Continuations for Parallel Logic Programming

Eneia Todoran
Technical University of Cluj-Napoca
Dept. of Computer Science
Parallel and Distributed Systems Laboratory
Baritiu Str. 28, 3400, Cluj-Napoca, Romania.
Eneia.Todoran@cs.utcluj.ro

Nikolaos S. Papaspyrou
National Technical University of Athens
Dept. of Electrical and Computer Engineering
Software Engineering Laboratory
Polytechnioupoli, 15780 Zografou, Greece.
nickie@softlab.ntua.gr

ABSTRACT

This paper gives denotational models for three logic programming languages of progressive complexity, adopting the "logic programming without logic" approach. The first language is the control flow kernel of sequential Prolog, featuring sequential composition and backtracking. A committed-choice concurrent logic language with parallel composition (parallel AND) and don't care nondeterminism is studied next. The third language is the core of Warren's basic Andorra model, combining parallel composition and don't care nondeterminism with two forms of don't know nondeterminism (interpreted as sequential and parallel OR) and favoring deterministic over nondeterministic computation. We show that continuations are a valuable tool in the analysis and design of semantic models for both sequential and parallel logic programming. Instead of using mathematical notation, we use the functional programming language Haskell as a metalanguage for our denotational semantics, and employ monads in order to facilitate the transition from one language under study to another.

Keywords

Parallel logic programming, basic Andorra model, denotational semantics, continuations, monads, Haskell.

1. INTRODUCTION

The theory and practice of sequential logic programming is now considered well-established, having been studied for several decades [1, 17]. Over these years, researchers have distinguished at least two basic categories of semantics for logic programming: *declarative* and *operational*. Following the idea that logic programming is logic + control [16], a number of researchers have found it convenient to dedicate their investigation not to the declarative semantics of logic programming, but rather to the study of the various control flow concepts encountered therein, an approach usually called "logic programming without logic" that is advocated in [5, 6]. Focusing on control flow, it is possible to apply techniques used in the theory of programming language semantics and to develop operational and denotational models for both sequential Prolog and parallel logic languages.

PPDP'00, Montreal, Canada.

One of the first models of parallel logic programming was implemented in the family of *committed-choice* languages [26], well known representatives of which are Concurrent PROLOG [25], PARLOG [13] and Guarded Horn Clauses (GHC) [32]. Committed-choice languages support don't care nondeterminism and the parallel composition of goals. Further descendants of these languages, such as Flat Concurrent Prolog [19] and Flat GHC [23], are based on flat guards.

In an attempt to combine the sequential logic programming model of Prolog, which provides don't know nondeterminism, with the model of committed-choice languages, the basic Andorra model (BAM) was proposed [34]. BAM, which is also based on flat guards, has been implemented in Andorra-I [4], PANDORA [2] and owes much to P-Prolog [36]. Andorra provides both don't care and don't know nondeterminism. The execution of a don't know choice can be very expensive in practice when it contains more than one non failing alternatives, a phenomenon called *nondeterministic promotion* in which all AND-parallel goals are replicated for each non failing alternative. For this reason, such don't know goals are suspended in Andorra, until all other parallel goals have been reduced.

Our aim in the present paper is to show that continuations are very well suited for designing denotational semantic models for the control flow kernel of both sequential and parallel logic programming. We define three languages of progressive complexity, largely representing the three categories of logic programming languages discussed in the previous three paragraphs, and provide denotational semantics for each one of them, based on continuations. Our approach follows the principles of "logic programming without logic". Thus, in the definition of the three languages we abstract from any articulation of the basic computation steps and represent the elementary operations encountered in logic programming (e.g. unification, substitution generation, etc.) as uninterpreted atomic actions. The logical connectives are modelled by appropriate operators on processes.

Instead of using mathematical notation for the definition of the denotational semantics, we use the functional programming language *Haskell* [22]. In this way, we allow our denotational semantics to be directly implementable, in the

form of an interpreter for the languages under study, and thus to be easily tested and evaluated. At the same time, we avoid unnecessary complexities accompanying the use of domain theory or the theory of metric spaces, which could have been adopted alternatively.

One of the most important drawbacks of classic denotational semantics is its lack of modularity: small changes in a language's definition often imply a complete rewrite of its formal semantics. The use of monads has been proposed as a remedy and has become quite popular both in the denotational semantics and the functional programming community [20, 33]. Monads, which are directly supported in Haskell, are used in this paper in order to facilitate the definition of a modular and elegant semantics for the three languages under study in a unified way.

A brief description of the three languages is given below, together with our comments concerning the techniques used in defining their semantics.

- **Language $\mathcal{L}0$:** Pure sequential Prolog with backtracking (don't know nondeterminism). We have named this language $\mathcal{L}0$ because we consider it only an appetizer for the parallel logic languages that follow. In the denotational semantics for $\mathcal{L}0$ we use the classic technique of continuations [29, 18], which has been advocated for capturing the semantics of backtracking of sequential Prolog in various papers [11, 12, 5]. It should be mentioned that numerous other operational and denotational semantics for sequential Prolog have been proposed; many of these approaches include extra-logical features, such as cuts and side effects, not considered in the present paper.

- **Language $\mathcal{L}1$:** Commited-choice concurrent logic language, providing parallel composition and don't care nondeterminism with flat guards. $\mathcal{L}1$ is basically equivalent to the core of Flat Concurrent Prolog [19] and Flat GHC [23]. The denotational semantics of $\mathcal{L}1$ can be defined using the classic direct approach to concurrency semantics as in [7, 5]. In our definition, we use the "continuation semantics for concurrency" (CSC) technique introduced in [30]. This technique can model both sequential and parallel composition in interleaving semantics, while providing the general advantages of the classic technique of continuations [10].

- **Language $\mathcal{L}2$:** The basic Andorra model [34] which incorporates parallel composition, don't care nondeterminism and Prolog-like don't know nondeterminism. To the best of our knowledge, no denotational semantics for Andorra has been published and all our attempts to model the behaviour of the model by using various combinations of the direct approach to concurrency and the classic technique of continuations have failed. We consider the denotational model that we propose in section 4 as the most significant result of this paper. It is based on the CSC technique, which can capture the semantics of all features present in $\mathcal{L}2$: AND parallelism, Prolog's backtracking (implemented as sequential OR) and OR parallelism. Moreover, our semantics models the process suspension mechanism which follows from the Andorra principle: deterministic computation is given priority over nondeterministic computation. A more detailed description of the basic Andorra model [34] is given in section 4.

  In $\mathcal{L}2$, OR parallelism is implemented as a "true" parallel composition operator, an interleaving semantics for which has been studied in [31], in the absence of AND parallelism. All other denotational models known to us, e.g. [9, 8], implement OR parallelism in the context of concurrent (constraint) logic programming as don't care nondeterminism, and each trace corresponds to (at most) one logical solution of the program. Obviously, this approach is no longer appropriate for a language with don't know nondeterminism, like Andorra.

The rest of the paper is structured as follows. The next three sections define the three languages $\mathcal{L}0$, $\mathcal{L}1$ and $\mathcal{L}2$, study their denotational semantics using continuations and provide implementations of the semantics in Haskell. In the last two sections we discuss related work and present some concluding remarks and directions for future research.

2. SEQUENTIAL LOGIC PROGRAMMING

The first language under study, called $\mathcal{L}0$, is intended as a simplified approximation of sequential Prolog. The features that it combines are: failure, atomic actions, recursion, sequential composition (sequential AND) and backtracking (sequential OR). A simple grammar for the language $\mathcal{L}0$ is given below.

$$
\begin{array}{lcl}
p & ::= & (x{=}s\,;)^*\ s \\
s & ::= & g \mid \mathrm{call}(x) \mid s\,.\,s \mid \langle l \rangle \\
g & ::= & \mathrm{fail} \mid a \\
l & ::= & \epsilon \mid g?s\ (+\,g?s)^*
\end{array}
$$

A program is a sequence of declarations followed by a statement. Declarations associate statements with procedure variables, i.e. elements of the syntactic class x, and can be recursive. Statements consist of elementary statements, i.e. elements of the syntactic class g, recursive calls and applications of the sequential composition and backtracking operators. Elementary statements are failure and interpretation of a single atomic action, i.e. an element of the syntactic class a. Finally, the backtracking operator takes a (possibly empty) list of operands, i.e. an element of the syntactic class l, each operand being a statement guarded by an elementary statement (which can model head unification).

The abstract syntax of $\mathcal{L}0$ can be implemented in Haskell as follows:

```
type Act  = String
type PVar = String

data Stmt = A Atomic
          | Rec PVar
          | Sand (Stmt, Stmt)
          | Sor LStmt
```

```
data Atomic = Fail | Act Act
type LStmt  = [(Atomic, Stmt)]
```

It is not necessary to implement declarations and programs directly. Declarations can be modelled in Haskell as functions of type:

```
type Decl = PVar -> Stmt
```

In order to simplify the definition of the semantics, we assume that there is a distinguished element of this type which contains all the declarations of a given program.

```
decl :: Decl
```

The main semantic function maps each statement of $\mathcal{L}0$ to a *computation*, i.e. an element of the semantic class D. Although the selection of an appropriate D for $\mathcal{L}0$ is relatively easy, it is our goal to make this selection as general and abstract as possible, in order to simplify the transition to the languages of the following sections. We will use the technique of continuations, therefore it is reasonable to assume that computations are functions, mapping the current continuation to the final answer of the program. The semantic class C, as yet unspecified, represents *continuations* and the semantic class P represents *program answers*.

```
sem :: Stmt -> D

type D = C -> P
```

In the study of logic programming without logic, a reasonable choice for a program answer is a sequence of *observations* that result from the program's execution. Finding an appropriate P for $\mathcal{L}0$ is again easy, however our tendency for generalization leads us to the introduction of three monads, under the following rationale.

- The performance of some atomic actions may not be observable in the program answer. It is therefore reasonable to distinguish between the observations, which are present in a program answer, and the atomic actions, which are present in the abstract syntax. The relation between the two can be best modelled by monad `ObsM`, which is used below to map the syntactic class `Act` of atomic actions to the semantic class `Obs` of observations.

  ```
  type Obs = ObsM Act
  ```

- As mentioned above, sequences of observations are required in program answers. The monad `SeqM` helps in the abstraction of implementation details and maps `Obs` to the semantic class `Seq` of sequences of observations.

  ```
  type Seq = SeqM Obs
  ```

- Are logic programs deterministic? The answer varies: the language $\mathcal{L}0$ is certainly deterministic, but we will not expect the same from the languages of the following sections. In general, program answers are expected to consist of a set of elements of `Seq` and, again, we find it reasonable to hide the implementation details behind the monad `PosM` of possible program answers. This monad maps `Seq` to `P` as shown below.

  ```
  type P = PosM Seq
  ```

The reader is referred to the appendix for a brief introduction to monads and how they are used in Haskell. Apart from hiding implementation details, the use of monads improves the semantics in terms of modularity and elegance and smoothens the transition from $\mathcal{L}0$ to the languages of the following sections.

For the semantics of $\mathcal{L}0$, observations need not be differentiated from atomic actions. Also, $\mathcal{L}0$ is deterministic and program answers consist of one sequence. Therefore, monads `ObsM` and `PosM` can be substituted by the identity monad `Id`, which is defined in the appendix. Haskell's list monad is a perfect candidate for monad `SeqM`. Moreover, in the study of $\mathcal{L}0$ the class C of continuations needs not be different from the class P of program answers, i.e. a continuation is the answer of the fragment of the program that remains to be executed.

```
type ObsM a = Id a
type SeqM a = [a]
type PosM a = Id a

type C = P
```

The operations on sequences of observations can easily be implemented in terms of Haskell's operations on lists. The empty sequence `s0`, the sequence prefixing operator `prefixS` and the sequence concatenation `seqS` are defined below.

```
s0 :: Seq
s0 = []

prefixS :: Obs -> Seq -> Seq
prefixS = (:)

seqS :: Seq -> Seq -> Seq
seqS = (++)
```

Operations on program answers can be defined using operations on sequences and the properties of monad `PosM`. The empty answer `p0` contains an empty sequence of observations. Function `prefixP` implements the prefixing of an observation to a program answer. Finally, function `lsorP` takes as argument a list of program answers $p_1, \ldots, p_n$ and returns a new program answer p. If $s_1, \ldots, s_n$ are sequences of observations in $p_1, \ldots, p_n$ respectively, then the concatenation of all $s_1, \ldots, s_n$ is a sequence of p.

```
p0 :: P
```

```
p0 = return s0

prefixP :: Obs -> P -> P
prefixP o p = p >>= (return . prefixS o)

lsorP :: [P] -> P
lsorP [] = p0
lsorP (p:ps) = do {
   s1 <- p;
   s2 <- lsorP ps;
   return (s1 `seqS` s2)
}
```

According to our selection of $\mathsf{C} = \mathsf{P}$, the empty continuation is simply an empty program answer.

```
c0 :: C
c0 = p0
```

Function `cont` implements continuation *completion*. It maps a continuation to the program answer that would result if the continuation alone was left to execute. In the semantics of $\mathcal{L}0$, `cont` is the identity function, but this will not be true in the languages studied in the following sections.

```
cont :: C -> P
cont = id
```

The *failure* computation `d0` ignores the current continuation and returns an empty program answer. Function `lsorD` implements the backtracking (sequential OR) operator, which takes a list of computations, passes the current continuation to each of them and concatenates the results using `lsorD`. Function `addCont` takes a computation d and a continuation c and returns an extended continuation that performs both d and c. In its implementation below, `addCont` simply applies d to c, thus resulting in the sequential execution of first d then c.

```
d0 :: D
d0 c = p0

lsorD :: [D] -> D
lsorD l c = lsorP (map (\d -> d c) l)

addCont :: D -> C -> C
addCont d c = d c
```

After the definition of the auxiliary operations on sequences, program answers, continuations and computations, the definition of the semantics of $\mathcal{L}0$ is straightforward. The meaning of fail is the failure computation, whereas the meaning of an atomic action is the prefixing of the associated observation to the completion of the current continuation. Recursive calls are easily handled and so is the sequential composition operator, which extends the current continuation with the computation of the second statement and then passes the result to the computation of the first statement. Finally, the meaning of the backtracking operator

```
NL0> d (read "a . b")
["a","b"]

NL0> d (read "a . fail . b")
["a"]

NL0> d (read "<a ? b + fail ? c + d ? e>")
["a","b","d","e"]
```

Figure 1: Example, semantics of $\mathcal{L}0$.

is implemented using function `lsorD`, where candidates for execution are the statements with non-failing guards.

```
sem (A Fail) = d0
sem (A (Act a)) = prefixP (return a) . cont
sem (Rec x) = sem (decl x)
sem (Sand (s1, s2)) = sem s1 . addCont (sem s2)
sem (Sor ls) = lsorD
   [prefixP (return a) . cont . addCont (sem s)
      | (Act(a), s) <- ls]
```

The meaning of a complete program is simply the meaning of its statement applied to the empty continuation.

```
d :: Stmt -> P
d s = sem s c0
```

The semantics of a few example programs in $\mathcal{L}0$ are shown in Figure 1. The first two examples illustrate sequential composition and the `fail` statement. The last example illustrates the backtracking mechanism.[1]

3. CONCURRENT LOGIC PROGRAMMING

In concurrent logic programming, sequential composition and backtracking give way to *parallel composition* (parallel AND) and general (don't care) *nondeterministic choice*. The second language under study, called $\mathcal{L}1$, is intended as a core concurrent logic programming language and combines these two with the basic features of $\mathcal{L}0$ (failure, atomic actions, recursion). Other features present in $\mathcal{L}0$, such as sequential composition and backtracking, have been removed from $\mathcal{L}1$. The possibility of retaining them is discussed in section 6.

In the grammar for $\mathcal{L}1$ the rule for statements has been replaced by:

$$s ::= g \mid \mathrm{call}(x) \mid \ll o \gg \mid s \parallel s$$
$$o ::= \epsilon \mid g{:}s \ (+ g{:}s)^*$$

where, in addition to features already present in $\mathcal{L}0$, $\ll$o$\gg$ is the nondeterministic choice between the guarded statements

[1] Additional Haskell code supports the parsing of programs. The appendix contains information on how to obtain the complete code.

of o, and $s_1 \parallel s_2$ is the (interleaved) parallel execution of s_1 and s_2.

At this point, we should notice the semantic difference between the symbol ?, used in the rule for the syntactic class l in the previous section, and the symbol : used in the rule for the syntactic class o in the definition above. The symbol ? can be viewed as a simple sequential composition operator, prefixing a flat guard to a statement. If the guard succeeds, the statement is executed and other alternatives in l may follow. In this way, ? provides don't know nondeterminism. On the other hand, the symbol : can be viewed as a "commit" operator and provides don't care nondeterminism. If the guard succeeds, the statement is executed but no other alternative in o can follow.

The implementation of the abstract syntax for $\mathcal{L}1$ requires the following modification in the Haskell code. The type `LStmt` is also used for implementing the syntactic class o.

```
data Stmt = A Atomic
          | Rec PVar
          | Ned LStmt
          | Pand (Stmt, Stmt)
```

The nondeterministic features of $\mathcal{L}1$ compel a redefinition of monad `PosM`, which must support multiple program answers. Haskell's list monad is again a reasonable choice; however, since our intention is to use Haskell's lists to model sets, it is necessary to implement a set union operation for removing multiply occurring elements. Function `unionP` returns the union of two program answers, whereas `lunionP` returns the union of a list of program answers. Notice that the empty program answer is returned if the list is empty.

```
type PosM a = [a]

unionP :: P -> P -> P
unionP [] ys = ys
unionP (x:xs) ys = if x `elem` ys then
                      xs `unionP` ys
                   else
                      x : (xs `unionP` ys)

lunionP :: [P] -> P
lunionP [] = p0
lunionP l = foldl1 unionP l
```

In the presence of interleaved execution, we are forced to reconsider our notion of continuation. Following the CSC technique [30], we define continuations to be multisets of computations that are executed in parallel, and we again implement multisets as Haskell's lists. The empty continuation `c0` is now indeed an empty list of computations and `addCont` adds a computation to the continuation.

```
newtype C = C [D]

c0 :: C
c0 = C []

addCont :: D -> C -> C
addCont d (C l) = C (d : l)
```

Finally, the completion function `cont` returns the union of all possible interleaved executions of the computations included in the continuation. Operator `plusP` combines two alternative program answers and is, for the time being, a synonym to `unionP`. Its presence is necessary if the computation model is biased towards specific program answers and will be justified in the following section.

```
cont :: C -> P
cont (C []) = p0
cont (C l) =
   let cont' :: [D] -> [D] -> P
       cont' [] l2 = []
       cont' (d:l1) l2 =
          d (C (l1++l2)) `plusP` cont' l1 (d:l2)
   in  cont' l []

plusP :: P -> P -> P
plusP = unionP
```

The implementation of the nondeterministic choice operator is simply based on the union of program answers of the alternatives, which are all applied to the current continuation. Similarly, the implementation of parallel composition is based on the (possibly) biased combination of two alternative computations: one starting from the first statement and one starting from the second. The semantics of the two constructs and the definitions of the two auxiliary functions, `lunionD` and `plusD`, are given below.

```
sem (Ned ls) = lunionD
   [prefixP (return a) . cont . addCont (sem s)
      | (Act a, s) <- ls]
sem (Pand (s1, s2)) =
   (sem s1 . addCont (sem s2)) `plusD`
   (sem s2 . addCont (sem s1))

lunionD :: [D] -> D
lunionD l c = lunionP (map (\d -> d c) l)

plusD :: D -> D -> D
plusD d1 d2 c = d1 c `plusP` d2 c
```

A number of example programs in $\mathcal{L}1$ are shown in Figure 2 with the corresponding semantics. The first two examples illustrate the semantics of parallel composition, according to which the atomic actions can be executed in any order, whereas the third example illustrates the semantics of nondeterministic choice. The last example is a combination of the two features.

4. THE BASIC ANDORRA MODEL

The third language under study, called $\mathcal{L}2$, is the core of Warren's Basic Andorra Model. The language contains all the features that were studied in the previous sections, with the exception of sequential composition (sequential AND),

```
NL1> d (read "a || b")
[["a","b"],["b","a"]]

NL1> d (read "a || b || c")
[["a","b","c"],["a","c","b"],["b","c","a"],
 ["b","a","c"],["c","b","a"],["c","a","b"]]

NL1> d (read "<<a : b + fail : c + d : e>>")
[["a","b"],["d","e"]]

NL1> d (read "a || <<fail : b + c : d>>")
[["a","c","d"],["c","d","a"],["c","a","d"]]
```

Figure 2: Example, semantics of $\mathcal{L}1$.

plus an additional don't know nondeterministic operator $\#\langle l \rangle$, implemented as parallel OR. The grammar rule for statements in $\mathcal{L}2$ is given below,

$$s \ ::= \ g \mid \text{call}(x) \mid \langle l \rangle \mid \ll o \gg \mid s \parallel s \mid \#\langle l \rangle$$

and the Haskell implementation of the abstract syntax for $\mathcal{L}2$ is the following.

```
data Stmt = A Atomic
          | Rec PVar
          | Sor LStmt
          | Ned LStmt
          | Pand (Stmt, Stmt)
          | Por LStmt
```

The basic characteristic of this language is the *Andorra principle*, which gives priority to deterministic computation over nondeterministic computation. The rationale behind this principle is that nondeterministic reduction steps (nondeterministic promotion) are likely to multiply work. By following this principle, a logic programming language equipped with parallel AND can indeed reduce the number of steps required for the execution of a program, as compared to a sequential version of the language like $\mathcal{L}0$.

Following the Andorra principle, execution in $\mathcal{L}2$ favors *determinate* goals over *nondeterminate* ones in parallel conjunctions. Elementary goals and don't care goals are determinate and can always be reduced. On the other hand, don't know goals can only be reduced if they are determinate, i.e. if at most one of the alternatives has a non failing guard. The reduction of nondeterminate goals is delayed as much as possible. When only nondeterminate goals remain to be reduced, then the alternatives of a don't know goal are tried either in order (sequential OR) or concurrently (parallel OR).

Following this behaviour, the semantic model for $\mathcal{L}2$ needs to distinguish between determinate and nondeterminate reduction steps. This can be achieved by modifying our notion of observation. In the case of a determinate reduction step, the observation is always a single atomic action, as it has been in the semantics of the previous sections. However, in the case of a nondeterminate reduction step, i.e. when a don't know goal with more than one alternatives must be reduced, the observation is a multiset containing the atomic actions guarding the alternatives of the nondeterminate goal. We need to modify the definition of monad `ObsM`, and again we use Haskell's lists to implement multisets.

```
type ObsM a = [a]
```

In order to model the semantics of biased execution behaviour in $\mathcal{L}2$, we need to modify the definition of function `plusP`. The new version of this function distinguishes between determinate and nondeterminate goals. If either of the two program answers p_1 and p_2 that must be combined contains a determinate goal, i.e. contains a sequence starting with a simple observation, then the combined answer contains only the determinate goals of p_1 and p_2. Otherwise, all nondeterminate goals are combined.[2]

```
plusP :: P -> P -> P
plusP p1 p2 =
   let det :: P -> P
       det [] = []
       det ([] : p) = [] : (det p)
       det (([a] : s) : p) = ([a] : s) : (det p)
       det (_ : p) = det p
   in  case det p1 `unionP` det p2 of
          [] -> p1 `unionP` p2
          p  -> p
```

The implementation of the backtracking operator requires a small modification from its original definition in section 2. The reason is that the set union operation needs to be applied to the resulting program answer, in order to eliminate multiple answers.

```
lsorP :: [P] -> P
lsorP [] = p0
lsorP (p:ps) = lunionP (do {
   s1 <- p;
   s2 <- lsorP ps;
   return [s1 `seqS` s2]
})
```

The semantics of the parallel OR construct requires a number of additional operations in our semantics. Function `parS` takes two sequences of observations and generates a program answer that contains all possible interleaved executions of the two sequences. Functions `lporP` and `lporD` extend the same operation to lists of program answers and lists of computations respectively.

```
parS :: Seq -> Seq -> P
```

[2] It is worthwhile noticing that our semantics does not decide the determinacy of goals based on the abstract syntax, but on program answers. The laziness of Haskell prevents our implementation of the semantics from computing the whole program answer, before deciding which goals are determinate, and this leads to improved performance.

```
parS q1 q2 =
   let parS' :: Seq -> Seq -> P
       parS' [] s = [s]
       parS' (o:s) s' = prefixP o (s `parS` s')
   in  (q1 `parS'` q2) `unionP` (q2 `parS'` q1)

lporP :: [P] -> P
lporP [] = p0
lporP (p:ps) = lunionP (do {
   s1 <- p;
   s2 <- lporP ps;
   return (s1 `parS` s2)
})

lporD :: [D] -> D
lporD l c = lporP (map (\d -> d c) l)
```

To complete the semantics of $\mathcal{L}2$, we only need to address the two don't know nondeterministic operators of the language. In both cases, for each alternative with a non failing guard the corresponding statement is added to the current continuation. The list of computations formed in this way is passed to the appropriate function, `lsorD` or `lporD`, and the prepended observation contains all the non failing guards.

```
sem (Sor ls) =
   let sor :: [(Act, D)] -> D
       sor [] = d0
       sor l = prefixP (map fst l) .
               lsorD (map snd l)
   in  sor [(a, cont . addCont (sem s))
            | (Act a, s) <- ls]
sem (Por ls) =
   let por :: [(Act, D)] -> D
       por [] = d0
       por l = prefixP (map fst l) .
               lporD (map snd l)
   in  por [(a, cont . addCont (sem s))
            | (Act a, s) <- ls]
```

Examples from our implementation of the semantics of $\mathcal{L}2$ are given in Figure 3. The first six examples are also contained in either Figure 1 or Figure 2. With the exception of the first and fourth example, which contain nondeterminate sequential OR goals, the obtained results are the same with those obtained in the previous figures. In the first and fourth example, the results are also similar but all non failing guards of the sequential OR goals are combined in one observation.

The last three examples illustrate the biased execution behaviour towards determinate goals. In the first one, the reduction of the nondeterminate sequential OR goal is delayed in the presence of a and b. After these two have been executed, in any order, the nondeterminate goal can be reduced. Notice that the same example would produce 46 different program answers, if the Andorra principle was not followed. Two of these answers would correspond to the answers obtained here, and the rest would be the result of the nondeterministic promotion that would follow the reduction of the sequential OR goal. The next example is similar, only the nondeterminate goal is a parallel OR.

The last example is the most complex one, where four goals are executed in AND parallel. After the interleaved execution of the two determinate goals `a || b`, a state is reached where the only goals that remain are the two nondeterminate ones: `<C ? c + D ? d> || <E ? e + F ? f>`. In this state an arbitrary nondeterminate goal is selected and execution can only proceed by nondeterministic promotion, i.e. by making copies of the other goal for each alternative of the selected one. For example the first trace shown in Figure 3 is obtained when the nondeterminate goal `<E ? e + F ? f>` is selected for nondeterministic promotion. After performing the nondeterministic promotion step, which produces the observable `["E","F"]`, the execution proceeds with two computations: first `e || <C ? c + D ? d>`, which produces the sequence of observables `["e"],["C","D"],["c"],["d"]`, and upon backtracking `f || <C ? c + D ? d>`, which produces `["f"],["C","D"],["c"],["d"]`.

5. RELATED WORK

The classic technique of continuations [29, 18] has been advocated for capturing the semantics of backtracking of sequential Prolog in various papers [11, 12, 5]. More recently, the list monad has been used for describing the semantics of Prolog's clause unfolding and the advantages of adopting monads in the higher-order logic programming framework of λProlog have been investigated [3].

Our continuation-based approach to the semantics of parallelism in logic programming seems to be new. All papers known to us (including [5, 6, 7, 8, 9]) follow the classic (direct) approach to concurrency, where the semantic designer defines the various operators for parallel composition on processes as functions that manipulate final semantic values. The CSC technique [30] used in the present paper provides a different way of computing the interleaved execution of parallel processes, which is based on manipulating continuations. Even though the two approaches are different, all our experiments show that they behave the same in the case of a language like $\mathcal{L}1$ which provides parallel composition (AND parallelism) and (don't care) nondeterminism.

To the best of our knowledge no denotational semantics for languages based on the Andorra model like $\mathcal{L}2$ has been published and all our attempts to model the behaviour of $\mathcal{L}2$ by using various combinations of the direct approach to concurrency and the classic technique of continuations have failed. This confirms our belief expressed in [30] that the CSC technique can provide more flexibility than the classic (direct) approach to concurrency in handling complex operations on processes.

Operational semantics for languages similar to $\mathcal{L}2$, combining reactive behaviour (related to AND parallelism in logic programming) and search (backtracking and OR parallelism), are given in [15, 27]. Both papers formalize implementations of the extended Andorra model, i.e. the extension of the basic Andorra model with deep guards. The first gives an operational semantics for the Andorra Kernel Language (AKL) [14], whereas the second presents a calculus (also operational in nature) intended as a semantic foundation for Oz [28].

```
NL2> d (read "<a ? b + fail ? c + d ? e>")
[[["a","d"],["b"],["e"]]]

NL2> d (read "a || b")
[[["a"],["b"]],[["b"],["a"]]]

NL2> d (read "a || b || c")
[[["a"],["b"],["c"]],[["a"],["c"],["b"]],[["b"],["c"],["a"]],
 [["b"],["a"],["c"]],[["c"],["b"],["a"]],[["c"],["a"],["b"]]]

NL2> d (read "<A ? a || b + B ? d || e>")
[[["A","B"],["a"],["b"],["d"],["e"]],[["A","B"],["a"],["b"],["e"],["d"]],
 [["A","B"],["b"],["a"],["d"],["e"]],[["A","B"],["b"],["a"],["e"],["d"]]]

NL2> d (read "<<a : b + fail : c + d : e>>")
[[["a"],["b"]],[["d"],["e"]]]

NL2> d (read "a || <<fail : b + c : d>>")
[[["a"],["c"],["d"]],[["c"],["d"],["a"]],[["c"],["a"],["d"]]]

NL2> d (read "a || <c ? d + e ? f> || b")
[[["a"],["b"],["c","e"],["d"],["f"]],[["b"],["a"],["c","e"],["d"],["f"]]]

NL2> d (read "a || #<c ? d + e ? f> || b")
[[["a"],["b"],["c","e"],["d"],["f"]],[["a"],["b"],["c","e"],["f"],["d"]],
 [["b"],["a"],["c","e"],["d"],["f"]],[["b"],["a"],["c","e"],["f"],["d"]]]

NL2> d (read "a || <C ? c + D ? d> || b || <E ? e + F ? f>")
[[["a"],["b"],["E","F"],["e"],["C","D"],["c"],["d"],["f"],["C","D"],["c"],["d"]],
 [["a"],["b"],["C","D"],["c"],["E","F"],["e"],["f"],["d"],["E","F"],["e"],["f"]],
 [["b"],["a"],["C","D"],["c"],["E","F"],["e"],["f"],["d"],["E","F"],["e"],["f"]],
 [["b"],["a"],["E","F"],["e"],["C","D"],["c"],["d"],["f"],["C","D"],["c"],["d"]]]
```

Figure 3: Example, semantics of $\mathcal{L}2$.

A recent work worth mentioning is presented in [24], where the authors propose an axiomatization of the semantics of logic languages similar to $\mathcal{L}2$, focusing on control flow concepts. Using a functional library of primitive operators and the equational machinery of functional languages, the authors find that the primitive scheduling operators of logic programming obey the laws found in the categorical theory of monads. The authors identify three such monads, one corresponding to a depth-first scheduling strategy, one to breadth-first strategy and one that allows both strategies. The semantics that they propose is denotational in nature and implemented in Haskell. Their language features two basic control flow operators, conjunction and disjunction, which can be interpreted sequentially or in parallel, depending on the scheduling strategy. Our approach differs in two ways. First, the sequential and parallel forms of the control flow operators are distinct and present in the language simultaneously. Second, our approach models the Andorra principle.

6. CONCLUDING REMARKS AND FUTURE RESEARCH

In this paper, we have shown that continuations can be used as a tool for the systematic specification and design of both sequential and parallel logic programming languages. We have studied three languages of progressive semantic complexity in a unified way and shown that a careful selection of monads can indeed enhance the modularity and elegance of the semantics and facilitate the introduction of additional features or additional execution principles. Furthermore, we have implemented our semantics in Haskell, thus providing a directly executable prototype for the three languages under study. With minor modifications, our Haskell implementation can serve as the basis of a prototype interpreter, which would randomly choose between alternative execution paths.

The main contribution of our work is an accurate denotational semantics for the basic Andorra model, using the CSC technique. To the best of our knowledge, no other denotational semantics for Andorra has been previously published. The basic Andorra model is incorporated in various logic and constraint-based languages and we believe that, in the future, parallel and distributed languages will combine reactive behaviour and search mechanisms in their design. For this combinations of concepts, we have shown that the CSC technique is an adequate specification tool. Moreover, little previous experience exists with building concurrent languages systematically from their denotational description. Our Haskell implementation of Andorra using the CSC technique is an important step in this direction.

It would be possible to include the backtracking operator of

$\mathcal{L}0$ (sequential OR) as a form of don't know nondeterministic operator in $\mathcal{L}1$, by modifying the definition of `lsorP` as we do in section 4. This would not add semantic complexity to $\mathcal{L}1$. However, retaining sequential composition (sequential AND) in either $\mathcal{L}1$ or $\mathcal{L}2$ would complicate a bit the semantics of these two languages. The reader is referred to [30], where the semantics of sequential composition is defined in a concurrent language by using partial ordering relations on continuations.

In this paper we have found it convenient to follow the "logic programming without logic" approach advocated in [5] and to focus on the control flow kernel of (parallel) logic programming. Accordingly, we have abstracted from any articulation in the basic computation steps and ignored concepts like unification, substitution generation, etc. However, it is not difficult to add such concepts to the semantic models given in this paper and to obtain full denotational descriptions of (parallel) logic languages. As explained in [5], in general this can be achieved by interpreting the elementary actions as computational steps that are relevant for the logic language under consideration, while leaving the already available (abstract) control flow kernel intact.

The semantic framework presented in this paper is very flexible, allowing for further refinements. In the future we are mainly interested in the application of the CSC technique in the specification and design of concurrent constraint (logic) programming languages. In [30] it is shown that the CSC technique can express the asynchronous communication mechanism encountered in concurrent constraint (logic) programming (see also [8]) in a simple and elegant way, by augmenting the semantic function with an additional parameter (that can be used to model a constraint store) shared by all processes in the continuation. We know how to obtain full descriptions of concurrent constraint (logic) languages, by adding this (asynchronous) communication mechanism to the semantic models given in this paper for $\mathcal{L}1$ and $\mathcal{L}2$. However, the subject is not trivial, and thus we defer its treatment to a forthcoming paper.

In the present work we have only considered languages with flat guards. One of our next aims is to apply the CSC technique to parallel logic programming languages with deep guards. In doing so, we intend to move from the basic Andorra model to the *extended Andorra model* [35], which has been implemented in languages (that incorporate the constraint programming paradigm) like AKL [14, 15] and Oz [27, 28].

Future research will also be directed to the tighter combination of continuations and monads, in order to improve the flexibility, elegance and modularity of the semantic descriptions. Based on our previous results concerning the interleaved semantics of expression evaluation under an unspecified evaluation order [21], we believe that it is possible to define monads for the CSC technique which would abstract the underlying execution behaviour.

7. ACKNOWLEDGEMENTS

We would like to thank the anonymous referees for their useful comments on the first version of this paper.

8. REFERENCES

[1] K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of theoretical computer science*, volume B, pages 493–574. Elsevier Science Publishers B.V., 1990.

[2] R. Bahgat. *PANDORA: nondeterministic parallel logic programming*. World Scientific, 1993.

[3] Y. Bekkers and P. Tarau. Monadic constructs for logic programming. In J. Lloyd, editor, *Proceedings of ILSP'95*, pages 51–65, Portland, OR, Dec. 1995. MIT Press.

[4] V. Costa, D. H. D. Warren, and R. Yang. Andorra-I: a parallel Prolog system that transparently exploits both and- and or-parallelism. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, 1991.

[5] J. W. de Bakker. Comparative semantics for flow of control in logic programming without logic. *Information and Computation*, 94:123–179, 1991.

[6] J. W. de Bakker and E. P. de Vink. *Control flow semantics*. Foundations of Computing Series. MIT Press, Cambridge, MA, 1996.

[7] J. W. de Bakker and J. N. Kok. Comparative metric semantics for concurrent Prolog. *Theoretical Computer Science*, 75:15–43, 1990.

[8] F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. A paradigm for asynchronous communication and its application to concurrent constraint programming. In K. R. Apt, J. W. de Bakker, and J. J. M. M. Rutten, editors, *Logic programming languages: constraints, functions and objects*, pages 82–114. MIT Press, 1993.

[9] F. S. de Boer, A. Piero, and C. Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science*, 151:37–78, 1995.

[10] A. de Bruin. *Experiments with continuation semantics: jumps, backtracking, dynamic networks*. PhD thesis, Vrije Universiteit, Amsterdam, 1986.

[11] A. de Bruin and E. P. de Vink. Continuation semantics for Prolog with cut. In *Proceedings of TAPSOFT'89*, volume 351 of *LNCS*, pages 178–192. Springer Verlag, 1989.

[12] E. P. de Vink. Comparative semantics for Prolog with cut. *Science of Computer Programming*, 13:237–264, 1989.

[13] S. Gregory. *Parallel logic programming in PARLOG*. Addison-Wesley, 1987.

[14] S. Haridi and S. Janson. Kernell Andorra Prolog and its computation model. In *Proceedings of the International Conference on Logic Programming (ICLP'90)*. MIT Press, 1990.

[15] S. Haridi and C. Palamidessi. Structural operational semantics of kernel Andorra Prolog. In *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE'91)*. Springer Verlag, 1991.

[16] R. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22:424–435, 1979.

[17] J. W. Lloyd. *Foundations of logic programming*. Springer Verlag, 2nd edition, 1987.

[18] A. Mazurkiewicz. Proving algorithms by tail functions. *Information and Control*, 18:220–226, 1971.

[19] C. Mierkovsky, S. Taylor, E. Shapiro, J. Levy, and M. Safra. The design and implementation of flat concurrent Prolog. Technical Report CS85-09, Weizmain Institute, Department of Applied Mathematics, Israel, 1985.

[20] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, University of Edinburgh, Laboratory for Foundations of Computer Science, 1990.

[21] N. S. Papaspyrou and D. Maćoš. A study of evaluation order semantics in expressions with side effects. *Journal of Functional Programming*, 2000. to appear.

[22] S. Peyton Jones and J. H. (editors). *Report on the programming language Haskell 98: a non-strict purely functional language*, 1999. Available from `http://haskell.org/`.

[23] G. A. Ringwood. PARLOG 86 and the dinning logicians. *Communications of the ACM*, 31:10–25, 1988.

[24] S. Seres and M. Spivey. Algebra of logic programming. In *Proceedings of the International Conference on Logic Programming (ICLP'99)*, 1999.

[25] E. Y. Shapiro. A subset of concurrent Prolog and its interpreter. Technical Report TR-003, ICOT, Tokyo, 1983.

[26] E. Y. Shapiro. The family of concurrent logic programming languages. *ACM Computer Surveys*, 21:412–510, 1989.

[27] G. Smolka. A calculus for higher-order concurrent constraint programming with deep guards. Research Report RR-94-03, Deutsches Forschungszentrum für Kunstliche Intelligenz (DFKI), Saarbrucken, Germany, 1994.

[28] G. Smolka. The Oz programming model. In *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer Verlag, 1995.

[29] C. Strachey and C. P. Wadsworth. Continuations: a mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University, Programming Research Group, 1974.

[30] E. Todoran. Metric semantics for synchronous and assynchronous communication: a continuations based approach. *Electronic Notes on Theoretical Computer Science*, 28:119–146, 2000.

[31] E. Todoran, J. den Hartog, and E. P. de Vink. Comparative metric semantics for commit in or-parallel logic programming. In *Proceedings of the International Logic Programming Symposium 97*, pages 101–115. MIT Press, 1997.

[32] K. Ueda. Guarded Horn clauses: a parallel logic programming language with the concept of a guard. In M. Nivat and K. Fuchi, editors, *Proceedings of Programming of Future Generation Computers*, pages 441–456. North Holland, 1988.

[33] P. Wadler. The essence of functional programming. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages (POPL'92)*, Jan. 1992.

[34] D. H. D. Warren. The Andorra principle. Talk given at the Gigalips Workshop, Swedish Institute of Computer Science (SICS), Stockholm, Sweden, 1988.

[35] D. H. D. Warren. Extended Andorra model with implicit control. Talk given at a Parallel Logic Programming Workshop, Eilat, Israel, 1990.

[36] R. Yang. *P-Prolog: a parallel logic programming language*. World Scientific, 1987.

APPENDIX

A. COMPLETE CODE

Due to space restrictions, the Haskell code presented in this paper does not implement the pretty-printing and parsing of $\mathcal{L}0$, $\mathcal{L}1$ and $\mathcal{L}2$ programs. Apart from this, however, the paper is self contained. The complete code can be obtained from `ftp://ftp.softlab.ntua.gr/pub/users/nickie/papers/ppdp00.code.tar.gz`.

B. MONADS IN HASKELL

The rest of the appendix contains a brief introduction to monads and their use in Haskell. More detailed introductions to the theory and practice of monads can be found in [20, 33]. Almost all there is to know about Haskell [22], including several papers on monads, can be found in Haskell's home page (`http://haskell.org/`).

A monad is a triple of the form (`m`, `return`, `>>=`), where the first element `m` is a type constructor of kind $* \to *$, mapping an arbitrary type `a` to a new type `m a`. The following code contains the definition of the standard Haskell class `Monad`, based on the previous description. The types of the polymorphic functions `return` and `>>=` are specified and the last line defines `>>=` as an infix left associative operator of very low precedence.

```
class Monad m where
   return :: a -> m a
   (>>=)  :: m a -> (a -> m b) -> m b

infixl 1 >>=
```

Types constructed by monad `m` can be considered as denoting *computations*, e.g. the type `m a` denotes computations returning values of type `a`. Therefore, the definition

of a monad `m` reflects a notion of computation. The result of `return v` is a trivial computation, simply returning the value `v`. Assuming `z :: m a` and `f :: a -> mb`, the result of `z >>= f` is the combined computation of `z`, returning `v`, followed by the computation `f v`. Thus, `return` can be seen as a way of inserting values in computations, whereas `>>=` can be seen as a way of extracting values from computations, in order to use them in subsequent computations.

Three monad laws, which every monad is required to satisfy, insure the soundness of this correspondance between monads and computations:

```
Law 1:            return v >>= f  =  f v
Law 2:              z >>= return  =  z
Law 3:  z >>= (\v -> f v >>= g)   =  z >>= f >>= g
```

The identity monad `Id` can be implemented as follows. It is trivial to check that it satisfies the monad laws.

```
newtype Id a = Id a

instance Monad Id where
   return      = Id
   Id a >>= f  = f a
```