概念制約式を用いたプログラミングを可能にするコンパイル手法*

1G－8

小林弘明† 高木浩光† 有田隆也† 川口喜三男† 曽和将容‡
†名古屋工業大学 電気情報工学科
‡電気通信大学大学院 情報システム学研究科

1 はじめに

知識モジュールやシステムが状況変化に対して柔軟に適応できるかどうかという、柔らかさと呼ばれる概念が注目を集めはじめている [1][2]。我々はプログラムを柔らかく記述する方法として概念制約式と呼ぶプログラム記述要素を提案している [3]。本稿では概念制約式を用いて表現されたプログラムから実行可能なプログラムを生成するコンパイル手法について述べる。

2 態度保留を含んだ要求仕様と概念制約

まず概念の代数の必要性について述べる。「プログラムを状況変化に対して適応させる」とは本質的にはプログラムに対する代数的変換であると考えられる。すなわち「元のプログラム±状況変化→結果プログラム」という変換である。したがってプログラムの最小構成要素のレベルで代数的変換が定義されていればこの変換に有利であると考えられる。この時「元の概念±差分」という記述を有効に利用するためには、元の概念に“態度保留”が多く含まれていることが望ましい。なぜなら元の概念の中で決められたことが多過ぎた場合、不適切な記述を除去するために概念上の引き算 (一度言ったことを覆す) を書く必要性が増大するからである。

次に概念の分類体系の必要性と問題点について述べる。一般的に人間が有用性を認める概念は大抵共通している。このため、オブジェクト指向プログラミングや ψ 項 [4] に見られるように概念の分類体系を用意し、継承機能を用いてこれを利用すれば、人間が一度に把握しなければならない問題は確かに減少する。しかし、本質的には、分類とは物事の性質に関する質問の集合が作り出す包含関係の半順序である。このため質問の集合を明示せずに分類を扱った場合、分類の要素どうしの関係が不明確になり、代数的変換に不利である。

このように態度保留を含んだ (概念) 記述どうしの合成演算と、質問を明示された概念分類体系との必要性は明確である。もちろん完全な分類体系を予め用意することを仮定することは非現実的である。しかし我々が直面する問題は既に分類体系と代数抜きでは対応しづらい巨大さになりつつある。したがって不完全なままで利用できるような代数つき分類体系を構築することが肝要である。

そこで実用性を得るために以下のような妥協を行なう。

(a) 既存のプログラムどうしを分類することを分類体系構築の目安とする。新たに書いたプログラムの性質記述が既存のプログラムの性質記述と同じになってしまう時のみ、分類体系を詳細化させる。分類体系の完全性についてはプログラマの余力に任せる。

(b) 質問に答えないことを許す。答え集合には真、偽に加えて「未回答 (未知の質問に直面した)」を意味する記号を含ませる。「未回答」は「偽に準ずるもの」として扱う。これにより重要でない問題に明示的に偽と答える必要をなくし、なおかつ明示的に偽と答えることも許す。

以上の考察に基づいて以下のように概念制約を定義する。

定義 1 予め定められた命題の集合 Q (問い集合と呼ぶ) と、それに対する答の集合 $A=\{T,*,u,F\}$ を考える。この時、全射写像 $C_q: Q \to A$ を概念制約と呼ぶ。概念制約の数式的な表現全般を概念制約式と呼ぶ。

ここで答集合 A 中の各記号の意味を明確にするために、必然性を表す様相記号□と可能性を表す様相記号◇ とを用いて表現すると、ブール代数における真と偽を t, f で表すとして、$T=\Box t$, $*=\Diamond t$, $u=\Diamond f$, $F=\Box f$ である。例えば 要求側が T, $*$を使った場合、それぞれ「必ずその能力が必要である」、「必ずしも必要ではない」を意味する。また供給側の場合は、「(するなと言われても) 必ずその仕事を達成してしまう」、「達成できるが、するなと言われたら、しないで済ませることができる」という意味になる。

概念制約上には通常のブール代数と同様の論理和 (＝単一化) や論理積 (＝一般化) のような演算が定義される。ただし記号 u と他の記号との間に演算が生じた時は、演算の結果として未知の概念が作られるので、そのような概念が妥当かどうかプログラマに対して問い合わせる。

以下概念制約式を四角括弧で囲って“[関数, 入力=[文字列]]”のように表現する。この例ではこの概念制約式の書かれた場所に実際に入るものが関数として機能する能力を持つものであることと、特に [入力] として [文字列] をとり得るものであることを表している。概念制約を定義する際には継承機構を利用する。例えば概念 “sort” を定義する必要が生じた時に、既に permutation (並べ換え全般) という概念が定義されていた場合、それを継承し、差分情報を付加する。

```
declare [sort]
 isa [permutation ,
      result.ordering=[value - dependent]];
end
```

3 概念制約の実現方法

3.1 概念制約の内部表現

概念制約の表現方法としては図1のようにベクタの index を個々の問いに対応させた表現 (問/答ベクタと呼ぶ) を用いる。

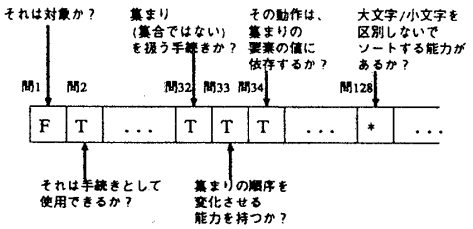

図 1: [ソート] を表す問/答ベクタ

問/答ベクタの全長は例えば数千要素以上にもなることが予想されるが、実際にはランレングス圧縮を利用できる

*A Compile Method which enables Programming with Concept Qualifier Expression
Hiroaki Kobayashi†, Hiromitsu Takagi†, Takaya Arita†, Kimio Kawaguchi†, Masahiro Sowa‡
†Nagoya Institute of Technology
‡University of Electro-Communications

こと、答が u になる所が多いこと、同じ答が連続しやすいように問の配置を並べ替えることが可能であることなどから、記憶容量の面で大きな問題になることはないと考える。

3.2 プログラムの内部表現

プログラムは問/答ベクタを頂点とするグラフとなる。グラフの辺に相当するものは問/答ベクタの表現する概念間の(フロー従属、等値制約などの)関係である。このプログラムグラフの辺を表現する方法としては隣接行列表現にマスク情報を付随させたものを用いる。マスクは概念どうしの相互作用を定義する際に、定義に態度保留を含ませるために使われる。例えば [分割. 並べ方] = [sort. 並べ方] と書いた時の "並べ方" がマスクに相当する。これにより問/答ベクタ上で等値制約の付加される index が制限される。

3.3 例データベースの構成と例の検索方法

概念制約式を用いて書かれたプログラムのことを例プログラム、もしくは単に『例』と呼ぶ。例は例データベース中に head (プログラム名と仮引数名に相当) と body (プログラム本体に相当) の 2 つに分けて格納される。

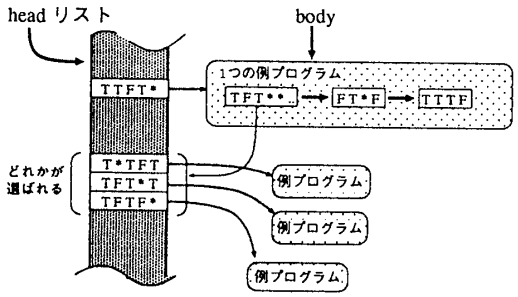

図 2: 例データベース

4 コンパイル手法

コンパイル手順を疑似コードを用いて記述すると、図 3 のようになる。

基本的には、要求を引数として受けとり要求を満たす例を探し出す作業 (展開と呼ぶ。関数 "FindExample()" に対応) を幅優先戦略に基づいて繰り返している。終了条件は、待ち行列中から要求記述がなくなることで、この時点でプログラムグラフが実コード断片 (C 言語などの他言語で書かれた終端要素) のみになっていれば、実行可能なプログラムを生成できる。

5 適用例

一般的なソートのプログラムはしばしばキーどうしの比較を行なう関数へのポインタを受けとるように作られる。このような一部機能の外在化を概念制約の上で表現すると図 4のようになる。

トップレベルから [ソート] の例プログラムへと与えられた要求のうち、並べ方に関するものは全て下位に対する要求の中へと埋め込み直される。この結果、もし既に適切な関数が例データベースの上に登録されていれば、それが選ばれることになる。

6 まとめ

概念制約の定義を従来より簡潔な形とし、その上で、概念制約の計算機上における表現方法と、概念制約の形で与えられた要求をもとにプログラムを生成する方法とを明らかにした。今後は以上の手法を実装、検証していく予定である。

```
/*start*/
EnQueue(initialRequirement);
while(QueueIsNotEmpty()){
 req = DeQueue();       /* 要求を 1 つ取り出す */
 ex = FindExample(req); /* 要求を満たす例を探す */
 diff = CqDifference(req, ex.head);
 /* 例の頭部と要求の概念制約の差分を計算する */
 for each link in FindLink(ex, diff) {
      /* 差分を元にリンクをたぐり… */
   PropagateParameters(link, ex.body);
      /* パラメータの伝播処理を行なう */
 }
 for each node in ex.body.node'list {
      /* 例の中の各ノードごとに、既定の順序で… */
   InsertToCodeGraph(node); /* コードグラフへ挿入 */
    if (nodeIsNotARealCodeFlagment(node)){
      /* 終端要素でないなら… */
      EnQueue(node);  /* 待ち行列にも入れる */
    }
 }
}
StoreCodeGraph(); /* 他言語ソースコードを生成 */
ForkCompiler();   /* 他言語コンパイラを起動 */
/*end*/
```

図 3: コンパイル手順の疑似コード

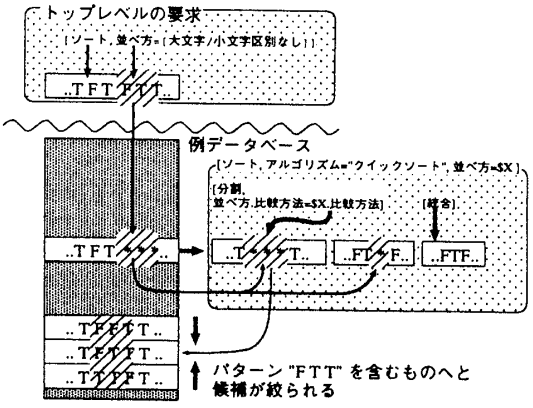

図 4: ソートの並べ方をパラメータ化した例

参考文献

[1] 蓬莱 尚幸: モジュール間協調に基づく軟らかい形式的仕様記述, 第 47 回 情報処理学会 全国大会論文集, 第 5 分冊,pp25-26,1993.

[2] 白鳥則郎, 菅原研次: やわらかいネットワークの開発に向けて — 知識型設計方法論 — , 電子情報通信学会 技術研究報告,AI93-46, 1993.

[3] 小林、有田、川口、曽和: 概念制約式を用いたプログラミングとプログラム合成, 第 47 回 情報処理学会 全国大会論文集, 第 5 分冊,pp19-20,1993.

[4] Hassan Aït Kaci: An Algebraic Semantics Approach to the Effective Resolution of Type Equations, Theoretical Computer Science 45,pp293–351,1986.