On logic programming and the refinement calculus: semantics based program transformations

Joost N. Kok

RUU-CS-90-39
December 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

On logic programming and the refinement calculus: semantics based program transformations

Joost N. Kok

Technical Report RUU-CS-90-39
December 1990

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN:0924-3275

On Logic Programming and the Refinement Calculus: Semantics Based Program Transformations

Joost N. Kok

Utrecht University*

Abstract

This paper tries to bridge the gap between logic programming and a refinement calculus for imperative programs. In order to make an embedding of logic programming possible, we have to extend the refinement calculus in several ways. The result is a notion of stepwise refinement for logic programs and the advantages of the refinement calculus become available for logic programming. For example, we can justify program transformation rules for logic programming in the calculus.

1 Introduction

The weakest precondition calculus as proposed by Dijkstra in [Dij76] associates with statements in an imperative language a predicate transformer. Later the approach was extended by Back (for an overview see [BvW90, Bac90]) for specifications. A more general class of statements can be handled and a notion of refinement based on weakest preconditions is introduced. This notion of refinement amounts to the following: we say that statement S_1 is refined by statement S_2 (denoted

*Department of Computer Science, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, the Netherlands. Email: joost@cs.ruu.nl. Part of this work was carried out in the context of ESPRIT Basic Research Action (3020) Integration.

by $S_1 \sqsubseteq S_2$) if for all predicates P, Q

$$P\langle S_1\rangle Q \Rightarrow P\langle S_2\rangle Q$$

where $P\langle S\rangle Q$ denotes total correctness: that is, if execution of S starting in a state σ that satisfies P then this execution is guaranteed to terminate, and moreover, to terminate in a state satisfying Q. There is no division between statements and specifications: they both are treated in the same way. Program development consists of refinements that transform specifications to statements. In Back's refinement calculus (BRC) we find unbounded nondeterminism, angelic nondeterminism and miracles. This kind of constructs is hard (or even impossible) to implement, but they are useful in specifications, i.e. in the initial stages of the program design. Other groups of researchers use similar generalized specification languages, for example in [MRG88, Mor87, Nel87].

The purpose of BRC is total correctness, that is, develop programs that deliver certain results and terminate. There is a strong mathematical foundation: a lattice theoretic framework and it is suited for program development of imperative programs with asynchronous communication via shared variables. Around BRC a lot of work has been done, including work on

1. Laws of Programming: a large set of rules that can be used for refinements are given in [Ser90a].

2. Transformation via Action Systems to Occam: the theory provides a path towards a distributed implementation. First a general statement (possibly containing specification statements) is transformed in several steps to an action system. An action system is a generalized while-loop. There are at least two basic ways to go from an action system to a distributed implementation:

   (a) Shared action model: distribute the variables over processors

   (b) Shared memory model: distribute actions (components of an actions system) over processors

   Implementations are more distributed if in the first case actions only refer to variables on a few processors, and in the second case if actions have only a few shared variables. Again the refine-

ment notion is used to carry out transformations on action systems to action systems that are more distributed. There has been singled out a subset of action systems (called Occam-like action systems) that are well suited for implementation on a multi-transputersystem. One of the refinement strategies is to try to refine to these Occam-like action systems ([BS90, Ser90b]).

3. Data Refinement: based on the notion of refinement for statements, there is also a theory for the refinement of data. For this a notion of inverse statements is studied. Changing from one datarepresentation to another can be seen as a refinement if the sequential composition of the decoder, a new statement and the inverse of the decoder is a refinement of the original statement ([Bac89]).

4. Mechanization in Higher Order Logic. The HOL system is a proofassistent for doing proofs in Higher Order Logic. A first attempt has been made to embed the refinementcalculus into this system in [vW90].

5. Reactive Programs, Action Refinement. Although BRC was originally designed for program development of terminating programs, there are now extensions for reactive programs. Also action refinement (splitting of atomic actions in smaller parts) is possible. For more details consult [Bac90].

The mixture of specification and implementation in BRC calls for a different kind of intuiition about programs. Execution of a program is seen as a game between a demon and an angel. In a program there are several choices to be made and each choice is assigned either to the demon or to the angel. Given a predicate Q, the angel tries to win the execution game by trying to terminate in a state satisfying Q. The demon tries to prevent this and the angel wins if it has a winning strategy. Angelic choices are hard to implement, because in principle it needs to know the behavior of the rest of the program. This is one of the reasons that the semantics is given by a continuation semantics (predicate transformers).

The main goal of this paper is to extend BRC to cover also Logic Programming and thus providing LP with a framework for studying termination, transformation rules, paralization and

program development. For this we need to extend BRC with a notion of deadlock. Classically, we associate with each statement S with a predicate transformer this has the following meaning: the predicate $S(Q)$ characterizes all states σ such that, when execution of S starts in σ, we are guaranteed to terminate, and moreover, in a state satisfying Q. The termination of a logic program depends in general also on the deadlock behavior of the components. For example, in the case of Prolog when we deadlock, we start backtracking. Therefore, it may be not surprising that if we want to define predicates $S(Q)$ we also need some information about the deadlock behavior of the components. Therefore we introduce a second kind of predicate transformers $S[Q]$: all states σ such that, when execution starts in σ, we are guaranteed either to terminate or to deadlock, and moreover, when we terminate, we do this in a state satisfying Q.

Also the intuition of BRC (game between angel and demon) has to be extended for Logic Programming. Again, we need a generalization to cope with deadlock. The angel first tries to terminate in the right state, but it if this is not possible, it tries to deadlock (and avoid termination in a wrong state and avoid divergence).

With these extensions, it is possible to embed logic programming in BRC. When we do this, we can some nice correspondences between the LP world and BRC:

1. unification can be seen as an update to the variables such that the equality of terms is satisfied. In general, there are more possibilities to do this. If we assign this choice to the angel, it makes such a choice that the program will terminate. The angel can not do better than taking the most general unifier. Hence we have the correspondence: unification = updates, most general unification = angelic updates.

2. for logic programming there exist several kinds of models. For example, there are declarative semantics (emphasizing to the logic component) and operational semantics (emphasizing flow of control). In the proposed framework we see that the embedding of Horn Clause Logic with fair selection rule (roughly corresponding to the declarative semantics) makes much more use of specification like constructs (like angelic choice) than the embedding of Horn Clause Logic with left-most selection ("Pure Prolog").

3. there is a correspondence between angelic choice and backtracking.

4. In concurrent logic languages it is too expensive to implement backtracking. Therefore committed choice is introduced. We will show that committed choice corresponds to a generalization of the guarded commands of Dijkstra.

More important is that we have given Logic Programming a new kind of semantics (axiomatic semantics) and that LP can be embedded in a broader framework, in which we can give study combinations of different paradigms (like backtracking, committed choice). In this way we could extend LP in a different direction, for example by adding specification constructs, miracles and theories for data refinement. Moreover, this extension seems to be orthogonal to the extension with constraints and a combination of the two extensions seems to be promising.

Also in this framework we can justify transformations on logic programs like the ones suggested in [Kom89], and the work on termination of [AP90] and [Dev90].

2 Refinement Calculus

In this section we give a short introduction to the refinement calculus and discuss some extensions that are needed in order to embed logic programs into this calculus.

The refinement calculus is based on a command language. The semantics of statements in this language is given by predicate transformers. It is even possible to identify statements with their meaning, i.e. do not distinguish between statements and their predicate transformers. We will also follow this approach. We will follow the notation of [BvW90] in which also an overview of the theory is given. For our purposes, we can take over the following constructs: Let v be a vector of variables and P be a predicate. Then statements are given by

$$S \ ::= P \mid \langle \forall v.P \rangle \mid S_1; S_2 \mid S_1 \vee S_2 \mid S_1 \wedge S_2 \mid |[con\ v; S]|$$

We discuss the operatorts briefly:

- P is a predicate and if execution reaches this predicate then it is checked if this predicate is true. If so, then it behaves like a skip statement, and otherwise it deadlocks.

- $\langle \vee v.P \rangle$ is an angelic update (cf. the intuition about the angel and demon in the introduction. It assigns values to the variables v in such a way that P becomes true. Which values are assigned is upto the angel (whose goal it is to terminate). If it is not possible to find such values, the statement deadlocks.

- $S_1; S_2$ the sequential composition of S_1 and S_2.

- $S_1 \vee S_2$ choice between S_1 and S_2 to be made by the angel.

- $S_1 \wedge S_2$ choice between S_1 and S_2 to be made by the demon.

- $|[con\ v; S]|$ block construct with local variables v.

In order to embed Logic Programs we add the following construct:

$$S ::= \cdots \mid S_1 \square S_2$$

The intention is to model the choice operator of Prolog. Such an operator is also considered in [dB88]. First statement S_1 is tried and only if this statement S_1 deadlocks then statement S_2 is considered.

We turn to the meaning of statements S. The semantic domains are:

Booleans:

$$\mathbf{B} \triangleq \{ff, tt\}$$

Vectors of variables:

$$v \in Var^*$$

Values of variables:

$$\alpha \in Value^*$$

States:

$$\sigma \in \Sigma = Var \rightarrow Value$$

Expressions:

$$Exp = \Sigma \rightarrow Value$$

$$t \in Exp^*$$

Predicates

$$P, Q \in Pred \triangleq \Sigma \rightarrow \mathbf{B}$$

Predicate Transformers:

$$PTran = Pred \rightarrow Pred$$

We also need a variant notation for predicates

$$P[t/v] \triangleq \lambda\sigma.P(\sigma\{t(\sigma)/v\})$$

where $\sigma\{\alpha/v\}$ is a state like σ except for the values of v which are defined to be α.

We take the following orderings on $\mathbf{B}$ and $Pred$:

$$ff \sqsubseteq tt$$

$$P \sqsubseteq Q \text{ if } \forall\sigma \in \Sigma.P(\sigma) \sqsubseteq Q(\sigma)$$

Next we associate with each statement a predicate transformer.

$$P(Q) \triangleq P \wedge Q$$

$$\langle \vee v.P \rangle(Q) \triangleq \exists v.(P \wedge Q)$$

$$S_1; S_2(Q) \triangleq S_1(S_2(Q))$$

$$S_1 \vee S_2(Q) \triangleq S_1(Q) \vee S_2(Q)$$

$$S_1 \wedge S_2(Q) \triangleq S_1(Q) \wedge S_2(Q)$$

$$|[con\ v; S]|(Q) \triangleq \exists v.S(Q)$$

$$S_1 \Box S_2(Q) \triangleq S_1(Q) \vee (S_1[Q] \wedge S_2(Q))$$

Here we have introduced a new kind of predicate transformer (denoted by square brackets). The intended meaning of $S[Q]$ is as follows: if a state σ satisfies $S[Q]$ then execution of S starting in state σ is guaranteed either to deadlock or to terminate. In case of termination the final state σ satisfies Q. Note that $S(Q) \Rightarrow S[Q]$ for any S and Q.

$$P[Q] \triangleq P \Rightarrow Q$$

$$(\vee v.P)[Q] \triangleq (\exists v.P) \Rightarrow (\exists v.P \wedge Q)$$

$$S_1; S_2[Q] \triangleq S_1[S_2[Q]]$$

$$|[con\ v; S]|[Q] \triangleq \exists v.S[Q]$$

$$S_1 \vee S_2[Q] \triangleq S_1(Q) \vee S_2(Q) \vee (\neg S_1(Q) \wedge \neg S_2(Q) \wedge (S_1[Q] \vee S_2[Q]))$$

$$S_1 \wedge S_2[Q] \triangleq S_1[Q] \wedge S_2[Q]$$

$$S_1 \square S_2[Q] \triangleq S_1(Q) \vee (S_1[Q] \wedge S_2[Q])$$

In this semantics we do not take care of the renaming of variables. If clashes of variables appear (for example in the block construct) we should do a renaming. A cleaner way to do this is to follow the approach of [BvW90]: use typing of statements and create variables only when they are necessary. The semantics given by predicate transformers is a continuation semantics: given that we know how the rest of the program behaves, we can tell the behavior of the program.

There is some freedom in the definition of the angelic choice $\vee$. The interpretation above works in the game-theoretic interpretation as follows: The angel tries to make choices in such a way that it wins (i.e. terminate in Q). If this is not possible, the angel tries to deadlock (and avoids divergence or termination in a state satisfying $\neg Q$). An alternative would be to define:

$$S_1 \vee S_2[Q] \triangleq S_1[Q] \vee S_2[Q]$$

This gives rise to a different interpretation: There are now to different games: game 1 is the old termination in state satisfying Q game and game number 2: the angel wins if the statement terminates in Q or deadlocks (i.e. no preference for termination).

Note that we now have two refinement relations on statements: statement S_1 is refined by statement S_2 if for all Q we have $S_1(Q) \Rightarrow S_2(Q)$ (first notion) or $S_1[Q] \Rightarrow S_2[Q]$ (second notion). The first notion we denote by $S_1 \sqsubseteq S_2$ and the second notion by $S_1 \ni S_2$. The next theorem summarizes when we can do refinements in a context. It is important in program development to be able to do refinements in a context.

Theorem 2.1 *For any statements S_1, S_1', S_2, S_2' we have that*

- *if $S_1 \sqsubseteq S_1'$ and $S_2 \sqsubseteq S_2'$ then $S_1 * S_2 \sqsubseteq S_1' * S_2'$ for $* \in \{\vee, \wedge, ;\}$.*
- *if $S_1 \ni S_1'$ and $S_2 \ni S_2'$ then $S_1 * S_2 \ni S_1' * S_2'$ for $* \in \{\vee, \wedge, ;\}$.*
- *if $S_1 \sqsubseteq S_1'$ and $S_1 \ni S_1'$ and $S_2 \sqsubseteq S_2'$ then $S_1 \Box S_2 \sqsubseteq S_1' \Box S_2'$.*
- *if $S_1' \ni S_1$ and $S_2 \ni S_2'$ then $S_1 \Box S_2 \ni S_1' \Box S_2'$.*
- *if $S \sqsubseteq S'$ then $|[con\ v; S]| \sqsubseteq |[con\ v; S']|$.*
- *if $S \ni S'$ then $|[con\ v; S]| \ni |[con\ v; S']|$.*

It can be shown that every S is monotonic:

Theorem 2.2 *All predicate transformers are monotonic: if $Q_1 \sqsubseteq Q_2$ then $S(Q_1) \sqsubseteq S(Q_2)$ and $S[Q_1] \sqsubseteq S[Q_2]$.*

This theorem enables us to add recursion to the command language. We will so this in the next subsection.

2.1 Recursion

In this subsection we add simultaneous recursion to the command language. In simultaneous recursion we have a separate set of procedure declarations, in which the procedure variables are defined. This is different from the approach taken in [BvW90] in which μ-recursion is added. We take simultaneous recursion because this is more close to logic programming. Hence we extend the command language with procedure calls:

$$S ::= \cdots \mid p(t)$$

where $p \in ProcVar$ is a procedure variable and $t \in Exp$ is a vector of the actual parameters. Each procedure variable should have a declaration of the form

$$p = \lambda v.S$$

A full program is made of a set of declarations d and a statement S and is denoted by $\langle d, S \rangle$. We will define a notion of refinement on programs. To this end we introduce environments which assign meaning to procedure calls:

$$\gamma \in Env = (ProcVar \rightarrow PTran) \times (ProcVar \rightarrow PTran)$$

An environment γ has two components: one component assigns meaning to procedure calls for the termination case and the other component for the deadlock case (cf. the difference between $S(Q)$ and $S[Q]$). Given a statement in the extended language and an environment γ we obtain a predicate transformer as is shown in the next definition:

$$P(\gamma)(Q) = P(Q)$$

$$\langle \vee v.P \rangle(\gamma)(Q) = \langle \vee v.P \rangle(Q)$$

$$S_1; S_2(\gamma)(Q) = S_1(\gamma)(S_2(\gamma)(Q))$$

$$S_1 \vee S_2(\gamma)(Q) = S_1(\gamma)(Q) \vee S_2(\gamma)$$

$$S_1 \wedge S_2(\gamma)(Q) = S_1(\gamma)(Q) \wedge S_2(\gamma)(Q)$$

$$|[con\ v; S]|(\gamma)(Q) = \exists v.S(\gamma)(Q).$$

$$S_1 \Box S_2(\gamma)(Q) = S_1(\gamma)(Q) \vee (S_1(\gamma)[Q] \wedge S_2(\gamma)(Q))$$

$$p(t)(\gamma)(Q) = \gamma_1(p)(Q)[t/v]$$

The deadlock case is handled in a similar, except for the procedure call where we use the second component of γ:

$$p(t)(\gamma)[Q] = \gamma_2(p)(Q)[t/v]$$

Given a fixed program with declarations d, how do we determine the associated γ. It is defined as the least fixed point of the following operator.

$$\Psi_d : Env \rightarrow Env$$

$$\Psi_d(\gamma)(p) = \langle \lambda Q.S(\gamma)(Q), \lambda Q.S(\gamma)[Q] \rangle$$

We take the least fixed point to obtain the meaning of a set of declarations. Because all predicate transformers are monotonic this is well defined.

$$\gamma_d = \mu \Psi_d$$

Now we are ready to define the notion of refinement of programs with recursion.

$$\langle d_1, S_1 \rangle \sqsubseteq \langle d_2, S_2 \rangle \ \textit{iff} \ S_1(\gamma_{d_1}) \sqsubseteq S_2(\gamma_{d_2})$$

Note that we refine a statement in an environment of procedures. Hence we are able to perform specializations in the calculus: we can start with a general program that works for all statements, but because we know what statement we are going to execute we can optimize with respect to this specific goal. This is one of the main goals of partial evaluation, see for example [Kom89, ?].

2.2 Committed Choice

In this subsection we give an extension of the refinement calculus which will enable us to use the committed choice mechanism of concurrent logic languages. It is a generalization of the normal guarded commands $P \rightarrow S$ where P is a predicate: we allow full statements at the place of P. The idea is that we can commit if execution of this statement terminates. Hence we extend the command language by

$$S \ ::= \cdots \mid S_1 \rightarrow S_2$$

with semantics

$$S_1 \rightarrow S_2(Q) \triangleq S_1(true) \Rightarrow S_1(S_2(Q))$$

$$S_1 \rightarrow S_2[Q] \triangleq S_1[true] \Rightarrow S_1[S_2[Q]]$$

3 Embedding of Logic Programming into the Refinement Calculus

The embeddings will be based on the denotational semantics. Take $\Sigma = \mathit{Subst}$: the set of idempotent substitutions. We then have $\mathit{Pred} = \mathit{Subst} \rightarrow \mathbf{B}$, i.e. $\mathit{Pred} = 2^{\mathit{Subst}}$. We first discuss how to embed Horn Clause Logic. There are several ways do this, depending on the selection rule. For more information about the semantics of logic programs consult [Llo87] or [Apt90] and various models for HCL are discussed in [dBKPR90]. We consider here a fair selection rule (the standard one for HCL) and a left-most selection rule (pure Prolog).

The embedding for the left-most selection rule is based on a continuation semantics given by [JM84]. It is interesting to note that we can have mixtures of selection rules in programs in the command language. After these two embeddings we give an outline how to model committed choice in the command language. Committed choice is the choice mechanism of concurrent logic languages, for which it is not feasible to implement backtracking. Then we discuss another embedding which is based on a different notion of states.

3.1 Embedding of Horn Clause Logic

We start by giving an embedding of Horn Clause Logic with a fair selection rule. A selection rule is called fair if it selects eventually every atom in the goal. As long as we use a fair selection rule, it does not matter what particular choice we make (see [Llo87]).

Consider in the rest of this subsection a fixed HCL program

$$H_i \leftarrow C_{1i}, \ldots, C_{n_i i}$$

for $i = 1, \ldots, k$ and fixed goal $\leftarrow C_1, \ldots, C_m$. For simplicity we take a program with only two clauses, i.e. $k = 2$. Before giving the embedding, we have to introduce one more notion. Our states map variables to terms, which can serve as parts of programs. If we use *deref* around an expression this means that we should replace the variables inside the expression by their values, and then consider the result as the statement to be executed.

Define

$$p = \lambda v.$$

$$|[con\ v_1 w\ ;\ \langle \vee w.w = variables(v)\rangle;\ \langle \vee deref(w)v_1.deref(p(v) = H_1)\rangle;\ C_{11};\ \ldots;\ C_{n_1 1}]|$$

$$\vee$$

$$|[con\ v_2 w\ ;\ \langle \vee w.w = variables(v)\rangle;\ \langle \vee deref(w)v_2.deref(p(v) = H_2)\rangle;\ C_{12};\ \ldots;\ C_{n_2 2}]|$$

for each predicate p appearing in the HCL program. The variables v_i $(i = 1, 2)$ should include the variables of $H_i \leftarrow C_{1i}, \ldots, C_{n_i i}$. We observe that each clause of the HCL program corresponds to a block. Inside such a block, we first determine the variables which should be updated and then we do this in such a way that the head and the atom in the goal become equal. We do this update in an angelic way, that is we choose the values of the variables such that we are guaranteed to terminate (if possible). This is weaker than taking the most general unifier. Also the choice between clauses is done by the angel.

The complete program is the declaration d as explained above together with statement $C_1;\ \ldots;\ C_m$ yielding the program

$$\langle d, C_1;\ \ldots;\ C_m\rangle.$$

We can do an optimization: above we compare an atom in the goal with every head of clauses but we can restrict this to only those heads which have the same predicate symbol. Note that all the choices in the program are angelic and hence the suggested program is almost a specification (i.e. the starting point of a program derivation).

A first step towards an implementation is to replace the angelic updates by most general unifiers and the angelic choice between clauses by the Prolog selection rule. This is the topic of the next subsection.

3.2 Embedding of Pure Prolog

We replace the angelic choice $\vee$ by the box $\Box$ and the angelic update by taking the most general unifier *mgu*. In this way we get a less specification oriented embedding. Take the following

declarations:

$$p = \lambda v.$$

$$[\![con\ v_1 w\ ;$$

$$\langle \vee w.w = variables(v)\rangle;\ \langle \vee deref(w)v_1.deref(w)v_1 = mgu(deref(p(v) = H_1))(wv_1)\rangle;$$

$$C_{11};\ \ldots;\ C_{n_1 1}]\!]$$

$$\square$$

$$[\![con\ v_2 w\ ;$$

$$\langle \vee w.w = variables(v)\rangle;\ \langle \vee deref(w)v_2.deref(w)v_2 = mgu(deref(p(v) = H_2))(wv_2)\rangle;$$

$$C_{12};\ \ldots;\ C_{n_2 2}]\!]$$

These embeddings look complicated at first sight, but by introducing some abbreviations and syntactic sugar they can be made more attractive.

3.3 Embedding of Committed Choice

Concurrent Logic Languages use committed choice because backtracking is too expensive to implement. Moreover, they have synchronization primitives and explicit parallelism (for more details about concurrent logic languages we refer to [Sha89]). The BRC calculus is used for detecting implicit parallelism, and one of the reasons for this is to keep correctness proofs and refinements simple. If we would also consider explicit parallelism the predicate transformer approach has to be extended in such that it can reason about intermediate states. Without such extension it is not possible to embed the concurrent logic languages. What we can do is to take over the committed choice mechanism and leave the detection of the possibilities for parallelization to the calculus. For the committed choice mechanism we extendeded BRC with an operator $\rightarrow$ (see the last subsection of the section on the refinement calculus). This extension is needed is because in concurrent logic languages it is allowed to have non-flat guards, that is, instead of checking boolean conditions, we

have to check if execution of a statement terminates. A concurrent logic program has clauses of the form

$$H \leftarrow G|B$$

where H is the head, G the guard and B the body of the clause. The guard and the body are sets of atoms. We give an outline of the embedding. Let ***unification*** stand for the two statements which establish the unification (see above), let ***guard*** be the sequence of procedure calls made out of the guard G and let ***body*** the sequence of procedure calls of the body B. When we subscript with i we mean the i-th clause.

$$p = \lambda v.$$

$$(\exists i \in \{1,2\}.|[con\ v_i;\ unification_i;\ guard_i]|(true));$$

$$|[con\ v_1;\ unification_1;\ guard_1 \rightarrow body_1]|$$

$$\wedge$$

$$|[con\ v_2;\ unification_2;\ guard_2 \rightarrow body_2]|$$

Note that we now can use demonic choice, but that we have to check in advance that at least one of the guards is terminating.

3.4 Alternative Embedding of Logic Programming

By playing around with the set of states, we can obtain a total different embedding. This embedding is based on the declarative semantics (see [Apt90]). Take $\Sigma = 2^{Atom}$: sets of ground atoms. We then have $Pred = \Sigma \rightarrow \mathbf{B}$, i.e. $Pred = 2^{2^{Atom}}$. Define

$$H \leftarrow C_1, \ldots C_n(Q) =$$

$$\{X \subseteq \Sigma\ :\ \exists \theta \in Subst,\ Y \in Q[Y = X \cup \{H\theta\} \wedge C_1\theta \in X \wedge \cdots \wedge C_n\theta \in X\}$$

This embedding is based on the intermediate consequence operator. Here we can obtain directly an action system by placing all the actions corresponding to clauses in a while loop with guards

that are always true. Action systems are well suited intermediate forms for parallelization (see [Ser90b]).

4 Conclusion

We have proposed several ways to embed logic programming in the refinement calculus. The advantages are both ways: the refinement calculus is extended with ways to reason about deadlock and has got some new operators. On the other hand, because logic programming can be embedded into the calculus, all the techniques of the calculus become available. For example, it seems to be possible to justify certain transformation rules, termination proofs and specialization. The embedding also paves the way for extensions to logic programming (for example with specification constructs, data structures) and also mixed formalisms (different selection rules, logic/imperative) seem to be feasible.

References

[AP90] K.R. Apt and D. Pedreschi. Studies in pure prolog: Termination. In J.W. Lloyd, editor, *Proc. Symposium on Computational Logic*, Lecture Notes in Computer Science, 1990.

[Apt90] K.R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland, 1990.

[Bac89] R.-J.R. Back. Changing data representations in the refinement calculus. In *Hawaii International Conference on System Sciences*, 1989.

[Bac90] R.-J.R. Back. Refinement calculus, part ii: Parallel and reactive programs. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, number 430 in Lecture Notes in Computer Science, pages 67–93, 1990.

[BS90] R.-J.R. Back and K. Sere. Deriving an occam implementation of action systems. Technical Report 99, Abo Akademi, 1990.

[BvW90] R.-J.R. Back and J. von Wright. Refinement calculus, part i: Sequential nondeterministic programs. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, number 430 in Lecture Notes in Computer Science, pages 42–66, 1990.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[dB88] J.W. de Bakker. Comparative semantics for flow of control in logic programming without logic. Technical Report CS-R8840, Centre for Mathematics and Computer Science, Amsterdam, 1988.

[dBKPR90] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.M.M.M. Rutten. From failure to success: Comparing a denotational and declarative semantics for horn clause logic. In M.Z. Kwiatkowska, M.W. Shields, and R.M. Thomas, editors, *Semantics for Concurrency*, Workshops in Computing, pages 38–60, 1990.

[Dev90] Y. Deville. *Logic Programming. Systematic Program Development*. International Series in Logic Programming. Addison-Wesley, 1990.

[Dij76] E.W. Dijkstra. *A discipline of Programming*. Prentice-Hall, 1976.

[JM84] N.D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for prolog. In *Proc. 1984 Int. Symp. on Logic Programming*, 1984.

[Kom89] J. Komorowski. Synthesis of programs in the framework of partial deduction. In *Proc. XIth International Joint Conference on Artificial Intelligence*, Detroit, 1989.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1987. second edition.

[Mor87] J. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, (9):287–306, 1987.

[MRG88] C.C. Morgan, K.A. Robinson, and P.H.B. Gardiner. On the refinement calculus. Technical Report PRG–70, Programming Research Group, 1988.

[Nel87] G. Nelson. A generalization of dijkstra's calculus. Technical Report 16, Digital Systems Research Center, 1987.

[Ser90a] K. Sere. Laws of action system programming. Technical Report 100, Abo Akademi, 1990.

[Ser90b] K. Sere. *Stepwise Derivation of Parallel Algorithms.* PhD thesis, Abo Akademi, 1990.

[Sha89] E.Y. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

[vW90] J. von Wright. *A Lattice-theoretical Basis for Program Refinement.* PhD thesis, Abo Akademi, 1990.