Dealing with Explicit Exceptions in Prolog

Luigi Liquori, Maria Luisa Sapino

Dipartimento di Informatica - Università di Torino

C.so Svizzera 185 - 10149 TORINO

E-mail: (liquori, mlsapino)@di.unito.it

Abstract

Existing logic languages provide some simple "extra-logical" constructs for control manipulation, such as the *cut* of standard Prolog and the exception handling constructs of other versions of Prolog (e.g. *SICStus* Prolog). Aspects specifically concerning the flow of control in a language can be quite naturally modelled by means of the Denotational Semantics, and in particular the Denotational Semantics with Continuations. In this paper we define a Denotational Semantics with Continuations to model the flow of control of a small fragment of a logic language with an explicit exception handling mechanism. Finally we show how the cut operator can be simulated by an appropriate use of the characterized exception handling constructs.

Keywords: Continuations, denotational semantics, exception handling, cut.

1 Introduction

One of the main features of Logic Programming is the explicit distinction between the logic component and the control component of programs [Kow74, Kow79]. The meaning of a program is completely specified by its logic component, which defines *what* the program does, independently of *how* the results are computed. According to this distinction, standard semantics have been defined for logic programs without taking into account the control component; these semantics characterize the intended meaning of programs by means of the set of ground atoms they imply.

However, when logic programming is actually used as a programming tool, i.e. when a logic language such as Prolog is used for real applications, not only must the control component be taken into account because of efficiency matters, but also to enlarge the expressive power of the language and to solve some problems which could not be expressed in a pure Horn logic language. This is the case, for example, when logic languages are used for significant applications in Artificial Intelligence, where the control strategy plays an important role, and problems concerning termination and exception handling must be dealt with.

Existing logic languages provide some simple *extra-logical* constructs for control manipulation, such as the *cut* of standard Prolog or the exception handling constructs of other versions of Prolog, such as *SICStus* Prolog [AAB+93]. These extra-logical constructs allow the user to explicitly modify the flow of control of the execution, and to avoid visiting some branches of the search space: branches which are known to be useless, or to be less convenient than other branches, are not explored. Due to this pruning mechanism within the search space, the resulting refutation strategy looses the completeness property with respect to the logical semantics. The standard declarative characterizations defined for pure logic programming are not well suited to model languages enriched with control operators. Other declarative semantics have been defined to model Prolog control: Barbuti et al. [BCGL93] proposed to model Prolog's behaviour in a simple constraint logic language; Bossi et al. [BBF93] presented a fixed point semantics which allows to capture both the left-to-right selection rule and the depth-first search strategy underlying Prolog's evaluation mechanism. Both of these proposals allow to maintain the usual declarative approach to the semantic characterization of logic programs.

The aspects concerning the flow of control in a language can be quite naturally modelled by means of other formal tools, like the denotational semantics [Sch86], and in particular the denotational semantics with *continuations* [dBdV89]. Continuations are functions that denote the part of the computation which still has to be executed in order to complete the modelled evaluation. They are a formal tool well suited to characterize the semantics of any language with exceptions, and have been used to model the behaviour of imperative, functional and logic languages [Sch86]. Starting from a given continuation semantics for a language, a compiler for the same language can be automatically defined [App92]; this means that continuations clearly relate the aspects concerning formal specifications of languages to those regarding the implementation of the languages themselves.

Some existing functional languages allow the user to directly manipulate continuations, and therefore to change the flow of control during the evaluation of a function. This is the case for the functional language *Scheme* [RC86] which provides a primitive, *call/cc*, to catch the current continuation and to install a previously catched one, respectively. While a single continuation is enough to model the behaviour of functional programs, logic programs need two of them, a *success continuation* (by means of which the control mechanism of the AND construct is formalized) and a *failure continuation* (to deal with the OR construct).

An interesting application of denotational semantics with continuations in logic programming is the use of Continuation Passing Style (CPS) [ST89, TB90, Ued87] programming to define Prolog translations into imperative languages [MH84].

Many authors have studied the problem of modelling the control mechanism of Prolog in a formal way. Jones and Mycroft [JM84] gave an operational characterization of Prolog as a deterministic language. Their approach is based on the SLD procedure, and it models the "cut" control operator, too. Arbab and Berry [AB87] provided both an operational and a denotational semantics for Prolog; they considered both the "cut", the database and some extra-logical facilities. Their approach is substantially different from the one in [JM84], since it is not based on the SLD procedure. The definition of the operational semantics is an interpretive one, defined by means of a Vienna Definition Language, and it is quite near to the way Prolog interpreters (written in Prolog) proceed. In particular, the backtracking mechanism is made explicit. Starting from the operational semantics, Arbab and Berry defined an equivalent continuation denotational one. De Vink [dV88] provided a direct denotational model for logic programs, and proved its soundness with respect to an operational semantics defined by means of a transition system. De Bruin and de Vink [dBdV89] gave a denotational semantics with continuations for Prolog with "cut" (or, more precisely, for a uniform language extracted from the Prolog with "cut"), and they proved its equivalence with respect to an operational semantics based on a deterministic transition system. Nicholson and Foo [NF89] defined a continuation based denotational semantics for Prolog. Brisset and Ridoux [BR93] introduced new built-in constructs in λ-Prolog [NM88], in order to provide new control mechanisms by means of explicit replacement of continuations. They chose λ-Prolog as the language to be extended since both the definition and the use of their new built-ins require higher-order syntax and implication goals[1].

In this paper we define a *uniform* logic language $\mathcal{L}$ [dBKM$^+$86, dB88, dBdV89] which can be regarded as an essential language extracted from Prolog with exception handling. More precisely, we will define two built-in constructs, $on\_exc(I, G_1, G_2)$ and $raise\_exc(I)$, which essentially behave like the analogous built-ins *on_exception(?Pattern, :ProtectedGoal, :Handler)* and *raise_exception(+Exception)* of *SICStus* Prolog.

All the logical aspects of Prolog are filtered out from our uniform language, in order to

[1]Actually, Brisset and Ridoux called their semantics "operational", probably because control flow is usually regarded as an operational aspect of the interpretation of programming languages. Nonetheless they adopted denotational semantics with continuations.

stress the fact that we concentrate on control mechanism aspects. The only explicit control mechanism introduced is exception handling. We do not include the "cut" operator in the language since, as we will show in section 4, its behaviour can be simulated quite easily by means of the exception handling constructs. We will define a denotational semantics with continuations [Sch86] for the uniform language $\mathcal{L}$.

The paper is organized as follows. In section 2 we define the uniform language $\mathcal{L}$, and informally show the behaviour of its control operators by means of an example. Section 3 is devoted to the definition of the denotational semantics with continuations for $\mathcal{L}$; in the same section we also introduce a notion of formal equivalence between programs with exceptions. In Section 4 we show how the "cut" operator can be simulated through the exception handling mechanism. Section 5 contains some conclusions and a few remarks about future work.

2 The Uniform Declarative Language $\mathcal{L}$

In this section we introduce the uniform propositional language $\mathcal{L}$, a simple language which only contains issues related to the flow of control of a declarative language. We can think of $\mathcal{L}$ as representing the control aspects of Prolog extended with explicit exception built-ins, like *on_exception(?Pattern, :ProtectedGoal, :Handler)* and *raise_exception(+Exception)* of *SICStus Prolog* [AAB+93]. The main choices underlying the language definition are borrowed from [dBdV89], where the abstract backtracking language $\mathcal{B}$ is defined as a uniform version of Prolog with cut. In particular, we do not deal with the logical aspects of programs, such as variables bindings: here inter-goal and intra-goal communication, usually realized by means of shared variables, are obtained through a notion of *store*, i.e. a global structure containing all the information representing the state of the computation. An oversimplified notion of store can be thought of as a collection of cells containing elementary values, such as numbers, characters, boolean values, etc. The language $\mathcal{L}$ is defined as follows:

Definition 2.1

i) Syntactic Categories:

$P \in Program$
$D \in Declaration$
$G \in Goal$
$X \in Proc$
$a \in Action$
$n \in Numeral$

ii) Abstract Syntax:

$P ::= D?G$

$D ::= X : -G \mid D.D$

$G ::= G, G \mid G; G \mid X \mid true \mid fail \mid a \mid on\_exc(X, G, G) \mid raise\_exc(X)$

$a ::= incr \mid decr \mid (equal?\, n)$

Notational conventions: In this paper A, B, C, G ..., range over *Goal*; X, Y, Z ..., range over *Proc*; a, b, t ..., range over *Action*, n, m ... range over *Numeral*. All symbols can appear indexed. The symbol $\equiv$ denotes the syntactic identity of goals. The priorities of the conjunction and disjunction operators are defined as usual, that is the goal $A, B; C$ must be read as $(A, B); C$.

Throughout the paper we will implicitly assume the procedural interpretation for logic programs [Kow74]; in particular we will use the notions of procedure name, exception name and procedure call.

The first BNF definition of the abstract syntax characterizes programs; they consist of a declaration followed by a goal. A declaration is a procedure name declaration, i.e. an expression that binds a procedure name to its body, or (recursively) a juxtaposition of two declarations. Different elementary declarations define different procedure names: as usual, alternative behaviours can be associated to a given procedure name by means of the disjunction construct (;). Goals can be simple goals, such as *true* and *fail* (which always succeeds and fails respectively), single procedure calls, actions[2], or compound goals. A compound goal can be a conjunction of goals, a disjunction of goals, or one of the two built-in predicates $on\_exc(X, G_1, G_2)$ and $raise\_exc(X)$ which are introduced to deal with exceptions and error conditions. Most of the compound goals have a straightforward meaning; in what follows we give an explanation of the goals $on\_exc(X, G_1, G_2)$ and $raise\_exc(X)$.

2.1 The Compound Goals $on\_exc(X, G_1, G_2)$ and $raise\_exc(X)$

The goal $on\_exc(X, G_1, G_2)$ dynamically defines an exception condition and its exception handler. The first argument of the exception declaration is a procedural name playing the role of an exception name, while the second and the third arguments are goals. While the procedure names are statically bound to their meaning (i.e. to the the meaning of their body), and clause declarations define a global environment accessible by any goal evaluation,

[2]As we have already pointed out, only control aspects are represented in the language $\mathcal{L}$. Therefore actions can be any elementary operations accessing or modifying the store; the actions we have defined (number increment, number decrease and test) are very simple, and they only play a role in the examples

the exception names are dynamically bound: the scope of X is *local* to the evaluation of the goal $on\_exc(X, G_1, G_2)$.

When $on\_exc(X, G_1, G_2)$ is evaluated, X is stacked on the current environment, i.e. the stack containing the (static) global environment which defines the procedure names, enriched with the exception names declared in the ancestor goals. After entering the stack, the name X becomes available to the following goal evaluations, and it remains available until the evaluation of $on\_exc(X, G_1, G_2)$ is completed.

After stacking X, $on\_exc(X, G_1, G_2)$ evaluates the goal G_1 in the extended environment. If the evaluation of G_1 succeeds, then $on\_exc(X, G_1, G_2)$ does; analogously, the failure of G_1 implies the failure of $on\_exc(X, G_1, G_2)$. The goal G_1 is *protected*: only G_1 and the goals recursively called by G_1 can access the exception name X. During the evaluation of the protected goal, a call to the procedure $raise(X)$ can occur. The evaluation of $raise(X)$ accesses the environment stack, looking for the first occurrence of the exception name X, then evaluates G_2, the exception handler, in an environment obtained from the current one by popping out X and all the exception names following it, and in the same store in which the exception has been declared by $on\_exc(X, G_1, G_2)$. Hence all the remaining subgoals in G_1 are discarded, and G_2 becomes the current goal in the environment and in the store where the last call to $on\_exc(X, G_1, G_2)$ occurred. This simply means that the computation from the $on\_exc(X, G_1, G_2)$ call on is rolled back, by restoring its definition context.

Notice that, during the evaluation of the protected goal G_1, more than one X can appear in the stack (this is the case when G_1 recursively calls $on\_exc(X, G_1, G_2)$: in this case the goal $raise\_exc(X)$ chooses the innermost one (i.e. the first one encountered when the stack is scanned from top to bottom). If, during the evaluation of the protected goal G_1, a call to $raise(Y)$ is found and Y is not accessible from G_1, an error condition arises, and the overall evaluation fails.

The following example shows, in abstract, the behaviour of the explicit control constructs. Let $P = D?A$ be a simple program matching the following declarations:

$A : -on\_exc(X, B_1, H), C.$
$B_1 : -B_2, B_3.$
$B_3 : -(equal?n), raise\_exc(X); B_1.$
$B_2 : -\ldots$
$H : -\ldots$
$C : -\ldots$

When the goal A is evaluated, $on\_exc(X, B_1, H)$ is called: the exception name X is stacked, and the goal B_1 is called in the resulting environment; the goal $on\_exc(X, B_1, H)$ will succeed if B_1 does. The goal B_1 first calls B_2. We did not specify the definition of B_2; let us suppose that it always succeeds after modifying the store by means of some actions. After B_2, B_1 calls B_3. The goal B_3 is a disjunction, therefore its first disjunct is explored. A test action $(equal?n)$ is called: if the test fails, the first disjunct of B_3 fails. In this case the second disjunct in the body of B_3 is evaluated, that is B_1 is recursively evaluated. If $(equal?n)$ succeeds then $raise\_exc(X)$ is called, i.e. the explicit exception mechanism is activated.

The interpreter accesses the stack, looking for the exception name X. Since X has been previously stacked, its associated exception handler, i.e. the goal H, is evaluated. The environment and the store in which the goal H is evaluated will be those in which the exception has been defined (in particular, H cannot access the exception name X). The success of H will transfer the control to C, while its failure would cause the overall goal A to fail.

3 Denotational Semantics with Continuations for the Language $\mathcal{L}$

Standard denotational semantics define *what* a program computes by formally defining the mathematical function denoted by the program. Usually the meaning of a statement is an abstract value in a Scott domain [Sco76], defined by means of a *semantic function*, expressed in λ-notation [Bar84], The semantic function takes as arguments the statement to be interpreted and the *semantic context* of interpretation, usually consisting of an environment and a store. The denotation of a program is given in terms of the meanings of its components.

Standard denotational semantics [Sch86] do not specify *how* the semantic functions have to be computed, therefore they are not well suited to model control paradigms, such as unconditional jumps, block exits, or exception handling. Aiming at characterizing control mechanism aspects, denotational semantics with continuations can be used.

A continuation is just a function denoting "the rest of the computation". To model the behaviour of logic programs two continuations are needed, a *success continuation* (by means of which the control mechanism of the AND construct is formalized) and a *failure continuation* (to deal with the OR construct).

Like in standard denotational semantics, the interpretation function takes as arguments the statement to be interpreted (enclosed in $[\![\;]\!]$) and the semantic context of interpretation. In the case of denotational semantics with continuations the semantic context consists of four elements: an environment (which binds procedure names to their meaning), a store (which models communications among goals) and two continuations, to represent the control flow. The first continuation is the success continuation, which is a function denoting the way the computation will proceed if the interpreted statement ends with success. The second one is the failure continuation, that is a function modelling the evolution of the computation in case of failure of the interpreted statement. Thanks to the failure continuation the *backtracking* mechanism can be modelled.

Informally, we can say that the success and the failure continuations represent the content of the control stacks (containing the current goal and the suspended goals respectively) in any operational semantics for $\mathcal{L}$. Notice that only goals in conjunctions are allowed to communicate, while disjuncted goal are completely independent from each other, and do not communicate at all.

The central idea of a denotational semantics with continuations is to reduce the interpretation of $\mathcal{L}$ to the interpretation of Plotkin's λ_v-calculus [Plo75], obtained by restricting the β-rule in the λ-calculus as follows:

$$(\lambda x.\, M)N \rightarrow_{\beta_v} M[N/x] \text{ provided } N \text{ is a value}$$

In what follows we give the semantic algebra and the semantic functions that characterize our denotational semantics with continuations for the language $\mathcal{L}$.

Definition 3.1 The *Semantic Algebras* are defined as follows:

i) $\mathit{Den\_Value} = \mathit{Store} + \mathit{String}$

ii) $\mathit{Answer} = \mathit{Den\_Value} \cup \{\bot\}$

iii) $\mathit{String} = \{\mathit{failure}, \mathit{error}\}$

iv) $\sigma \in \mathit{Store} = \mathit{Nat}$
$\sigma_0 = \underline{0}$
$\mathit{plus\_one} : \mathit{Store} \rightarrow \mathit{Store}$
$\mathit{plus\_one} = \lambda\sigma.\, \sigma + \underline{1}$
$\mathit{minus\_one} : \mathit{Store} \rightarrow \mathit{Store}$
$\mathit{minus\_one} = \lambda\sigma.\, \sigma - \underline{1}$
$\mathit{eq?} : \mathit{Store} \rightarrow \mathit{Nat} \rightarrow \mathit{Bool}$
$\mathit{eq?} = \lambda\sigma\underline{n}.\, \mathit{if}\ \sigma = \underline{n}\ \mathit{then}\ \underline{t}\ \mathit{else}\ \underline{f}$

v) $\xi \in \mathit{Fail\_cont} = \mathit{Store} \rightarrow \mathit{Answer}$
$\xi_0 = \lambda\sigma.\, \mathit{failure}$

vi) $\kappa \in \mathit{Succ\_cont} = \mathit{Den\_Env} \rightarrow \mathit{Fail\_cont} \rightarrow \mathit{Fail\_cont}$
$\kappa_0 = \lambda\xi\sigma.\, \sigma$

vii) $\phi \in \mathit{Strategy} = \mathit{Succ\_cont} \rightarrow \mathit{Succ\_cont}$

viii) $\rho \in \mathit{Den\_Env} = \mathit{Proc} \rightarrow \mathit{Strategy}$
$\rho_0 = \lambda X \kappa\rho\xi\sigma.\, \xi\sigma$
$\rho[X \leftarrow \phi] : \mathit{Den\_Env}$
$\rho[X \leftarrow \phi](Y) = \mathit{if}\ X = Y\ \mathit{then}\ \phi\ \mathit{else}\ \rho(Y)$
$\mathcal{D}om(\rho)$ is the domain of ρ

Den_Value is the set of denotable values, that is the set of values which can result from any statement evaluation. *Answer* is the set of outputs possibly returned by the interpretation of a program. A program can return either a store or a *failure/error* message; we have chosen the lifted domain of *Den_Value* in order to model nonterminating programs too.

A failure continuation is a unary function mapping stores, into answers. Intuitively this function provides the information which is needed when a failure occurs, to throw away the current store and to roll back the control flow to a "safe" point.

A success continuation is a function which manipulates failure continuations: when the current goal is satisfied, the success continuation communicates to the next goal to be evaluated the current failure continuation; in such a failure continuation the instantaneous description which will be restored by the next backtracking point is codified.

A strategy is a function that modifies the flow of control, by installing or discarding instantaneous descriptions of the current state of interpretation. A stragegy can be stored in a structure, the environment (*Den_Env*), that contains all the strategies bound to procedure and exception names. Examples of simple strategies are clause bodies and exception handlers.

Definition 3.2 The *Semantic Functions* are defined as follows:

i) $\mathcal{P}[\![\bullet]\!] : Program \rightarrow Answer$

$$\mathcal{P}[\![D?G]\!] \equiv \mathcal{G}[\![G]\!]\kappa_0(\mathcal{D}[\![D]\!]\rho_0)\xi_0\sigma_0$$

ii) $\mathcal{D}[\![\bullet]\!] : Declaration \rightarrow Den\_Env \rightarrow Den\_Env$

$$\begin{aligned}
\mathcal{D}[\![D_1, D_2]\!] &\equiv \lambda\rho.\,(\mathcal{D}[\![D_2]\!] \circ \mathcal{D}[\![D_1]\!])\rho \\
\mathcal{D}[\![I : -G]\!] &\equiv \lambda\rho.\,\rho[I \leftarrow (\lambda\kappa\rho\xi\sigma.\,\mathcal{G}[\![G]\!]\kappa\rho\xi\sigma)]
\end{aligned}$$

iii) $\mathcal{G}[\![\bullet]\!] : Goal \rightarrow Succ\_cont \rightarrow Den\_Env \rightarrow Fail\_cont \rightarrow Store \rightarrow Answer$

$$\begin{aligned}
\mathcal{G}[\![G_1, G_2]\!] &\equiv \lambda\kappa\rho\xi\sigma.\,\mathcal{G}[\![G_1]\!]\rho(\lambda\rho'\xi'\sigma'.\,\mathcal{G}[\![G_2]\!]\rho'\kappa\xi'\sigma')\xi\sigma \\
\mathcal{G}[\![G_1; G_2]\!] &\equiv \lambda\kappa\rho\xi\sigma.\,\mathcal{G}[\![G_1]\!]\rho\kappa(\lambda\sigma'.\,\mathcal{G}[\![G_2]\!]\rho\kappa\xi\sigma)\sigma \\
\mathcal{G}[\![X]\!] &\equiv \lambda\kappa\rho\xi\sigma.\,\rho(X)\kappa\rho\xi\sigma \\
\mathcal{G}[\![incr]\!] &\equiv \lambda\kappa\rho\xi\sigma.\,\kappa\xi(plus_one\,\sigma) \\
\mathcal{G}[\![decr]\!] &\equiv \lambda\kappa\rho\xi\sigma.\,\kappa\xi(minus_one\,\sigma) \\
\mathcal{G}[\![(equal?\,n)]\!] &\equiv \lambda\kappa\rho\xi\sigma.\,if\ (eq?\sigma\underline{n})\ then\ \kappa\rho\xi\sigma\ else\ \xi\sigma \\
\mathcal{G}[\![true]\!] &\equiv \lambda\kappa\rho\xi\sigma.\,\kappa\rho\xi\sigma \\
\mathcal{G}[\![fail]\!] &\equiv \lambda\kappa\rho\xi\sigma.\,\xi\sigma \\
\mathcal{G}[\![on_exc(X, G_1, G_2)]\!] &\equiv \lambda\kappa\rho\xi\sigma.\,\mathcal{G}[\![G_1]\!]\kappa\rho[X \leftarrow (\lambda\kappa'\rho'\xi'\sigma'.\,\mathcal{G}[\![G_2]\!]\kappa\rho\xi\sigma)]\xi\sigma \\
\mathcal{G}[\![raise_exc(X)]\!] &\equiv \lambda\kappa\rho\xi\sigma.\,if\ X \in \mathcal{D}om(\rho)\ then\ \rho(X)\kappa\rho\xi\sigma else\ error
\end{aligned}$$

Each function takes four arguments: a success continuation κ, an environment ρ, a failure continuation ξ and a store σ. The tuple $\kappa\rho\xi\sigma$ can be regarded as an *instantaneous description* of the state of the interpretation: when we give a meaning to a logic language, both the current binding of logic variables (here modelled by the store) and the contents of the control stacks specifying the current and the suspended goals, are denoted by the tuple $\kappa\rho\xi\sigma$.

The function $\mathcal{P}$ gives a meaning to a program $D?G$ by interpreting the goal G in the environment resulting from the evaluation of the declaration D; the evaluation of D is realized in the basic continuations κ_0, ξ_0 and in the empty store σ_0. The function $\mathcal{D}$ evaluates the declarations one at a time, and produces as output an environment in which each declared name has been bound to its meaning. When a declaration $X : -G$ is evaluated, the current environment is modified to also include the binding of the procedure name X with the λ-term $(\lambda\kappa\rho\xi\sigma. \mathcal{G}[\![G]\!]\kappa\rho\xi\sigma)$, representing the evaluation of the goal G in the current instantaneous description $\kappa\rho\xi\sigma$.

The function $\mathcal{G}$ evaluates a goal to produce an answer. Let us analyze the most interesting cases of this functions.

If the current goal is the conjunction of goals (G_1, G_2), the goal G_1 is first evaluated in the given ρ, ξ, σ, but in a new success continuation κ'. The continuation κ' specifies how the overall computation will proceed after the evaluation of G_1: the interpretation of G_1 must be followed by the interpretation of G_2, which in turns must be performed in the instantaneous description returned by the interpretation of G_1.

If the current goal is a procedure name X, the strategy bound to X in the given environment ρ is applied to the instantaneous description $\kappa\rho\xi\sigma$.

Basic actions a require to modify or check the store argument σ occurring in the instantaneous description $\kappa\rho\xi\sigma$; if the action is a test and the test fails, the current failure continuation must be applied.

The interpretation of the *true* construct simply applies the instantaneous description $\kappa\rho\xi\sigma$, while the interpretation of *fail* discards the current success continuation κ and applies the failure continuation ξ to the store σ.

The exception handling goal $on\_exc(I, G_1, G_2)$ deals with the declaration of an exception: it evaluates G_1 in a new instantaneous description, obtained from the given one by increasing the given environment ρ with the binding of X to the λ-term $(\lambda\kappa'\rho'\xi'\sigma'. \mathcal{G}[\![G_2]\!]\kappa\rho\xi\sigma)$. The meaning of this term is: "evaluate the goal G_2 in the previously catched description $\kappa\rho\xi\sigma$".

The interpretation of the $raise\_exc(X)$ construct deals with the raising of exceptions; if the exception name X has been declared in the environment ρ, $raise\_exc(X)$ discards the current instantaneous description, and it evaluates the exception handler G_2 in the instantaneous description $\kappa\rho\xi\sigma$ previously catched (the installed description is the one catched when the exception was declared). Otherwise it raises a system exception by returning a trivial error message.

3.1 Formal Equivalence Between Programs

The denotational semantics with continuations introduced in the previous sections can be used to state an operational equivalence property between programs using $on\_exc(X, G, G)$ and $raise\_exc(X)$. Intuitively, two goals are operationally equivalent when they can be replaced to each other in any program without affecting the final result of the program itself. We state two equivalences between goals in which the exception handling mechanism is used.

If an exception is declared and the protected goal unconditionally raises the exception, the exception handler is immediately called. The goal $\mathcal{G}[\![\ on\_exc(X, raise\_exc(X), G)\ ]\!]$ is operationally equivalent to G. Formally:

$$\begin{aligned}
\mathcal{G}[\![\ on_exc(X, raise_exc(X), G)\ ]\!]\kappa\rho\xi\sigma &= \mathcal{G}[\![\ raise_exc(X)\ ]\!]\kappa\rho[X \leftarrow (\lambda\kappa'\rho'\xi'\sigma'.\mathcal{G}[\![\ G\ ]\!]\kappa\rho\xi\sigma)]\xi\sigma \\
&= (\lambda\kappa'\rho'\xi'\sigma'.\mathcal{G}[\![\ G\ ]\!]\kappa\rho\xi\sigma)\kappa\rho[X \leftarrow (\lambda\kappa'\rho'\xi'\sigma'.\mathcal{G}[\![\ G\ ]\!]\kappa\rho\xi\sigma)]\xi\sigma \\
&= \mathcal{G}[\![\ G\ ]\!]\kappa\rho\xi\sigma
\end{aligned}$$

If an exception is declared and the protected goal never raises the declared exception, the exception handler is never called: The goal $\mathcal{G}[\![\ on\_exc(X, G, \_)\ ]\!]$ is operationally equivalent to G, provided G does not call $raise\_exc(X)$. Formally:

$$\begin{aligned}
\mathcal{G}[\![\ on_exc(X, G, _)\ ]\!]\kappa\rho\xi\sigma &= \mathcal{G}[\![\ G\ ]\!]\kappa\rho[X \leftarrow (\lambda\kappa'\rho'\xi'\sigma'.\mathcal{G}[\![\ _\ ]\!]\kappa\rho\xi\sigma)]\xi\sigma \\
&= \mathcal{G}[\![\ G\ ]\!]\kappa\rho\xi\sigma
\end{aligned}$$

4 Simulating the cut operator in the language $\mathcal{L}$

The cut operator (!) is the only construct that allows the control flow in a standard Prolog to be explicitly modified, by pruning the search space of the SLD procedure. We have not included the cut operator in the definition of the uniform language $\mathcal{L}$, and up to now we have not taken it into any account. Our choice is due to the fact that adding the cut operator to $\mathcal{L}$ would not increase the expressive power of the language, since its behaviour can be quite naturally simulated by means of an appropriate use of the (more powerful) exception handling mechanism.

In order to obtain the behaviour of the cut operator appearing in the body of a program clause, it is sufficient to replace each clause $A : -B$ such that B contains the cut, with the clause

$$A :- on\_exc(!, B, fail)$$

add to the program the following definition

$$! :- true; raise\_exc(!)$$

This transformation makes it clear that the cut operator is a locally declared exception, acting within the scope of the body of A. The first evaluation of cut always succeeds: if, due to the backtracking mechanism, the second disjunct of its definition is evaluated, then an exception is raised. If this is the case, then the exception handler (*fail*) is called, causing A to fail.

5 Conclusions

In this paper we have defined a denotational semantics with continuations to model the flow of control in a logic language. The considered language is a *uniform* logic language $\mathcal{L}$ [dBKM+86, dB88, dBdV89] which can be thought of as a fragment of Prolog with exception handling. In particular $\mathcal{L}$ contains two built-in constructs, $on\_exc(I, G_1, G_2)$ and $raise\_exc(I)$ which are specifically designed to deal with exception handling, and whose behaviour essentially corresponds to the way the analogous built-ins *on_exception(?Pattern, :ProtectedGoal, :Handler)* and *raise_exception(+Exception)* of *SICStus* Prolog are interpreted.

The approach adopted in this paper is quite similar to some of the denotational characterizations we have mentioned in the Introduction. The nearest one is the denotational semantics with continuations proposed by Brisset and Ridoux [BR93]; they extend λ-Prolog with new built-in constructs, which provide new control mechanisms based on the explicit replacement of continuations.

Brisset and Ridoux's characterization differs from ours in some essential aspects. First, the considered language: Brisset and Ridoux have introduced continuations in λ-Prolog, hence they have taken into account implication goals and higher order syntax. Their characterization also models some logical aspects such as variable unification, while we deal with a logic language focused on control aspects.

The semantics by Brisset and Ridoux characterizes the flow of control by means of three continuation functions, the success continuation, the failure continuation and the cut continuation; due to our choice of avoiding the explicit definition of the cut operator, the success continuation and the failure one are sufficient to model the flow of control in our language.

We plan to further develop the work presented in this paper; in particular we are interested in analyzing different control mechanism which can be easily modeled by means of our exception handling constructs, and in extending the semantics to model logical aspects such as unification.

Acknowledgements: We are very grateful to Laura Giordano and Alberto Martelli for their helpful comments and suggestions.

Luigi Liquori has been partially supported by HCM project No. ERBCHRXCT920046 "Typed Lambda Calculus", while Maria Luisa Sapino has been partially supported by CNR - Progetto Finalizzato "Sistemi Informatici e Calcolo Parallelo" under grant n. 91.00893.PF69.

References

[AAB+93] J. Andersson, S. Andersson, K. Boortz, M. Carlsson, H. Nillson, T. Sjoland, and J. Widen. *SICStus Prolog User's Manual, version 2.1 ♯ 8*, 1993.

[AB87] B. Arbab and D. M. Berry. Operational and denotational semantics of Prolog. *Journal of Logic Programming*, 4:309–329, 1987.

[App92] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[Bar84] H.P. Barendregt. *The λ-calculus: its syntax and semantics.* North-Holland, 1984. revised edition.

[BBF93] A. Bossi, M. Bugliesi, and M. Fabris. A new fixpoint semantics for prolog. In David S. Warren, editor, *Proc. of the 10th International Conference on Logic Programming*, pages 374 –389. MIT Press, 1993.

[BCGL93] R. Barbuti, M. Codish, R. Giacobazzi, and G. Levi. Modelling Prolog control. *Journal of Logic and Computation*, 3(6):579–603, 1993.

[BR93] P. Brisset and O. Ridoux. Continuations in λ-Prolog. In David S. Warren, editor, *Proc. of the 10th International Conference on Logic Programming*, pages 27 –43. MIT Press, 1993.

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

[dBdV89] A. de Bruin and E. P. de Vink. Continuation semantics for Prolog with cut. In *Proc. TAPSOFT-89, LNCS*, pages 178–192, Berlin, 1989. Springer-Verlag.

[dBKM+86] J. W. de Bakker, J. N. Kok, J.-J.Ch. Meyer, E.-R Olderog, and J.I Zucker. Contrasting themes in the semantics of imperative concurrency. In J. W. de Bakker, W. P. de Rover, and G. Rozenberg, editors, *Current Trends in Concurrency: Overviews and Tutorials*, pages 51–121. Springer, 1986.

[dV88] E. P. de Vink. Equivalence of an operational and a denotational semantics for a Prolog-like language with cut. Technical report, Free University of Amsterdam, Amsterdam, 1988.

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

[Kow74] R.A. Kowalski. Predicate logic as a programming language. In *proc. IFIP 74*, pages 569–574, Amsterdam, 1974.

[Kow79] R. A. Kowalski. Algorithm= logic + control. *Communications of the ACM*, 22(7):424–436, 1979.

[MH84] C. Mellish and S. Hardy. Integrating Prolog in the POPLOG environment. In J. A. Campbell, editor, *Implementations of Prolog Programming.* Ellis Horwood, 1984.

[NF89] T. Nicholson and N. Foo. A denotational semantics for Prolog. *ACM Transactions on Programming Languages and Systems*, 11(4):650–665, 1989.

[NM88] G. Nadathur and D. Miller. An overview of λ-prolog. In *Proc. Symp. Logic Programming*, pages 810–827, Seattle, 1988.

[Plo75] G. Plotkin. Call by name, call by value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[RC86] J. Rees and W. Clinger. *Revised^3 Report on the Algorithmic Language Scheme, SIGPLAN Notices, vol. 21, n. 12*, 1986.

[Sch86] D. A. Schmidt. *Denotational semantics: a methodology for language development.* Allyn and Bacon Inc, 1986.

[Sco76] D. S. Scott. Data types as lattices. *SIAM Journal of Computing*, 5:522–587, 1976.

[ST89] T. Sato and H. Tamaki. Existential continuations. *New Generation Computing*, (6):421–438, 1989.

[TB90] P. Tarau and M. Boyer. Elementary logic programs. In P. Deransart and J. Maluszynski, editors, *Proc. International Workshop on Programming languages implementation and Logic Programming - LNCS 456*. Springer-Verlag, 1990.

[Ued87] K. Ueda. Making exhaustive search programs deterministic: Part II. In J.L Lassez, editor, *Proc. 4-th International Conference on Logic Programming*, Melbourne, 1987. MIT Press.