リフレクション原理によるフレームシステムの拡張

1F−1

○木島勝弘，西谷泰昭 (群馬大学)

1. はじめに

ある対象領域をフレームシステムで記述しようとする場合、純粋に既存のフレームシステムの枠組だけでは、知識をうまく記述できないことが多い。そのため、フレームシステムに次のような変更を加えることになる。

1. 対象領域独自の関数の追加
2. 既存の関数の変更
3. 新しいインヘリタンスの追加
4. 特定のフレーム、スロットに対する機能の追加

これらの機能は、既存のフレームシステムのインタープリタを修正するか、あるいは与えられたフレームシステムの枠組(デーモンなど)を利用する方法がある。前者は、使用者がフレームシステムのソースコードを理解し、修正できる能力がなければならない。これではフレームシステムを使用する前段階のコストが大きくなり、生産性は向上しない。後者は対象領域の知識とフレームシステムの定義が混在してしまう形になり、対象領域の本来の知識が不明確になってしまう。本研究においては、上で述べた機能をメタレベルの機能と考え、メタレベルにおける柔軟なシステム構築機能を持つフレームシステムを実現する。

2. リフレクション

リフレクションは Smith[1] や Ferber[2] によって研究された計算メカニズムで、より強力なプログラミングを可能にするものである。リフレクションは次のように定義されている [3]。

1. その計算システムが自己表現を持つ
2. その表現とシステムが因果的に結合されている。

これはシステムがシステム自身の状態を表現するメカニズムを持ち、表現された状態を操作することで実際のシステムを変更することができることを述べている。

リフレクションを導入したシステムでは、システム自身の状態を感知できることから、より高度な操作が可能となり、従来のシステムではプリミティブであった手続きも使用者が定義できるようになる。ここでいう使用者とは本システムを利用して要求されたフレームシステムを作成する人も、実際に作成されたフレームシステムを利用する人も含む。フレームシステム作成のレベルを概念的に次の4段階に分ける。

1. 核となるフレームシステムを作成するレベル
2. 1 を用いて、要求されたフレームシステムの骨格を作成するレベル(インヘリタンスや低レベルのアクセス関数など、一般的な部分)
3. 要求されたフレームシステムに独自なもの(スロットに固有の手続きなど)を作成するレベル
4. 実際にフレームシステムを利用し、対象領域の知識を記述するレベル

1,2,3 で対象レベルを記述するためのシステムができる。1 が本研究で作成するフレームシステムである。従来のシステムでは 1,2 は固定されていた。3 については値を設定したり、別の言語で作成し、埋め込むなどの方法で実現されており、拡張性の高いものではなかった。リフレクションを用いることの利点は、2,3,4 のどのレベルにおいてもその前の段階のフレームシステムを利用することができ、いつでも前のレベルに戻り、修正を施すことができることである。

従来のフレームシステムは、

インタープリタ + フレーム群 (1)

で構成されている。インタープリタはフレーム群に対するアクセス関数とそれらを制御する実行器からなり、フレーム群は対象領域の事象を表す。フレーム群の性質(インヘリタンスや各スロット、ファシットの機能など)はインタープリタによって決定し、この性質を変えるにはインタープリタを修正する必要がある。そこで、インタープリタのアクセス関数が自己を決定するものと考え、関数定義をデータ(フレーム)として表現する。よって (1) 式を次の (2) 式のように変更する。

(メタ - インタープリタ + 定義フレーム群)
+ フレーム群 (2)

定義フレームもフレームであるので、メタ - インタープリタによって修正が可能である。またメタ - インタープリタは変更された定義フレームを参照し、フレーム群を操作するので、(2) のシステムはリフレクションの原理を満たしている。また、メタ - インタープリタも定義フレーム群を対象領域としたフレームシステムのインタープリタとなるので、これも (メタ - メタ - インタープリタ + メタ - インタープリタのための定義フレーム群) とすることができ、インタープリタの無限階層ができる。

本システムでは、さらに拡張性を高め、個々のフレームに対して手続きが定義できるようにする。そのために、フレームシステムを次のように定義する。

メタ - インタープリタ
+ (定義フレーム + 対象フレーム) 群 (3)

3. メタレベルの構成

前節の (3) のシステムを実現するために、対象フレームの各々にメタフレームを設け、メタフレームに関数定義を書き込むことにする。また、フレーム群の管理のために USER フレームを作り、対象フレーム群の管理を行なう。USER フレームもひとつのフレームであるので、メタフレームが存在するが、それにはフレームシステム全体のデフォルトとなる関数を登録する。これらのフレームを図 1 に示す。

フレームにはメタフレームが必ず存在する。よってメタフレームのメタフレームも存在することになる。メタ - メタフレームはメタフレームの操作方法を表すので、メタ - メタ - インタープリタに相当する。また meta リンクと system リンクは本システムでプリミティブに用意したものであり、これらについてはその性質を変えることはできない。対象レベルと

Extension of frame system based on reflection principle

Katsuhiro Kijima , Yasuaki Nishitani
Gunma University

メタレベルの関係を表したのがmetaリンクで、各レベルのクラス-インスタンス関係に相当するのがsystemリンクである。

レベル1で作成される関数はプリミティブなものであり、フレームへのプリミティブなアクセス関数は基本的にp-fget, p-fput,p-fremoveだけである。レベル2では、USER-METAに記述されるべきデフォルト関数を作成する。レベル3では、対象領域に固有のスロットに関する手続きや、レベル4で使用される手続きをUSER-METAに定義する。レベル4では、実際に対象領域の知識を記述してゆくわけだが、レベル4のユーザは、USERフレームやメタフレームの存在を知る必要はない。

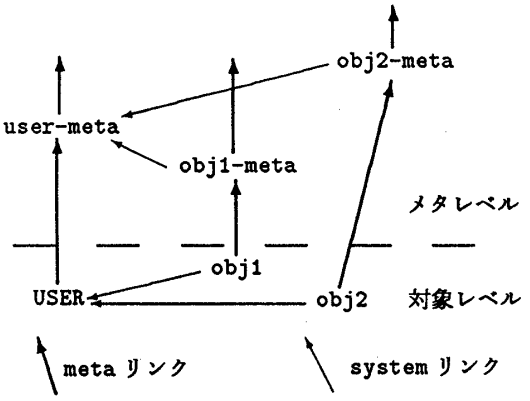

図1 フレームとメタフレームの関係

4. インタープリタ

対象レベルに対するインタープリタは、メタフレーム群とその中の関数定義を処理する実行器である。対象フレームの値を操作するような命令を受けると、実行器は次のような動作をする。

1. その命令がプリミティブなものならそれを実行する
2. 操作されるフレームが存在するなら、そのフレームからメタフレームを得て、関数定義を探す
3. フレームが存在しなかったり、定義がなかった場合は、USER-METAからデフォルト関数を探す
4. 3においてもなかった場合にはエラーメッセージを出力する
5. 定義が見つかった場合はそれを解釈、実行する

この中で使われる"定義を探す"ということは、メタフレームにアクセスすることになるので、メタ-メタフレームに記述してある手続きが呼ばれることになる。この手続きを取り出すにはさらにメタなフレームが必要になり、無限に続くことになるが、システムで用意したプリミティブを呼び出す場合には、メタを呼ばずにシステムが処理することで無限の探索を避ける。

5. 応用例

このようにして実現したリフレクティブなフレームシステムは、1で述べたような機能を実現している。1-1,1-2についてはメタフレームの関数定義を変更すれば良いし、1-4については対象となるフレームのメタフレームに、望みの関数を定義すれば良い。1-3に関しては、本システムでは対象レベルにおけるインヘリタンスはひとつも定義されていないので、利用者が全て定義することになる。その方法は、インヘリタンスをとるスロット名を決め、インヘリタンスを持たせたい関数を対象レベル全体の関数として、USER-METAフレームに定義すれば良い。例えば、指定のスロットの値がない場合はakoリンクをたどり、上位フレームの値を探す関数としてfget* を定義する。

```
fget*:
 (lambda (frame slot facet)
     (let ((value (p-fget frame slot facet)))
          (if value  value
              (if (fget frame 'ako 'val)
                  (fget* (fget frame 'ako 'val)
                         slot facet)
                  (error)))))
```

この定義をUSER-METAフレームに格納することでインヘリタンスが定義できる。

もうひとつの応用例を示す。デバグ問題として、あるフレームにどのような値が格納されたか、を表示するためのfputを定義してみる。

```
fput:
 (lambda (frame slot facet value)
     (p-fput frame slot facet value)
     (fprint-terminal
         "value ~S is added to ~S"
         value slot))
```

これをトレースしたいフレームのメタフレームに定義すれば、他のフレームに対しては通常の fput が実行され、定義したフレームに値が追加されると、メッセージが端末に表示される。この定義は従来のフレームシステムにある if-added といったデーモンでも記述できるが、この様なデーモンは対象領域についての知識を表現するべきものであり、デバグは異なるレベルに記述すべきである。

このように、トレース手続きのような従来プリミティブであったような部分もフレームシステムから定義できる。メタレベルにおいてインヘリタンスを導入したり、メタリンクを張り変えたりすることも可能であるので、より高度なフレームシステムを構築することができる。

6. まとめ

本研究では、フレームシステムの拡張性を高める手段としてリフレクションを利用し、メタレベルにおける手続きの修正能力を向上させた。これにより、低レベルなフレームシステムからより高度なフレームシステムを構築することができる。

また、デーモンやインヘリタンスを知識の記述のみに使用し、フレームシステムとしての機能拡張を区別して、メタレベルで記述することができる。

処理速度の低下はこのシステムの問題点である。この解決には、求めるフレームシステムが完成した時点で、メタレベルのコンパイルが必要だと思われる。

参考文献

[1] Smith B.C.: "Reflection and Semantics in Lisp",Proc. 11th ACM Symposium on Principles of Programing Language, 1984.

[2] Jaques Ferber : "Computational Reflection in Class Based Object Oriented Languages" , Proc. ACM Conference on Object-Oriented Programing: Systems, Languages and Applications, 1989.

[3] Pattie Maes : "Concepts and Experiments In Conmutational Reflection" , Proc. ACM Conference on Object-Oriented Programing: Systems, Languages and Applications, 1987.