list given as the first argument as the position in the list given as the third argument where to do the replacement. The fourth argument is the replacement result. The second argument is inserted and the removed element is in the fifth argument (the elements of the list given as the first argument are not used, since we use an anonymous variable in the second clause).

```
rep([],I,[H|T],[I|T],H).
rep([_|F],I,[H|T],[H|U],J) :- rep(F,I,T,U,J).
```

5.6 Differ

The auxiliary predicate *dif* simply checks for different indices (non-empty lists differ from empty lists and also if they differ with respect to their tails).

```
dif([],[_|_]).
dif([_|_],[]).
dif([_|F],[_|G]) :- dif(F,G).
```

6 Conclusions and Related Work

To our knowledge the operational semantics of imperative languages like NIL has not been given in definite clauses, although it is well-known that it is possible. We hope to have shown that it can be done in an elegant way using the approach described here, in particular using the de Bruijn notation and using substitutions instead of environments. The notion of environment would seem to be considerably less natural for our long-term aim: to formalize and reason about specific actions and planning tasks for rational agents, cf. [15], and we believe that the simplicity of the operational semantics of NIL is an important advantage here — in particular when we at the same time allow mutually recursive definitions of procedures and non-deterministic choices in order to "scaling-up" programming in NIL. Related works on operational semantics, using e.g. higher order logic programming or logical frameworks, have a different focus [5,4,6,10], but are of course more general.

Notice that it is not our purpose as such to enhance an imperative language in order to support declarative programming, cf. Alma-O [3] (except with non-deterministic choices).

Many important aspects of the operational semantics of various imperative languages with "while"-statements have been considered in relation to both logic programs [14] and constraint logic programs [13]).

In future work we plan to describe how the design of NIL makes it possible to give it a concise declarative semantics equivalent to its operational semantics (with the integrated possibility of making correctness assertions). We consider the operational semantics of NIL in definite clauses as the reference semantics (using logic program and not the specific Prolog search). In operational semantics we specify exactly how actions change the state. In order to validate that the reference semantics is "right" we have also investigated the denotational semantics of NIL. In denotational semantics we specify only the effect of the change — not how it is obtained — using various mathematical objects [11]. As emphasized by [9] the usual use of reflexive domains seems to be an unnecessary complication in connection with the denotational semantics of imperative languages like NIL. Instead we translate the actions of NIL into classical higher order logic [1] (type theory) and since the logic comes with a model-theoretic interpretation, we get an interpretation for NIL too. An advantage of such a translation into higher order logic is that partial and total correctness properties can be asserted directly

in the logic; the result is known as a Hoare logic [11,12]. We hope to show that because of the simplicity of the semantics of the imperative language NIL it is very well suited for the recent approach to program semantics in classical higher order logic [9]. A similar approach using a translation into higher order logic has also been used to formalize the dynamics of natural language processing [8].

Acknowledgements

Thanks to John Gallagher and the anonymous reviewers for useful comments. This research was partly sponsored by the IT-University of Copenhagen.

References

1. P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory*. Academic Press, 1986. Rev. ed. Kluwer Academic Publishers, 2002.
2. K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B*, pages 493–574. Elsevier, 1990.
3. K. R. Apt, J. Brunekreef, A. Schaerf, and V. Partington. Alma-0: An imperative language that supports declarative programming. *ACM TOPLAS*, 20(5):1014–1066, 1998.
4. T. Despeyroux. Typol: A formalism to implement natural semantics. INRIA Research Report 94, 1988.
5. J. Hannan and D. Miller. From operational semantics to abstract machines. *Journal of Mathematical Structures in Computer Science*, 2(4):415–459, 1992.
6. G. Kahn. Natural semantics. In *Symposium on Theoretical Aspects of Computer Science*. Springer-Verlag, 1987. LNCS 247.
7. F. Kamareddine. Reviewing the classical and the de Bruijn notation for λ-calculus and pure type systems. *Journal of Logic and Computation*, 11(3):363–394, 2001.
8. R. Muskens. Anaphora and the logic of change. In J. van Eijck, editor, *Logics in AI, European Workshop JELIA '90, Amsterdam*, pages 412–427. Springer-Verlag, 1991. LNCS 478.
9. R. Muskens. Program semantics and classical logic. 27 pages. CLAUS-Report 86, 1997. `http://www.coli.uni-sb.de/claus`
10. G. Nadathur. The metalanguage λProlog and its implementation. In *International Symposium on Functional and Logic Programming*, pages 1–20. Springer-Verlag, 2001. LNCS 2024.
11. H. R. Nielson and F. Nielson. *Semantics with Applications*. John Wiley & Sons, 1992. Rev. ed. 1999.
12. T. Nipkow. Hoare logics in Isabelle/HOL. In *Proof and System-Reliability*. Kluwer Academic Publishers, 2002.
13. J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of imperative programs through analysis of constraint logic programs. In G. Levi, editor, *International Symposium on Static Analysis*, pages 246–261. Springer-Verlag, 1998. LNCS 1503.
14. B. J. Ross. The partial evaluation of imperative programs using Prolog. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming*, pages 341–363. MIT Press, 1989.
15. J. Villadsen. On programs in rational agents. In M. R. Hansen, editor, *Nordic Workshop on Programming Theory*, page 8, 2001. IMM-TR-2001-12.