[9] F. Toni. *Abductive logic programming.* PhD Thesis, Imperial College, London, 1995.

[10] F. Toni and R.A. Kowalski. Reduction of abductive logic programs to normal logic programs. To appear in *Proc. ICLP*, 1995.

Appendix

First, we define partial orders $>$ and $\gg$ between (abductive and consistency) derivations, by appropriately modifying the definitions in [3]. Given an abductive derivation α: $(G_1, \Delta_1^\alpha), \ldots, (\Box, \Delta_n^\alpha)$, a consistency derivation κ: $(L, \Delta_1^\kappa), \ldots, (\emptyset, \Delta_m^\kappa)$, where $n, m \geq 1$, and an assumption A,

$\kappa > \alpha$ via A at step i, where $1 \leq i < m$, if A is selected in the chosen goal in S_i and a step of type (C4)(i) takes place, invoking α,
i.e. $S_i = \{\leftarrow L', \mathcal{Q}\} \cup S'$, $\Re$ selects L' in $\leftarrow L', \mathcal{Q}$, $L' = A$,
$L' \notin \Delta_i^\kappa$, $G_1 = \leftarrow \overline{A}$, $\Delta_1^\alpha = \Delta_i^\kappa$, $S_{i+1} = S'$ and $\Delta_{i+1}^\kappa = \Delta_n^\alpha$;

$\alpha > \kappa$ via A at step i, where $1 \leq i < n$, if A is selected in G_i and a step of type (A3) takes place, invoking κ,
i.e. $G_i = \leftarrow L', \mathcal{Q}$, $\Re$ selects L' in G_i, $L' = A$, $L' \notin \Delta_i^\alpha$,
$L = A$, $\Delta_1^\kappa = \Delta_i^\alpha \cup \{A\}$, $G_{i+1} = \leftarrow \mathcal{Q}$ and $\Delta_{i+1}^\alpha = \Delta_m^\kappa$;

$\gg$ is the transitive closure of $>$.

Let α_0 denote the successful abductive derivation $(\leftarrow \mathcal{Q}, \emptyset)$ to $(\Box, \Delta)$ as in theorem 4.1. By definition of successful abductive derivation, α_0 is finite and there exists a finite number of successful and finite derivations δ such that $\alpha_0 \gg \delta$.

Lemma 6.1 For each derivation δ: $(U_1, \Delta_1), \ldots, (U_n, \Delta_n)$ with $\alpha_0 \gg \delta$:
$\Delta_i \subseteq \Delta_j$, for all $i, j = 1, \ldots, n$ with $i < j$, and
$\Delta_i \subseteq \Delta$, for all $i = 1, \ldots, n$.

This lemma can be easily proved by induction on the number of derivations δ' such that $\delta \gg \delta'$ (see [9]).

Let $\mathcal{HB}_{not}$=$\{not\ A | A$ is an atom in the Herbrand base of $\mathcal{P}\}$. The **assumption set of an abductive derivation** α: $(G_1, \Delta_1), \ldots, (G_n, \Delta_n)$ is the set $ass(\alpha) = \{A \mid \exists G_i$ such that $A \in G_i \cap (\mathcal{AB}_{KM} \cup \mathcal{HB}_{not})\}$. (We interpret goals, i.e. conjunctions of literals, as sets of literals.)

Lemma 6.2 Given an abductive derivation α:$(\leftarrow \mathcal{Q}', \Delta_1), \ldots, (\Box, \Delta_n)$ with $\alpha_0 \gg \alpha$,

1. $ass(\alpha) \subseteq \Delta$, and
2. $\mathcal{P} \cup ass(\alpha) \vdash \mathcal{Q}'$.

Proof:

1. By definition, each $A \in ass(\alpha)$ occurs as a subgoal in some G_i in α. By definition of abductive derivation, for each such A there exists Δ_i such that $A \in \Delta_i$. Therefore, directly from Lemma 6.1, each such A is in Δ. As a consequence, $ass(\alpha) \subseteq \Delta$.
2. Directly from the definition of abductive derivation, $\mathcal{P} \cup ass(\alpha) \vdash_{SLD} \mathcal{Q}'$. By soundness of SLD resolution, $\mathcal{P} \cup ass(\alpha) \vdash \mathcal{Q}'$.

Corollary 6.1 Given an abductive derivation α: $(\leftarrow L, \Delta_1), \ldots, (\Box, \Delta_n)$ with $\alpha_0 \gg \alpha$ and L an atom or the negation of an abducible atom, Δ attacks the assumption $\overline{L}$ via case (i) or case (ii).

Let the **assumption set of a consistency derivation** κ: $(L, \Delta_1), \ldots, (S_n, \Delta_n)$ be the set $ass(\kappa) = \{ass(\mathcal{B}) \mid \mathcal{B}$ is a branch of the tree $\mathcal{T}(\kappa)$ corresponding to $\kappa\}$, where the **assumption set of a branch** $\mathcal{B}$ in $\mathcal{T}(\kappa)$ is the set $ass(\mathcal{B}) = \{A \mid \exists G_i$ in $\mathcal{B}$ such that $A \in G_i \cap (\mathcal{AB}_{KM} \cup \mathcal{HB}_{not})\}$, and the **tree $\mathcal{T}(\kappa)$ corresponding to κ** is $\mathcal{T}_n$, such that, assuming that for each $i = 1, \ldots, n$, $S_i = \{\leftarrow L_i, \mathcal{Q}_i\} \cup S_i'$, and for each $i = 1, \ldots, n-1$, S_{i+1} is obtained by choosing the goal $\leftarrow L_i, \mathcal{Q}_i$ and by selecting the literal L_i in it:
$\mathcal{T}_0$ consists only of the root, which is the empty goal;
for each $i \geq 1$, given $\mathcal{T}_i$ and S_{i+1}, then $\mathcal{T}_{i+1}$ is obtained as follows:

1. if L_i is an atom and case (C1) applies, then $\mathcal{T}_{i+1}$ is $\mathcal{T}_i$ where all resolvents of $\leftarrow L_i, \mathcal{Q}_i$ on L_i in $\mathcal{P}$ are children of $\leftarrow L_i, \mathcal{Q}_i$; if there is no such resolvent, failure is the only child of $\leftarrow L_i, \mathcal{Q}_i$;
2. if L_i is an assumption and $L_i \in \Delta_i$, i.e. case (C2) applies, then $\mathcal{T}_{i+1}$ is $\mathcal{T}_i$ where $\leftarrow \mathcal{Q}_i$ is the only child of $\leftarrow L_i, \mathcal{Q}_i$;
3. if L_i is an assumption and $\overline{L_i} \in \Delta_i$, i.e. case (C3) applies, then $\mathcal{T}_{i+1}$ is $\mathcal{T}_i$ where failure is the only child of $\leftarrow L_i, \mathcal{Q}_i$;
4. if L_i is an assumption, $L_i \notin \Delta_i$ and $\overline{L_i} \notin \Delta_i$, i.e. case (C4) applies, then, if there exists an abductive derivation from $(\leftarrow \overline{L_i}, \Delta_i)$ to $(\Box, \Delta')$ (case (i)), then $\mathcal{T}_{i+1}$ is $\mathcal{T}_i$ where failure is the only child of $\leftarrow L_i, \mathcal{Q}_i$; otherwise (case (ii)), $\mathcal{T}_{i+1}$ is $\mathcal{T}_i$ where $\leftarrow \mathcal{Q}_i$ is the only child of $\leftarrow L_i, \mathcal{Q}_i$.

Lemma 6.3 Given a consistency derivation $\kappa : (L, \Delta_1), \ldots, (\emptyset, \Delta_n)$ with $\alpha_0 \gg \kappa$, $ass(\kappa) \supseteq \{\Delta' | \Delta'$ is a minimal (with respect to set inclusion) attack against $L\}$.
Proof: By definition of consistency derivation, since all resolvents are considered at any step of kind (C1), $ass(\kappa)$ contains all sets of assumptions Δ' such that
$\mathcal{P} \cup \Delta' \vdash_{SLD} \overline{L}$, and
$\mathcal{P} \cup \Delta' \vdash_{SLD} L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m$, for some $\neg[L_1 \wedge \ldots \wedge L_m]$ in $\mathcal{I}_{KM}$ with $L_i = L$ for some $i = 1, \ldots, m$.
But note that, for any (definite) program $\mathcal{P}'$ and query $\mathcal{Q}'$, if there exists a minimal

subset $\mathcal{P}''$ of $\mathcal{P}'$ such that $\mathcal{P}'' \vdash \mathcal{Q}'$ then $\mathcal{P}' \vdash_{SLD} \mathcal{Q}'$. As a consequence, directly from the definition of (minimal) attack, each minimal attack Δ' against L belongs to $ass(\kappa)$.

Lemma 6.4 For each consistency derivation κ such that $\alpha_0 \gg \kappa$:

1. there exists an abductive derivation α such that $\alpha > \kappa$;
2. for any abductive derivation α such that $\alpha > \kappa$, via L at step i, for each branch $\mathcal{B} \in \mathcal{T}(\kappa)$, if $ass(\mathcal{B})$ is a minimal attack against L then for some $L' \in ass(\mathcal{B})$, (a) Δ attacks L' via case (i) or case (ii), and (b) $L' \notin \Delta$.

Proof: Part 1. is obvious. Let us prove part 2. To prove (a), note that each $\mathcal{B} \in \mathcal{T}(\kappa)$ terminates with failure. If $ass(\mathcal{B})$ (minimally) attacks L, then, by completeness of SLD resolution, failure cannot be generated via case (C1). Therefore, failure must be generated via case (C3) or (C4)(i), due to the selection of an assumption L' at some step j. (Note that necessarily $L' \in ass(\mathcal{B})$.) In the first case Δ attacks L' via case (i) or via case (ii) trivially holds. In the second case, there must exist an abductive derivation α' such that $\kappa > \alpha'$ via L'. By definition, α' is a derivation from $(\leftarrow \overline{L'}, \Delta_j)$, where Δ_j is the set of assumptions accumulated at step j in κ. By corollary 6.1, Δ attacks L' via case (i) or via case (ii).

To prove (b), suppose by contradiction that $L' \in \Delta$. Necessarily, $L' \notin \Delta_j$, otherwise a step of type (C2) would have taken place. Therefore, there must exist an abductive derivation α'' such that $L' \in ass(\alpha'')$ and $\alpha' \gg \alpha''$, and a consistency derivation κ' such that $\alpha'' > \kappa'$ via L'. Since Δ attacks L' (see (a)), by completeness of SLD resolution there must exist a branch $\mathcal{B} \in \mathcal{T}(\kappa')$ such that $ass(\mathcal{B}) = \Delta'$ for some $\Delta' \subseteq \Delta$, and $\mathcal{B}$ terminates with $\square$. As a consequence, α_0 cannot terminate successfully. This gives a contradiction.

Lemma 6.5 For all sets of assumptions $\mathcal{A}$ and Δ in the assumption-based framework corresponding to a KM abductive logic program $\langle \mathcal{P}, \mathcal{AB}_{KM}, \mathcal{I}_{KM} \rangle$, the following statements are equivalent:

1. if $\mathcal{A}$ attacks Δ, then Δ attacks $\mathcal{A} - \Delta$;
2. if $\mathcal{A}$ is a minimal attack against Δ, then Δ attacks $\mathcal{A} - \Delta$.

Proof: 1. $\Rightarrow$ 2. trivially holds. To prove 2. $\Rightarrow$ 1., note that if $\mathcal{A}$ is a non-minimal attack against Δ, then there exists $\mathcal{A}' \subset \mathcal{A}$ such that $\mathcal{A}'$ is a minimal attack against Δ. Then, by 2., Δ attacks $\mathcal{A}' - \Delta$. Therefore, Δ attacks $\mathcal{A} - \Delta$.

Proof of theorem 4.1: Part 1. holds by lemma 6.2. Part 2. can be proved as follows: by lemma 6.5 we only need to consider minimal attacks against Δ. By definition of attack, for each (minimal) attack $\mathcal{A}$ against Δ, there must exist $L \in \Delta$ such that $\mathcal{A}$ attacks L. Moreover, by construction of Δ, for each such L, there must exist an abductive derivation α such that $\alpha_0 \gg \alpha$ and $L \in ass(\alpha)$, and a consistency derivation κ such that $\alpha > \kappa$ via L. By lemma 6.3, $\mathcal{A} \in ass(\kappa)$. By lemma 6.4 Δ attacks $\mathcal{A} - \Delta$ via case (i) or via case (ii). This concludes the proof.

Implementing Higher-Order Term-Rewriting for Program Transformation in λProlog *

Francesca Arcelli, Ferrante Formato
DIIIE– Universitá di Salerno
84084 Fisciano (Salerno), Italy.
Fax:+39-89-964194, Ph.: +39-89-964254
e-mail: arcelli@ponza.dia.unisa.it, formato@columbia.diima.unisa.it

Abstract

Higher-order programming languages, as λProlog can be used with success in implementing program transformation systems, exploiting the capability offered by higher-order unification and by the availability of λ-terms. Through these features many operations on programs can be naturally expressed as higher-order term-rewriting systems. In this paper we apply higher-order term-rewriting techniques in the context of program transformation using λProlog as metalevel language. In particular we give a theoretical setting and we propose an implementative solution to the execution of the *rewrite-by-lemma* and *fold* steps of a program transformation strategy proposed in [17].

We propose a solution involving algebraic specification for dealing with the rewrite-by-lemma steps, which encapsulates the complete knowledge of algebraic properties of a data-structure into a finite set of axioms. These steps are carried out through term-rewriting techniques, and we propose two different interpretational approches of them in λProlog. Moreover the execution of folding steps is performed through rewriting techniques in the setting of recursive program schemes.

Keywords: higher-order logic programming, program transformation, term-rewriting techniques, algebraic specifications.

1 Introduction

Several program transformation methods based on rewriting techniques have been proposed in the literature. One of the most famous is due to Burstall and Dar-

*This work has been partially supported by MURST 60%.

lington ([5]), where program transformations are expressed as rewriting systems. In [7] the connections between program transformations and compiled techniques are described where, through the application of suitable rewriting-steps (semantic-preserving) new programs can be derived.

In this context, different programming languages paradigms have been adopted. In particular, the uselfuness in using higher-order logic programming languages, such as λProlog [15], to specify and implement program transformation techniques, has been outlined for example in [9], [13]. The main advantages achieved in using this language are related to the declarative features of logic programming, to the higher-order intensional features of λ-terms for representing programs and to the availability of higher-order unification. Through the λ-abstraction built into λ-terms it is possible to easily represent quantification in formulas or abstraction in functional programs, and so many operations on formulas and programs can be naturally expressed as higher-order rewrite systems, as we will see in this paper.

The main goal of this research is the implementation of rewriting techniques using λProlog (we have used the MALI implementation of λProlog developed by [4]) for the automation of a particular program transformation strategy, called "higher- order generalization strategy" or "λ-abstraction strategy ([17]). Higher-order expressions and higher-order reasoning arise naturally in meta-level manipulation of program code. Exploiting modularity of λProlog we can symbolically rewrite the object-level program expressions through a rewrite-rule system originated by an algebraic specifications.

In this paper, to the extent of supplying the program transformation strategy with as much uniformity as possible, we decided to use the *Recursive Program Schemes* ([6]), which show how splitting a program into its control and its data-structures represents a powerful method for investigating on the general properties of programs. To specify these properties of data-types involved in programs, we use algebraic specifications and so we focus our attention on the way to implement them in a higher-order context.

Term-rewriting systems represent an adequate tool for execute algebraic specifications and a way to implement them is to translate algebraic specifications directly to λProlog programs. The translation from term-rewriting systems to λProlog programs benefits from the fact that higher-order unification, in connection with a decomposition of λ-terms is very successfull to describe the mechanism of subterm rewriting. In particular, through higher-order unification we apply transformation steps involving symbolic rewriting of different parts of the programs. The precise mathematical semantics of algebraic specifications can then serve as a basis for a formal verification of the translation schemes. In this way we describe an interpretational approach to term-rewriting using λProlog. We apply higher-order rewrite rules techniques for the manipulation of program expressions, where semantically safe rewrite rules are necessary for passing smoothly from one program to another, as suggested by [5], through fold/unfold style transformation steps.

How directly implement through λProlog rewriting techniques have been shown

by Heering in [10]. These techniques, although appealing for their simplicity of implementation, introduce spurious solutions due to higher-order unification. For this reason we developed quite a new approach to the implementation of rewriting techniques, based on the "context-closure" of a term.

Other related works, in the literature, on the definition and the application of higher-order extensions of rewriting techniques, are for example [11], [16], [8], [19]. The paper is organized through the following sections.

In section 2 we introduce algebraic specifications by which we specify the properties of data-types involved in programs, introducing modularity in the implementation of the program transformer. We then briefly introduce the principal features of the λ-generalization strategy and we show how fold/unfold transformational steps could be uniformly dealt as rewritings of program expressions, if such transformations are seen in the light of recursive program schemes. In section 3 we describe how through the interpretational approach of algebraic specifications coupled with the declarative power of a logic programming language, such as λProlog, it is possible to directly implement term-rewriting systems as a modular structure of the architectural design of the program tranformer, and we show how higher-order rewriting techniques are useful in automatic execution of folding steps. Two different implementations in λProlog of an algebraic specification of a *list of objects* are given, one exploiting higher- order unification for direct rewriting, and the other obtained through the transitive closure of a set of rewrite rules. Finally we conclude and briefly discuss some future research directions.

2 Using Algebraic Specifications and Recursive Program Schemes in Program Transformation

One of the major problems connected with strategies for program transformation is enlarging the range of their uniform applicability. Besides, since the software maintenance and developement of complex systems is greately improved by a modular design, it is preferable to have modularity in the architecture of the program transformer. We then decided to choose two combined theoretical approaches, to make the transformation strategy as uniform as possible and to reflect the modular features of λProlog in the modular design of the implementation: *Recursive Program Schemes* and *Algebraic Specifications*. There are several reasons to introduce algebraic specifications in program transformations. They incapsulate knowledge relative to the algebraic properties of data-types in a finite number of axioms, so it is possible to demand the execution of some particular transformation steps, i.e. *rewrite by lemma* steps, to a module implementing the algebraic specification. Further algebraic properties could be deduced from the axioms of the specification.

2.1 Algebraic specifications

In our approach, the data structures instantiating the recursive program schemes, i.e. the interpretations, are represented by means of algebraic specifications, which abstract the declarative features of data-types from the implementation details (see [18] for a complete survey on the subject). An algebraic specification is a couple $\langle \Sigma, E \rangle$ where Σ is a first-order signature $\langle F, S \rangle$, E is a finite set of equational axioms on Σ and F is a sorted signature with set of sorts S. In the following we use algebraic specifications, expressed in the formalism of ASF of [3], to represent the data structure *list of objects* :

```
module ListofObjects
begin
parameters
        Object    begin
                      sorts   OBJECT
                  end  Object
exports
     begin
                  sorts  LIST
                  functions
                      [] :                      ->    LIST
                      : :   OBJECT    #  LIST ->    LIST
                      tl :                LIST - >   LIST
                      hd:                 LIST - >   OBJECT
                      @ :   LIST      #  LIST ->    LIST
     end
variables
          x, y, z  :  -> LIST
                 l  :  -> OBJECT
equations
[s1]   hd[l : x]          =    l
[s2]   tl[l : x]          =    x
[s3]   [] @ x             =    x
[s4]   x @ []             =    x
[s5]   [l : y] @ z        =    [l : (y @ z)]
[s6]   x @ (y @ z)        =    (x @ y) @ z

end ListofObjects
```

Algebraic specification can also be used at object-level in program developement, as part of a prototyping language for input programs, by means of interpretational implementation of rewriting systems, originated by suitable orientation of equational axioms. Beside, when the existence of an initial algebra is granted, the semantics of the algebraic specifications is easily at hand.

2.2 A Program Transformation Technique: λ-generalization strategy

The program transformation strategy given in [17] avoids term mismatch useless for folding steps by means of λ-conversion of λ-terms. Basically, this strategy called λ-generalization strategy represents a higher-order variant of the classical generalization strategy of Aubin described in [2] and consists of a sequence of applications of the following elementary transformation steps:

- define a suitable auxiliary function
- instantiate the auxiliary function
- unfold the occurrences of left hand side of input program equations
- rewrite program terms according to data-type properties
- fold the occurrences of right hand-side of the equation defining the auxiliary function equation

The λ-generalization strategy is used, when in the input program, a mismatch between an expression $E[e]$ and a subexpression e occurs. In this case the folding steps are not applicable. In order to circumvent the problem, an auxiliary function $\lambda x.\, E[x]$ is defined, linked to $E[e]$ through λ-conversion relation $E[e] = (\lambda x.E[x])\, e$, allowing folding operations which are not otherwise enabled. This auxiliary function meets the flexibility and the free advantages of λ-conversion rules, making fold/unfold transformation steps easier to obtain. In [1] we applied the strategy to transform the naive version of reverse of lists into a less time-consuming iterative one. The fold/unfold transformational steps could be uniformly dealt as rewritings of program expressions if we see such trasformations in the light of recursive program schemes. Given a set of sorts S and two disjoint S-signatures F and Φ, a *system of equations over* F with set of unknowns Φ is an N-tuple of equations of the form $\Sigma = \langle \phi_i(x_{i,1}, \ldots, x_{i,n_i}) = t_i, i = 1..n \rangle$, where for each $i = 1, N$, $t_i \in M(F \cup \Phi, \{(x_{i,1}, \ldots, x_{i,n_i}\})$. A *recursive applicative program scheme* is a pair $\langle \Sigma, t \rangle$,where Σ is as above and t is a particular term that can be interpreted as an *applicative entry point* in the computation of a program. An applicative program is an instantiation of a recursive program scheme under an interpretation which is modelized by an F-algebra M. It is nice keeping M in the category of models of an equational set of axioms E, so we can benefit of the initiality properties of the associated algebraic specification $\langle F, E \rangle$. With such an approach an unfolding step could be seen as a rewrite-rule system $\Sigma = \{\phi_i(x_{i,1}, \ldots, x_{i,n_i}) \longrightarrow t_i, i = 1 \ldots n\}$, while a folding step could be seen as the reverse of such a relation $\Sigma^{-1} = \{t_i \longrightarrow \phi_i(x_{i,1}, \ldots, x_{i,n_i})\, i = 1 \ldots n\}$. In the higher-order case, we have to join to systems Σ and Σ^{-1} the rules of λ-conversion. The advantages to use recursive program schemes in program transformation are stated in the following points:

1. the separation of the data structures of the input program from its control structures allows focusing on the interpretation modelized by an algebraic specification;

2. the class of input programs, where the program transformation strategies are uniformly applied, is enlarged (as outlined in [1]).

Hence, as pointed out in this section, the fold/unfold steps in the transformation of applicative programs have proven to be a particular case of term-rewriting. As such, in their λProlog implementation, they could be dealt with the same general rewrite-rule based techniques we elaborated for the algebraic specifications.

3 Term-Rewriting Techniques in the Context of Program Transformation

We design the program transformer according to a top-down methodology which allows us to distribute the architecture of the program into several modules. Term-rewriting is ubiquitous in fold/unfold strategies, expecially in *rewrite-by-lemma* steps. Following a modular style of programming, we have written a module in λProlog containing a specification of a data-type, in our case the data-type *list*. λ-bounded variables in program expressions need a higher- order extension of rewriting-rule; for these reasons we embedded the algebraic specification into a higher-order specification language such as λProlog which gives it for free.

The implementation of algebraic specifications exports a predicate, called `reduce`, invoked every time the strayegy must perform *rewrite-by-lemma* steps. The main module, containing the program transformer, imports the predicate `reduce` through the top-level commands `use, import, export` of the λProlog compiler.

In the following we give two different implementations of an algebraic specification of a *lists of objects* in λProlog.

3.1 Direct Rewriting Through Higher-Order Unification

According to the interpretational approach, a rewrite rule is described as a di-adic predicate instantiated with the two sides of the rewrite rule. The main advantage offered by higher-order λProlog implementation, is that a suitable higher-order variable, namely H, allows the higher-order matching of the extended left hand side of the equation with the full input term, performing the subterm lookup implicitly. The major drawback with this technique is due to the introduction of spurious bindings for the variable H to λ-terms of kind $\lambda x.s$, originated by the application of the *imitation rule* of the higher-order unification algorithm. Consider the following example:


```
kind int type.
type 0   int .
type plus int → int → int.
type reduce A → A → o.

   reduce H (plus X 0) (H X).
```

By trying the goal `?-reduce (plus 3 0) Xs`, higher-order unification will produce a disagreement set of the kind { ⟨ (plus 3 0) , H (plus X 0) ⟩, ⟨ Xs, (H X) ⟩ }. Now, the *matching* procedure will yield two substitutions for H, $\{H \rightarrow x\backslash x\}$ and $\{H \rightarrow x\backslash x\ plus(H1\,X)(H2\,X)\}$. Only the first substitution will give a correct answer $\{X\,s \rightarrow 3\}$, the other one results in a spurious one. A way to solve this problem would be turning the rewrite rules expressed in form of facts in λProlog into rules expressing *conditional* rewriting. Actually, the body of the rule contains the specification of the correct binding for H; as for the above example, the program line would be substituted by the following:

```
reduce (H (plus X 0)) (H X) :- context(H).
context(x\x).
```

The definition of higher-order rewriting system given in [19] is consistent with the implementation technique given above, to the extent that the matching between a subterm and the left hand side of a rewriting rule is automatically computed by instantiating the variable H. Of course this implementation technique is not acceptable from the computational point of view, since many solutions produced by higher-order unification mechanism would be discarded by the `context` predicate definition, producing useless work of λProlog interpreter, but in anyway, for "small" signatures there are no significative problems.

Another major drawback of these techniques is the unadequacy of dealing with bounded variables for truly higher-order rewriting rules. To show this, consider this example:

```
type [] list A.
type @  list A → list A → list A.
type :  A → list A → list A.

type reduce A → A → o.

   reduce (H (Xs @ [])) (H Xs).
```

When we try the goal, `?- reduce (x\(x @ [])) Xs`, the disagreement set is not able to produce the right substitution since the computed answer will never bind Xs to a closed term. A partial solution to this problem would be substituting the above program line with the following line:

```
reduce (Xs \ H (Xs @ [])) (Xs \ (H Xs)).
```

However such a solution would result in a loss of generality because a kind of first-order/higher-order *switch* would be necessary. We adopted this solution in [1] because only higher-order, bounded variables were used in program expressions. We present in the following the implementation of the whole `list` module that is invoked by the program transformer.

```
module lists.
kind list type → type.
type []   list A.
type :    A → (lst A) → (lst A).
type hd   (lst A) → A.
type tl   (lst A) → (lst A).
type @    (lst A) → (lst A) → (lst A).
type extrule (lst A) → (lst A) → o.


extrule   (Xs\ H (hd [X : Xs]))  (H X)          :- context(H).(1)
extrule   (Xs\ H (tl [X : Xs]))  (Xs\ H Xs)     :- context(H).(2)
extrule   (Xs\ H (Xs @ []))      (H Xs)         :- context(H).(3)
extrule   (Xs\ H ([] @ Xs))      (H Xs)         :- context(H).(4)
extrule   (Zs\ H((Xs @ Ys) @ Zs))(H (Xs @ (Ys @ Zs)))
                                                :- context(H).(5)
extrule   (Zs\ H ( [X : Xs] @ Zs)) H ( [X : (Xs @ Zs)]))
                                                :- context(H).(6)
context   (x\x).                                              (7)
context   (x\y\x).                                            (8)
context   (x\y\y).                                            (9)
reduce    X Y :- extrule X Y.    %this is the predicate to be exported
```

The module `list` exports to the main module, which manages the program transformation strategy, the predicate `reduce`.

We have also found that higher-order rewriting techniques, are useful in automatic execution of folding steps. To clarify our idea, let us consider the following recursive applicative version of the program computing the reverse of a list:

$$rev([]) = []$$
$$rev([hd(l) : tl(l)]) = rev(tl(l))@[hd(l)]$$

The application of the λ-generalization strategy to the above function will first yield the auxiliary higher-order function $g(l) = \lambda x.\, rev(l)@x$. Through the application of a sequence of rewrite-by-lemma steps, we get the program expression $\lambda x\, rev(tl(l))@[hd(l) : x]$. At this point, we exploit the following relation among λ-terms:

$$\lambda x.\, rev(tl(l)) \,@\, [\, hd(l) : x] =_\lambda \lambda x.\, [\lambda y.\, rev(tl(l)) \,@\, y]\ [hd(l) : x] \qquad (1)$$

Then we apply a folding step to the λ-term $\lambda y.\, rev(tl(l)) \,@\, y$ which, matched against the equation $g(l) = \lambda x.\, rev(l) \,@\, x$, gives the term $g(tl(l))$. Substituting the latter term with the left hand side of equation (1) and considering this equation as the leftmost part of a chain of λ-conversions beginning with $g([hd(l) : tl(l)])$ and obtained through the application of the λ-generalization strategy, we have a new recursive expression of the auxiliary program function g, that will yield a new iterative version of the reverse function. In this way the folding process could be seen as a sort of *conditional term-rewriting* process, and so logic programming-style implementation is immediate. Besides, higher-order unification mechanism inherent to λProlog provides λ-term probing for performing folding steps of subterms and for replacing them in the originary contexts. In our implementation we obtain the folding steps through the following λProlog lines:

```
   fold X\ (K M (N X) ) (X \ Y (N X)) :- rewritefun Y (X\K M X).
rewritefun (H (g L)) (H (X\ (rev L) ) @ X).
```

In these lines the search for foldable expressions is demanded to the instantiation of variable M, while folding step is up to the body predicate `rewritefun` (which is invoked with the first argument uninstantiated) through the instantiation of the variable Y. Higher-order variables K and N control the context of the program expression. In the case we studied, higher- order unification, on invocation of the following goal:

```
?- fold x \ ( rev( tl (L)) @ [ hd(L) : x] (x \ Y ( N x))
```

produces the bindings K → x \ y\(x @ y), N → x\ [hd(L) : x], M → rev(tl (L)) and Y → g(tl (L)), instantiating the second argument of the goal to x\ g(tl (L)) [hd(L) : x] , i.e. the left hand side of the new recursive program equation. Now, observe that the λProlog clause `fold`, does not depend on the particular input program. Also `rewritefun` is a predicate that could be replaced by a set of predicates that depend only from the recursive program scheme associated to the input program. In this way the scope of the strategy can be enlarged to the class of programs instantiated the same recursive program scheme.

3.2 Congruence Closure of a Set of Rewrite Rules

On account of the drawbacks described in the previous section, we have formulated an alternative version of the `list` module. In this latter approach, we formulate a set of predicates specifying the elementary rewriting steps and then we expand them as respect to the signature. It is a method which grants uniformity as respect to signature, avoiding the problems originated by the use of higher-order variables.

```
module lists
kind lst type → type.
type []  lst A.
type :   A → (lst A) → (lst A).
```

```
type hd  (lst A) → A.
type tl  (lst A) → (lst A).
type @   (lst A) → (lst A) → (lst A).


rev X @ [] X .                                  (1)
rev [] @ X .                                    (2)
rev ( [X : Xs] @ Ys) ([X : (Xs @ Ys)]).         (3)
rev X @ ( Y @ Z) (X @ Y) @ Z .                  (4)
rev hd ([X : Xs]) X .                           (5)
rev tl ([X : Xs]) Xs .                          (6)
rev (x \ M x) (x\ N x)
    :- pi x \[rev x x => rev (M x) (N x)].      (7)
rev (A1 A2) (B1 B2)
    :- rev(A1 B1) , rev(A2, B2).                (8)
reduce (X1 @ Y2) (X2 @ Y2)   :- reduce X1 X2.   (9)
reduce (X1 @ Y1) (X1 @ Y2)   :- reduce Y1 Y2.  (10)
reduce (hd [X1 :  Xs1]) (hd [X1 :  Xs2])
                             :-reduce Xs1 Xs2.(11)
reduce (tl [X1 :  Xs1]) (tl [X2 :  Xs1])
                             :- reduce X1 X2 .  (12)
reduce X Y :-                rev X Y .         (13)
```

The main advantage of this module is that the code lines specifying the congruence closure of the term-rewriting system (lines 9 through 13) are uniform respect to the signature of the data- structure. Line (7) allows dealing with λ-bounded variables by means of $\forall$-bounded variables at propositional level, turning the former class of variables into a sort of *local parameters* that could be dynamically reduced by means of first-order rules. Informally, we can think of line (7) and (8) as an "higher-order closure" of the first-order term- rewriting system.

4 Conclusions and Future Developments

In this paper we have investigated a particular application of rewriting techniques in solving program transformation tasks. We have used the higher-order language λProlog as metalanguage for implementing rewriting techniques, exploiting in this way the higher-order features of the language and the availability of λ-terms to express different operations on programs.

Beside, our work has pointed out that the use of algebraic specification and recursive program schemes in program transformation reduces the problem of folding programs expressions to a problem of term-rewriting, to be solved into the theory of generalized equational unification. We realized that three kinds of unification problems have been involved in our research: the higher-order unification of λ-terms, the higher-order equational specification of algebraic specification and the

Σ-unification of recursive program schemes. The study of the decidability problem in the general theory of unifcation will be the theoretical support for a successful application of rewrite- by-lemma steps leading to signifcant foldings of object-level programs.

Acknowledgements

We would like to thank D.Miller, A. Pettorossi and M. Proietti for their advices and their hints in undertaking this research.

References

[1] F.Arcelli and F.Formato, "Higher-Order Implementation of Program Transformations using Algebraic Specifications", in *PTELP Workshop*, Post-ICLP94 Conference, A.Momigliano and M.Ornaghi, editors, S.Margherita Ligure, june, 1994.

[2] R.Aubin, "Mechanizing Structural Induction", Ph.D. Thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1976.

[3] J.A.Bergstra, J.Heering, and P.Klint, eds., *Algebraic Specification*, ACM-PRESS/Addison-Wesley, 1989.

[4] P.Brisset and O.Ridoux, "The compilation of λProlog and its execution with Mali", in Rapport de Recherche n.1831, INRIA, jan. 1993.

[5] A.Burstall and R.M.Darlington, "A Transformation System for Developing Recursive Programs" , in *Journal of the ACM*, 24,1, pp. 44-67, 1977.

[6] B.Courcelle, "Recursive Applicative Program Schemes", in *Handbook of Theoretical Computer Science*, Chapter 9, Elsevier pb B.V.,1990.

[7] N.Dershowitz, "Synthesis by completion", *Proceedings of 9th.International Joint Conference on Artifical Intelligence*, 1, pp.208-214, 1985.

[8] A.Felty, "A Logic Programming Approach to Implementing Higher-Order Term Rewriting", in *Proceedings of the 1991 International Workshop on Extensions of Logic Programming*, in L.H Eriksson, L.Hallnas and P.Schroeder-Heister, editors, Springer-Verlag Lectures Notes in Artifical Intelligence, 1992.

[9] J.Hannan and D.Miller, "Uses of Higher-Order Unification for Implementing Program Transformers", in *Proceedings of Fifth Int.Conf. on Logic Programming*, MIT Press, Seattle, pp. 942-959, 1988.

[10] J.Heering, "Implementing Higher-Order Algebraic Specifications", in *Proceedings of the 1992 Workshop on the λProlog Programming Language*, D.Miller, editor, University of Pennsylvania, pp. 141-157, 1992.

[11] G.Huet and B.Lang, "Proving and Applying Program Transformations Expressed with Second-Order Patterns", in *Acta Informatica*, 11, pp. 31-55, 1978.

[12] D.Miller, "A logic programming languge with lambda-abstraction, function variables, and simple unification", in *Extensions of Logic Programming*, P.Schroeder-Heister, editor, Lectures Notes in Artificial Intelligence, 475, Springer-Verlag, pp.253-281, 1991.

[13] D.Miller and G.Nadathur, "A logic programming approach to manipulating formulas and programs", in *Proceedings of the IEEE Fourth Symposium on Logic Programming*, IEEE PRESS, 1987.

[14] G.Nadathur, " A higher-order logic as a basis for logic programming", Ph.D. Dissertation, University of Pennsylvania, may, 1987.

[15] G.Nadathur and D.Miller, "An Overwiew of λProlog", *Fifth International Logic programming Conference*, K.A.Bowen and R.A.Kowalski, editors, MIT PRESS, pp. 810- 827, 1988.

[16] T.Nipkow, "Higher-Order Critical Pairs", in *Sixth Annual Symposium on Logic in Computer Science*, pp. 342-349, Amsterdam, july 1991.

[17] A. Pettorossi and A. Skowron, "The Lambda Abstraction strategy for program derivation", in *Fundamenta Informaticae*, North Holland pb, XII, pp. 541-562, 1989.

[18] M.Wirsing, "Algebraic Specification ", in *Handbook of Theoretical Computer Science*, Chapter 13, Elsevier pb B.V.,1990.

[19] D.A.Wolfram, "Rewriting, and equational Unification: the Higher-Order Cases", in *Proceedings of RTA-91*, Lectures Notes in Computer Science, Springer-Verlag, Berlin, 1991.

The Undefined Function Differs From the Pointwise Undefined Function

Walter Dosch
Institut für Mathematik
Universität Augsburg
D–86135 Augsburg

Phone +49-821-598-2170
Fax +49-821-598-2200
Email dosch@informatik.uni-augsburg.de

Abstract

In the framework of a simply typed higher order λ-calculus, we study one particular possibility of extending the call-by-value semantics from first order to higher order functions. The basic assumption is to discriminate the undefined function from the pointwise undefined function. The calculus is elaborated under two basic aspects, viz. reduction and denotational semantics.

Keywords λ-calculus, call-by-value semantics, strictness, higher order functions

1 Introduction

The foundations of functional programming were laid by the λ-calculus (for the historic origin see [6] and [8], for recent books [3] and [13]): its conversion rules, viz. the α-, β- and η-conversion, define the basic equivalence of applicative expressions, its models like D^∞ or P^ω provide a semantic universe for interpreting them. At a closer look, however, pure λ-calculus does not adequately formalize the semantics of an applicative language with a call-by-value semantics.

In this paper we investigate one particular aspect of the semantics of a simply typed higher order λ-calculus: *How to extend the call-by-value semantics from first order to higher functions?* This can be rephrased in denotational or operational terms as follows:

- What is the appropriate semantic domain associated with higher types under a call-by-value semantics?
- How can the β-conversion and the η-conversion (extensionality) of pure λ-calculus be modified to meet a call-by-value semantics for higher order functions?

For a first order function the call-by-value regimen evaluates the argument before the function is applied. Hence the evaluation of an application diverges, if the evaluation of its argument does. In the semantic model this leads to strict functions which yield undefined whenever their argument is undefined.

This suggests to consider strict functions also for higher types. However, this straightforward denotational semantics allows no simple operational rules. The difficulty lies in checking an argument of higher type whether it is different from the least element. For first order functions, the argument is of ground type and can be completely evaluated. For a higher type, the test could only be performed by evaluating the function at all its arguments in a fair manner in parallel.

This inadequacy originates from taking the *pointwise undefined function* as the least element in the function space in which the functionals are required to be strict. To overcome this difficulty, we enlarge the function space by the *undefined function* which is properly less defined than all genuine functions. The undefined function models nontermination when computing the function as the result of a higher order function. The pointwise undefined function models a well-defined functional object which is undefined at each argument.

In the sequel, we explore the consequences of this design decision to the semantics of a simply typed higher order λ-calculus under two major aspects, viz. reduction and denotational semantics. In particular, we characterize the notion of extensionality of higher order expressions under a call-by-value semantics and establish the call-by-value η-reduction. In summary, the discrimination between the undefined function and the pointwise undefined function allows a coherent framework for higher order call-by-value programming.

The reader should be familiar with the foundations of functional programming, in particular with term rewriting (for overviews see [9] and [14]), denotational semantics (see, for example, [17] and [20]) and the underlying domain theory (see, for example, [12] and [22]).

2 Syntax

In this section we introduce the syntax of the language together with the syntactic functions to manipulate expressions. By incorporating suitable typing rules into the

inductive definition, we only generate well-typed expressions. Hence we can focus on the deduction of expressions without having to treat simultaneously the deduction of types (as in [4] or [11]).

2.1 Basis

We base the language on the fixed signature $\Sigma = (\mathcal{S}, \mathcal{K}, \mathcal{F})$ of natural numbers and Boolean values comprising the *sorts* $\mathcal{S} = \{\mathbf{bool}, \mathbf{nat}\}$, the *constants* $\mathcal{K}^{\mathbf{bool}} = \{\mathit{true}, \mathit{false}\}$ and $\mathcal{K}^{\mathbf{nat}} = \{\mathit{zero}\}$, as well as the *operators* $\mathcal{F}^{\mathbf{nat}\to\mathbf{nat}} = \{\mathit{succ}, \mathit{pred}\}$ and $\mathcal{F}^{\mathbf{nat}\to\mathbf{bool}} = \{\mathit{iszero}\}$.

Types are syntactic attributes that impose a discipline on forming expressions.

1. Definition The set $\mathcal{T}$ of *types* over $\mathcal{S}$ is defined inductively:

(1) *Ground types* $\mathcal{S} \subseteq \mathcal{T}$.
(2) *Higher types* If $\mathbf{r}, \mathbf{s} \in \mathcal{T}$, then $(\mathbf{r} \to \mathbf{s}) \in \mathcal{T}$.

Types will be denoted by the bold face letters $\mathbf{r}, \mathbf{s}, \mathbf{t}$. In applicative expressions variables are bound by abstractions and by (recursive) declarations.

2. Definition A $\mathcal{T}$-typed *variable family* $X = (X^{\mathbf{t}})_{\mathbf{t}\in\mathcal{T}}$ consists of countably infinite, pairwise disjoint sets $X^{\mathbf{t}}$ of *variables*. A *basis* (Σ, X) consists of a signature Σ and a variable family X disjoint to Σ.

2.2 Syntax of Applicative Expressions

The expression language follows the applicative style of functional programming. Expressions of higher types are built over a first order signature.

3. Definition The family $EXPR = (EXPR^{\mathbf{t}})_{\mathbf{t}\in\mathcal{T}}$ of (*applicative*) *expressions* over the basis (Σ, X) is defined inductively:

(1) *Constants* $\mathcal{K}^{\mathbf{s}} \subseteq EXPR^{\mathbf{s}}$ $(\mathbf{s} \in \mathcal{S})$.
(2) *Operators* $\mathcal{F}^{\mathbf{r}\to\mathbf{s}} \subseteq EXPR^{\mathbf{r}\to\mathbf{s}}$ $(\mathbf{r}, \mathbf{s} \in \mathcal{S})$.
(3) *Variables* $X^{\mathbf{t}} \subseteq EXPR^{\mathbf{t}}$.
(4) *Conditional* If $E \in EXPR^{\mathbf{bool}}$ and $F, G \in EXPR^{\mathbf{t}}$, then **if** E **then** F **else** G **fi** $\in EXPR^{\mathbf{t}}$.
(5) *Application* If $F \in EXPR^{\mathbf{r}\to\mathbf{s}}$ and $E \in EXPR^{\mathbf{r}}$, then $F(E) \in EXPR^{\mathbf{s}}$.
(6) *Abstraction* If $x \in X^{\mathbf{r}}$ and $E \in EXPR^{\mathbf{s}}$, then $(\lambda x.E) \in EXPR^{\mathbf{r}\to\mathbf{s}}$.
(7) *Recursion* If $x \in X^{\mathbf{t}}$ and $E \in EXPR^{\mathbf{t}}$, then $(\mu x.E) \in EXPR^{\mathbf{t}}$.

The language supports the explicit typing paradigm (originally introduced in [6]) — and not the implicit typing paradigm ([7], [8]). The language corresponds to the deterministic kernel of CIP-L ([5]) and to the programming language PCF for computable

functions ([21]).

4. Notation Constants will be denoted by c, d, operators by f, g, h, variables by x, y, z, and expressions by E, F, G, H. The syntactic equality of expressions is denoted by $\hat{=}$. In writing expressions parentheses may be dropped with the usual conventions.

5. Examples

a) Natural numbers: $succ^n(zero) \hat{=} \underbrace{succ(\ldots(succ}_{n}(zero))\ldots)$ $(n \in \mathcal{N})$

b) Addition of natural numbers $(x \in X^{\mathbf{nat}\to(\mathbf{nat}\to\mathbf{nat})},\ y, z \in X^{\mathbf{nat}})$:
$add \hat{=} \mu x.\lambda y.\lambda z.\,\mathbf{if}\ iszero(y)\ \mathbf{then}\ z\ \mathbf{else}\ x(pred(y))(succ(z))\ \mathbf{fi}$

c) Syntactic error element representing non-termination: $error^{\mathbf{t}} \hat{=} \mu x.x$ $(x \in X^{\mathbf{t}})$. In particular, $error^{\mathbf{r}\to\mathbf{s}}$ will represent the "undefined function".

d) Expression representing the "pointwise undefined function": $omega^{\mathbf{r}\to\mathbf{s}} \hat{=} \lambda y.error^{\mathbf{s}}$ $(y \in X^{\mathbf{r}})$.

e) Constant zero function $(x \in X^{\mathbf{t}})$: $constzero^{\mathbf{t}\to\mathbf{nat}} \hat{=} \lambda x.zero$.

Every expression of the language possesses a unique type. The finite type structure of the language excludes the self application of expressions.

We will not build the consistent renaming of bound variables into the conversion rules ([6]). Rather we identify alphabetically equivalent expressions on the syntactic level ([3], [4], [11]).

6. Definition The family $EXP = (EXP^{\mathbf{t}})_{\mathbf{t}\in\mathcal{T}}$ of (α-*congruent*) *expressions* comprises the α-congruence classes of $EXPR$. to avoid problems of α-variant.

The free variables of an expression are not changed by a consistent renaming.

7. Definition The set of *free variables* of an expression $E \in EXP$ is denoted by $free(E)$. The family $CMB = (CMB^{\mathbf{t}})_{\mathbf{t}\in\mathcal{T}}$ of *combinators* consists of *closed expressions* having no free variables.

The substitution $[E]^F_x$ replaces all free occurrences of the variable x in the expression E by the expression F of appropriate type.

2.3 Values

To meet the strictness constraints of a call-by-value semantics, we introduce a subfamily of expressions the syntactic form of which will ensure that they have a defined interpretation.

8. Definition The family $VL = (VL^{\mathbf{t}})_{\mathbf{t}\in\mathcal{T}}$ of *values* is defined inductively:

(1)	*Constants*	$\mathcal{K}^{\mathbf{s}} \subseteq VL^{\mathbf{s}}$ $(\mathbf{s} \in \mathcal{S})$.
(2)	*Operators*	$\mathcal{F}^{\mathbf{r}\to\mathbf{s}} \subseteq VL^{\mathbf{r}\to\mathbf{s}}$ $(\mathbf{r}, \mathbf{s} \in \mathcal{S})$.
(3)	*Non-zero Numbers*	If $w \in VL^{\mathbf{nat}}$, then $succ(w) \in VL^{\mathbf{nat}}$.
(4)	*Abstractions*	If $x \in X^{\mathbf{r}}$ and $E \in EXP^{\mathbf{s}}$, then $(\lambda x.E) \in VL^{\mathbf{r}\to\mathbf{s}}$.

Values of ground types denote Booleans and natural numbers, values of higher types are operators and abstractions denoting genuine functions. Values are closed under substitution.

3 Denotational Semantics

In the denotational semantics we concentrate on the strictness properties of the expression language.

3.1 Flat Domains

With the underlying data structure we associate flat domains with strict functions.

1. Definition The continuous Σ-algebra $\mathcal{A}$ consists of the *carrier sets* $\mathbf{bool}^{\mathcal{A}} = \{\bot^{\mathbf{bool}}, T, F\}$ and $\mathbf{nat}^{\mathcal{A}} = \{\bot^{\mathbf{nat}}, 0, 1, 2, \ldots\}$, the *constants* $true^{\mathcal{A}} = T$, $false^{\mathcal{A}} = F$ and $zero^{\mathcal{A}} = 0$, and the *operations* $succ^{\mathcal{A}}, pred^{\mathcal{A}}$: $\mathbf{nat}^{\mathcal{A}} \to \mathbf{nat}^{\mathcal{A}}$ and $iszero^{\mathcal{A}}$: $\mathbf{nat}^{\mathcal{A}} \to \mathbf{bool}^{\mathcal{A}}$ with the usual interpretation.

3.2 Domain of Strict and Continuous Functions

With function types one usually associates the domain of continuous functions.

2. Definition For complete partial orders $\mathcal{D} = (D, \sqsubseteq^D, \bot^D)$ and $\mathcal{R} = (R, \sqsubseteq^R, \bot^R)$, $([D \to R], \sqsubseteq, \Omega^{D\to R})$ denotes the *cpo of continuous functions* from D to R with the order $f \sqsubseteq g$ iff $f(d) \sqsubseteq^R g(d)$ for all $d \in D$. The *pointwise undefined function* $\Omega^{D\to R}$: $D \to R$ with $\Omega^{D\to R}(d) = \bot^R$ for all $d \in D$ is the least element.

To meet the call-by-value semantics we select the subspace of strict and continuous functions. For complete partial orders $\mathcal{D}$ and $\mathcal{R}$, the set $[D \xrightarrow{s} R]$ of strict and continuous functions forms a subcpo of $[D \to R]$. Moreover, the retraction *strict*: $[D \to R] \to [D \xrightarrow{s} R]$ given by $strict(f) = f[\bot^D/\bot^R]$ is continuous; here $.[./.]$ denotes the update operation on functions.

3.3 Lifting the Function Space

The lift attaches a new least element to a domain.

3. Definition For a complete partial order $\mathcal{D} = (D, \sqsubseteq^D, \bot^D)$ the *lifted complete partial order* $\mathcal{D}_\bot$ consists of the set $D_\bot = (D \times \{0\}) \cup \{\bot\}$ endowed with the order relation $x \sqsubseteq y$ iff $x = \bot$ or $x = (d, 0)$ and $y = (e, 0)$ and $d \sqsubseteq^D e$.

For a complete partial order $\mathcal{D} = (D, \sqsubseteq^D, \bot^D)$ the lift $\mathcal{D}_\bot = (D_\bot, \sqsubseteq, \bot)$ is a complete

partial order. The mappings in: $D \to D_\perp$ with $in(d) = (d, 0)$ as well as out: $D_\perp \to D$ with $out(\perp) = \perp^D$ and $out(d, 0) = d$ are continuous with $out \circ in = id_D$ and $in \circ out \sqsupseteq id_{D_\perp}$.

4. **Definition** For cpos $\mathcal{D}$ and $\mathcal{R}$, the *lifted function space* $[D \xrightarrow{s} R]_{\perp^{D\to R}}$ is the space of strict and continuous functions extended by the *undefined function* $\perp^{D\to R}$. The apply operation *apply*: $[D \xrightarrow{s} R]_{\perp^{D\to R}} \times D \to R$ is given by $apply(f, d) = (out(f))(d)$.

The undefined function $\perp^{D\to R}$ is properly less defined than every genuine function, in particular less defined than the pointwise undefined function $\Omega^{D\to R}$. The undefined function denotes expressions of functional type with non-terminating computations whereas the pointwise undefined function denotes abstractions that lead in every application to a non-terminating computation. The lifted function space integrates two orders: a function can be applied as an operator to an argument — this leads to the pointwise order on the space of genuine functions. But a function can also be the argument of a higher order function — this leads to a flat order on the extended function space.

3.4 Interpretation of Expressions

First we associate complete partial orders with the type system.

5. **Definition** The *interpretation of the types* is defined inductively:

$$(1)\quad [\![\mathrm{s}]\!] = \mathrm{s}^{\mathcal{A}} \quad (\mathrm{s} \in \mathcal{S})$$
$$(2)\quad [\![\mathrm{r} \to \mathrm{s}]\!] = [\,[\![\mathrm{r}]\!] \xrightarrow{s} [\![\mathrm{s}]\!]\,]_{\perp^{\mathrm{r}\to\mathrm{s}}}$$

Valuations record the binding of variables to semantic elements.

6. **Definition** A *valuation* $\rho = (\rho^{\mathrm{t}}\colon X^{\mathrm{t}} \to [\![\mathrm{t}]\!])_{\mathrm{t}\in\mathcal{T}}$ is a family of mappings associating elements to variables. The *environment* ENV denotes the set of all valuations.

7. **Definition** The *interpretation* $[\![\,.\,]\!] = ([\![\,.\,]\!]^{\mathrm{t}})_{\mathrm{t}\in\mathcal{T}}$ *of applicative expressions* is a family of mappings $[\![\,.\,]\!]^{\mathrm{t}}\colon EXP^{\mathrm{t}} \to [ENV \to [\![\mathrm{t}]\!]]$ defined as follows:

$$(1)\quad [\![c]\!]^{\mathrm{s}}(\rho) = c^{\mathcal{A}}$$
$$(2)\quad [\![f]\!]^{\mathrm{r}\to\mathrm{s}}(\rho) = in(f^{\mathcal{A}})$$
$$(3)\quad [\![x]\!]^{\mathrm{t}}(\rho) = \rho^{\mathrm{t}}(x)$$
$$(4)\quad [\![\textbf{if } E \textbf{ then } F \textbf{ else } G \textbf{ fi}]\!]^{\mathrm{t}}(\rho) = \begin{cases} \perp^{\mathrm{t}} & \text{if } [\![E]\!]^{\mathbf{bool}}(\rho) = \perp^{\mathbf{bool}} \\ [\![F]\!]^{\mathrm{t}}(\rho) & \text{if } [\![E]\!]^{\mathbf{bool}}(\rho) = T \\ [\![G]\!]^{\mathrm{t}}(\rho) & \text{if } [\![E]\!]^{\mathbf{bool}}(\rho) = F \end{cases}$$
$$(5)\quad [\![F(E)]\!]^{\mathrm{s}}(\rho) = (out([\![F]\!]^{\mathrm{r}\to\mathrm{s}}(\rho)))([\![E]\!]^{\mathrm{r}}(\rho))$$
$$(6)\quad [\![\lambda x.E]\!]^{\mathrm{r}\to\mathrm{s}}(\rho) = in(strict(d \mapsto [\![E]\!]^{\mathrm{s}}(\rho[x/d]))) \quad (d \in [\![\mathrm{r}]\!])$$
$$(7)\quad [\![\mu x.E]\!]^{\mathrm{t}}(\rho) = \mu(d \mapsto [\![E]\!]^{\mathrm{t}}(\rho[x/d])) \quad (d \in [\![\mathrm{t}]\!])\,.$$

Above μ denotes the least fixpoint operator. Two expressions $E, F \in EXP^{\mathrm{t}}$ are called (*semantically*) *equivalent* (denoted by $E \approx^{\mathrm{t}} F$), if for all $\rho \in ENV$ we have

$[\![E]\!]^{\mathrm{t}}(\rho) = [\![F]\!]^{\mathrm{t}}(\rho)$.

We illustrate the interpretation assigned to the expressions denoting the 'undefined function' and the 'pointwise undefined function'.

8. **Example**

$$\begin{aligned} & [\![error^{\mathrm{r}\to\mathrm{s}}]\!](\rho) \\ = \ & [\![\mu x.x]\!](\rho) \\ = \ & \mu(f \mapsto [\![x]\!](\rho[x/f])) \\ = \ & \mu(f \mapsto f) \\ = \ & \perp^{[\![\mathrm{r}]\!]\to[\![\mathrm{s}]\!]} \end{aligned} \qquad \begin{aligned} & [\![omega^{\mathrm{r}\to\mathrm{s}}]\!](\rho) \\ = \ & [\![\lambda y.error^{\mathrm{s}}]\!](\rho) \\ = \ & in(strict(d \mapsto [\![error^{\mathrm{s}}]\!](\rho[y/d]))) \\ = \ & in(strict(d \mapsto \perp^{[\![\mathrm{s}]\!]})) \\ = \ & in(strict(\Omega^{[\![\mathrm{r}]\!]\to[\![\mathrm{s}]\!]})) \\ = \ & in(\Omega^{[\![\mathrm{r}]\!]\to[\![\mathrm{s}]\!]}) \\ = \ & (\Omega^{[\![\mathrm{r}]\!]\to[\![\mathrm{s}]\!]}, 0) \end{aligned}$$

Values have a defined meaning in all environments.

9. **Proposition** For all $\rho \in ENV$ and $W \in VL^{\mathrm{t}}$ we have $[\![W]\!](\rho) \neq \perp^{\mathrm{t}}$.

4 Reduction

The reduction aims at simplifying an expression completely to some normal form. The confluence (termination) of the reduction relation ensures that every expression has at most (at least) one normal form.

4.1 Basic Reduction

The reduction comprises the execution of basic operators (δ-reduction), the simplification of the conditional (γ-reduction), the application of an abstraction to an argument (β-reduction), and the unfolding of (recursive) declarations (μ-reduction).

1. **Definition** The *one-step r-reduction* $\to_r = (\to_r^{\mathrm{t}})_{\mathrm{t}\in\mathcal{T}}$ with $r \in \{\delta, \gamma, \beta, \mu\}$ is the least family of compatible relations $\to_r^{\mathrm{t}}$ on EXP^{t} with

δ-*reduction*	$pred(zero) \to_\delta^{\mathbf{nat}} zero$ $pred(succ^{n+1}(zero)) \to_\delta^{\mathbf{nat}} succ^n(zero) \quad (n \in \mathcal{N})$ $iszero(zero) \to_\delta^{\mathbf{bool}} true$ $iszero(succ^{n+1}(zero)) \to_\delta^{\mathbf{bool}} false \quad (n \in \mathcal{N})$
γ-*reduction*	$\textbf{if } true \textbf{ then } F \textbf{ else } G \textbf{ fi} \to_\gamma^{\mathrm{t}} F$ $\textbf{if } false \textbf{ then } F \textbf{ else } G \textbf{ fi} \to_\gamma^{\mathrm{t}} G$
β-*reduction*	$(\lambda x.E)(W) \to_\beta^{\mathrm{s}} [E]_x^W$
μ-*reduction*	$\mu x.E \to_\mu^{\mathrm{t}} [E]_x^{\mu x.E}$.

The *r-reduction* $\twoheadrightarrow_r$ is the reflexive transitive closure of $\to_r$; the *r-equivalence* $\equiv_r$ is the symmetric transitive closure of $\twoheadrightarrow_r$.

In a β-reduction the argument W is substituted for the bound variable x into the body E of the abstraction $\lambda x.E$. For a call-by-value semantics, it is essential to confine the argument expression to a value.

2. Notation For $R \subseteq \{\delta, \gamma, \beta, \mu\}$ the relation $\to_R$ denotes $\cup_{r \in R} \to_r$; moreover $\twoheadrightarrow_R$ is the reflexive transitive closure of $\to_R$, $\equiv_R$ is the equivalence closure of $\twoheadrightarrow_R$.

The elementary properties of the reduction are summerized in

3. Proposition

a)	(Substitutivity)	If $E \to_r F$, then $[E]_y^G \to_r [F]_y^G$ $(r \in \{\delta, \gamma, \beta, \mu\})$.
b)	(Substitutability)	If $E \twoheadrightarrow_r F$, then $[G]_y^E \twoheadrightarrow_r [G]_y^F$ $(r \in \{\delta, \gamma, \beta, \mu\})$.
c)	(Free Variables)	If $E \twoheadrightarrow_{\delta\gamma\beta\mu} F$, then $free(E) \supseteq free(F)$.
d)	(Values)	If $W \in VL^{\mathbf{t}}$ and $W \twoheadrightarrow_{\delta\gamma\beta\mu} E$, then $E \in VL^{\mathbf{t}}$.

The reduction properties of this language, extended by the smash product, multiary functions and the treatment of the finite error, but without extensionality were studied in [10]; we cite two relevant results:

4. Theorem $\to_{\delta\gamma\beta\mu}$ is confluent and $\to_{\delta\gamma\beta}$ is Noetherian.

As an immediate consequence the expression for the undefined function and the pointwise undefined function are not $\delta\gamma\beta\mu$-convertible.

5. Corollary $error^{\mathbf{r}\to\mathbf{s}} \not\equiv_{\delta\gamma\beta\mu} omega^{\mathbf{r}\to\mathbf{s}}$.

The reduction relations respect the assignment of meanings to expressions.

6. Theorem (Soundness) If $E \equiv_{\delta\gamma\beta\mu} E'$, then $E \approx^{\mathbf{t}} E'$.

4.2 Extensionality

The extensionality states that a function is completely determined by its application to arguments. In the λ-calculus the extensionality can equivalently be characterized by the η-conversion or the rule (ext).

4.2.1 η-Reduction

When adding the classical η-reduction to the calculus, we loose the confluence of the overall reduction relation. Hence we suitably modify the η-reduction rule to meet the call-by-value semantics for higher order functions.

7. Definition The *one-step call-by-value η-reduction* $\to_\eta = (\to_\eta^{\mathbf{t}})_{\mathbf{t}\in\mathcal{T}}$ is the least family of compatible relations $\to_\eta^{\mathbf{t}}$ on $EXP^{\mathbf{t}}$ with

$$\text{If } x \notin free(W), \text{ then } \lambda x.W(x) \to_\eta^{\mathbf{r}\to\mathbf{s}} W.$$

The basic reduction properties are summarized in

8. Proposition

a)	(Substitutivity)	If $E \to_\eta F$, then $[E]_y^G \to_\eta [F]_y^G$.
b)	(Substitutibility)	If $E \twoheadrightarrow_\eta F$, then $[G]_y^E \twoheadrightarrow_\eta [G]_y^F$.
c)	(Free Variables)	If $E \twoheadrightarrow_\eta F$, then $free(E) = free(F)$.
d)	(Values)	If $W \in VL^{\mathbf{t}}$ and $W \twoheadrightarrow_\eta E$, then $E \in VL^{\mathbf{t}}$.
e)	(Soundness)	If $E \equiv_\eta E'$, then $E \approx E'$.
f)	(Confluence)	$\to_\eta$ is strictly confluent.

4.2.2 $\delta\gamma\beta\mu\eta$-Confluence

Next we aim at establishing the confluence of the overall reduction relation. To this end we investigate how the η-reduction cooperates with the other reduction relations. For pure λ-calculus, the $\beta\eta$-confluence was first proved in [8].

9. Definition Let $\to_r, \to_s$ be two reduction relations on EXP. The relation $\to_r$ is said *to commute with* $\to_s$, if for all $E, F, G \in EXP$ with $E \twoheadrightarrow_r F$ and $E \twoheadrightarrow_s G$ there is an $H \in EXP$ with $F \twoheadrightarrow_s H$ and $G \twoheadrightarrow_r H$.

Thus a reduction relation on EXP is confluent iff it commutes with itself.

10. Proposition $\to_{\delta\gamma\beta\mu}$ commutes with $\to_\eta$.

Using the 'Lemma of Hindley and Rosen', we obtain the main result of this section:

11. Theorem $\to_{\delta\gamma\beta\mu\eta}$ is confluent.

This deduction result shows that the call-by-value η-reduction neatly fits into the calculus.

4.3 Relating Denotational and Reduction Semantics

The simple results from first order functions over flat cpos (compare, for example, [15], Section 5.3) do not carry over to higher order functions, since the corresponding function spaces are non-flat domains. First we note that an expression with an undefined meaning cannot reduce to a value.

12. Proposition If $[\![E]\!](\rho) = \bot^{\mathbf{t}}$ and $E \twoheadrightarrow^{\mathbf{t}}_{\delta\gamma\beta\mu(\eta)} E'$ then $E' \notin VL^{\mathbf{t}}$.

In particular, a closed expression with an undefined meaning has no normal form.

13. Proposition No $E \in CMB$ with $[\![E]\!](\rho) = \bot^{\mathbf{t}}$ for some (and hence for all) $\rho \in ENV$ has a $\delta\gamma\beta\mu(\eta)$-normal form.

On the contrary, if a closed expression with a defined meaning has a normal form, then it is a value.

14. Proposition If $E \in CMB^{\mathbf{t}}$ has a $\delta\gamma\beta\mu(\eta)$-normal form, then $[\![F]\!](\rho) \neq \bot^{\mathbf{t}}$ for some (and hence for all) $\rho \in ENV$.

A closed expression of higher type with a defined meaning need not have a normal form.

15. Example The expression $constzero^{t\to nat}$ for the constant zero function has a $\delta\gamma\beta\mu$-normal form, since it is in $\delta\gamma\beta\mu$-normal form. The pointwise undefined function $omega^{r\to s}$ has no $\delta\gamma\beta\mu$-normal form, since it μ-reduces only to itself. Both expressions are closed and have a defined meaning. The same argument holds for their $\delta\gamma\beta\mu\eta$-normal forms.

In summary, we get the following adequacy result for closed expressions of arbitrary type.

16. Theorem For all $E \in CMB^t$ with $[\![E]\!](\rho) \neq \bot^t$ for some (and hence for all) $\rho \in ENV$ there exists $U \in VL^t$ with $E \twoheadrightarrow_{\delta\gamma\beta\mu} U$.

For ground types, the adequacy result can be strengthened.

17. Corollary Let $E \in CMB^s$ with $s \in \mathcal{S}$. If $[\![E]\!](\rho) = [\![U]\!](\rho)$ for some $U \in VL^s$ and $\rho \in ENV$, then $E \twoheadrightarrow_{\delta\gamma\beta\mu} U$.

Cor. 17 does not hold for higher types since functions can be represented by equivalent, but non-convertible expressions.

18. Example The expression $constzero^{nat\to nat}$ for the constant zero function is a value and semantically equivalent with $(z \in X^{nat\to nat}, x \in X^{nat})$

$$reczero^{nat\to nat} \equiv \mu z.\lambda x.\,\textbf{if}\ iszero(x)\ \textbf{then}\ zero\ \textbf{else}\ z(pred(x))\ \textbf{fi}\ .$$

The expressions $constzero^{nat\to nat}$ and $reczero^{nat\to nat}$ are not convertible. In particular, we do not have $reczero^{nat\to nat} \twoheadrightarrow_{\delta\gamma\beta\mu} constzero^{nat\to nat}$.

5 Conclusion

The reduction can be narrowed to an *operational semantics* by endowing it with an evaluation strategy. The operational semantics is a non-compatible relation usually defined by structural transition rules.

The *evaluation relation* $\Rightarrow = (\Rightarrow^t)_{t\in\mathcal{T}}$ of expressions comprises the local transitions

$pred(zero) \Rightarrow^{nat} zero$ $pred(succ^{n+1}(zero)) \Rightarrow^{nat} succ^n(zero)\quad (n \in \mathcal{N})$ $iszero(zero) \Rightarrow^{bool} true$ $iszero(succ^{n+1}(zero)) \Rightarrow^{bool} false\quad (n \in \mathcal{N})$
$\textbf{if}\ true\ \textbf{then}\ F\ \textbf{else}\ G\ \textbf{fi} \Rightarrow^t F$ $\textbf{if}\ false\ \textbf{then}\ F\ \textbf{else}\ G\ \textbf{fi} \Rightarrow^t G$
$(\lambda x.E)(W) \Rightarrow^s [E]_x^W$
$\mu x.E \Rightarrow^t [E]_x^{\mu x.E}$

equipped with the strategy

If $E \Rightarrow^{bool} E'$, then $\textbf{if}\ E\ \textbf{then}\ F\ \textbf{else}\ G\ \textbf{fi} \Rightarrow^t \textbf{if}\ E\ \textbf{then}\ F\ \textbf{else}\ G\ \textbf{fi}$;
If $F \Rightarrow^{r\to s} F'$, then $F(E) \Rightarrow^s F'(E)$;
If $E \Rightarrow^r E'$, then $F(E) \Rightarrow^s F(E')$.

Thus in a conditional expression, the condition is evaluated first. In an application, the operator is evaluated and — according to the call-by-value strategy — also the operand.

The evaluation relation is confluent, since it contracts only disjoint redexes. In contrast to reduction, the evaluation simplifies an expression of higher type only partially, since the body of an abstraction is not evaluated. Therefore an expression may have an $\Rightarrow$-normal form, although is has no $\delta\gamma\beta\mu$-normal form.

In particular, the expression $omega^{r\to s}$ denoting the pointwise undefined function is in $\Rightarrow$-normal form, although it has no $\delta\gamma\beta\mu$-normal form. On the contrary, the expression $error^{r\to s}$ denoting the undefined function has neither a $\Rightarrow$-normal form nor a $\delta\gamma\beta\mu$-normal form. This different evaluation behaviour of the expressions $omega^{r\to s}$ and $error^{r\to s}$ motivated the presented semantic model which carefully discriminates between the undefined function and the pointwise undefined function.

References

[1] S. Abramsky: *The Lazy λ-Calculus.* In: D.E. Turner (ed.): Logical Foundations of Functional Programming. Addison-Wesley 1990, 16–65

[2] A. Asperti: *Integrating Strict and Lazy Evaluation: the λ_{sl}-Calculus.* In: P. Deransart, J. Małuszynski (eds.): Programming Language Implementation and Logic Programming (PLIP '90). Lecture Notes in Computer Science *456*. Berlin: Springer 1990, 238–254

[3] H.P. Barendregt: *The Lambda Calculus — Its Syntax and Semantics.* Revised edition, second printing. Studies in Logic and the Foundations of Mathematics *103*. Amsterdam: North-Holland 1985

[4] H.P. Barendregt: *Lambda Calculi with Types.* In: S. Abramsky, D.M. Gabby, T.S.E. Maibaum (eds.): Handbook of Logic in Computer Science, Vol. 2. Background: Computational Structures. Oxford: Clarendon Press 1992, 117–309

[5] F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson (†), M. Wirsing, H. Wössner: *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L.* Lecture Notes in Computer Science *183*. Berlin: Springer 1985

[6] A. Church: *The Calculi of Lambda Conversion.* Annals of Mathematical Studies *6*. Princeton: Princeton University Press 1941

[7] H.B. Curry: *Functionality in Combinatoric Logic.* Proceedings National Academy of Science 1934, 584–590

[8] H.B. Curry, R. Feys, W. Craig: *Combinatoric Logic.* Volume I. Amsterdam: North-Holland 1958

[9] N. Dershowitz, J.-P. Jouannaud: *Rewrite Systems.* In: J. van Leeuwen (ed.): Handbook of Theoretical Computer Science, Vol. B. Formal Methods and Semantics. Amsterdam: North-Holland/Cambridge, Mass.: The MIT Press 1990, 243–320

[10] W. Dosch: *Reduction Relations in Strict Applicative Languages.* In: D. Dolev, Z. Galil, M. Rodeh (eds.): Theory of Computing and Systems. Lecture Notes in Computer Science *601*. Berlin: Springer 1992, 55–66

[11] C.A. Gunter: *Semantics of Programming Languages — Structures and Techniques.* Cambridge, Mass.: The MIT Press 1992

[12] C.A. Gunter, D.S. Scott: *Semantic Domains.* In: J. van Leeuwen (ed.): Handbook of Theoretical Computer Science, Vol. B. Formal Methods and Semantics. Amsterdam: North–Holland/Cambridge, Mass.: The MIT Press 1990, 655–674

[13] J.R. Hindley, J.P. Seldin: *Introduction To λ-Calculus and Combinators.* London Mathematical Society Student Texts *1*. London: Cambridge University Press 1986

[14] J.W. Klop: *Term Rewriting Systems.* In: S. Abramsky, D.M. Gabby, T.S.E.Maibaum (eds.): Handbook of Logic in Computer Science, Vol. 2. Background: Computational Structures. Oxford: Clarendon Press 1992, 1–116

[15] J. Loeckx, K. Sieber: *The Foundations of Program Verification.* Stuttgart: Teubner/Chicester: John Wiley et Sons 1984

[16] R. Milner: *Fully Abstract Models of Typed Lambda-Calculi.* Theoretical Computer Science *4*, 1–22 (1977)

[17] P. Mosses: *Denotational Semantics.* In: J. van Leeuven (ed.): Handbook of Theoretical Computer Science, Vol. B. Formal Methods and Semantics. Amsterdam: North–Holland/Cambridge, Mass.: The MIT Press 1990, 577–632

[18] G.D. Plotkin: *Call-By-Name, Call-By-Value and the λ-Calculus.* Theoretical Computer Science *1*, 125–159 (1975)

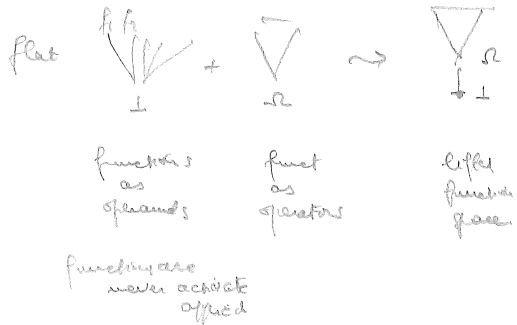
[19] G.D. Plotkin: *LCF Considered as a Programming Language.* Theoretical Computer Science *5*, 223–256 (1977)

[20] D. Schmidt: *Denotational Semantics: A Methodology for Language Development.* New York: Allyn and Bacon 1986

[21] D.S. Scott: *A Type-Theoretical Alternative to* CUCH, ISWIM, OWHY. Unpublished manuscript 1969. Theoretical Computer Science *121*, 411–440 (1993)

[22] D.S. Scott: *Lectures on a Mathematical Theory of Computation.* In: M. Broy, G. Schmidt (eds.): Theoretical Foundations of Programming Methodolgy. NATO Advanced Institute Series D. Dordrecht: Reidel 1982, 145–292

[23] G. Winskel: *The Formal Semantics of Programming Languages.* Cambridge: MIT Press 1993



EXTENSIONS AND INTEGRATION

- loose extensionality.

Then: generally extensional

2 funct expressions of the same type are exten. equiv.

they both denote the undefin function

& they both denote proper function and they are extensionally equivalent.

LOO: An Object-Oriented Logic Programming Language

P. Mancarella, A. Raffaetà and F. Turini
Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
Phone: +39 50-887253 Fax: +39 50-887226
{paolo,raffaeta,turini}@di.unipi.it

Abstract

Object-oriented programming has proven to be appropriate for the construction of complex software systems. On the other hand, logic programming stands out for its declarative flavor, built-in inference capabilities and well defined semantics.

We present a language, called LOO, which combines object-oriented programming and logic programming. We model classes as sets of clauses which represent their methods. An object is an instance of a class and it is identified by a unique name. We use a set of operators over theories for handling state changes and for modeling inheritance. A message sent to an object is translated into a goal which is solved with respect to a dynamic composition of clauses representing its class and its current state.

The challenge lies in avoiding the superimposition of a complex syntactic and semantic structure over the simple structure of logic programming. We have tried to extend logic programming in a conservative way as much as possible, in order to retain a simple and clear semantics.

Keywords : Logic programming, Object-Oriented programming, Program composition, Semantics

1 Introduction

Our long term goal is the definition of a unified language aimed at integrating the functionalities of programming, program specification, databases, knowledge representation and problem solving. There are at least two reasons for this endeavor: integrating different proposals in a common setting in order to have a common semantic foundation which allows a deeper comparison, and having a single language which can be used during the various phases of software production, from the requirements analysis down to the program solving the initial problem. Nowadays such a language does not exist, but we believe that the merging of two complementary paradigms, i.e. Object-Oriented Programming (OOP) and Logic Programming (LP), can yield it.

Logic programming combines clean and simple semantics, ease of use and expressive power. Its development, however, has shown that the basic formalism of Horn clauses does not suffice to deal with several computing problems. In this paper, we focus upon the limitation of logic programming deriving from the lack of abstraction mechanisms for structuring and modularizing programs. A logic program consists of a flat set of clauses and does not embody abstraction mechanisms which help in mastering the complexity of realistically sized applications. On the other hand, the OOP (see, e.g. [11, 12, 13]) is recognized as an excellent vehicle to simulate physical worlds, since physical entities and mental concepts can be directly represented as objects. Interactions among objects are modeled by message exchange. Every object belongs to a class

specifying the interface of the object with the external world (that is the set of messages accepted by the objects of the class). Classes also serve as templates for creating objects with the specified interface and implementation behavior. Inheritance is a mechanism for sharing the code common to a collection of classes. Inheritance collects shared properties of classes into superclasses and reuses them in the definition of subclasses. This allows an incremental style of programming. In fact, it is not necessary to modify the code of existing classes but it is possible to create new classes by specifying how they differ from those already defined. It is evident that, via inheritance, we can have forms of non-monotonic reasoning.

The main advantage of extending LP with OOP features is the acquisition of the OOP paradigm as a guiding principle for writing programs. This offers a simple but powerful model for encoding applications as computational entities, or objects, which communicate each other via message passing. An object is encapsulated: a client can access objects only by issuing requests for services. Clients cannot directly access or manipulate data associated with objects. In this way we increase LP with a form of abstraction and with the possibility of having information hiding. The class construct removes one of the weaknesses of LP, namely the lack of structuring mechanisms for programming in the large.

Nevertheless, most OO programming languages do not have formal semantics. Hopefully, valid semantics insights can be gained by integrating an OO programming language with a logic based language. Moreover, even if OO languages provide good abstraction mechanisms for structuring software, they do not support a declarative specification of software. If we combine OOP with LP we can use an OO approach for the representation of the problem domain and we can use logic to set constraints and rules.

The main obstacle to the integration seems to be that the LP paradigm does not support the notion of a mutable state which is an essential feature of the OO paradigm since it allows to model the inherent dynamics of physical entities. A mutable state implies a certain form of non-monotonic reasoning, since assertions which are provable in a given state may become invalid after a state change. The standard model theoretic semantics of LP disallows this non-monotonic behavior.

In the literature, many approaches to the integration of logic and object-oriented programming have been proposed. The survey by Davison [6] is a good, general introduction to these languages, showing how they deal with the representation of objects and classes, message passing and inheritance. The first proposal we mention here is McCabe's *objects as theories* [9], where objects and classes are represented by means of a single construct, the *class template*. A class template is formed by a set of clauses labeled with a Prolog term and a set of class rules which model inheritance. However, McCabe does not give state changes a logical characterization. On the contrary, Conery [7] wants just to model objects with changing states. He has adopted *objects as atoms*. Classes are defined through *object clauses*, i.e. clauses containing a conjunction of two atoms in their head. Roughly, the first head atom represents the object and its state, and the second atom represents a method. Finally, *objects as processes* were first proposed by Shapiro and Takeuchi [10]. An object is represented as a process which calls itself recursively and holds its internal state in unshared arguments. Objects communicate with each other by instantiating shared variables. This approach deals with the representation of state in a simple manner, and allows concurrency at the inter-object level. However, the semantics of languages based on this approach are quite complex because of the interaction between concurrency and Horn clause logic.

The proposal of this paper builds upon a long stream of research on the use of algebras of logic programs as a means for implementing common sense reasoning and program structuring [1, 3, 8]. In order to combine OOP and LP, we extend the language presented by Brogi et al. [4], which already offers basic mechanisms for building object-oriented features into logic programming. The advantage of our approach with respect to many others which have been proposed [6] is in the firm rooting of the semantics in a conservative extension of the semantics of pure logic programming.

2 Program expressions

The starting point of our work is the language of program expressions defined in [1, 3, 8]. This language is a conservative extension of logic programming consisting of moving from a single logic program to a collection of logic programs (theories), identified by names. Besides it provides a set of composition operators over such programs.

The language of program expressions Exp is defined by the following abstract syntax:

$$Exp ::= P \mid Exp \cup Exp \mid Exp \cap Exp \mid (Exp)^* \mid Exp \lhd Exp \mid Exp \prec P$$

where P is a plain program, i.e. a collection of clauses.

The semantics of a plain program P is taken to be its immediate consequence operator T_P. This choice allows one to give the semantics of the operators, $\cup \cap * \lhd \prec$, in a compositional way by expressing the meaning of a program expression in terms of the meanings of its component sub-expressions. Informally, union ($\cup$) and intersection ($\cap$) of program expressions lead to behaviors in which programs either cooperate or constrain each other step by step [1, 3]. Encapsulation ($*$) is a unary operation which supports a form of implementation hiding. The code of an encapsulated program is hidden to other programs, thus making the program behave differently when combined with others via binary operators. The *import* operation ($\lhd$) allows one to have a fine grained notion of information hiding/export [1, 3]. Finally, the restriction operator, $Q \prec P$, discards the clauses of a program expression Q which define predicates already defined in the program P.

These operators can be exploited to support notions of module and module compositions which encompass the essence of conventional modular programming languages, such as Ada [3]. Moreover, they can support forms of hierarchical reasoning, that is the definitions of inheritance relations between programs. For instance, the relation *isa*, which will be used in our object-oriented language, can be defined as follows. Let P and Q be logic programs, then P *isa* Q means that P inherits all the predicate definitions from Q, except for the predicates defined in P. This hierarchical relation can be modeled by the composition $P \cup (Q \prec P)$ (see [1]).

This language of program expressions is employed as a meta-language for composing programs written in a separate language, namely definite programs. In [4] a single language amalgamating the language in which programs are written (object language) and the language of program expressions (meta-language) is presented. Namely programs are *extended* definite programs in which clause bodies may contain meta-level calls to program expressions. More precisely, a program is a finite set of extended definite clauses of the form $A \leftarrow B_1, \ldots, B_n$ where each B_i is either an atomic formula or a meta-level formula of the form B *in* E, where B is an atomic formula and E is a program expression.

The *in* feature can be interpreted as a means of sending messages from a virtual program to another virtual program. Here, by virtual program we mean the collection of clauses which can be proved from a program expression.

Therefore, the amalgamated language can model message passing and inheritance. However, it is not suited to represent knowledge evolution because programs are still static and the language does not support any mechanisms to change them. We must extend the amalgamated language with constructs to deal with classes and with a notion of state in order to cope with the dynamic evolution of knowledge bases.

3 Syntax of LOO

The main constructs of the language LOO are:

- **class** *IdeCl*(*Pred_list*) {*Clauses*}
  with *Initial_State*
  to declare a class;

- *new(IdeCl,IdO)* to create objects;
- *update*($\{X \leftarrow\}$) to change the state of an object;
- *A in O* to model message passing.

IdeCl is a constant which identifies the class. A class has three components: a set of fresh state predicate names (*Pred_list*), a set of extended clauses ($\{Clauses\}$) and a set of unit clauses (*Initial_State*).

A state predicate is a predicate that can be modified by the *update* predicate. It cannot be defined inside $\{Clauses\}$, where, however, it can be used. Indeed state predicates are the only predicates whose definitions can be changed. They fulfill a role similar to instance variables in conventional OO programming languages.

$\{Clauses\}$ is a theory with *extended* clauses of the form: $A \leftarrow B_1, \ldots, B_n (n \geq 0)$. A is an atom, whose predicate name is neither *update*, nor *new*, nor a state predicate, and B_i is either:

- an atom $p(t_1, \ldots, t_m)$ where all t_j are terms which might include the self keyword and p can be a state predicate, or
- a meta-level formula B *in* O where B is an atom and O can be a variable, an object identifier or the self keyword, or
- a meta-level formula $new(IdeCl, IdO)$ where *IdeCl* is a class name or a variable and *IdO* is an object identifier or a variable, or
- a meta-level formula $update(\{q(t_1, \ldots, t_k) \leftarrow\})$ where q is a state predicate. We suppose that if *update* is the predicate of B_i then *update* is the predicate of every B_j $(i \leq j \leq n)$, too. As a shorthand, $update(\{X_1 \leftarrow, \ldots, X_r \leftarrow\})$ stands for $update(\{X_1 \leftarrow\}), \ldots, update(\{X_r \leftarrow\})$.

The constraint that requires updates to appear only at the end of clause bodies, goes in the direction of keeping method definitions as declarative as possible by compelling assignments to occur only at the end of the computation of methods.

The self keyword is used to permit self communication. It is a special term which has a value dependent on the context: it stands for the identifier of the active object. We shall further explore the role of self in Section 4.3.

Initial_State is a set of unit clauses which define the state predicates of the class. We can consider these definitions as default definitions: when an object of the class *IdeCl* is created its initial state is just this set of unit clauses.

An object is created by the *new* predicate. *new(IdeCl,IdO)* means that an object identified by *IdO* and belonging to the class *IdeCl* is created. *IdO*, a ground term, is the unique name of the object: this name models object identity. Indeed an object is an instance of a class: it has the methods defined in its class and its own state represented by a virtual program of unit clauses defining the current values of its class state predicates. In this way the set of clauses modeling an object is obtained by a meta-level composition of these two theories. This view of an object is similar to [5], where, however, there is no notion of class.

It is worth observing that the theory $\{Clauses\}$ of a class is a sort of parametric theory where parameters are just the state predicates. The fact that state predicates have no definition in $\{Clauses\}$ allows us to modify their values simply by combining $\{Clauses\}$ with different sets of unit clauses defining the current values of state predicates. In fact, this is how we implement state changes. The resolution of $update(\{X \leftarrow\})$ modifies only the state theory.

The meta-level formula A *in* O is used to send the message A to the object O. Conceptually, the message is treated as a goal to be solved by using the clauses that model the object O, that is A must be solved in the composition of the class of O with the current state of O.

In order to relate different classes to each other by means of inheritance, inside $\{Clauses\}$ we use unit clauses of the kind: $link(Super) \leftarrow$ where *Super* is a class name. *Super* is a superclass of

the class *IdeCl* which identifies $\{Clauses\}$. It is worth noting that this language naturally supports multiple inheritance by allowing more than one *link* clause in $\{Clauses\}$.

The inheritance rules state that a subclass inherits state predicates and methods from its superclass(es). In addition to the inherited state predicates, a subclass can declare fresh state predicate names, different from inherited ones. Methods in the subclasses override inherited methods. According to these rules, we define the function $StatePred : Class \rightarrow Pred$ as follows:

For each $IdCl \in Class$ declared by
class $IdCl(Pred\_list)$ $\{Clauses\}$
with *Initial_State*

$$StatePred(IdCl) = \begin{cases} Pred\_list & \text{if no } link \text{ clause belongs to } \{Clauses\} \\ Pred\_list \cup \Pi & \text{if } link(Super) \leftarrow \text{ belongs to } \{Clauses\} \\ & \text{and } StatePred(Super) = \Pi \end{cases}$$

For each class, this function returns, as its value, the set of all its state predicates, both proper ones and inherited ones. *Class* is the set of all class names and *Pred* is the powerset of state predicate names. In the above definition we support only single inheritance, even though it is easy to extend it in order to deal also with multiple inheritance. In fact, in this case we have more than one *link* clause. Therefore, Π is the union of all the state predicates of these superclasses.

Finally, a **LOO program** is a set of class declarations along with a main program. A main program is an extended program, where the meta-predicate *new* is used to create objects, and the meta-predicate *in* is used to send messages to the newly created objects. In the main program we cannot use the *update* predicate and the self keyword because such a program is not an object, whereas *update* and self are only related to object management.

With respect to the amalgamated language of [4] we have a further level of abstraction represented by objects. Besides, the operators for building program expressions are used here as implementation means for the realization of the object-oriented features of LOO. On the other hand, program expressions are not allowed in the *in* construct where one can refer only to objects. In the next section we will see how program expressions are used to implement the object-oriented features of our language.

Example 1 *We present the classes* stack *and* stacknum *that is a subclass of* stack. *In* stacknum *we have added a state predicate storing the number of elements in the stack.*

class *stack(list)*
 $\{push(X) \leftarrow list(S),$
  $update(\{list([X \mid S]) \leftarrow\})$
  $pop(X) \leftarrow list([X \mid S]),$
  $update(\{list(S) \leftarrow\})$
  $top(X) \leftarrow list([X \mid S])\ \}$
with $\{list([]) \leftarrow\}$

class *stacknum(num)*
 $\{push(X) \leftarrow list(S), num(N),$
  $update(\{list([X \mid S]) \leftarrow, num(s(N)) \leftarrow\})$
  $pop(X) \leftarrow list([X \mid S]), num(s(N)),$
  $update(\{list(S) \leftarrow, num(N) \leftarrow\})$
  $link(stack) \leftarrow\}$
with $\{num(0) \leftarrow, list([]) \leftarrow\}$

The main program is:

P: $r(a) \leftarrow$
 $go(X) \leftarrow new(stack, st), push(X)$ *in* st
 $p(X, Z) \leftarrow pop(Y)$ *in* $X, top(Z)$ *in* X
 $q(X) \leftarrow new(stacknum, X), r(Y), push(Y)$ *in* X

In order to illustrate the use of StatePred, notice that
 $StatePred(stack) = \{list\}$ *and* $StatePred(stacknum) = \{list, num\}$

4 Semantics of LOO

We present a semantics for the language by means of a transition system, called S_{ogg}. It has two types of configurations:

- $\langle \mathcal{E}, \mathcal{P}, IdO, G \rangle$ representing that the goal G has to be solved from the multi-object environment $\mathcal{E}$ and by using the clauses of $\mathcal{P}$;
- $\langle \mathcal{E}, \mathcal{P}, IdO \rangle$ representing a terminal configuration.

Formally, the set of configurations of the S_{ogg} system is:

$$\Gamma_{ogg} = \{\langle \mathcal{E}, \mathcal{P}, IdO, G \rangle | \ \mathcal{E} \in Env,\ \mathcal{P} \in PExp,\ IdO \in Obj \cup \{main\},\ G \in Goal\}$$
$$\cup\ \{\langle \mathcal{E}, \mathcal{P}, IdO \rangle \mid\ \mathcal{E} \in Env,\ \mathcal{P} \in PExp,\ IdO \in Obj \cup \{main\}\}$$

We now describe the various components of a generic configuration $\langle \mathcal{E}, \mathcal{P}, IdO, G \rangle$.

$\mathcal{E}$ is the multi-object environment which represents the set of objects that have been created up to now. For each object, we are interested in its name, the class it belongs to, and its current state. This set is denoted by a sequence defined by the following abstract syntax:

$$Env ::= \emptyset \mid (IdO, Cl, S) \mid Env :: Env$$

where $Cl \in Class$, IdO is an object identifier and S is a program expression formed by theories defining only state predicates and composed together via the *isa* operation. $\emptyset$ represents the empty sequence: no object has yet been created. Via the *new* predicate we add new tuples, that is objects, to the environment, whereas via the *update* predicate we replace tuples of the environment with new ones where only the third component, that is the state, has changed.

$\mathcal{P}$ is a program expression which represents the theory where the goal G is executed. Initially, $\mathcal{P}$ is the main program. Then, when a message is sent to an object O, $\mathcal{P}$ becomes a program expression representing the theory associated with the object O. The set of program expressions $PExp$ is defined by the following abstract syntax:

$$PExp ::= P \mid PExp \cup PExp \mid P\ isa\ PExp \mid PExp \prec P \mid (PExp)^*$$

where P is a collection of extended clauses as defined in Section 3. When we use a class name in a program expression, this name stands for the theory $\{Clauses\}$ of this class. In the following, program expressions are ranged over by calligraphic capital letters, such as $\mathcal{P}$ or $\mathcal{Q}$, while plain programs are ranged over by plain capital letters such as P or Q. We have considered a subset of the language of program expressions defined in Section 2 because $\cup$, $\prec$, $*$ and *isa* suffice to support the object-oriented features.

IdO is *main* when $\mathcal{P}$ is the main program. Otherwise it is the identifier of the object in which we solve G. Any change, determined by G provability, affects the state of IdO. Obj is the set of object identifiers which are ground terms.

G is the goal to solve. G may contain atoms whose predicates are defined in the main program, or which can create objects by *new*, or which can send messages to those objects via meta-level constructs of the form A *in* O. $Goal$ is the set of clauses of the form $\leftarrow B_1, \ldots, B_m$ where B_i can be:

- an atom $p(t_1, \ldots, t_n)$;
- A *in* O where A is an atom and O can be a variable or an object identifier or the keyword **self**.

Terminal configurations are characterized by the lack of the *Goal* component: if we have $\langle \mathcal{E}, \mathcal{P}, IdO, G \rangle$, as initial configuration, and we derive a terminal configuration from it, this derivation is a refutation for G.

Given a generic program with main program P and a general goal G, then the initial configuration of the system S_{ogg} is: $\langle \emptyset, P, main, G \rangle$.

A critical issue is the kind of inheritance supported by the language. In fact, in logic programming, the knowledge about a predicate is available at two different levels: the *intensional* and the *extensional* level. The former is represented by the collection of clauses defining the predicate. The latter is the set of atomic formulae provable for that predicate. For example, consider the following program (the main program is empty) :

class $Cl_1()$	**class** $Cl_2()$
$\{r(a) \leftarrow$	$\{r(b) \leftarrow$
$p(X) \leftarrow r(X)\}$	$link(Cl_1) \leftarrow\}$
with $\{\}$	**with** $\{\}$

and the goal $\leftarrow new(Cl_2, ob), p(X)$ *in* ob.
If we use atomic inheritance (extensional level) the computed answer substitution is $\{X \leftarrow a\}$ because ob inherits the relations p, r from the superclass Cl_1. On the other hand, if we adopt inheritance at an intensional level, ob inherits the definition of p, i.e. the clause $p(X) \leftarrow r(X)$, from Cl_1 and hence computes $\{X \leftarrow b\}$ as an answer substitution since the derived goal $r(X)$ is solved using the definition of r in Cl_2, which overrides the definition in Cl_1.

In the next section we show how the inference rules can accommodate both approaches to the semantics of inheritance.

4.1 Atomic Inheritance

The definition of the transition relation is given by rules of the form

$$\frac{C_1\ C_2\ \ldots\ C_n}{\gamma \rightarrow \gamma'}$$

where $\gamma \rightarrow \gamma'$ (*Conclusion*) holds whenever $C_1, C_2, \ldots, C_n$ (called *Premises*) hold. Besides we use the notation $\gamma \xrightarrow{\theta} \gamma'$ to indicate that θ is the computed answer substitution for the variables of the goal in γ configuration and $\gamma \xrightarrow{\theta^*} \gamma'$ to express that exists a derivation $\gamma_0 \xrightarrow{\theta_1} \gamma_1 \xrightarrow{\theta_2} \ldots \xrightarrow{\theta_i} \gamma_i$ satisfying $\gamma_0 = \gamma$, $\gamma_i = \gamma'$ where γ' is a terminal configuration, and $\theta = \theta_1 \circ \ldots \circ \theta_i$.

Let us first present the transition rules ((1)-(5)) to derive new configurations when $\mathcal{P}$ is a complex expression. The meaning of the operators $\cup$, $*$, $\prec$ and *isa* is not modified, but it is simply adapted to the multi-object environment.

$$\frac{\langle \mathcal{E}, \mathcal{P}, Ogg, A \rangle \xrightarrow{\theta} \langle \mathcal{E}', \mathcal{P}, Ogg, G \rangle}{\langle \mathcal{E}, \mathcal{P} \cup \mathcal{Q}, Ogg, A \rangle \xrightarrow{\theta} \langle \mathcal{E}', \mathcal{P} \cup \mathcal{Q}, Ogg, G \rangle} \quad (1)$$

$$\frac{\langle \mathcal{E}, \mathcal{Q}, Ogg, A \rangle \xrightarrow{\theta} \langle \mathcal{E}', \mathcal{Q}, Ogg, G \rangle}{\langle \mathcal{E}, \mathcal{P} \cup \mathcal{Q}, Ogg, A \rangle \xrightarrow{\theta} \langle \mathcal{E}', \mathcal{P} \cup \mathcal{Q}, Ogg, G \rangle} \quad (2)$$

These rules state that either program may be used to derive a new configuration. Since any subexpression ($\mathcal{P}$ or $\mathcal{Q}$) can contain an encapsulated subexpression of the form $\mathcal{R}^*$, the overall environment can be modified, as we are going to explain next.

$$\frac{\langle \mathcal{E}, \mathcal{P}, Ogg, A \rangle \xrightarrow{\theta^*} \langle \mathcal{E}', \mathcal{P}', Ogg \rangle}{\langle \mathcal{E}, \mathcal{P}^*, Ogg, A \rangle \xrightarrow{\theta} \langle \mathcal{E}', \mathcal{P}^*, Ogg, empty \rangle} \quad (3)$$

This rule states that we can derive a new configuration if A is provable in $\mathcal{P}$. The resolution of A may involve state changes which may modify the overall environment.

$$\frac{\langle \mathcal{E}, \mathcal{Q}, Ogg, A\rangle \xrightarrow{\theta} \langle \mathcal{E}', \mathcal{Q}, Ogg, G\rangle \qquad pred(A) \notin defs(P)}{\langle \mathcal{E}, \mathcal{Q} \prec P, Ogg, A\rangle \xrightarrow{\theta} \langle \mathcal{E}', \mathcal{Q} \prec P, Ogg, G\rangle} \tag{4}$$

Given a program expression $\mathcal{Q}$ and a plain program P, if we can rewrite A with G in $\mathcal{Q}$, then it is possible to rewrite A with G in $\mathcal{Q} \prec P$ only if the predicate of A is not a predicate defined in P. *defs(P)* is the set of predicates defined in P.

$$\frac{\langle \mathcal{E}, P \cup (\mathcal{Q} \prec P), Ogg, A\rangle \xrightarrow{\theta} \langle \mathcal{E}', P \cup (\mathcal{Q} \prec P), Ogg, G\rangle}{\langle \mathcal{E}, P \; isa \; \mathcal{Q}, Ogg, A\rangle \xrightarrow{\theta} \langle \mathcal{E}', P \; isa \; \mathcal{Q}, Ogg, G\rangle} \tag{5}$$

Recall that P *isa* $\mathcal{Q}$ (see Section 2) means that P inherits all the predicate definitions from $\mathcal{Q}$, except for the predicates defined in P. Rule (5) states that the *isa* relation can be modeled by union and restriction operators.

The following four rules model SLD resolution.

$$\frac{}{\langle \mathcal{E}, \mathcal{P}, X, empty\rangle \xrightarrow{\{\}} \langle \mathcal{E}, \mathcal{P}, X\rangle} \tag{6}$$

This rule states that the empty goal is solved in any program expression and it allows us to obtain a terminal configuration.

Given a set of predicate names π, let 1_π denote the program: $\{p(X_1, \ldots, X_n) \mid p \in \pi\}$. We use this kind of programs in the following rule:

$$\frac{\langle \mathcal{E}, \mathcal{P}, Ogg, G_1\rangle \xrightarrow{\theta}{}^{*} \langle \mathcal{E}', \mathcal{P}', Ogg\rangle \qquad \mathcal{E}' = E_1 :: (Ogg, Cl, S) :: E_2}{\langle \mathcal{E}, \mathcal{P}, Ogg, (G_1, G_2)\rangle \xrightarrow{\theta} \langle \mathcal{E}', S \cup (\mathcal{P} \prec 1_{StatePred(Cl)}), Ogg, G_2\theta\rangle} \tag{7}$$

$$\frac{\langle \mathcal{E}, P, main, G_1\rangle \xrightarrow{\theta}{}^{*} \langle \mathcal{E}', P, main\rangle}{\langle \mathcal{E}, P, main, (G_1, G_2)\rangle \xrightarrow{\theta} \langle \mathcal{E}', P, main, G_2\theta\rangle} \tag{8}$$

These rules allow us to solve a conjunction of goals (G_1, G_2). The leftmost goal is removed from the current goal statement and the system tries to solve it. Our system is sensitive to the order in which methods are called. We lose *referential transparency* and independence of selection rule. This is the price to pay for allowing objects with changing states.

It is worth observing that the environment $\mathcal{E}$ can change, as a side effect of the successful resolution of G_1. If the third component of the configuration is an object identifier, it is necessary to change $\mathcal{P}$, too, because $\mathcal{P}$ "contains" the old state of Ogg. Instead G_2 will have to be solved by using the current object state S. The new program expression will be $S \cup (\mathcal{P} \prec 1_{StatePred(Cl)})$: $\prec$ is the restriction operator which "retracts" from $\mathcal{P}$ the state predicates definitions. This is not the case if the conjunctive goal is computed with respect to the main program, which has no state, as it is axiomatized by rule (8).

$$\frac{A' \leftarrow G \in P \qquad \theta = mgu(A, A') \qquad P \text{ is a set of clauses}}{\langle \mathcal{E}, P, Ogg, A\rangle \xrightarrow{\theta} \langle \mathcal{E}, P, Ogg, G\theta\rangle} \tag{9}$$

This rule states that, to solve an atomic goal A, choose a clause $(A' \leftarrow G)$ from P whose head is unifiable with A. Let θ be the m.g.u. among A and A' then recursively solve $G\theta$ in P.

The following rule models atomic inheritance:

$$\frac{\mathcal{E} = E_1 :: (Ogg, Cl, S) :: E_2 \qquad link(C) \leftarrow\in P \qquad \langle \mathcal{E}, (C \cup S)^* \prec P, Ogg, A\rangle \xrightarrow{\theta} \langle \mathcal{E}', (C \cup S)^* \prec P, Ogg, empty\rangle}{\langle \mathcal{E}, P, Ogg, A\rangle \xrightarrow{\theta} \langle \mathcal{E}', P, Ogg, empty\rangle} \tag{10}$$

This rule enriches the set of atoms provable in P with relations inherited from a superclass except for predicates defined in P. By means of the $*$ operation we import the atomic formula $A \leftarrow$ from $C \cup S$. Recall that a class C is a sort of parametric theory. Therefore, in order to derive the class relations, we must join it with S, that contains the current definitions of its parameters. The inherited relations vary according to the current state of the active object.

We now define the rule dealing with *update*.

$$\frac{\mathcal{E} = E_1 :: (Ogg, Cl, S) :: E_2}{\langle \mathcal{E}, \mathcal{P}, Ogg, update(\{X \leftarrow\})\rangle \xrightarrow{\{\}} \langle E_1 :: (Ogg, Cl, \{X \leftarrow\} \; isa \; S) :: E_2, \mathcal{P}, Ogg\rangle} \tag{11}$$

This rule states that solving a goal of the form $update(\{X \leftarrow\})$ leads the system to a terminal configuration in which $\mathcal{E}$ has changed: the tuple related to Ogg in $\mathcal{E}$ is replaced by a new one where the current state becomes $\{X \leftarrow\}$ *isa* S. The definition of the predicate of X, which belongs to S, is overridden by $X \leftarrow$.

The next rule deals with the creation of objects.

$$\frac{undefined(O, \mathcal{E}) \qquad O \in Obj}{\langle \mathcal{E}, \mathcal{P}, Ogg, new(Cl, O)\rangle \xrightarrow{\{\}} \langle (O, Cl, InitState(Cl)) :: \mathcal{E}, \mathcal{P}, Ogg\rangle} \tag{12}$$

This rule states that solving a goal of the form $new(Cl, O)$ leads the system to the terminal configuration $\langle (O, Cl, InitState(Cl)) :: \mathcal{E}, \mathcal{P}, Ogg\rangle$ where we have added a tuple, representing the new object, to the environment. This tuple contains the object identifier (O), the class name Cl and the initial state of every object belonging to Cl (*InitState(Cl)*). The object identifier O must be ground ($O \in Obj$) and unique (*undefined(O,E)*). The predicate *undefined(X, Y)* is true if no tuple with the first component X exists in the environment Y.
When an object is created, it becomes visible to any other object, to the main program and to the top-level, because we have a unique multi-object environment.

$$\frac{\mathcal{E} = E_1 :: (O, Cl, S) :: E_2 \qquad \langle \mathcal{E}, (Cl \cup S)^* \prec 1_{(StatePred(Cl) \cup \{update\})}, O, A\rangle \xrightarrow{\theta} \langle \mathcal{E}', \mathcal{P}', O, empty\rangle}{\langle \mathcal{E}, \mathcal{P}, Ogg, A \; in \; O\rangle \xrightarrow{\theta} \langle \mathcal{E}', \mathcal{P}, Ogg\rangle} \tag{13}$$

This rule states that a goal of the form A *in* O is provable in $\mathcal{P}$ if the goal A is provable in the program expression $(Cl \cup S)^* \prec 1_{(StatePred(Cl) \cup \{update\})}$. $Cl \cup S$ is the theory currently associated with the object O: Cl is the class that O belongs to and S is the current state of O. We use the $*$ operator to hide the way A is solved in $Cl \cup S$. It is worth observing that the restriction $\prec 1_{(StatePred(Cl) \cup \{update\})}$ ensures that the predicate of A is neither a state predicate nor the *update* predicate. This ensures that the state is encapsulated: it cannot be directly read or modified. The sender of a message does not know which definitions are used to solve it. Moreover, notice that in the proof of A, the current object is O and no longer Obj.
By exploiting unification and non-determinism, we can use the above rule to search for an object where the goal A can be solved. This exploitation of the logical variable offers a mechanism much more powerful than the ones supported by traditional object-oriented languages.

$$\frac{}{\langle \mathcal{E}, \mathcal{P}, Ogg, A \; in \; \text{self}\rangle \xrightarrow{\theta} \langle \mathcal{E}, \mathcal{P}, Ogg, A \; in \; Ogg\rangle} \tag{14}$$

This rule states that the self keyword stands for the active object. The consequences of this definition are shown by an example in Section 4.3.

4.2 Inheritance at an intensional level

To model inheritance at an intensional level we must only change the transition rule (10).

$$\frac{link(C) \leftarrow \in P \qquad \langle \mathcal{E}, C \prec P, Ogg, A\rangle \xrightarrow{\theta} \langle \mathcal{E}', C \prec P, Ogg, G\rangle}{\langle \mathcal{E}, P, Ogg, A\rangle \xrightarrow{\theta} \langle \mathcal{E}', P, Ogg, G\rangle} \tag{15}$$

Via this rule we inherit the definitions of the superclass except for the predicates defined in P. In contrast with atomic inheritance, the set of inherited clauses does not depend on the state of the active object.

It is worth noting that, in particular, rules (10), (11), (13) and (15) show the expressive power of the operators $\cup$, $*$, $\prec$ and *isa*. They allow us to model naturally different forms of inheritance, state changes and message passing.

4.3 Discussion of inheritance schemes

The main difference between inheritance at the intensional level and atomic inheritance is that the former puts together clauses belonging to different classes, whereas the latter hides the code of a class to other classes, even to its descendants. Therefore we can use the superclass only as a black box, by querying the relations defined in it without accessing its clauses. This is why we talk about *delegation* [13].

A serious problem with intensional inheritance is that it can behave incorrectly when multiple inheritance is used, since classes may start to interact in unexpected ways. For example, consider the class C which is a subclass of the classes A and B that are not related to each other.

class $A()$
$\{r(X) \leftarrow q(X)$
$q(a) \leftarrow\}$
with $\{\}$

class $B()$
$\{q(b) \leftarrow\}$
with $\{\}$

class $C()$
$\{p(X) \leftarrow r(X)$
$link(A) \leftarrow$
$link(B) \leftarrow\}$
with $\{\}$

The goal $\leftarrow new(C, o), r(b)$ *in* o is solved, although A is unrelated to B.

The drawback of atomic inheritance is that only atomic consequences of a theory are inherited. However, a judicious use of the **self** keyword can have the effect of inheriting all the necessary consequences. Furthermore this is under programmer control. Consider the following example [9].

We want to define a class *animal* with a predicate *mode* expressing the fact that if an animal has two legs then it runs. Otherwise, if it has four legs, it gallops. This rule for mode of travel is valid for all types of animals but it depends on sub-relations which are specific to each class denoting particular species. To resolve this conflict we use **self** reference.

Let *animal* and *bird* be the following classes:

class *animal*()
$\{mode(run) \leftarrow n\_legs(2)$ *in* **self**
$mode(gallop) \leftarrow n\_legs(4)$ *in* **self**
with $\{\ \}$

class *bird*()
$\{covering(feathers) \leftarrow$
$n\_legs(2) \leftarrow$
$link(animal) \leftarrow\}$
with $\{\ \}$

To find out if the bird *tweety* runs or gallops, we solve the goal $\leftarrow$ *mode(X) in tweety*. Since *mode* is not defined in *bird*, via rule (10), it is delegated to the superclass *animal*. Here, we can use the clause *mode(run)* $\leftarrow$ *n_legs(2) in* **self** and by applying the rule (14) we bind **self** with *tweety*. This allows to "root" us back into the class to which the object belongs, that is at the root of the inheritance hierarchy. Therefore, we can correctly solve the goal $n\_legs(2)$.

It is worth noticing that if a class refers to **self** then it can only have a "complete" meaning in terms of its various specializations. This is a situation similar to the notion of an *abstract class* in conventional OO languages, such as *C++*.

Moreover, thanks to **self**, the inheritance at an intensional level can be considered as a particular case of atomic inheritance.

5 Conclusions

We have presented an integration of logic and object-oriented programming which combines fundamental features of both, taking the object-oriented view to model the problem domain, and logic programming to describe the entities and to provide the computation engine.

On the one side, the resulting language provides a logical understanding of object-oriented features. Typical object-oriented mechanisms, such as inheritance and message passing, can be understood in terms of deduction processes. On another side, it exemplifies a proper extension of typical object-oriented formalisms by allowing the logical definition of methods and hierarchical links. It overcomes the imperative flavor usually associated with messages, and it gives objects a more "intelligent" appearance. Finally, this language provides logic programming with abstraction mechanisms and a notion of mutable state. State is given a logical characterization, and this is one of the most interesting points. In fact, many logic programming-based object-oriented languages use the extra-logical Prolog primitives *assert* and *retract* to deal with state changes or they simply ignore state changes, thus lacking a vital feature of object-oriented programming.

LOO can be easily extended with a primitive *kill* for destroying objects. *kill(O)* destroys the object O and its semantics is simply obtained by deleting the tuple associated with O from the multi-object environment. Moreover, thanks to the restriction operator ($\prec$), we can allow private clauses in a class, that is clauses defining a predicate that is kept hidden (this corresponds to *C++ private* functions). Another possible extension of LOO consists in having meta-level clauses of the form A *in* E where E is a program expression of objects. For example, let O_1 and O_2 be object identifiers, then we can compose them in the expression $O_1 \cup O_2$ with the intended meaning of joining the theories modeling these objects. Therefore, on one side classes allow us to create objects dynamically with a well-defined behavior, and on the other side expressions allow us to combine them at execution time.

This language has a meta-logical implementation obtained by simply turning the inference rules of Section 4 into meta-level axioms. The axioms have the form of an extension to the vanilla metainterpreter. The meta-logic preserves the simple and concise description of the inference rules, and it provides an "executable specification" of LOO. This implementation is however costly in terms of efficiency. As a step towards a real system, we are trying to extend the compiler and the extended WAM proposed in [2] with the object-oriented features.

Acknowledgements

This research was partly supported by the ESPRIT BRA 6810 (Compulog 2) and by the EEC "Keep in Touch Activity" KITO11.

References

[1] A. Brogi. *Program Construction in Computational Logic.* PhD thesis, Univ. Pisa, March 1993.

[2] A. Brogi, A. Chiarelli, P. Mancarella, V. Mazzotta, D. Pedreschi, C. Renso, and F. Turini. Implementations of program composition operations. In M. Hermenegildo and J. Penjam, editors, *Proceedings PLILP 94*, pages 292–307. Springer-Verlag, 1994.

[3] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular Logic Programming. *ACM Transactions on Programming Languages and Systems*, 16(4):1361–1398, 1994.

[4] A. Brogi, C. Renso, and F. Turini. Amalgamating Language and Meta-Language for Composing Logic Programs. In M. Alpuente, R. Barbuti, and I. Ramos, editors, *Proc. of Gulp-Prode '94*, pages 408–422, 1994.

[5] A. Brogi and F. Turini. Metalogic for State Oriented Programming. In E. Lamma and P. Mello, editors, *Proc. ELP'92*, LNAI 660, pages 187–204. Springer-Verlag, Berlin, 1992.

[6] A. Davison. A Survey of Logic Programming-based Object Oriented Languages. In P. Wegner and G. Agha, editors, *Research directions in concurrent object-oriented programming*, pages 42–106. MIT Press, 1994.

[7] J.S.Conery. Logical Objects. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 420–434. The MIT Press, Cambridge, Mass., 1988.

[8] P. Mancarella and D. Pedreschi. An algebra of logic programs. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Inter. Conf. on Logic Programming*, p.1006-1023, 1988. MIT Press.

[9] F. G. McCabe. *Logic & Objects*. Int. Series in Computer Science. Prentice-Hall, 1992.

[10] E. Shapiro and A. Takeuchi. Object Oriented Programming in Concurrent Prolog. *New Generation Computing*, pages 25–48, 1983.

[11] A. Snyder. The Essence of Objects: Concepts and Terms. *IEEE Software*, p. 31-42, Jan. 93.

[12] D. Thomas. What's in an object. *BYTE*, pages 231–240, 1989.

[13] P. Wegner. Dimensions of Object-Based Language Design. In *Proc. of the OOPSLA '87, SIPLAN Notices*, volume 22:12, pages 168–182, 1987.

- Logic.
- Objects oriented
- constraints.

Forum & Objects

at Penn State Univ.

Giorgio Delzanno and Maurizio Martelli

D.I.S.I. - Università di Genova.

Viale Benedetto XV, 3, Genova 16132, Italy

phone:+39-10-3538034, fax:+39-10-3538028

email:{giorgio,martelli}@disi.unige.it

Abstract

The notions of **computation** and **state** are essential in computer science.

An operational semantics can easily describe the history of state changes occurring in a computation, while a declarative semantics, which is adequate and corresponding to the operational one, is much more difficult to obtain.

Linear Logic [10] gives some hints to solve this problem. We can look at a *proof* as the witness of a *computation*, i.e., as describing, in logical terms, the behavior of a system. It is interesting to study *object-oriented* based systems from this point of view; in this setting the state of the system consists of the set of the currently active objects.

In our work we will reconsider some proposals for the integration of object-oriented and linear logic programming. For this purpose we will exploit **Forum** [20], a presentation of *Higher-Order Linear Logic*, which is endowed with a characterizing property of logic programming languages, namely the uniformity of proofs.

In this paper we will outline our proposal by presenting a specialization of Forum for state-based computations and by showing how to exploit it for representing objects.

Keywords: Logic programming, object oriented and higher-order linear logic.

1 Introduction

Programming languages are based on the intuitive idea of *computation*. Assigning an operational semantics to the constructs of a language allows to describe the behavior, or the possible behaviors, of a program.

The more powerful is the language, the more complex becomes the operational description. As a consequence, the operational semantics can be unsuitable for studying properties of programs and it can be very difficult to find a more abstract semantics.

The **object-oriented** programming paradigm is probably the most important example of this situation. Here the challenge is to assign a clear meaning to the notion of **object**, an individual entity having a complex data structure, and to its interactions with the external world.

In the last years, many efforts have been spent in studying this problem from a logical point of view; there have been proposals both to give a formal logical meaning to some features of o.o. programming and to understand how to *integrate* them in a logic programming setting.

In our opinion, this second aspect is very important in order to develop logic programming tools able to deal with large applicative programs, i.e., able to perform reasoning over powerful data representations. In the following we will address some general considerations about o.o. and l.p..

Some basic notions of the o.o. methodology

We list here the basic properties the standard notion of *object* is endowed with. First of all, objects have unique individual names. Objects encapsulate both *data* and *methods*. Methods must be the only vehicles to access and modify the data of an object, i.e., to perform operations over them. Public methods can be invoked from outside an object, while private ones can be only invoked by the other methods of the considered object. The set of public methods represent the external interface of the object. Some o.o. languages also provide shared methods and public data variables.

An o.o. language provides constructs to create objects according to a given pattern, to kill them and, possibly, to modify their structure.

Definitely, **encapsulation** makes the difference among objects and other data structures. For instance, having methods inside an object allows to dynamically modify its behavior according to the conditions of the environment. A clear example is the *Object Calculus* [1] by Abadi and Cardelli, where objects consist of only method declarations.

Also, objects have been interpreted as perpetual processes communicating via message passing. See, for instance, [14, 17] where methods are nothing but rewriting rules transforming a configuration into another one.

A *class* can be considered as the description of a uniform set of objects. This notion can be complicated introducing, for instance, inheritance relationships and declarations of shared instance variables. Notice that inheritance can be considered as a specialization of the *delegation* mechanism.

In [26] Peter Wegner suggested the following classification: *object-based*, languages which provide the very basic notion of objects (e.g. the Actors language); *class-based*, object-based languages supporting the notion of *class*; *object-oriented*, class-based languages providing inheritance (e.g. Smalltalk). In this preliminary work we will focus our attention on *class-based* languages.

Some of the above aspects have been already studied from the logic programming

point of view. We will discuss some proposals in the following paragraph, with particular regard to the representation of the objects.

Integration between o.o. and l.p.

The first representation we mention here is *objects* as *variables*, the approach presented in [25] by Chen and Warren. They assigned a *dynamic* meaning to logical variables, i.e., a function from states to values, focusing on the problem of state updating.

The next one is *objects* as *terms*. For instance, Aït-Kaci and Nasr proposed the languages Login and Life [3], where terms have a record-like structure induced by their types, see also [23].

Objects as *atoms* is the representation closer to our approach. It was already adopted by Conery in [9]. In his approach the evolution of an object was determined by applying Prolog-like clauses with *conjunction* of atoms in the conclusion, in order to simultaneously rewrite the atom representing the object and the atom representing the message. The resulting operational mechanism was quite involved.

Linear Logic revealed itself well-suited for such a kind of approach, as shown by Andreoli and Pareschi in [7]. They substituted the classical conjunction in the head of methods with the linear logic multiplicative disjunction, considering an object as a *multiset of atoms*. We will return to their approach later.

The final representation that we consider is *objects* as *theories*. The original idea was due to MacCabe [16], which considered objects as named Prolog programs, without the possibility of changing their state. His approach stimulated the study of *modules* and *inheritance* as shown in Contextual Logic Programming [21]. Other approaches to modularity exploited connectives like *intuitionistic implication* and *universal quantification* for handling local variables [8].

Linear Logic and Forum

Girard's **Linear Logic** [10] allows to study the very basic notion of *computation* by different perspectives.

For instance, in the functional interpretation where *proofs* can be reduced to λ-terms by the Curry-Howard isomorphism, *cut elimination* is considered as the basic computational mechanism [2, 11].

It is also possible to interpret *proofs* as *computations*. The main point is to find the analogies between the structure of the rules of a *cut free* system and the operational steps of a computation, see for example [22] for a survey.

Linear logic sequent calculus does not allow the use of *contraction* and *weakening*. Consequently, each formula in a proof can be *used* only once, yielding the notion of bounded-use *resources*. Re-usable formulas are re-introduced in the logic by means of the exponentials, namely ! and ?.

Another consequence of the elimination of the above mentioned structural rules

is that two versions of connectives must be considered, namely additive and multiplicative ones.

In the Logic Programming setting, these aspects have been investigated among the others by Miller and Hodas [13, 20]. In particular Miller's work results in the definition of a powerful specification language called **Forum**, a presentation of full Higher-Order Linear Logic, see also [4], where the cut-elimination theorem holds. This language is based on the linear connectives $\parr$ (multiplicative disjunction), & (additive disjunction), $\circ\!\!-$ (linear implication), $\Rightarrow$ (intuitionistic implication), $\top$, $\bot$ and $\forall$.

The rules of the Forum proof system are designed to reflect the idea of **uniform proof**, i.e., a proof where left introduction rules are applied only after having decomposed all the formulas in the right hand side of sequents.

Considering simply typed λ-terms as elements of the language allows to have a higher-order theorem prover. This is very important in order to use the language as *meta-level* specification for programs and proofs. Using the previous notions we can briefly explain what is meant for *proof* as *computation*.

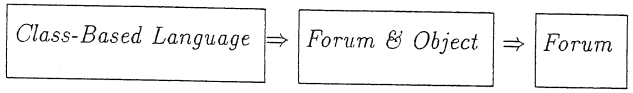
Let us consider a generic computation starting from a state S_i and ending with state S_f. The idea is to mimic it by a proof for the linear formula $S_i^* \circ\!\!- S_f^*$, where S_f^*, S_i^* are logical representations of S_f and S_i. Thus, one or more applications of Forum rules should mimic a single step in the computation.

Our approach

In our current work we are using Forum as a specification language for state-based systems, i.e., we have defined a simplified version of Forum, called $F\&O$, in order to express an explicit notion of state in the sequent and modeling its evolution using a subset of linear logic formulas.

The fragment of linear logic considered in our language consists of the connectives $\forall$, &, $\parr$, $\circ\!\!-$, $\top$ and $\bot$, with a further restrictions on the form of the formulas, in order to have multi-conclusion clause with $\parr$ of atomic formulas in the head and allowing the other connectives to occur in the body, see [7].

In the following scheme we show how $F\&O$ fits into the context of the integration between o.o. and l.p.



Thus, it is possible to give a logical counterpart to some object-oriented languages using an adequate embedding in Forum, using $F\&O$ as an intermediate stage.

Andreoli and Pareschi, in their basic paper on the language *LO* [5], represented an **object** as a multiset of atoms, i.e., the collection of its attributes, *floating* in parallel ($\parr$) in the right-hand side of a sequent. Using the additive conjunction (&) they *cloned* objects spreading them on different branches of a proof. In order to let

the objects communicate through messages, a common logical variable, acting as a blackboard, was added to the sequents [6].

We are going to reconsider such an approach using a different representation of objects (*objects* as *atoms*), in order to capture features like *encapsulation* and *data hiding* that were not treated there.

These features can be captured using the **higher-order** nature of Forum, and thus of $F\&O$, see, for instance, [18], as we will explain in the final part of the paper. Furthermore, we will simplify the communication model, representing in a sequent the *global current state* of a computation in an o.o. system.

To be fair to them, in their approach *inheritance* is very natural, while, for the time being, we have restricted our study to **class-based** systems.

In the paper, Section 2 will be devoted to describe the $F\&O$ formulas, while in Section 3 we will discuss the corresponding proof system. In Section 4 we will present the o.o. aspects of $F\&O$ and, finally, Section 5 will be devoted to the conclusions.

2 The language $F\&O$

Our staring point is a formulation of Forum [20] in which the right hand side of a sequent is a multiset. As previously mentioned, our aim is to isolate a notion of *state* in a linear logic sequent. The simplest approach is to use data constructors (i.e. predicate symbols) and to embed state values inside them. In the following we will assume to have a signature $\Sigma_{\mathcal{S}}$ containing a given set of *state constructors*. We will consider simply typed λ-calculus as the basis of our language to uniformly handle terms and formulas. Given a signature Σ containing $\Sigma_{\mathcal{S}}$, simply typed λ-terms can be defined as usual by induction. It is important to notice here that formulas are nothing but λ-terms with a particular type o.

We consider only the fragment of LL consisting of $\parr$, &, $\forall$, $\circ\!\!-$, $\top$ and $\bot$. The linear connectives can be represented as constants with an ad hoc type, thus, for instance, $\parr$ has type $o \to o \to o$ and $\forall_\tau$ has type $(\tau \to o) \to o$. The set of atomic formulas, whose top level constant symbol is in $\Sigma_{\mathcal{S}}$, will be referred as $\Pi_{\Sigma_{\mathcal{S}}}$.

$F\&O$ Formulas

As mentioned above, formulas are λ-terms with type o. We are interested in a subset of Linear Logic formulas, namely the smallest one generated by the following grammar:

$$\begin{aligned}
\mathcal{D} &::= \forall(\mathcal{H} \circ\!\!- \mathcal{G}) \mid D_1 \& D_2 \\
\mathcal{G} &::= G_1 \& G_2 \mid G_1 \parr G_2 \mid \forall G \mid \top \mid \bot \mid \mathcal{H} \circ\!\!- \mathcal{D} \mid \mathcal{A} \\
\mathcal{H} &::= \mathcal{H}_1 \parr \mathcal{H}_2 \mid \mathcal{A}
\end{aligned}$$

where $\mathcal{A}$ denotes an atomic formula. A $\mathcal{D}$-formula can be considered as a set of multi-headed clauses which extend the usual logic programming ones. Under the

same perspective $\mathcal{G}$-formulas can be considered as extensions of usual goals, with implication and universal quantification. It is important to observe that $\mathcal{D}$-formulas cannot contain $\bot$ and $\top$ in their *heads*, i.e., the consequence of the top level linear implication.

We are interested in a proof system which reflects the similarities between our language and a logic programming one. Essentially, we will eliminate the *decide* rule and all the left introduction rules (except the & ones), see [20], and we will substitute them with two *backchaining* rules.

3 The $F\&O$ Proof System

In this section, we will try to *customize* Forum, see [20] for the complete set of rules, in order to highlight its use as a specification language for state-based systems. In the formulation of Forum that we are going to consider the right hand side of a sequent is a multiset of formulas.

First of all, we slightly modify the structure of sequents:

$$\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S}} \mathcal{M}$$

where Σ is a signature containing all typed constants of the sequent, $\mathcal{P}$ is a set of $\mathcal{D}$-formulas in **clausal form**, namely $\forall(\mathcal{H} \multimap \mathcal{G})$, and using the logical equivalence $!(A\&B) \equiv !A \otimes !B$ we can avoid to explicitly express the conjunction between the clauses. $\mathcal{R}$ is a multiset of $\mathcal{D}$-formulas, $\mathcal{M}$ is a multiset of $\mathcal{G}$-formulas and $\mathcal{S}$ is a multiset of atoms in $\Pi_{\Sigma_{\mathcal{S}}}$. The intuition is that $\mathcal{P}$ corresponds to a set of global definitions (always usable), $\mathcal{R}$ is a set of *bounded* definitions which can be consumed only once, $\mathcal{S}$ can be considered as the current state of the system that we want to simulate and, finally, $\mathcal{M}$ is the multiset of pending activities in the system, e.g., messages. In terms of linear logic the previous sequent can be read as follows:

$$(\otimes_{c \in \mathcal{P}} !\, c) \ \otimes (\otimes_{c \in \mathcal{R}} c) \rightarrow (\parr_{m \in \mathcal{M}} m) \parr (\parr_{s \in \mathcal{S}} s)$$

In figure 1, we have depicted the set of $F\&O$ rules.

The letter $\mathcal{M}$ denotes an arbitrary multiset of $\mathcal{G}$-formulas, while the letters $\mathcal{N}$ and $\mathcal{Q}$ denote multisets of atomic formulas. In the backchaining rules we also require that $\mathcal{N}$ and $\mathcal{Q}$ consist of only *non-state* atoms, furthermore, $\{A_1[\bar{t}/\bar{x}], ..., A_n[\bar{t}/\bar{x}]\} = \mathcal{N} \uplus \mathcal{S}$ where the left side of the equation is a multiset and $\uplus$ denotes the multiset union, i.e, the merging of the elements of the two multisets.

Besides the two backchaining rules, which clarify the role of a $\mathcal{D}$-clause in a computation, we have introduced the axiom *final* to isolate the final state of a computation.

Such axiom has not a direct Forum counterpart. As explained in the introduction, the idea is to prove implications of the form $S_i \multimap S_f$ in Forum which implies to

$$\frac{}{\Sigma : \mathcal{P}; \rightarrow_{S_f}} \ \textit{final} \qquad \frac{}{\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S}} \top, \mathcal{M}} \ \textit{all}$$

$$\frac{A \in \Pi_{\Sigma_{\mathcal{S}}} \quad \Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{A,\mathcal{S}} \mathcal{M}}{\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S}} A, \mathcal{M}} \ \textit{add} \qquad \frac{\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S}} A, B, \mathcal{M}}{\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S}} A \parr B, \mathcal{M}} \ \textit{par}$$

$$\frac{\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S}} A, \mathcal{M} \quad \Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S}} B, \mathcal{M}}{\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S}} A\&B, \mathcal{M}} \ \textit{rwith} \qquad \frac{\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S}} \mathcal{M}}{\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S}} \bot, \mathcal{M}} \ \textit{anti}$$

$$\frac{\Sigma : \mathcal{P}; B, \mathcal{R} \rightarrow_{\mathcal{S}} A, \mathcal{M}}{\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S}} A \multimap B, \mathcal{M}} \ \textit{impl} \qquad \frac{y : \tau, \Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S}} B[y/x], \mathcal{M}}{\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S}} \forall_\tau x.B, \mathcal{M}} \ \textit{forall}$$

$$\frac{\Sigma : \mathcal{P}; A, \mathcal{R} \rightarrow_{\mathcal{S}} \mathcal{N}}{\Sigma : \mathcal{P}; A\&B, \mathcal{R} \rightarrow_{\mathcal{S}} \mathcal{N}} \ \textit{lwith} \qquad \frac{\Sigma : \mathcal{P}; B, \mathcal{R} \rightarrow_{\mathcal{S}} \mathcal{N}}{\Sigma : \mathcal{P}; A\&B, \mathcal{R} \rightarrow_{\mathcal{S}} \mathcal{N}} \ \textit{lwith}$$

$$\frac{\bar{t} \in \Sigma\text{-}terms \quad \Sigma : \forall \bar{x}.(\parr_{i:1..n} A_i \multimap B), \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{T}} B[\bar{t}/\bar{x}], \mathcal{Q}}{\Sigma : \forall \bar{x}.(\parr_{i:1..n} A_i \multimap B), \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S},\mathcal{T}} \mathcal{N}, \mathcal{Q}} \ \textit{backchaining-1}$$

$$\frac{\bar{t} \in \Sigma\text{-}term \quad \Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{T}} B[\bar{t}/\bar{x}], \mathcal{Q}}{\Sigma : \mathcal{P}; \forall \bar{x}.(\parr_{i:1..n} A_i \multimap B), \mathcal{R} \rightarrow_{\mathcal{S},\mathcal{T}} \mathcal{N}, \mathcal{Q}} \ \textit{backchaining-2}$$

Figure 1: Proof system

guess the final state S_f at the beginning of the proof. The above mentioned axiom lays emphasizes on the fact that the final state S_f is reached *after* performing the computation.

4 An Application for $F\&O$

The structure of the $F\&O$ proof system can help us to find a natural representation of the o.o. features discussed in the introduction. Both higher-order and linear logic aspects are useful for such purpose. For the sake of this brief presentation we will consider a **class-based** system, where messages are processed in a completely *asynchronous* way, without any order upon the communication channel. With this simplifications, it is easier to show how *encapsulation* and *data abstraction* can be expressed in $F\&O$. In the sequel, we will define three basic rules to manipulate objects showing how to derive them in $F\&O$.

We can consider each $F\&O$ sequent as a logical representation of a *snapshot* of a computation in an o.o. system.

The essential choices in our representation are: *objects* are *atomic state formulas* embedding attributes and methods and possibly hiding some of them. Messages, i.e. method calls and primitive operations, are also atomic formulas, and methods are $\mathcal{D}$-

formulas. Classes will be represented by *universally quantified formulas*, expressing the structure of the objects. We also consider auxiliary operations expressed by atomic formulas, for instance, we assume to have the *new*, *send* and *kill* operations.

According to these ideas, an $F\&O$ sequent assumes the following form:

$$\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{o_1,\ldots,o_n} \mathcal{M}, \Omega$$

where $\mathcal{P}$ consists of the set of classes and auxiliary operations definitions; $o_1, \ldots, o_n$ is a multiset of atoms representing all the living objects in the system, i.e., the global state; $\mathcal{R}$ contains the currently fired methods; $\mathcal{M}$ is the set of pending messages; Ω is the set of remaining invocations, i.e., auxiliary operations.

The main point to notice here is how we **represent** every **single object**. We consider object constructors, i.e., typed constants, acting as individual names. Let i be a generic type and id a symbol with type $\tau = i \rightarrow o \rightarrow o$, thus, objects will be represented by atoms having the form *id(Attrs,Meth)*.

It is necessary to have distinguished identifiers for distinguished objects, for this reason there must be a demon process in our system able to generate new names: a message asking for the creation of a new object must send a message to get a new identifier, too. For the sake of brevity, we will omit this aspect in the rest of the discussion.

Notice that to have such kind of *predicative* identifiers it is enough to consider another constant *name* with type $i \rightarrow \tau$ and implement them as $name(t)$ with a different t for each object, e.g., numerals.

While we assume *Attrs* to be any term, representing in the proper way the data of the object, we must fix the form of the **methods**. $Meth$ is a conjunction of D-formulas $Meth_1 \& \ldots \& Meth_n$ where each $Meth_i$ has the following form

$$\forall \bar{x}.\ id(Attrs, Ms) \parr (id : Head) \ \circ\!\!- \ id(Attrs', Ms') \parr Msg_1 \parr \ldots \parr Msg_n$$

where id is the identifier of the object that is fixed at the moment of the creation, for brevity $\bar{x}$ denotes all the free variables of the clause and $Head$ is a term expressing the name of the method and the parameters. We need to consider id, the name of the object, together with $Head$ in order to avoid that messages with the same name are delivered to the wrong objects, hence the constant ':' has type $\tau \rightarrow i \rightarrow o$.

Msg_i $(i : 1..n)$ can be a method invocation, $send(Id : Msg)$, a creation message, $new(Class, Id)$, a killing message, $kill(Id)$ or the invocation of auxiliary operations, where either Id is variable in $\bar{x}$ or a constant name.

Rules for objects

Let us forget the sequent representation, i.e., hiding Σ and $\mathcal{R}$ and design three operational rules for our objects.
The first one concerns the **creation of an object**:

$$< \mathcal{M}; new(classname, id), \Omega; \mathcal{S} > \underset{\mathcal{P}}{\longrightarrow} < \mathcal{M}; \Omega; id(as, ms), \mathcal{S} > \ (new)$$

provided *classname* is defined in $\mathcal{P}$, i.e., it is a template for objects having structure (as, ms), and $id(as, ms)$ is a new instance with unique name id. A unique identifier id allows us to refer to the object $id(as, ms)$ using its name. We could also consider id as a structured term containing *classname*. To be able to mimic such a rule in $F\&O$, it is enough to express classes in $\mathcal{P}$ by the following $\mathcal{D}$-formula:

$$\forall_\tau Id.\ \ new(classname, Id) \ \circ\!\!- \ \ Id(as, ms)$$

where as and ms are two given patterns for the attributes and the methods. It is easy to see that the previous rule corresponds to the following inferencial figure in $F\&O$:

$$\dfrac{\dfrac{\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{id(as,ms),\mathcal{S}} \mathcal{M}, \Omega}{\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S}} id(as, ms), \mathcal{M}, \Omega}\ (\text{add})}{\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S}} \mathcal{M}, new(classname, id), \Omega}\ (\text{backchaining-1})$$

Notice that we use higher-order quantification in order to deal with newly created identifiers.
The second rule concerns how to handle **message passing**:

$$< send(id : m), \mathcal{M}; \Omega; id(as, ms), \mathcal{S} > \underset{\mathcal{P}}{\longrightarrow} < \mathcal{M}'; \Omega'; id(as', ms'), \mathcal{S} > \ (send)$$

where a method m_i, **non-deterministically** chosen from $m = m_1 \& \ldots \& m_n$ matching with $id : m$, is executed yielding $id(as', ms')$, $\mathcal{M}'$ and Ω' (i.e. the operations in the body of m_i are added to $\mathcal{M}$ and Ω). Let us recall that we allow the maximal degree of concurrency between messages.

In order to handle such a representation we must define a clause governing the method calls. Let us include the following formula in $\mathcal{P}$:

$$\begin{array}{l} \forall_\tau Id. \forall_i Msg, As. \forall_o Ms. \\ \quad send(Id : Msg) \parr Id(As, Ms) \ \ \circ\!\!- \\ \quad (\ ((Id : Msg) \ \parr \ Id(As, Ms)) \ \ \circ\!\!- Ms\ ) \end{array}$$

This clause is used every time a *send* has to be processed. It synchronizes the method invocation with the corresponding object moving its methods into the bounded context on the left (the meaning of $\circ\!\!-$ in the body of the clause), so that a *backchaining-2* rule can be applied. Notice that we use higher-order quantification to select the methods. Hence, the *send* rule has the corresponding $F\&O$ figure:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{id(as',ms'),\mathcal{S}} \mathcal{M}', \Omega'}{\ldots}\ (add)}{\Sigma : \mathcal{P}; m_i, \mathcal{R} \rightarrow_{\mathcal{S}} id(as', ms'), \mathcal{M}', \Omega}\ (\text{backchaining-2})}{\Sigma : \mathcal{P}; ms, \mathcal{R} \rightarrow_{id(as,ms),\mathcal{S}} (id : m), \mathcal{M}, \Omega}\ (\text{add})}{\Sigma : \mathcal{P}; ms, \mathcal{R} \rightarrow_{\mathcal{S}} (id : m), id(as, ms), \mathcal{M}, \Omega}\ (\text{par})}{\Sigma : \mathcal{P}; ms, \mathcal{R} \rightarrow_{\mathcal{S}} \mathcal{M}, (id : m) \parr id(as, ms), \Omega}\ (\text{impl})}{\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{\mathcal{S}} \mathcal{M}, ((id : m) \parr id(as, ms)) \ \circ\!\!- ms, \Omega}\ (\text{backchaining-1})}{\Sigma : \mathcal{P}; \mathcal{R} \rightarrow_{id(as,ms),\mathcal{S}} \mathcal{M}, send(id : m), \Omega}$$

where we suppose that $ms = m_1 \& \ldots \& m_n$. For the sake of brevity, we have omitted the obvious passages consisting of $\parr$ and *add* applications that allow to simplify the body of the method. There is another important aspect to consider. Since methods definitions are fired in the bounded context and thus used only once, each recursive call must be embedded into a *send* call.
The last rule concerns **killing an object**:

$$< \mathcal{M}; kill(id), \Omega; id(as, ms), \mathcal{S} > \underset{\mathcal{P}}{\longrightarrow} < \mathcal{M}; \Omega; \mathcal{S} > \quad (kill)$$

This can be achieved specifying the following definition for the *kill* operator, whose only goal is to consume the object.

$$\forall_\tau Id. \forall_i As. \forall_o Ms.\ kill(Id) \parr Id(As, Ms) \multimap \bot$$

We resort to *anti*, the neutral element of $\parr$, since it does not influence the other resources in the right part of a sequent.

Regarding the objects, so far we have shown how to obtain *encapsulation*. Without modifying the previous rules, but only the formulas defining classes, it is also possible to obtain **data abstraction**. The idea is to use a universal quantified variable on the right hand side of the sequents, whose scope extends over the part of objects to be hidden:

$$\forall_\tau Id.\quad new(classname, Id) \multimap \forall \bar{v}.\ Id(as, ms)$$

This representation is very powerful. For example, let us consider quantifying over the *data constructors*. Since the scope is extended over all the object, its methods can use them, while external operations cannot access that name (because of the side condition on the constant introduced by a $\forall$ rule). In the same way, quantifying over a *method name*, we create private methods that can be used only by methods of the same object and quantifying over the name of an object created with *new* yields *local objects*.

We have not shown occurrences of *rwith* on the right, however, the idea is to exploit & for having auxiliary branches in the proofs where auxiliary properties can be proven independently from the main branch representing the state evolution.

5 Conclusions

This work is only a preliminary overview of how to model objects using an higher-order linear logic language like Forum. For the sake of brevity, we have only presented some simple aspects of the matter, i.e., how to represent objects and classes in the context of a specialization of Forum, that we call *F&O*.

There are many other aspects that have not been treated here, concerning other applications of *F&O*. For example, we can also model more complex object-based

systems with sequential computational models, using *continuations* to obtain sequentiality in linear logic.

Also, we have also sketched a translation of a subset of the object-oriented language Pool, which Walker in [24] translated in π-calculus.

Moreover, this kind of application of linear logic is also connected with many other topics like sequentiality in linear logic, different degrees of concurrency etc., e.g. see the works [12, 15, 19], that we will take into account in the future.

At the moment, a prototype is also being implemented using λ-Prolog.

References

[1] M. Abadi and L. Cardelli. Object Calculi with Override. Technical report, *DECS RC*, 1993.

[2] S. Abramsky. Computational Interpretations of Linear Logics. Technical report, *Department of Computing, Imperial College, Technology and Medicine, London*, 1990.

[3] H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16, 1993.

[4] J. M. Andreoli. Logic Programming with focusing proofs. *Journal of Logic and Computation 2(3)*, 1992.

[5] J. M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-In Inheritance. In D.H. Warren and P.Szeredi, editors, *Proceedings of the 7th International Conference on Logic Programming*, pp. 495–510. The MIT Press, Cambridge, MA, 1990.

[6] J. M. Andreoli and R. Pareschi. Communication as Fair Distribution of Knowledge. In *Proceedings of OOPSLA '91*, 1991.

[7] J. M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-In Inheritance. *New Generation Computing*, 9:445–473, 1991.

[8] M. Bugliesi, E. Lamma, and P. Mello. Modularity in Logic Programming. *Journal of Logic Programming*, 19:443–501, 1994.

[9] J. S. Conery. Logical Objects. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of 5th International Conference on Logic Programming*, pp. 420–434. The MIT Press, Cambridge, MA, 1988.

[10] J. Y. Girard. Linear Logic. *Theoretical Computer Science*, 50:1:1–102, 1987.

[11] J. Y. Girard. Towards a Geometry of Interaction. *Contemporary Mathematics*, 92:69–108, 1989.

[12] A. Guglielmi. Concurrency and plan generation in a logic programming language with a sequential operator. In *Proceedings of ICLP '94*, pp: 240–254, 1994.

[13] J.S. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Journal of Information and Computation*, 110(3):327-365, May 1994.

[14] K. Kahn, E.D. Tribble, M.S. Miller, and D.G. Bobrow. Vulcan: Logical Concurrent Objects, Research Directions in Object-Oriented Programming. Computer System Series. The MIT Press, Cambridge Ma, London, 1987.

[15] N. Kobayashi and A. Yonezawa. Linear Logic Programming as a Unifying Framework for Concurrent Programming. Technical report, Department of Information Science the University of Tokyo, 1992.

[16] F. G. McCabe. Logics ad Objects. Technical report, Department of Computing, Imperial College, London, 1988.

[17] J. Meseguer. Conditional Rewriting Logic. Technical report, *SRI International, Stanford University*, 1993.

[18] D. Miller. Lexical Scoping as universal quantification. In *Sixth International Logic Programming Conference*, pp. 268–283, Lisbon, Portugal, June 1989, MIT Press.

[19] D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P.Mello, editors, *Proceedings of the 1992 Workshop on Extension to Logic Programming*, LCNS Vol.660, pp. 242–265. Springer-Verlag, Berlin, 1993.

[20] D. Miller. A Multiple-Conclusion Meta-Logic. Technical report, Computer Science Department, University of Pennsylvania, 1994.

[21] L. Monteiro and A. Porto. Contextual Logic Programming. In G.Levi and M.Martelli, editors, *Proceedings of 6th International Conference on Logic Programming*, pp. 284–302. The MIT Press, Cambridge, MA, 1989.

[22] A. Scedrov. Linear Logic and Computation: A Survey. *To appear in: Proof and Computation* 17, 1994.

[23] G. Smolka. A feature logic with subsorts. Technical report, IWBS, IBM Deutschland, Stuttgard, Germany, 1988.

[24] D. J. Walker. π-calculus Semantics of Object-Oriented Programming Languages. Technical report, *University of Technology*, 1990.

[25] W. Chen and D.H. Warren. Objects as Intensions. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of 5th International Conference on Logic Programming*, pp. 404–419. The MIT Press, Cambridge, MA, 1988.

[26] P. Wegner. Dimension of Object-Based Language Design. In *Proceedings of OOPSLA '87*, pp. 168–182, 1987.

Three-Valued Semantics for Extended Logic Programs

Piero A. Bonatti and Laura Giordano
Dipartimento di Informatica, Università di Torino
Corso Svizzera 185, I-10149 Torino, Italy
email: {bonatti,laura}@di.unito.it tel: +39-11-7429 111

Abstract

In [4], the semantics of monotonic (i.e. not-free) extended logic programs (ELPs) has been rephrased in three-valued logic for two purposes: achieving tractable reasoning with incomplete information and understanding the relationships between the existing semantics and many-valued logics. In this paper, we generalize this approach to unrestricted ELPs. We obtain a unifying view of many formalisms, including the answer set semantics, the well-founded semantics, generalized stable models (as in [11]), default logic, autoepistemic logic (AEL) and some of its variants (three-valued AEL and Schwartz's reflexive AEL). Our framework highlights surprising similarities between previously unrelated formalisms, such as TMS's with dependency directed backtracking, the WFSX semantics by Alferes and Pereira, and reflexive AEL. Moreover, we obtain very interesting new semantics, which make it possible to solve many hard benchmark problems with a substantial gain in elegance and efficiency.

KEYWORDS: Three-valued Logic, Negation as Failure, Incomplete Information

1 Introduction

Research on extended logic programs (ELP's) has been focussed on the semantics of default negation, not, and on its interplay with explicit negation, $\neg$ (cf. [16]). While several semantics have been proposed for not, there is little variety of meanings for the not-free fragment of the language, hereafter called *monotonic*.

In [4], the semantics of monotonic ELP's has been rephrased in three-valued logic, for different purposes. The first goal was improving our understanding of the relationships between ELP's and multiple valued logics, which are not completely clear, despite numerous superficial similarities. The second goal was a purely semantic account of ELP's, whose meaning is often defined through pervasive syntactic manipulations that transform negative literals into new atoms, both within

programs and within interpretations. The final goal was finding a more powerful tractable semantics for ELP, capable of expressing and using incomplete knowledge without resorting to more complex forms of reasoning involving combinatorial search or noncomputable inferences. The approach of [4] tackles all the above problems and leads to a unifying view of different semantics for monotonic ELP's. The basic idea is considering a not-free program rule as a formula

$$(\ldots(L_0 \leftarrow L_1) \leftarrow \ldots) \leftarrow L_n\,.$$

By interpreting $\leftarrow$ as one of the standard three-valued implications illustrated in Fig. 1, different semantics can be captured.

It is interesting to see how this approach behaves when we introduce default negation through standard nonmonotonic constructions. This is the purpose of the present paper. For the sake of generality we shall model different possible meanings of not through a general construction (stable classes) that generalizes stable and well-founded semantics. By tuning two parameters, that is, the truth table of implication and the program transformation involved in the nonmonotonic construction (one of which is the familiar Gelfond-Lifschitz transformation), we obtain a unifying view of many formalisms and highlight surprising similarities between previously unrelated formalisms. We shall prove that Lukasiewicz's implication induces some very interesting new semantics, which make it possible to solve many hard benchmark problems with a substantial gain in elegance and efficiency. Moreover, a well-founded version of Lukasiewicz's semantics constitutes a natural semantics for Reason Maintenance, more powerful and more efficient than the skeptical belief revision model of [18].

2 Preliminaries

2.1 Three-Valued Logic

Three-valued interpretations are mappings from the set of ground atoms into the set $\{F, U, T\}$. As usual, three-valued interpretations will be represented as consistent sets of ground literals, so that A is T (resp. F) in I iff $A \in I$ (resp. $\neg A \in I$). In the literature there is general agreement about the meaning of $\neg$, while the meaning of implication is controvertial. Three of the major proposals are recalled in Fig. 1 where $\leftarrow_K$ is Kleene's implication, $\leftarrow_L$ is the one proposed by Lukasiewicz, and $\Leftarrow$ is a less famous but important implication which has been considered by several authors (cf. [1]) and has been applied to non-monotonic reasoning [5]. The classical equivalence $A \vee B \equiv A \leftarrow_K \neg B$ holds for Kleene's implication but not for $\leftarrow_L$. The latter corresponds to a disjunction, usually denoted by $\oplus$, defined by $A \oplus B \equiv A \leftarrow_L \neg B$.

A	$\neg A$
F	T
U	U
T	F

$\leftarrow_K$	F	U	T
F	T	U	F
U	T	U	U
T	T	T	T

$\leftarrow_L$	F	U	T
F	T	U	F
U	T	T	U
T	T	T	T

$\Leftarrow$	F	U	T
F	T	T	F
U	T	T	U
T	T	T	T

Figure 1: Truth tables for negation and various forms of implication

2.2 Default and Autoepistemic Logic

We assume the reader to be familiar with default logic (DL) and autoepistemic logic (AEL). For an extensive treatment, see [12, 13]. A few variants of DL and AEL are briefly recalled in this section.

Baral and Subrahmanian [3] generalized default extensions by introducing the notion of *extension class*, that is a family of sets $\mathcal{F}$ such that $\mathcal{F} = \{\Gamma_\Delta(E) \mid E \in \mathcal{F}\}$, where Γ_Δ is Reiter's operator. This approach is more robust than DL; in fact, every finite closed default theory has an extension class.

In [17], Schwartz introduced a variant of AEL based on the notion of *reflexive expansions*, which are the solutions of

$$E = \{\psi \mid T \cup (LE \equiv E) \cup \neg L\overline{E} \vdash \psi\}\,, \tag{1}$$

where $LE \equiv E$ is an abbreviation for $\{L\psi \leftrightarrow \psi \mid \psi \in E\}$ and $\neg L\overline{E} = \{\neg L\psi \mid \psi \notin E\}$. Many theories without stable expansions have a reflexive expansion. Moreover, reflexive expansions contain no weakly-grounded (i.e. self-supporting) beliefs.

Three-valued autoepistemic logic (3AEL) [5] tackles similar problems with a different technique. The basic idea is that agents may have doubts. For a large and expressive class of theories, which generalize Konolige's autoepistemic normal form, we have that every consistent theory has one minimal *generalized stable expansion* (GSE), which enjoys a fixpoint construction and contains no weakly grounded beliefs.

2.3 Extended Logic Programs

The set of objective literals, denoted by LIT, consists of all atoms and negated atoms of the form $\neg A$. For all sets of sentences S, $\mathrm{LIT}(S)$ denotes $S \cap \mathrm{LIT}$. Similarly, $\mathrm{AT}(S)$ denotes the set of atoms in S. As usual, we say that A and $\neg A$ are complementary, and let $\overline{L}$ denote the literal complementary to L. A default literal is a formula not L where L is an objective literal. We assume the reader to be familiar with extended logic programs (ELP's); see [10] for the definition of their syntax and semantics. *Notation*: the unique answer set of monotonic programs will be denoted by $\mathrm{ANS}(P)$; the Gelfond-Lifschitz transformation of P w.r.t. $S \subseteq \mathrm{LIT}$ will be denoted by P^S_{GL}.

Giordano and Martelli [11] introduced a different transformation, and a notion of generalized stable model (GSM) for normal logic programs with constraints, which

captures the dependency directed backtracking (DDB) mechanism of TMS's. Given a classical interpretation M, their transformation, hereafter denoted by P^M_{GM}, is obtained from P in three steps. First, all the literals not B such that $B \notin M$ are removed. Then the remaining default literals not B are replaced by $\neg B$. Finally, among the resulting rules, select those which are *strictly satisfied* by M, i.e. the rules $L_1 \leftarrow L_2, \ldots, L_n$ such that M evaluates $n-1$ of the literals $L_1, \overline{L_2}, \ldots, \overline{L_n}$ to F and one of them to T. The GSM's of P are the classical models of P (when default literals not B in P are replaced by $\neg B$) that are solutions of the equation

$$\mathrm{AT}(M) = \mathrm{AT}(\mathrm{Cn}(P^M_{\mathrm{GM}})).$$

In [4], the semantics of monotonic ELP's has been rephrased in three-valued logic. The basic idea is regarding ELP rules as sentences of the form

$$(\ldots(L_0 \leftarrow L_1) \leftarrow \ldots) \leftarrow L_n,$$

where $\leftarrow$ is one of the standard implications illustrated in Fig. 1. The semantics of a monotonic ELP P is captured by $\mathrm{ANS}_X(P)$, which denotes the set of literals that are logical consequences of P in X-valued logic $(X = 2, 3)$. It has been shown that $\Leftarrow$ yields the answer set semantics, while $\leftarrow_K$ captures classical logic.

Theorem 2.1 ([4]) *For all monotonic ELP's P,*

i) *If $\leftarrow$ is $\Leftarrow$, then $\mathrm{ANS}_3(P) = \mathrm{ANS}(P)$.*

ii) *If $\leftarrow$ is $\leftarrow_K$, then $\mathrm{ANS}_3(P) = \mathrm{ANS}_2(P)$.*

Lukasiewicz's implication yields a new interesting semantics, whose operational semantics is a restricted form of unit resolution ($\vdash_{\mathrm{UR}}$), where clauses should be treated as multisets, rather than sets. A similar restriction of input linear resolution ($\vdash_{\mathrm{IR}}$), constitutes an equivalent operational semantics, which models top-down, SLD-like computations.

Theorem 2.2 ([4]) *For all monotonic ELP's P where $\leftarrow$ is $\leftarrow_L$,*

i) *If P is consistent, then $\mathrm{ANS}_3(P)$ is the least model of P.*

ii) *If P is consistent, then its declarative, operational and fixpoint semantics coincide. If P is inconsistent, then the three semantics are all inconsistent.*

iii) *$\mathrm{ANS}_3(P) = \mathrm{ANS}(\mathrm{CNT}_P)$, where CNT_P is the contrapositive completion of P, that is, $P \cup \{\, \overline{L_k} \leftarrow L_1, \ldots, L_{k-1}, \overline{L_0}, L_{k+1}, \ldots L_n \mid L_0 \leftarrow L_1, \ldots, L_n \in P \,\}$.*

3 Three-Valued Semantics for ELP's

In this section, the three-valued semantics for monotonic ELP's introduced in [4] is extended to unrestricted ELP's. Default negation is interpreted through the construction underlying stable classes, which captures well-founded and stable semantics in a uniform way. In our framework, the meaning of an ELP is determined by two parameters, namely, the program transformation (e.g. GL, GM) and the truth table of implication (cf. Fig. 1). Accordingly, we replace the operator F_P investigated by Baral and Subrahmanian with $F_P^{\mathrm{TR},\leftarrow}(X) = \mathrm{ANS}_3(P^X_{\mathrm{TR}})$, where TR is a program transformation and implication is interpreted as $\leftarrow$, which should be one of the connectives illustrated in Fig. 1.

Definition 3.1 A nonempty family of sets $\mathcal{F}$, contained in the powerset of LIT, is a *TR($\leftarrow$)-answer class* of an ELP P iff $\mathcal{F} = \{\, F_P^{\mathrm{TR},\leftarrow}(X) \mid X \in \mathcal{F} \,\}$. If $\mathcal{F} = \{\, S \,\}$, i.e. if S is a fixpoint of $F_P^{\mathrm{TR},\leftarrow}$, then S is called a *TR($\leftarrow$)-answer set* of P. If $\mathcal{F} = \{\, S_1, S_2 \,\}$, where $S_1 \subseteq S_2$, $F_P^{\mathrm{TR},\leftarrow}(S_1) = S_2$ and $F_P^{\mathrm{TR},\leftarrow}(S_2) = S_1$, then we say that $\mathcal{F}$ is an *alternating TR($\leftarrow$)-answer set* of P, and denote $\mathcal{F}$ by (S_1, S_2).

The major results of [2] can be immediately extended to our framework. First of all, $F_P^{\mathrm{TR},\leftarrow}$ is anti-monotonic when TR is GL or the transformation RE which will be introduced in Sec. 3.2. Secondly, when $F_P^{\mathrm{TR},\leftarrow}$ is anti-monotonic, its square power is monotonic, and every program P has an alternating TR($\leftarrow$)-answer set

$$\mathcal{F} = \left(\mathrm{lfp}\left(\left[F_P^{\mathrm{TR},\leftarrow}\right]^2\right), \mathrm{gfp}\left(\left[F_P^{\mathrm{TR},\leftarrow}\right]^2\right)\right), \qquad (2)$$

which is also the least TR($\leftarrow$)-answer class under Hoare's ordering. The proofs of these claims are simple adaptations of the proofs in [2] and are left to the reader.

We say that an objective literal L is derivable from an answer class $\mathcal{F}$ iff, for all $S \in \mathcal{F}$, $L \in S$. We say that a default literal not L is derivable from an answer class $\mathcal{F}$ iff, for all $S \in \mathcal{F}$, $L \notin S$. In particular, if $\mathcal{F}$ consists of an answer set S, then L is derivable iff $L \in S$, and not L is derivable iff $L \notin S$. When $\mathcal{F}$ is an alternating answer set (S_1, S_2), L is derivable iff $L \in S_1$, and not L is derivable iff $L \notin S_2$. Under this interpretation, the least alternating answer set (2) induces well-founded semantics for ELP's. In the rest of this section we investigate the relations between the above framework and the semantics proposed so far.

3.1 GL-Answer Classes

First we study the relationships between GL-answer classes and the existing semantics of ELP's. In [4], it was proved that $\Leftarrow$, which behaves much like an inference rule, preserves the standard meaning of monotonic ELP's. This result can easily be extended to unrestricted ELP's. Actually, we shall prove a more general result, relating ELP's with default logic. By following the terminology in [14], given a program P, we will denote by $\mathrm{tr}_1(P)$ the default theory $\langle \emptyset, D \rangle$

where D is the set of defaults $L_1 \wedge \ldots \wedge L_m : \neg L_{m+1}, \ldots, \neg L_n \,/\, L_0$, such that $L_0 \leftarrow L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n$ is in P.

Theorem 3.2 *For all ELP's P, there is a one to one correspondence between extension classes of $\text{tr}_1(P)$ and GL($\Leftarrow$)-answer classes of P.*

As a special case of extension classes, we capture various semantics of normal logic programs (cf. [2, 10]). GL($\Leftarrow$)-answer classes capture stable classes; the well-founded model is captured by the least alternating GL($\Leftarrow$)-answer set of P. Moreover:

Corollary 3.3 *For all ELP's P, every answer set of P is a GL($\Leftarrow$)-answer set of P and vice-versa.*

The least alternating GL($\Leftarrow$)-answer sets of ELP's are a natural generalization of the well-founded semantics. By Theorem 3.2, they correspond to Baral and Subrahmanian's *well-founded semantics of default logic* [2].

Next we focus on Kleene's valuation, which yields a semantics whose monotonic inferences are exactly the ones supported by classical logic (cf. Theorem 2.1). The corresponding embedding in default logic is tr_2, introduced in [14], which translates $L_0 \leftarrow L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n$ into : $\neg L_{m+1}, \ldots, \neg L_n \,/\, L_0 \leftarrow L_1 \wedge \ldots \wedge L_m$.

Theorem 3.4 *For all ELP's P, there is a one to one correspondence between extension classes of $\text{tr}_2(P)$ and GL($\leftarrow_K$)-answer classes of P.*

Next we clarify the correspondence with 3AEL, which encompasses AEL as a special case.

Theorem 3.5 *Let P be an ELP and let AE(P) be the autoepistemic translation of P obtained by replacing* not *with $\neg L$ and $\leftarrow$ with $\Leftarrow$. There is a one to one correspondence between:*

i) *GSE's of AE(P) and consistent alternating GL($\leftarrow_K$)-answer sets of P.*

ii) *standard stable expansions of AE(P) and GL($\leftarrow_K$)-answer sets of P.*

Finally, we consider Łukasiewicz's implication. It induces new semantics where contraposition is allowed. In particular, for a given program P, let CNT_P be the *contrapositive completion of P*, consisting of P and all the contrapositives

$$\overline{L_i} \leftarrow L_1, \ldots, L_{i-1}, \overline{L_0}, L_{i+1}, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n, \qquad (3)$$

of the rules $L_0 \leftarrow L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n$ in P. The following proposition is an easy consequence of Theorem 2.2.(iii).

Proposition 3.6 *For all ELP's P, the GL($\leftarrow_L$)-answer classes of P and the GL($\Leftarrow$)-answer classes of CNT_P coincide. Moreover, the GL($\leftarrow_L$)-answer sets of P are the (standard) answer sets of CNT_P.*

1. $a(x) \leftarrow w(x).\quad a(x) \leftarrow f(x).\quad \ldots\quad a(x) \leftarrow s(x)$
2. $w(w_1).\quad f(f_1).\quad b(b_1).\quad c(c_1).\quad s(s_1)$
3. $g(g_1).\quad p(x) \leftarrow g(x)$
4. $sm(x,y) \leftarrow b(y), c(x)$
5. $sm(x,y) \leftarrow b(y), s(x)$
6. $sm(x,y) \leftarrow b(x), f(y)$
7. $sm(x,y) \leftarrow f(x), w(y)$
8. $\neg l(x,y) \leftarrow w(x), f(y)$
9. $\neg l(x,y) \leftarrow w(x), g(y)$
10. $\neg l(x,y) \leftarrow b(x), s(y)$
11. $l(x,y) \leftarrow b(x), c(y)$
12. $l(x, p_2(x)) \leftarrow c(x).\quad p(p_2(x)) \leftarrow c(x)$
13. $l(x, p_3(x)) \leftarrow s(x).\quad p(p_3(x)) \leftarrow s(x)$
14. $[\, l(x,y) \leftarrow p(x)\,] \oplus [\, l(x,z) \leftarrow a(z), sm(z,x), p(u), l(z,u)\,] \leftarrow a(x)$

Figure 2: An ELP for Schubert's steamroller

Contrapositives may not seem a significative extension, at first glance. On the contrary, contraposition makes it possible to solve in a natural and efficient way many difficult benchmark problems for automatic theorem provers. About 60% of the benchmarks without equality listed in [15] — including some of the most difficult ones, according to Pelletier's rating—can be solved through Łukasiewicz's semantics, and contraposition proves to be essential (cf. [4]). This is an astonishing result for a monotonic logic programming language. The problems that can be successfully solved include Schubert's Steamroller—one of the two most difficult problems of [15]—and the Dreadsbury Mansion Mistery. Both have been recently considered in [19], where a nonmonotonic version of the Steamroller is introduced and proposed as a benchmark. In the following example, we show how the non-monotonic Steamroller can be solved through Łukasiewicz's semantics. To enhance readability, we shall use $\oplus$ as syntactic sugar. Note that every formula in the following example can be turned into an equivalent ELP rule of the same size.

Example 3.7 The following is a description of the non-monotonic version of Schubert's Steamroller due to Wagner; numbers refer to the formalization illustrated in Fig. 2, where $\leftarrow$ should be interpreted as $\leftarrow_L$. Default rules like "Normally F if G" are expressed through semi-normal rules $F(x) \leftarrow G(x), \text{not } ab_R(x), \text{not } \neg F(x)$.

> Wolves, foxes, birds caterpillars and snails are animals, and there are some of each of them (1–2). Also, there are some grains and grains are plants (3). Caterpillars and snails are much smaller than birds, which are much smaller than foxes, which in turn are much smaller than wolves (4–7). Normally, wolves do not like to eat foxes or grains, while birds like to eat caterpillars