PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. 



# Logic Programs with Inheritance

Yaron Goldberg, William Silverman, and Ehud Shapiro

Department of Applied Mathematics and Computer Science
The Weizmann Institute of Science
Rehovot 76100, Israel

**Abstract**

It is well known that while concurrent logic languages provide excellent computational support for object-oriented programming they provide poor notational support for the abstractions it requires. In an attempt to remedy their main weaknesses — verbose description of state change and of communication and the lack of class-like inheritance mechanism — new object-oriented languages were developed on top of concurrent logic languages.

In this paper we explore an alternative solution: a notational extension to pure logic programs that supports the concise expression of both state change and communication and incorporates an inheritance mechanism. We claim that combined with the execution mechanism of concurrent logic programs this notational extension results in a powerful and convenient concurrent object-oriented programming language.

The use of logic programs with inheritance had a profound influence on our programming. We have found the notation vital in the structuring of a large application program we are currently building that consists of a variety of objects and interfaces to them.

## 1 Introduction

We share with the Vulcan language proposal [8] the view on the utility of concurrent logic languages as object-oriented languages:

> "The concurrent logic programming languages cleanly build objects with changeable state out of purely side-effect-free foundations. [...] The resulting system has all the fine-grained concurrency, synchronization, encapsulation, and open-systems abilities of Actors [4]. In addition, it provides unification, logic variables, partially instantiated messages and data, and the declarative semantics of first-order logic.
>
> Abstract machines and corresponding concrete implementations support the computational model of these languages, providing cheap, light-weight processes [...] Since objects with state are not taken as a base concept, but are built out of finer-grained material, many variations on traditional notions of object-oriented programming are possible. These include object forking and merging, direct broadcasting, message stream peeking, prioritized message sending, and multiple message streams per object."

See also [7] for a more recent account of the object-oriented capabilities of concurrent logic programs.

We also share with the designers of Vulcan the conclusion that:

> "While [concurrent logic languages] provide excellent computational support [for object-oriented programming], we claim they do not provide good notation for expressing the abstractions of object-oriented programming."

However, we differ in the remedy. While Vulcan and similar proposals [2,13] each offer a new language, whose semantics is given via translation to concurrent logic languages, we propose a relatively mild notational extension to pure logic

programs, and claim that it addresses quite adequately the needs of the object-oriented programmer. Specifically, our notation addresses the main drawbacks of logic programs for object-oriented programming: verbose description of objects with state, cumbersome notation for message sending and receiving, and the lack of a class-like inheritance mechanism that allows the concise expression of several variants of the same object. We explain the drawbacks and outline our solutions.

## Inheritance

In certain applications, most notably graphical user interfaces, many variants of the same object are employed to cater to various user needs and to support smooth interaction. In the absence of an inheritance mechanism, a variant of a given object must be defined by manually copying the description of the object and editing it to meet the variant specification. Both development and maintenance are hampered when multiple copies of essentially the same piece of code appear within a system. Class-based inheritance mechanisms provide the standard solution for defining multiple variants of the same basic object in a concise way, without replicating the common parts. In this paper we propose an inheritance mechanism for logic programs, called *logic programs with inheritance*, or *lpi* for short.

The idea behind *lpi* is simple. When a procedure $p$ inherits a procedure $q$ via the inheritance call $q(T_1, \ldots, T_k)$, add to $p$'s clauses all of $q$'s clauses, with the following two basic modifications:

1. Replace the head predicate $q$ by the head predicate $p$, with "appropriate arguments".
2. Replace recursive calls to $q$ by "corresponding calls" to $p$.

The formal definition of *lpi* will make precise the meaning of the terms "appropriate arguments" and "corresponding calls". The effect of inheritance is that behaviors realizable by $q$ are also realizable by $p$, as expected.

Using common object-oriented terminology, *lpi* can be characterized as follows:

- *Predicates are classes*: Logic program predicates are viewed as *classes*, and procedures (*i.e.*, predicate definitions) as *class definitions*. Classes can be executed as well as inherited; that is, superclasses are executable in their own right.
- *Clauses are methods*: In the concurrent reading of logic programs, each clause in a procedure specifies a possible process behavior. Inheriting a procedure means incorporating appropriate variants of the clauses of the inherited procedure into the inheriting procedure.
- *Multiple inheritance*: A procedure may inherit several other procedures.
- *Parameterized inheritance*: Inheritance is specified by "inheritance calls", which may include parameters. Hence an inheriting procedure may inherit the same procedure in several different ways, using different parameters in the inheritance calls.

We shall see examples for these features in the following sections.

We note that the inherent nondeterminism of logic programs (and the inherent indeterminacy of concurrent logic programs) can accommodate conflicts in inherited methods with no semantic difficulty. If necessary, an overriding mechanism can be incorporated to enforce a preference of subclass methods over superclass ones.

## Implicit Arguments

Objects with state, accessible via messages, are realized in concurrent logic programs by recurrent processes. Typically, such a recurrent process has one or more shared variables and zero or more private state variables. The expression of a recurrent process by a concurrent logic program has the general form:

$p(\ldots$ old state of process $\ldots) \leftarrow$
$\ldots$ message receiving and sending, $\ldots$
$p(\ldots$ new state of process $\ldots)$.

When a process has several variables, where only a few of them are accessed or changed on each process reduction, then the notation of plain logic programs becomes quite cumbersome. This is due to the need to specify explicitly all the variables that do not change in a process's state transition twice: once in the "old" state in the clause head, and once in the "new" state in the clause body. In addition, different names need to be invented for

the old and new incarnations of a state variable that did change in the transition.

This verbose syntax introduces the possibility of trivial errors and reduces the readability of programs, since it does not highlight the important parts (what has changed) from other details (repetition of the unchanged part).

We define a notational extension, independent of the inheritance notation, called *implicit arguments*, to support the concise expression of recurrent processes. The notation allows specifying what has changed in the process's state during a state transition, rather than the entire old and new states required by a plain logic program, thus effectively providing a frame axiom for the state of recurrent processes. The semantics of the extended notation is given in terms of its translation to plain logic programs.

Streams are the most commonly used data structure in concurrent logic programs. To describe sending or receiving a message $M$ on a stream $Xs$ one equates $Xs$ with a list (stream) cell whose head is $M$ and tail is a new variable, say $Xs'$, as in $Xs = [M—Xs']$. In this notation the "states" of the stream before and after the communication have to be named and referred to explicitly. We propose a notation that, by exploiting the implicit arguments notation, refers only once to the stream being used.

In practice we combine the two notational extension, inheritance and implicit arguments, into one language. We find the resulting language greatly superior to the "vanilla" syntax of (concurrent) logic programs.

In the rest of the paper we formally define the inheritance notation, the implicit arguments notation and give examples of their use.

# 2 Logic Programs with Inheritance

## 2.1 An Example

The inheritance notation is an extension to plain logic programs which allows *inheritance calls*, which are calls of the form $+p(T_1, \ldots, T_n)$. Using object-oriented terminology, we refer to a predicate as a *class* and to a procedure as a *class definition*. Each clause (or disjunct in a disjunctive form) of the procedure is viewed as a *method* which manipulates the class's state.

As an example of inheritance consider the following well known logic program which manipulates a simple counter:

```
counter(In) :-
    counter(In,0).

counter([clear|In],_) :-
    counter(In,0).
counter([add|In],C) :-
    C' := C + 1, counter(In,C').
counter([read(C)|In],C) :-
    counter(In,C).
counter([],_).
```

An alternative representation of a logic program can be in a disjunctive form, where all clauses of a predicate are written with separating semicolons and are under a single, simple head (an atom is *simple* if its arguments are distinct variables). The translation between plain logic programs and logic programs in disjunctive form is trivial.

Put in disjunctive form, the definition of *counter/2* would appear as:

```
counter(In, C) :-
    In = [clear|In'], counter(In',0);

    In = [add|In'], C' := C + 1,
        counter(In',C');
    In = [read(C)|In'], counter(In',C);

    In = [].
```

We illustrate the inheritance notation by adding a feature to *counter*, which enables us to retain a backup value of the counter, named *BackUp*. The *checkpoint* method backs up the counter value and *restore* restores its value from the backup. The syntactic changes from the previous *counter* version are an added argument and two new disjuncts. Using the inheritance notation we would write this as:

```
counter2(In) :-
    counter2(In, 0, 0).

counter2(In, C, BackUp) :-
    +counter(In, C);

    In = [checkpoint|In'],
        counter2(In', C, C);
    In = [restore|In'],
        counter2(In', BackUp, BackUp).
```

This procedure stands for:

```
counter2(In, C, BackUp) :-
    In = [checkpoint|In'],
        counter2(In', C, C);
    In = [restore|In'],
        counter2(In', BackUp, BackUp);
    In = [clear|In'],
        counter2(In',0, BackUp);
    In = [add|In'], C' := C + 1,
        counter2(In',C', BackUp);
    In = [read(C)|In'],
        counter2(In',C, BackUp);
    In = [].
```

## 2.2 Syntax

A *logic program with inheritance, lpi*, is a set of procedures, each having a unique head predicate. Each procedure is a disjunctive clause of the form:

$$p(X_1, \ldots, X_n) \leftarrow \alpha_1; \ldots; \alpha_k.$$

where $n, k \geq 0$, $X_i$'s are distinct variables and each $\alpha_i$ is either a conjunctive goal or an *inheritance call* of the form $+q(X_{i_1}, \ldots, X_{i_m})$, where the $i_j$'s are distinct and $1 \leq i_j \leq n$ for every $j$, $1 \leq j \leq m$.

Note that if $p/n$ inherits $q/m$ the definition implies that $m \leq n$.

An *inheritance graph* for an *lpi* program $P$ is a directed graph $(V, E)$ where $V$ is the set of predicates defined in $P$ and for every inheritance call to a predicate $q$ in the procedure of $p$ in $P$, $(p, q) \in E$.

An *lpi* program $P$ is *well-defined* if the graph $(V, E)$ is well-defined (*i.e.*, every predicate that occurs in $E$ is a member of $V$) and acyclic.

For convenience, we employ the following syntactic default. Suppose the predicate $q$ is defined by:

$q(Y_1, \ldots Y_m) \leftarrow \beta_1; \ldots; \beta_l.$

Then the inheritance call $+q$ is a shorthand for $+q(Y_1, \ldots, Y_m)$.

## 2.3 Semantics

The semantics of a well-defined logic program with inheritance $P$ is given by the following unfolding rule, whose application to completion to $P$ results in a logic program in disjunctive form. In the following rule $p$, $q$ are predicates, the $S$'s are terms, $X$ and $Y$ are logic variables, $\alpha_i$, $\beta_i$ are disjuncts.

**Lpi Rule:** Replace the clause:

$$p(X_1, X_2, \ldots, X_n) \leftarrow \alpha_1; \ldots; +q(X_{i_1}, \ldots, X_{i_m}); \ldots; \alpha_k.$$

where $q$ is defined by the (renamed apart) clause:

$$q(Y_1, \ldots Y_m) \leftarrow \beta_1; \ldots; \beta_l.$$

with the clause:

$$p(X_1, X_2, \ldots, X_n) \leftarrow \alpha_1; \ldots; \beta'_1; \ldots; \beta'_l; \ldots; \alpha_k.$$

where $\beta'_i$ is obtained from $\beta_i$ by the following transformation:

1. Apply the substitution $\theta \stackrel{def}{=} \{Y_j \rightarrow X_{i_j} \mid 1 \leq j \leq m\}$.
2. Replace each recursive call $q(S_1, \ldots, S_m)$ with the call $p(X_1, \ldots, X_n)\sigma$, where $\sigma \stackrel{def}{=} \{X_{i_j} \mapsto S_j \mid (1 \leq j \leq m)\}$.

This completes the definition of logic programs with inheritance.

Assume some fixed first-order signature $L$. Let $\mathcal{P}$ be the set of all well-defined logic programs with inheritance over $L$. Let $\rightarrow: \mathcal{P} \times \mathcal{P}$ be the relation satisfying $P \rightarrow P'$ iff $P'$ can be obtained from $P$ by an application of the *lpi* rule to a clause in $P$.

The pair $\langle \mathcal{P}, \rightarrow \rangle$ is not strictly a rewrite system according to the standard definitions [3], since logic programs with inheritance are sets, not terms, and since they are not closed under substitution. However, these differences do not affect the applicability of the relevant tools of rewrite systems, so we ignore them.

**Lemma 1** *The rewrite system $\langle \mathcal{P}, \rightarrow \rangle$ is terminating and confluent up to clause renaming, i.e. if $P'$ and $P''$ are two normal forms of $P$ then they are equivalent up to clause renaming. Furthermore, all normal forms are ordinary logic programs.*

**Proof outline:** Termination follows from the fact that any application of the *lpi* rule eliminates one inheritance call. Normal forms don't have inheritance calls since they can all be reduced by the requirement that $\mathcal{P}$ contains only well-defined logic programs with inheritance. Confluence follows from the associativity of substitution composition.□

**Corollary 1** *The semantics of* lpi *is well defined.*

### 2.4 An Example of Parameterized Multiple Inheritance

As an example of parameterized inheritance, suppose we have a "show_id" feature which waits on an input port *In* for a message *show_id* and then fills the incomplete message with the value *Id*:

```
id(In, Id) :-
    In = [show_id(Id)|NewIn],
    id(NewIn, Id).
```

And suppose we have a class containing two input ports *In1* and *In2*, where on each port the class can receive requests to show its id. Instead of copying the method twice, we shall write:

```
class(In1, In2, Name) :-
    +id(In1, Name);
    +id(In2, Name);

    <<class body>>
```

The result of applying the *lpi* rule would be:

```
class(In1, In2, Name) :-
    In1 = [show_id(Name)|NewIn],
        class(NewIn, In2, Name);
    In2 = [show_id(Name)|NewIn],
        class(In1, NewIn, Name);

    <<class body>>
```

The same *id* feature could be used when an object wishes to have different id's on different ports, *i.e.*, return different answers on different ports for the same incomplete message *show_id*, as in:

```
class(In1, In2, Name1, Name2) :-
    +id(In1, Name1);
    +id(In2, Name2);

    <<class body>>
```

The expansion is as follows:

```
class(In1, In2, Name1, Name2) :-
    In1 = [show_id(Name1)|NewIn],
        class(NewIn, In2, Name1, Name2);
    In2 = [show_id(Name2)|NewIn],
        class(In1, NewIn, Name1, Name2);

    <<class body>>
```

### 2.5 Integration with a Module System

The power of logic programs with inheritance is enhanced when integrated with a module system. We have integrated *lpi* with the hierarchical module system of Logix [11]. To simplify the description, we outline the principles behind the integration for a non-hierarchical module system.

When $p$ in module $M$ inherits $q$ from another module $M'$, the semantics of inheritance is that the definitions of all predicates in $M'$, called or inherited directly or indirectly by $q$, are incorporated in $M$, unless they are already defined in $M$.

This overriding capability, which gives some of the effects of higher-order programming, proves to be invaluable in practice. One can easily specify a variant of a module $M$ by inheriting its top-level procedure and overriding the definition of one or more of its subprocedures. For example, by inheriting a sorting module and overriding the comparison routine, one can turn an ordinary sort routine to a sort routine that operates on records with keys.

We note that although the semantics specifies "code copying", the following semantics-preserving optimization may apply. If $M'$ inherits from $M$, $P$ is a set of procedures in $M$ that do not call or inherit procedures outside of $P$, and none of the procedures in $P$ is redefined in $M'$, then the code for $P$ need not be included in $M'$, and any call to a procedure in $P$ may be served by $M$. This optimization achieves runtime code sharing among several modules inheriting from the same module.

## 3 Logic Programs with Implicit Arguments

### 3.1 Example

We illustrate the notation of implicit arguments via an example. The *counter* program (section 2.1) is a typical logic program specifying a recurrent process. A logic program with implicit arguments that corresponds to the plain logic program for *counter* is:

```
counter(In) + (C=0) :-
    In = [clear|In'], C' = 0, self;

    In = [add|In'], C' := C + 1, self;
```

```
    In = [read(C)|In'], self;

    In = [].
```

Similarly, a binary merge can be defined using implicit arguments by:

```
merge(In1,In2,Out) :-
    In1 = [X|In1'], Out=[X|Out'], self;
    In2 = [X|In2'], Out=[X|Out'], self;
    In1 = [], Out = In2;
    In2 = [], Out = In1.
```

## 3.2 Syntax

A logic program with implicit arguments is a set of clauses. A clause is composed of a predicate declaration and a disjunctive body. The predicate declaration has the form:

$$p(X_1,\ldots,X_n) + (X_{n+1} = V_1,\ldots,X_{n+k} = V_k) \leftarrow$$

where $n$, $k \geq 0$, the $X$'s are distinct variable names and the $V$'s are terms. We say that the predicate $p$ has $n$ global and $k$ local arguments, denote it by $p/n{+}k$ (or $p/n$ if $k = 0$), and call $V_1,\ldots,V_k$ the initial values of the local arguments of $p/n{+}k$. There can be at most one clause for any predicate $p$.

A call to $p/n{+}k$ in a procedure other than that of $p/n{+}k$ has the form $p(T_1,\ldots,T_m)$, where $m = n$ or $m = n + k$, where the $T$'s are terms. A call to $p/n{+}k$ that occurs in its own procedure may also have the form $p$, *i.e.*, the call may have no arguments whatsoever. Such a recursive call is called *implicit*. In addition, any call to $p/n{+}k$ that occurs in its own procedure may use the predicate name *self* as a synonym for $p$.

Variable names may be suffixed by a prime, e.g. $X'$, $Y'$. $X^n$ denotes the variable name $X$ suffixed by $n$ primes, $n \geq 0$. A primed version of a variable name denotes a new incarnation of the variable in the sense that the "most primed" occurrence of a variable name is considered the most updated version of the variable and hence is used in implicit recursive calls as explained below. We assume that the predicate $=/2$ is defined via the single unit clause $X = X$.

## 3.3 Semantics

The semantics of a logic program with implicit arguments $P$ is given by the following rewrite rules, whose application to completion results in a disjunctive logic program $P'$.

**Rule 1**: Expand local argument of calls. Replace each procedure call:

$$p(T_1,\ldots,T_n)$$

to a procedure $p/n + k$, by the call:

$$p(T_1,\ldots,T_n,V_{n+1},\ldots,V_{n+k})$$

where $V_n,\ldots,V_{n+k}$ are the initial values of the local arguments of $p/n{+}k$.

**Rule 2**: Expand implicit recursive calls. Replace each procedure call:

$$p \text{ or } self$$

in the clause of $p/n + k$ by the call:

$$p(U_1,\ldots,U_{n+k})$$

where $U_i$ is the most primed version of $X_i$ in the clause. We say that $X^n$ is *the most primed occurrence of $X$ in a clause $C$* if $X^n$ occurs in $C$, and for no $k > n$, does $X^k$ occur in $C$.

For example, applying the rewrite rules to the merge procedure results in:

```
merge(In1,In2,Out) :-
    In1 = [X|In1'], Out=[X|Out'],
        merge(In1',In2,Out');
    In2 = [X|In2'], Out=[X|Out'],
        merge(In1,In2',Out');
    In1 = [], Out = In2;

    In2 = [], Out = In1.
```

## 3.4 Special Notation for Stream Communication

Streams are the most commonly used data structure in concurrent logic programs. Recurrent processes almost always have one input stream and often have several additional input and/or output streams. Sending and receiving messages on a stream $Xs$ by a process $p$ can be specified by the clause schema:

```
p(...Xs...) :-
    Xs = [Message|Xs'],
    ...,
    self.
```

where the difference between sending and receiving is expressed using a language-specific synchronization syntax. Since this is such a common case, we found it worthwhile to provide it with a special notation. Our notation is reminiscent of CSP [5] and Occam [6] (and in logic programming the Pool language [2]):

```
Xs ! Message,
Xs ? Message,
```

These constructs specify, respectively, sending and receiving a message *Message* on a stream *Xs*. Each is equivalent to *Xs = [Message—Xs']* with the appropriate language-specific synchronization syntax added. The construct requires *n+4* fewer characters, where *n* is the length of the stream variable name, and hence is less liable to typing errors and probably also more readable.

Using this notation, a binary stream merger can be specified by:

```
merge(In1,In2,Out) :-
    In1 ? X, Out ! X, self;
    In2 ? X, Out ! X, self;
    In1 = [], Out = In2;
    In2 = [], Out = In1.
```

The predicate *append/3* can be specified using the first and third disjuncts of *merge/3*:

```
append(In1,In2,Out) :-
    In1 ? X, Out ! X, self;
    In1 = [], Out = In2.
```

It is interesting to note that this description of *append* facilitates its process reading . The program can be read as: "*append* is a process with three streams. Upon receiving an item on its first stream it sends that item on its third stream and iterates. If the first steam is closed, then the second and third streams are connected".

Using multiple primes allows multiple messages to be sent or received on the same stream, as the following example for the filtering of pairs of items on a stream shows:

```
remove_pairs(In, Out) :-
    In ? X, In' ? X,  Out ! X, self;
    In ? X, In' ? Y, X =\= Y,
        Out ! X, Out' ! Y, self.
```

### 3.5 Special Notation for Arithmetic

Arithmetic operations are quite common in ordinary and concurrent logic programs. Recurrent processes with a loop counter such as the following are abundant:

```
list(N,Xs) :-
    N > 0, Xs = [N|Xs'], N' := N - 1,
       list(N',Xs');
    N = 0, Xs = [].
```

Following C conventions we allow variable names to be suffixed by -- and ++, with the semantics of the expression `N--` given by replacing it with `N` and adding the conjunctive goal `N' := N-1`. Using the stream and arithmetic support, the above list generator can be written as:

```
list(N,Xs) :-
    N-- > 0, Xs ! N, self;
    N = 0, Xs=[].
```

Similarly, we define `+=` and `-=`, where `N += K` stands for `N' := N+K`, and
`N -= K` stands for `N' := N-K`.

## 4 Implicit Logic Programs with Inheritance

### 4.1 Concepts

The combination of inheritance and implicit arguments proves to be both highly succinct and more readable. For example, the program *counter2* of section 2.1 can now be rewritten as:

```
counter2(In) + (C=0, BackUp=0) :-
    +counter;
    In ? checkpoint, BackUp' = C, self;
    In ? restore, C' = BackUp, self.
```

The new style differentiates between global and local (hidden) arguments and also avoids copying *counter*'s code as well as specifying the arguments of the two recursive calls.

An implicit logic program with inheritance is translated into a pure logic program by applying the two previously defined rules. That of *lpi* (section 2.3) and that of implicit arguments (section 3.3). Minor changes in the rules are due. Those changes depend on the order in which we apply the two transformations.

## 4.2 Examples

A curious example in which *lpi* gives us some insight into a program, is the redefinition of *merge* (section 3.4) as:

```
merge(In1,In2,Out) :-
    +append(In1, In2, Out);
    +append(In2, In1, Out).
```

which means that merging is actually trying nondeterministically to append in both possible ways.

The following example implements a simple lookup table as a list of *key - value* pairs. The *create* predicate builds the list (named *Table*):

```
create(Table) :-
        Table=[].
```

i.e, a new table is an empty list. The following two predicates are not for direct usage. *search* iterates through the list as long as the key-value pair at the top of the list does not match a given *Key*. *find* inherits *search*, and adds a termination clause.

```
search(Key, Table, Table1) :-
    Table ? Key1 - Value1, Key=\=Key1,
        Table1 ! Key1 - Value1, self.

find(Key, Table, Table1, Ok) :-
    +search;
    Table = [],
        Table1 = [],
        Ok = false('key not found', Key).
```

The following *check* and *lookup* predicates inherit *find* and add a clause for the case where an identical key was found. The *replace* predicate inherits *search* directly since we want a different error message.

```
check(Key, Table, Table1, Ok) :-
    +find;
    Table ? Key - Value,
        Table1 = Table,
        Ok = true.

lookup(Key, Value1, Table, Table1, Ok) :-
    +find;
    Table ? Key - Value,
        Table1 = Table,
        Value1 = Value,
        Ok = true.

replace(Key, Value, NewValue, Table,
                            Table1, Ok) :-
    +search;
    Table ? Key - OldValue,
        Value = OldValue,
        Table1 = [Key - NewValue
                  | Table'],
        Ok = true;
    Table = [],
        Table1 = [],
        Ok = false('key not found',
                   Key - NewValue).
```

Finally *insert* and *delete* add and remove key–value pairs from the table.

```
insert(Key, Value, Table, Table1, Ok) :-
    +search;
    Table ? Key - Value1,
        Table1 = Table,
        Ok = false('key already exists',
                   Key - Value);
    Table = [],
        Table1 = [Key - Value],
        Ok = true.

delete(Key,Value,Table,Table1,Ok) :-
   +find;
   Table ? Key - Value1,
        Value = Value1,
        Table1 = Table',
        Ok = true.
```

As a third example we demonstrate the capabilities of the inheritance mechanism in a graphical environment by rewriting the window handling class from [10].

The first class defines a rectangular area with methods *clear* for painting the area specified by *Frame*, and *ask* to retrieve the rectangular's dimensions. *In* is an input port and *Frame* is a four-tuple of rectangle coordinates.

```
rectangular_area(In) +
         (Frame = {X, Y, W, H}) :-
    In ? clear,
        clear_primitive(Frame),
        self;
    In ? ask(Frame),
        self.
```

The following class *frame* is a rectangular area with some content, which means that apart from the methods *clear* and *ask*, one can *draw* the area boundaries, and *refresh* it. Note that *refresh* is just a combination of two previously defined methods *draw* and *clear*. This also fixes a subtle synchronization bug in Shapiro and Takeuchi [10] where the two methods were simultaneously activated, one by the class process and one by the *independent* superclass process, which could have caused drawing before clearing.

```
frame(In) + (Frame = {X,Y,W,H}) :-
    +rectangular_area;
    In ? draw,
        draw_lines(Frame), self;
    In ? refresh,
        self([clear, draw|In']).
```

The final class *labeledWindow* adds two more methods: *change*, to change a label, and *show* to show it. In addition we redefine the *refresh* method to show the label after refreshing (we thus require a method override mechanism). Another local variable *Label* is added.

```
labeledWindow(In) + (Frame = {X,Y,W,H},
            Label = default) :-
    +frame;
    In ? change(Label'), self;
    In ? show,
        show_label_primitive(Frame),
        self;
    In ? refresh,
        self([clear, draw, show|In']).
```

After we have the class *labeledWindow* we can subclass it to define our own window as in:

```
my_window(In,....) + (Frame = ...,
        Label = ...) :-
    +labeledWindow;

  <<my_window_additional_methods>>.
```

The generated code derived from the semantics of *lpi* and implicit arguments is not shown here due to space limitations.

# 5 Conclusions

## 5.1 Implementation

Both notations, *implicit* and *lpi*, have been implemented in FCP within the Logix system [11] by adding language preprocessors. The *lpi* preprocessor implements the combined notation of Section 4; *i.e.*, it translates FCP with inheritance and implicit arguments to FCP with implicit arguments. Another preprocessor translates implicit FCP to pure FCP. Each of the preprocessors is about 1000 lines of code. The *implicit* preprocessor was first written in FCP(:,?) [12]. That initial version was then used to bootstrap a new version written in the implicit notation. The *lpi* preprocessor is also written using the notation of implicit arguments.

## 5.2 Further work

A certain form of overriding is already available via the integration of *lpi* and a module system, described in Section 2.5. However, one may find useful also the ability of a subclass's method to override a method of the superclass. This can be achieved, for example, by stating that if several methods apply, then textual order dictates precedence. By appropriately placing inheritance calls, one can achieve the desired override effect.

Additional clarity and conciseness could be achieved by enabling an overriding method to also execute the overridden method (apart from doing some processing of its own). This feature, called *send to super* in the object oriented terminology, was easily implemented with Shapiro and Takeuchi's scheme [10] of a subclass having also an output stream to its super, by putting the method on the output stream. As an example of the send to super feature, suppose in *my_window* (section 4.2) we need to add functionality to the *draw* routine (e.g. drawing a grid on the rectangular_area), which means overriding the current *draw* method. Instead of copying the whole *draw* method, we would write:

```
In ? draw, send_to_super,
  draw_grid(Frame), my_window;
```

where *send_to_super* is a macro which copies the necessary code from the appropriate superclass.

A redundancy problem occurs when we want to use multiple inheritance but the generated inheritance graph is not a tree. For example, classes $b$ and $c$ both inherit $a$, and $d$ inherits both $b$ and $c$. Applying the transformation would result in $d$ having $a$'s methods twice. This (harmless) redundancy could be optimized later, e.g. by the decision graph compilation method [9].

### 5.3 Experience

The implicit arguments notation was incorporated into Logix more than two years ago, and has been used extensively by all members of our group. All of us found it preferable to the notation of plain logic programs.

Logic programs with inheritance were incorporated as an extension to the implicit arguments notation less than a year ago. It has been used by all of us extensively, and it has had a major effect on our programming style. One notable effect is that inheritance allows us to specify in a modular way processes with a dozen of arguments and dozens of clauses, by specifying multiple methods, each referring only to a subset of the process's arguments, and using multiple inheritance to specify the final process. This programming style meshes well with the decision graph compilation method to produce code which is readable, maintainable, and efficient.

We have implemented two large systems using *lpi*, each having several thousand lines of FCP code, and we find it hard to imagine how we could have written them without an inheritance notation.

## 6 Acknowledgments

The notation of implicit arguments was first described in an unpublished paper by Kenneth Kahn and the last two authors. We thank Yael Moscowitz and Marilyn Safran for comments on previous drafts.

## References

[1] Clark, K.L., Gregory, S., *PARLOG: Parallel Programming in Logic*, ACM Trans. on Programming Languages and Systems, 8(1), pp. 1-49, 1986.

[2] Davison, A., *POOL: A PARLOG Object Oriented Language*, Imperial College.

[3] Dershowitz, N., and Jouannaud, J.-P., Rewrite Systems, in *Handbook of Theoretical Computer Science*, J. van Leeuwen (Ed.), pp.243-320, Elsevier Science Publishing, 1990.

[4] Hewitt C., *A Universal, modular Actor formalism for artificial intelligence*, Proc. International Joint Conference on Artificial Intelligence, 1973.

[5] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall, New Jersey, 1985.

[6] INMOS Ltd., *OCCAM Programming Manual*, Prentice-Hall, New Jersey, 1984.

[7] Kahn K. M., *Objects - A Fresh Look*. Proceedings of the European Conference on Object-Oriented Programming, Nottingham, England, July 1989.

[8] Kahn K. M., Tribble D., Miller M. S., Bobrow D.G. *Vulcan: Logical Concurrent Objects*. in Concurrent Prolog: Collected Papers, Vol 2, Chapter 30, MIT press, 1987.

[9] Kliger, S., and Shapiro, E., From decision trees to decision graps, *Proc. of the 1990 North American Conf. on Logic Programming*, S. Debray and M. Hermenegildo (Eds.), MIT Press, pp. 97-116, 1990.

[10] Shapiro E., Takeuchi A., *Object Oriented Programming in Concurrent Prolog*. in Concurrent Prolog: Collected Papers, Vol 2, Chapter 29, MIT press, 1987.

[11] Silverman, W., Hirsch, M., Houri, A., and Shapiro, E., *The Logix System User Manual, Version 1.21*, in Concurrent Prolog: Collected Papers, Vol 2, Chapter 21, MIT press, 1987.

[12] Yardeni, E., Kliger, S., and Shapiro, E., The languages FCP(:) and FCP(:,?), *New Generation Computing*, 7(2-3), pp.85-87, 1990.

[13] Yoshida K., Chikayama T. *A'UM - A stream based Concurrent Object-Oriented Language* FGCS, Vol 2, 1988.