3Y-8 論理型言語における多重名前空間の実現について

小長谷　明彦　　新　淳
日本電気㈱　C&Cシステム研究所

1．はじめに

Prologに代表される論理型言語では、プログラム（クローズ）を名前（述語名）を用いて参照するため、大規模システムを開発する際に名前の衝突がシステム開発の大きなネックとなる。この名前の衝突を解決する方法の一つとして、Common Lisp[1]ではパッケージを利用した多重名前空間を提供している。本稿ではこのような多重名前空間を論理型言語に適用した際の利用法、設計上の問題点、ならびにマルチプロセス環境への拡張について述べる。本稿で述べる多重名前空間は第五世代計算機プロジェクトの一環として開発した逐次型推論マシンＣＨＩ[2]上の論理型言語SUPLOGに実装稼働している。

2．多重名前空間の利用法

多重名前空間を論理型言語に導入することにより以下の処理が可能となる。

①名前の隠ぺい
②システム定義述語の再定義
③オブジェクト指向的プログラミング

多重名前空間の最大の利点は、多人数で大規模システムを開発した際に、名前の衝突を容易に回避することができる点にある。特に論理型言語の場合、名前の衝突した述語は再定義ではなく、クローズのマージとなるため、バグは非常に見つけにくい。名前の隠ぺいはパッケージ毎にパッケージの外部から見える名前（外部アトム）とパッケージ内で局所的な名前（内部アトム）を用意することにより実現できる。例えば、SUPLOGではパッケージｆｏｏにおいて定義された述語ｐ，ｑ，ｒの内、ｐとｑを参照可能にしたいときは次のように定義する。

```
:- in_package(foo).
:- export([p,q]).
p :- ...
q :- ...
r :- ...
```

パッケージｆｏｏの定義した述語を使用するときは、ｆｏｏを継承するか、参照する述語名をimportすればよい。

多重名前空間のもうひとつの利点はシステム定義述語を容易に再定義できることである。このような再定義の必要性は、他の論理型言語のコンパチビリティ　パッケージを作る場合や、基本データ型を追加して論理型言語の仕様を拡張しようとしたときに生じる。例えば、無限精度整数（大数）を実現することを考える。大数の表現法として、"bignum"をファンクタとする構造体を用いた場合、この大数が整数と同じように使えるようにするためには"is"や、"integer"といった組み込み述語を大数を扱えるように拡張する必要がある。このような拡張は名前空間の継承機能、継承の制限機能および継承しなかった名前をアクセスする機能を用いて以下のように実現することができる。

```
:- in_package(with_bignum,$use(prolog)).
:- shadow([prolog$integer,prolog$is]).
integer(X) :- prolog$$integer(X),!;
              X = bignum([_|_]).
```

ここで、"in_package"はprologパッケージ（システム定義の述語名を保持する名前空間）の名前を継承した新しいパッケージwith_bignumを生成することを意味する。また、"shadow"はwith_bignumにおいてprologで定義された名前のうち、integer、isを継承から外すことを示す。また、"prolog$$integer"はprologパッケージにおけるintegerの定義を利用することを表している。

また、多重名前空間の新しい利用法として、パッケージを知識表現のフレームあるいは"オブジェクト"のように解釈するオブジェクト指向的プログラミングの実現がある。例えば、鳥とペンギンの関係は、パッケージbird、penguinを用いて次のように定義できる。

```
:- in_package(bird).
number_of_wings(2).
canfly.
:- in_package(penguin,$use(bird)).
:- shadow(bird$canfly).
canfly :- fail.
```

パッケージpenguinはパッケージbirdを継承することにより、「翼の数は２枚である」といった鳥に関する一般的な性質をそのまま受け継ぐ。一方、「飛べる(canfly)」という性質については、名前空間から削除し、新たに「飛べない（canfly :- fail）」という性質を再定義することにより、例外事象を扱うこともできる。この"オブジェクト指向プログラミング"はいわゆるクラスーインスタンス階層を持たないが、メッセージ（述語）の解釈をオブジェクト（パッケージ）側で行なうという点でオブジェクト指向的である。さらにオブジェクトがコンパイル時に分かっているときは述語を直接呼び出すように最適化することができるので、本方式では静的なメソッド結合が多い場合はかなりの性能向上が期待できる。

Design and implementation of multiple name spaces in logic programming
Akihiko Konagaya and Atushi Atarashi
NEC Corporation. C&C Systems Research Laboratories

3．設計上の問題点

論理型言語における多重名前空間の設計上の問題点は次の２点である。

①データとしてのアトムの取扱い。

②ファンクタの多重化ｖｓアトムの多重化。

論理型言語ではアトムをデータと述語の両方に用いている。ここで、述語名でないアトムまで多重化するか否かという問題が発生する。多重化した際の利点は名前のカプセル化が完全に実現できることである。例えば、前節で述べた大数を導入する例では、型の記述子としてbignumというアトムを用いていたが、名前が多重化されていれば利用者が定義したbignumという構造体を大数として処理してしまうようなエラーを防ぐことができる。一方、述語名でないアトムを多重化しないときの利点は、ユニフィケーションの際に名前の多重構造を意識せず済む点にある。しかしながら、名前空間の継承機能を用いればほとんどの場合、利用者は名前の多重構造を意識せずに済むし、エラーメッセージのように本質的に大域的なデータはむしろ「文字列」で表現する方が自然である。このような観点からSUPLOGでは、アトム名全体を多重化する方式を採用している。

もう一つの問題点は、論理型言語の述語の管理単位が述語名でなく述語名／引数（すなわちファンクタ）である点に起因している。すなわち、論理型言語における名前の多重化の単位としては、アトムとファンクタの２通りが考えられる。アトムを多重化した際の問題点は、引数の数が異なっていても述語名が同じ場合は名前の衝突がおきてしまうことである。例えば、パッケージｆｏｏでｐ／２をパッケージｂａｒでｐ／１を定義したときに、両者は別な述語として定義したにもかかわらず、ｆｏｏとｂａｒの両方を継承したパッケージでは名前の衝突が起きてしまう。

一方、ファンクタを多重化した際には、よりきめ細かな述語の管理が可能な代わりに、オペレータ宣言等で曖昧な記述が生ずる可能性がある。例えば、パッケージfooでファンクタf/1を後置型のオペレータに、パッケージbarでf/2を中間置型のオペレータに宣言したとする。f/1,f/2共に参照可能なパッケージにおいて

ｔ１　ｆ　ｔ２

を読み込んだ際に、オペレータは引数情報を持たないためｆがf/1なのかf/2なのか決定することができない。また、組込述語

```
functor(Term,Functor,Arity)　や
Term =.. [Functor|Args]
```

において、Termが与えられたとき、Functorとして何を返すべきかも明確ではない。このような事象がおきる一つの要因は、Prologにおけるアトム名と述語名の区別が中途半端な点にある。例えば、０引数の述語をｐ（）のように記述し、アトムｐと区別するようにすれば、ファンクタを多重化した際の問題点はかなり解決する。ただしSUPLOGでは、従来型Prologとの整合性を重視してアトムの多重化を行っている。

4．マルチプロセス環境への拡張

多重名前空間をマルチプロセス環境に拡張するときの実現法は共有するデータの度合により、大きく分けて以下の４つのケースに分けられる。

①共有データを持たない方式。

②プログラム本体（コード）のみを共有する方式。

③コードとアトムを共有する方式。

④コードとアトムとパッケージを共有する方式。

①の共有データを持たない方式は、プロセス間での名前空間アクセスの衝突を防ぐもっとも確実な方法であるが、システム定義のプログラムを全てコピーする必要があるためメモリの消費量が増え、かつ、プロセス生成のコストが非常に大きいという問題点を持つ。

次に、②のパッケージ、アトムをコピーし、コードを共有する方式では、プロセス生成時のオーバーヘッドは大幅に減少する。ただし、システムプログラムを正しく実行するためにはコード中に埋め込まれたアトムを各プロセス毎にコピーしたアトムとして解釈できるような機構が必要となる。すなわち、システムプログラムのデータとコードを分離し、データは必ず間接参照する「再入可能コード」として実現する必要がある。

③のパッケージのみをコピーし、アトム、コードを共有する方式ではプロセス生成のオーバーヘッドはさらに小さくて済む。この方式ではシステム定義のアトムおよびコードは全て共有することになるが、パッケージはプロセス毎に局所的なので、仮にあるプロセスでシステム定義のアトムを除去、再定義してもその影響は当該プロセスの中だけで済み、他のプロセスへの影響を排除することができる。

④のパッケージまで共有する方式では、プロセス生成のコストは最小となるが、複数のプロセスが並行して動作する環境下ではシステムとしての一貫性を保つのが難しくなる。例えば、プロセスＡとＢがある同一の名前をinternし、別のプロセスＣがその名前をuninternしようとしている状況を考える。プロセスＡ，Ｂの両方がプロセスＣの前か後に実行したときはプロセスＡとＢは同一の名前を共有するが、プロセスＣがプロセスＡとＢの間に実行したときはプロセスＡとＢは別の名前を参照することになる。

どの方式をとるかはシステムの目的、アーキテクチャサポートに依存する。例えば、マルチユーザ環境では①の方式が安全であるし、再入可能コードがサポートされていれば②の方式もとれる。また、並列プログラミングが目的であれば④の方式も魅力的である。ＣＨＩでは単一ユーザ環境における複数タスク処理を目的とすることから③の方式を採用している。

参考文献

[1] Common Lisp the language, Digital Press, 1984

[2] 幅田　他：逐次型推論マシン：ＣＨＩ、情処記号処理研究会、８７－ＳＹＭ－４２－１，１９８７年９月