XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

Un Esquema para la Programación Orientada a Objetos en PROLOG[1]

Javier Pinto
Depto. Ciencia de la Computación
Pontificia Universidad Católica de Chile
Casilla 6177, Santiago, CHILE
Correo Electrónico (UUCP): jpinto@juncal.puc.cl

Resumen. En este trabajo se presenta un esquema para incorporar la Programación Orientada a Objetos al lenguaje Prolog. Una ventaja importante de este esquema es el apoyo que ofrece para la *abstracción de datos*, permitiendo encapsular definiciones de Tipos Abstractos de Datos. Además de esta capacidad, ausente en Prolog, la Programación Orientada a Objetos incorpora el concepto de *herencia de atributos*, permitiendo la jerarquización de los Tipos Abstractos de Datos que se definan.

El esquema presentado expande Prolog, incorporando un conjunto de predicados para la definición de las clases y de sus atributos. Al definir una clase, se introduce su definición como "query" u objetivo, que al ser satisfecho produce el efecto lateral de instalar, vía asserts, las definiciones en la base de datos de Prolog.

Además de las características funcionales de la extensión propuesta, que permiten la definición y utilización de clases, se discute la integración semántica de la extensión al lenguaje Prolog y se presentan algunos aspectos relacionados con la implementación.

Introducción

Una desventaja importante de Programación en Lógica Clásica [Hogg84], ejemplificada por el lenguaje PROLOG [Cloc84], es que sólo permite colecciones de reglas y hechos absolutamente planas [Subr85]. Esta característica del lenguaje impide modularizar las reglas, así como también impide la definición de jerarquías entre conjuntos de reglas.

Por otra parte, se ha argüido que PROLOG es inadecuado para la construcción de aplicaciones de Inteligencia Artificial, por ejemplo Sistemas Expertos, debido a que propugna un solo estilo o paradigma de programación [Bobr85].

En este trabajo se presenta un esquema para mejorar la posición de PROLOG con respecto a estas deficiencias, construyendo sobre el lenguaje un conjunto de predicados, denominado CLASS, que permita incorporar "Programación Orientada a Objetos" [Stef86] al lenguaje. En primer término, se introducen conceptos de la Programación Orientada a Objetos en el marco general de PROLOG aumentado con CLASS. Se presenta el mecanismo de herencia de atributos y el manejo de excepciones en el contexto de un ejemplo. Posteriormente se analizan los problemas semánticos asociados a la integracion de CLASS con PROLOG, se discuten algunos aspectos de implementación, y finalmente se presentan las conclusiones y se discuten posibles líneas de trabajo futuro.

Aspectos Básicos.

El conjunto CLASS de predicados permite la definición de clases de objetos, de modo que todos los objetos que pertenecen a una misma clase comparten la definición de un conjunto de datos, o variables de

[1]Este trabajo ha sido financiado en parte por la Dirección de Investigacion de la Universidad Catolica, bajo el proyecto 13/87.

instancia y un conjunto de predicados. Los conjuntos de predicados definidos para una clase reciben el nombre de métodos de la clase. Así, es posible definir una clase usando:

```
?- defClass(persona,[class])[1].                                    (1)
?- defInstanceVariables(persona,[(nombre,[]),                       (2)
                                 (apellido,[]),
                                 (padre,[]),
                                 (madre,[]),
                                 (edad,0)
                                 (hijos,[])]).
```

El primer predicado, **defClass**, se satisface si su primer argumento, **persona**, puede definirse como una nueva clase; el segundo argumento será explicado más adelante. El segundo predicado, **defInstanceVariables**, define el conjunto de variables de instancia asociadas a los objetos de la clase especificada por el primer argumento, el segundo argumento es un conjunto de pares ordenados (<nombre de variable>, <valor inicial>). Para definir que un objeto pertenece a una determinada clase, es necesario especificar:

```
?- instanceOf(jorge, persona,[(nombre,"Jorge")]).                   (3)
```

mediante lo cual, el objeto **jorge** queda definido como una instancia de **persona** con sus variables de instancia inicializadas como en (2), exceptuando **nombre**, que queda definido como "Jorge".

Los métodos permiten asociar predicados a una clase, de modo que éstos se pueden invocar sólo con los objetos pertenecientes a dicha clase. Por ejemplo:

```
?- defMethod(persona,                                               (4)
        [(actualizarEdad:-  send(self,get(edad,Y)),
                            Z is Y+1,
                            send(self,set(edad,Z)))]).
```

define el método **actualizarEdad** para la clase **persona**, el cual modifica el valor asociado a la variable de instancia **edad** aumentándolo en uno. En la definición de este método se han introducido dos características importantes asociadas a la programación orientada a objetos (OOP), según se discute a continuación. En OOP, la interacción con los objetos se realiza a través del envío de mensajes, para lo cual se utiliza el predicado **send**, de modo que un objeto recibe mensajes que es capaz de interpretar solamente si se ha definido un método para ello. De este modo, el satisfacer predicados sobre determinados objetos es equivalente al envío de mensajes a ellos. Así, para invocar un método asociado a un objeto, se le envía un *mensaje* a este último. Por ejemplo, para actualizar la edad de "Jorge", se especifica:

```
?- send(jorge,actualizarEdad).                                      (5)
```

que tiene el efecto descrito por el método definido en (4). Por otra parte, el símbolo distinguido **self** es utilizado en la definición de métodos para referirse al objeto que recibe el mensaje. Además, en la definición del método **actualizarEdad** se envían los mensajes **get** y **put** a **self**, los métodos que responden a estos mensajes quedan asociados a las clases en forma automática, cuando se realiza la definición de la clase correspondiente. El método **get** permite obtener el valor de una variable de instancia, mientras que el método **put** permite alterarlo, para lo cual se retractan los valores previos. Entonces, cuando **jorge** recibe el mensaje **actualizarEdad**, se invoca el predicado especificado en (4), el cual envía el mensaje **get**, con argumentos **edad** e **Y**, al mismo objeto **jorge**. Luego actualiza el valor

---

[1] El símbolo '?-', indica que los términos a la derecha del mismo se introducen como objetivos. La razón para introducir estos términos como objetivos, y no como aserciones simples, es que la información registrada en la base de datos debe ser actualizada cada vez que se defina una nueva clase, método u objeto. La alternativa de aceptar estos términos como aserciones, obligaría a introducir un "overhead" en el momento de utilizar los objetos definidos.

asociado a **edad**, cuyo valor original es retornado como binding de **Y**, al satisfacer **get(edad,Y)**, y, finalmente, se actualiza el "binding" de la variable de instancia **edad**.

Comúnmente, al declarar que un cierto objeto pertenece a una determinada clase, es necesario ejecutar algún código que permita inicializar apropiadamente las estructuras de datos asociadas a éste. Por ejemplo, suponga que se define el siguiente objeto:

```
?- instanceOf(david,persona,[ (nombre,"David López"),          (6)
                              (hijos, [benjamin,luis])]).
```

En esta situación, se ha declarado que **david** tiene como **hijos** a **benjamin** y **luis**, los cuales son también objetos que están siendo manipulados por el programa. Dada la forma en que se ha decidido estructurar la información, es necesario que al declarar el conocimiento respecto de los hijos de **david**, se actualice la información asociada a ellos. Esto es, se debe agregar como atributo de **benjamin** y **luis** el valor **david** para sus variables de instancia **padre**. Para realizar esta operación, es posible exigir que el usuario de las clases las utilice en forma disciplinada. Una mejor solución permite que se especifique un método para ser ejecutado cada vez que se cree una nueva instancia de una determinada clase, de manera que los datos mantengan su consistencia en forma automática. Así, la definición:

```
?- defMethod(persona,                                          (7)
        [(beforeMethod:- send(self,get(hijos,L)).
                         compatHijos(self,L))]).
```

especifica el método **beforeMethod**[1] para la clase persona. Para satisfacer el objetivo:

```
?- instanceOf(<objeto>, <clase>, <inicialización>).
```

se intenta satisfacer el método **beforeMethod** dado en (7). Si este método falla, entonces, la definición de la instancia también falla. En el ejemplo, **beforeMethod** falla cuando **compatHijos** falla, lo que ocurre si el objeto **self** (en este caso **david**), no puede ser **padre** de **benjamin** o **luis**, por ejemplo por tener éstos un padre diferente previamente definido.

Herencia de Atributos.

La Programación Orientada a Objetos, originalmente introducida por el Lenguaje Simula67 [Dahl70], ha hecho aportes muy importantes para el desarrollo de metodologías de diseño de software. El concepto más importante introducido por OOP es el de herencia de propiedades, concepto que permite jerarquizar los tipos de datos definidos en forma tradicional. La utilización adecuada de esta herramienta de conceptualización, entre otras, facilita la reutilización de código [Meye87].

Las características de CLASS presentadas hasta el momento posibilitan la definición de clases de objetos, de modo que el conjunto de métodos asociado a cada clase permite formar módulos de software. Cada uno de estos módulos puede interpretarse como la implementación de un Tipo de Dato Abstracto [Bish86]. Además de esto, en CLASS también se incorpora la idea de herencia, permitiendo definir jerarquías de clases. Por ejemplo, es posible definir que la clase **niño** es una subclase de **persona**:

```
?- defClass(niño,[persona]).                                   (8)
```

Esta definición crea una nueva clase de objetos (**niño**), la cual *hereda* las propiedades asociadas a la clase **persona**, esto es, su definición de variables de instancia, más su definición de métodos. Así, al declarar:

```
?- instanceOf(hugo,niño,[]).                                   (9)
```

---

[1] El nombre **beforeMethod** queda reservado para ser utilizado en el contexto indicado.

el objeto **hugo** hereda el conjunto de definiciones asociadas a **persona**, incluyendo todo aquello que es, a su vez, heredado por **persona**.

En la práctica, todos los objetos definidos para el universo de un programa tendrán en común un conjunto de primitivas básicas. Por ejemplo, todos los objetos son capaces de responder a los mensajes **get** y **put**, los cuales reciben como argumentos nombres y valores de variables de instancia asociadas a los objetos. Para lograr que todos los objetos definidos incorporen efectivamente el conjunto de primitivas básicas, sería posible agregar éstas como métodos de cada una de las clases que se definan. Una solución de este tipo es obviamente inadecuada, por el "overhead" extremo de producir código de similares características para cada una de las clases. Una mejor solución, adoptada por la mayoría de los lenguajes para la Programación Orientada a Objetos, es definir una clase elemental que incorpora un conjunto básico de métodos y de variables de instancia. En CLASS, esta clase básica es llamada **class** y todas las clases definidas por el usuario heredan sus atributos[1].

Un Ejemplo.

Este trabajo fue desarrollado en el contexto de un proyecto para la construcción de software para la simulación cualitativa de procesos [Pint88]. PROLOG surgió naturalmente como una de las opciones para la construcción del sistema. Sin embargo, inmediatamente se tropezó con la dificultad de organizar el conocimiento asociado al modelo que se deseaba construir.

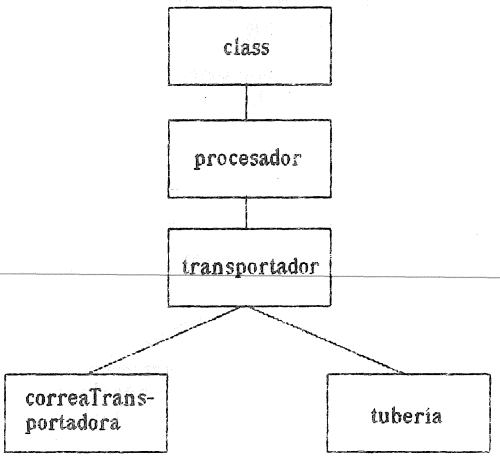

Figura 1. Una jerarquía de clases

Particularmente, en la aplicación mencionada interesa incorporar conocimiento relativo a ciertos dispositivos procesadores. Por ejemplo, existe un molino de mineral, correas transportadoras, tuberías, motores, etc. Todos estos dispositivos comparten entre sí un conjunto de características, lo que permite agruparlos en *clases* de objetos. Por ejemplo, las correas transportadoras y las tuberías pertenecen a la clase de dispositivos que permiten transportar material entre dos puntos. Sin embargo, correas transportadoras y tuberías no pueden aparecer como instancias de una misma clase, dado que estos dispositivos presentan diferencias importantes entre sí. En vez de esto, es posible definir independientemente las clases **tubería** y **correaTransportadora**, pero ambas como sub-clases de la clase superior **transportador**. Esta jerarquía de clases es ilustrada en la figura 1.

---

[1]En rigor, en CLASS es necesario especificar explícitamente cuando una clase será subclase de **class**. Por ejemplo, en (1), la clase **persona** fue declarada como subclase de **class** en forma explícita; en cambio, en (8), la clase **niño** simplemente hereda esta definición de su clase superior **persona**.

Para declarar este conocimiento en CLASS, se puede especificar:

```
?- defClass(procesador, [class]).                                    (10)
        % Se define que procesador es una clase de alto nivel (descendiente
        % directo de class).

?- defInstanceVariables(procesador,[ (vEndógena, []),                (11)
                                     (vExógena,[])]).
        % El estado del procesador es caracterizado por dos conjuntos de
        % variables, exógenas y endógenas, las que se introducen como
        % variables de instancia de la clase.

?- defClass(transportador,[procesador]).                             (12)
        % La clase transportador es descendiente de la clase procesador.
?- defInstanceVariables(transportador,[(material , [])]).            (13)
        % Los objetos transportadores transportan material específico
        % (mineral, pulpa, etc.).

?- defClass(molino,[procesador]).                                    (14)

?- defClass(correaTransportadora,[transportador]).                   (15)
?- defInstanceVariables(correaTransportadora,[(motor , [])]).        (16)
?- defClass(tuberia,[transportador]).
?- defInstanceVariables(tuberia,[(bomba , [])]).                     (17)
        % Tanto correaTransportadora como tuberia son subclases de
        % transportador. La clase correaTransportadora tiene una
        % variable de instancia motor. Análogamente, tuberia tiene asociada
        % una bomba.
```

En (16), al definir la variable de instancia **motor**, se tiene la intención de que ésta quede asociada a un objeto de clase **motor**[1], para lo cual es posible utilizar la siguiente secuencia:

```
?- chequearMotor(mot1), send(correa, set(motor, mot1)).              (18)
```

donde **chequearMotor**, es satisfecho sólo si **mot1** es una instancia de un motor y es un motor apropiado para mover una correa. Sin embargo esta solución es pobre, por cuanto se requeriría utilizar predicados de "chequeo" para todas las variables de instancia interesantes. En vez de esto, sería preferible utilizar:

```
?- send(correa, set(motor, mot1)).                                   (19)
```

siempre que se pueda especificar que cuando se realice un **set** de la variable de instancia **motor**, en forma automática se haga un chequeo previo del valor asociado a **mot1**. Para esto, es posible incluir un predicado *if-added* a las variables de instancia (solución común en lenguajes orientados a *frames* [Fike85]). En vez de esto, se ha preferido manejar este tipo de situaciones como *excepciones*. Una excepción se produce cuando en una jerarquía, un método de una clase superior no es adecuado para responder mensajes específicos de clases inferiores a ella. Típicamente, esta situación se produce cuando un método debe ser refinado o especializado. Para especializar un método, basta con redefinirlo para la clase inferior en cuestión. Este esquema funciona adecuadamente, pues al enviar un mensaje a un objeto, primero se revisa si existe un método para responder a él asociado a la clase más baja del objeto. Si dicho método no se encuentra, se busca un método asociado a una de sus clases superiores.

---

[1]Esta asociación no está relacionada con la asociación jerárquica entre clases. Más bien es una asociación entre instancias de diferentes clases, relación que debe ser manejada en forma explícita por el programador.

Entonces, en el ejemplo es posible redefinir el método **set** asociado a la clase **correaTransportadora** para la variable de instancia **motor**, de modo que además de asociar un valor a la variable de instancia, haga el chequeo correspondiente. Un primer intento, refinado más adelante, para realizar la redefinición es el siguiente:

```
?- defMethod(correaTransportadora,                          (20)
     [ (set(motor,Value):-
          !,                                               (21)
          chequearMotor(Value),
          send(self, set(motor, Value)))]).                (22)
```

En la redefinición de este método, es necesario destacar dos detalles importantes. El primero es el *cut* de la línea (21), el cual asegura que el objetivo asociado al mensaje **set**, para **motor**, no se verá satisfecho por un método **set** asociado a una clase superior a **correaTransportadora**. Este tema se discute más adelante. En segundo término, en (22) se pretende modificar el valor de la variable de instancia **motor** asociada a la instancia de **correaTransportadora** que recibe el mensaje. Sin embargo, el efecto real es que (22) queda como un llamado recursivo, donde, para satisfacer el mensaje **set**, se envía un mensaje **set** al mismo objeto y con los mismos argumentos, produciéndose un ciclo sin término. Una versión correcta para la redefinición será introducida en la sección subsiguiente, donde se discute en mayor detalle el problema de la redefinición de métodos.

Herencia Múltiple.

En algunos lenguajes orientados a objetos, es posible definir clases con herencia múltiple, permitiendo crear jerarquías representadas por grafos dirigidos. En general, el mecanismo de herencia múltiple no es difícil de implementar, la dificultad asociada a este esquema se deriva de la complejidad semántica resultante. En particular, sería posible tener una jerarquía como la que ilustra la figura 2.

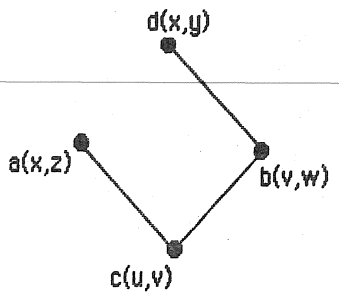

Figura 2. Una jerarquía con herencia múltiple.

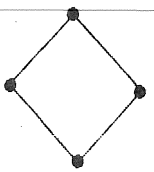

Figura 3. Un ciclo en el grafo no dirigido subyacente, en la jerarquía de clases.

Cada nodo en la jerarquía de la figura tiene una etiqueta **c(m,n)**, tal que **c** representa una clase con métodos definidos **m** y **n**. Cuando se envía un mensaje a un objeto perteneciente a la clase **c**, es necesario establecer un ordenamiento entre los predecesores de éste para decidir qué clase debe interpretarlo y responder. Por ejemplo, ante un requerimiento **send(c_1,X)**, con $c_1 \in$ **c**, es necesario determinar si c_1 debe comportarse como objeto de la clase **d**, o de la clase **a**. La situación se complica aún más cuando el grafo tiene ciclos en el grafo no dirigido subyacente, como ilustra la figura 3.

Una posible solución, [Moon86]), es recorrer el grafo *bottom-up* en *depth first* en algún orden, listando todas las clases con sus métodos, si alguna se repite (debido a ciclos) se eliminan todas sus ocurrencias, salvo la primera. Dicho esquema resuelve el problema, aunque en forma solamente parcial, puesto que el aporte semántico de la solución es bastante oscuro. ¿Qué diferencia tendría, desde

un punto de vista semántico, realizar un recorrido *breadth-first*, en vez de un *depth-first?*; en CLASS este problema aún no se ha abordado, únicamente se permite definir jerarquías con forma de árbol (sólo un ancestro por clase).

Manejo de Excepciones o Especialización de Métodos.

Típicamente, al recomponer métodos se requiere utilizar el método original, situación ejemplificada en (20). En este ejemplo, el refinamiento del método **set**, requiere del método original. Sin embargo, con las herramientas provistas hasta este momento, no es posible referirse a este método a través de un objeto de la clase **correaTransportadora**. Lo que se requiere es un mecanismo explícito para hacer referencia a la jerarquía de clases. En CLASS, al igual que en lenguajes con capacidades para OOP (ej. Flavors [Moon86] y Smalltalk [Gold83]), se provee el nombre distinguido **super**, el cual permite hacer referencias a clases superiores en el momento de la definición de métodos. Por ejemplo, el siguiente método redefine el método **set** definido en (20):

```
?- defMethod(correaTransportadora,                                    (23)
        [ (set(motor,Value):-
                !,
                chequearMotor(Value),
                send(super,set(motor,Value)))]).
```

Aquí se ha incluído una referencia al símbolo distinguido **super**, cuando una instancia de la clase **correaTransportadora** recibe un mensaje **set(motor,Value)**, el método refinado envía el mismo mensaje (**set**), pero utiliza el símbolo **super**. Esto significa que se debe buscar un método para responder al mensaje **set** a partir de las clases superiores a **correaTransportadora**, evitando así el fenómeno de recursión que se produce en (22). Así, a menos que se haya redefinido el método **set** para una clase superior a **correaTransportadora**, éste será atendido por el predicado original, definido para la clase **class**.

Consideraciones Semánticas.

En esta sección, se estudia la integración que existe entre la extensión CLASS que se propone para el lenguaje PROLOG, y la semántica asociada a este último. Para realizar el análisis es necesario considerar los aspectos declarativos y *procedurales* de PROLOG. Para una exposición más resumida del tema, se limitirá a considerar aquellos predicados que son más interesantes, desde el punto de vista del contenido semántico asociado a ellos.

La definición de clases, métodos y variables de instancia, implica necesariamente un efecto lateral que modifica el contenido de la base de datos de PROLOG. Por ejemplo, el requerimiento:

```
?- instanceOf(hugo, niño,[]).                                        (24)
```

establece que el objeto hugo pertenece a la clase **niño**. En CLASS, para satisfacer este requerimiento se modifica el contenido de la base de datos, realizando aserciones que declaran en forma explícita las clases de los objetos presentes en el sistema. De este modo, una interpretación *procedural* es simplemente que instanceOf produce un efecto lateral (instalando declaraciones en la base de datos). Desde un punto de vista declarativo, (24) puede interpretarse como: "¿Es cierto que **hugo** es **niño**?". Por otra parte, el requerimiento:

```
?- instanceOf(hugo,X,_).                                             (25)
```

podría interpretarse como:"¿a qué **clase** pertenece **hugo**?". Sin embargo, debido al efecto lateral introducido, indeseable desde un punto de vista puramente lógico, esta interpretación es inaceptable. El problema fundamental que se produce es que no es posible determinar cuáles son las intenciones del usuario (es decir, el requerimiento es ambiguo), por ejemplo, el requerimiento (25) podría también interpretarse como la aserción: "**hugo** es un objeto que pertenece a todas las clases". ¿cuál es la

interpretación correcta?, dependerá de la aplicación específica. En CLASS, este problema se evita exigiendo que todos los parámetros de instanceOf estén instanciados al momento de ser invocado. Problemas análogos al ilustrado con **instanceOf** se presentan con la definición de métodos y de variables de instancia.

El tratamiento de los mensajes tampoco está libre de problemas, pero, afortunadamente, la situación es más clara. Cuando se envía un mensaje utilizando **send**, como en:

?- **send(**‹**objeto**›**,**‹**mensaje**›**)**. (26)

send no es interpretado como un predicado cuya verdad dependa estrictamente de sus argumentos. Para una interpretación adecuada, es necesario escaparse del formalismo de lógica de primer orden, por cuanto el predicado que se invoca en (26) no está determinado por la declaración, sino que éste debe identificarse a partir del ‹**objeto**› y del nombre del ‹**mensaje**› específico. De modo que la interpretación de (26) está dada por:

send(‹**objeto**›**,**‹**mensaje**›**)** → (27)
predicado(clase(‹**objeto**›**),nombre(**‹**mensaje**›**))(**‹**objeto**›**)**.

donde **clase** es una función que denota la clase a la que pertenece su argumento, **nombre** es una función que denota el nombre de ‹**mensaje**›, y **predicado** es una función que denota un predicado (obviamente la función no es de primer orden), el cual se aplicará a ‹**objeto**› para obtener el valor de verdad resultante.

Por otra parte (27) puede interpretarse como una regla sintáctica (i.e. una macro), que transforma el lado izquierdo en el derecho. Si ambos parámetros de **send** se encuentran instanciados (situación más típica), entonces los argumentos a la función **predicado** son constantes, lo que implica que el literal **predicado(_,_)** es una constante. Entonces, si se exige que ambos parámetros de la función **send** sean constantes, el problema de introducir características fuera de la lógica de primer orden desaparece.

No obstante lo anterior, se ha preferido no introducir la exigencia referida. Fundamentalmente, porque la aplicación del send considerando sus argumentos variables, tiene interpretaciones que son interesantes. Particularmente, existen diferentes interpretaciones del mensaje (26), dependiendo de los objetos que se encuentren instanciados. Por ejemplo, el mensaje:

?- **send(Objeto,mensajeA)**. (28)

puede interpretarse como "¿Existe algún objeto que pueda responder a **mensajeA**?"[1]. Por otra parte, el mensaje:

?- **send(objeto,X)**. (29)

debiera satisfacerse con cualquier mensaje al que **objeto** pueda responder. Sin embargo, el significado del resultado es muy poco claro (e.g., ¿cómo interpretar un *yes* como respuesta a (29)?). Por esta razón, se ha decidido restringir el segundo parámetro a una función de primer orden, cuyo nombre debe aparecer como literal.

Finalmente, es posible interpretar el contenido semántico agregado a PROLOG con extensión CLASS, en el marco de la lógica de primer orden para el caso general en el que se restrinja el uso del predicado send a aquél en el cual sus argumentos estén siempre instanciados. En otro caso, sería necesario recurrir a interpretaciones bajo un esquema lógico diferente.

---

[1]Esta interpretación es análoga a aquella de [Fuku86].-

Aspectos de Implementación.

CLASS se ha desarrollado utilizando una versión de PROLOG de acuerdo al estándar de la Universidad de Edimburgo [Cloc84]. Cada uno de los predicados **defClass**, **defInstanceVariables** e **instanceOf** son predicados simples que instalan información en la base de datos. La información que se almacena es la relativa a cada clase definida, junto con la definición de la jerarquía correspondiente.

Para realizar la definición de métodos, el predicado **defMethod** realiza una traducción del método especificado en el segundo parámetro. De modo que ante una especificación como:

?- **defMethod(clase,** (30)
[(método(Arg$_1$,Arg$_2$,...,Arg$_n$):- ***Código*****(self,super)]).**

donde ***Código*****(self, super)**, representa el código del método, el cual puede tener referencias a los símbolos distinguidos **self** y **super**, se realiza una traducción a:

método(clase, Self,Arg$_1$,Arg$_2$,...,Arg$_n$):- ***Código*****(Self, super(clase)).** (31)

Así, el método original es transformado a un predicado estándar de PROLOG. En forma análoga, los mensajes (e.g. (26)) son traducidos consistentemente con la traducción de los métodos.

En la actualidad, CLASS funciona en forma interpretada sobre el sistema PROLOG, en el cual se leen las definiciones de CLASS previamente a su utilización. La ejecución interpretada de CLASS presenta la ineficiencia derivada de la búsqueda de los métodos en tiempo de ejecución. Es posible incorporar CLASS a PROLOG, de modo que los envíos de mensajes sean traducidos a invocaciones de los predicados correspondientes. Este esquema permitiría una ejecución libre del *overhead* propio a la interpretación de los mensajes.

Conclusiones y Trabajo Futuro

La extensión que se propone para PROLOG cumple con el objetivo de permitir Programación Orientada a Objetos dentro del marco de la programación en lógica. El sistema resultante hereda características importantes de ambas clases de lenguajes, esto es, lenguajes para programación en lógica y lenguajes para programación orientada a objetos.

Dentro de las ventajas más notables se encuentra la facilidad para la organización de reglas permitiendo construir estructuras conceptuales como aquellas propias de *frames*. Esta característica tiene gran importancia desde un punto de vista de Ingeniería de Software, puesto que permite la definición de módulos con encapsulamiento de predicados. Por otra parte, la programación orientada a objetos se ha reconocido como muy importante para el desarrollo de aplicaciones de IA, puesto que permite implementar con facilidad esquemas de representación basados en *frames* y redes semánticas.

En el futuro se espera especificar formalmente un lenguaje que extienda PROLOG con las ideas aquí expuestas, esperando producir la implementación de un intérprete para el mismo.

Por otra parte, es necesario ganar experiencia con la utilización de esta extensión, de modo de poder analizar, en base a casos concretos, los beneficios de la programación orientada a objetos en PROLOG.

Referencias:

[Bish86] J. Bishop, "Data Abstraction in Programming Languages", Addison Wesley, 1986.

[Bobr85] D.G. Bobrow, "If Prolog is the Answer, What is the Question? or What it Takes to Support AI Programming Paradigms", IEEE Trans. Software Eng., vol. SE-11, pp. 1401-1408.

[Cloc84] W.F. Clocksin y C.S. Mellish, "Programming in Prolog, Second Edition", Springer Verlag, 1984.

[Dahl70] O.J. Dahl, B. Myhrhang y K. Nygaard, "Simula67 Common Base Language", Norwegian Computer Center, S-22, 1970.

[Fike85] R. Fikes y T. Kehler, "The Role of Frame-Based Representation in Reasoning", Communications of the ACM, September 1985, pp. 904-920.

[Fuku86] K. Fukunaga, S. Hirose, "An Experience with a Prolog-based Object-Oriented Language". Proceedings de la 1986 Object Oriented Programming Systems, Languages and Applications, 1986, pp. 224-231

[Gold83] A. Goldberg, D. Robson, "Smalltalk-80: The Language and Its Implementation", Addison - Wesley, 1983.

[Hogg84] C.J. Hogger, "Introduction to Logic Programming", Academic Press, 1984.

[Meye87] B. Meyer, "Reusability: The Case for Object-Oriented Design" IEEE Software, March 1987, pp. 50-64.

[Moon86] D. Moon, "Object Oriented Programming with Flavors", Proceedings de la 1986 Object Oriented Programming Systems, Languages and Applications, 1986 pp. 1-8.

[Pint88] J. Pinto, "Arquitectura para Sistemas Expertos en Control de Procesos: Reporte Preliminar", Anales de la Conferencia Latinoamericana IEEE, Latincon '88, Buenos Aires Argentina, Abril 1988.

[Stef86] M. Stefik y D.G. Bobrow, "Object Oriented Programming: Themes and Variations", AI Magazine, Vol.6. No. 4, Winter 1986, pp. 40-62.

[Subr85] P.A. Subrahmanyam, "The 'Software Engineering' of Expert Systems: Is Prolog Appropriate?", IEEE Trans. Software Eng., vol. SE-11, pp. 1391-1400.