逐次型推論マシンCHIにおける動的述語の実現と評価

7Q-4

新　　淳、小長谷 明彦
日本電気(株) C&Cシステム研究所

1　はじめに

第5世代コンピュータ研究開発プロジェクトの一環として、逐次型推論マシンCHI[5]、およびその小型化版[3]を研究開発してきた(以下、CHI小型化版を単にCHIと呼ぶ)。CHIの開発ではコンパイル時に定義が定まる述語(静的述語)の高速実行に加え、assert/retractにより実行時に定義が変化する述語(動的述語)についても高速実行することを目指している。これは応用プログラムでは動的述語の利用が不可欠であり、応用プログラムのトータルとしての性能の向上のためには動的述語の高速実行が重要であるからである。このため、我々は、クローズをassertする際に直接実行可能な機械語列に展開する動的コンパイル方式を提案した[4]。

以下、本論文ではこの動的コンパイル方式のCHI上での実現方法と評価について述べる。

2　動的コンパイル方式

CHIで採用した動的コンパイル方式では、クローズがassertされる毎に、これを命令列にコンパイルし、得られたコード(これをダイナミックコードと呼ぶ)を所定のアトム / ファンクタのチェインにつなぐ。また、クローズがretractされる時には対応するダイナミックコードをこのチェインから外す。

動的述語の実現には、単にクローズを命令列に展開するだけでは不十分であり、clauseやretractを実現するために、登録された時のクローズイメージを保持しておく必要がある。この実現方法にはいくつかの方式が考えられる[2, 1, 4]。

1. クローズイメージを保持
2. 実行コードから逆コンパイル
3. クローズイメージを取り出す命令列を保持
4. 複合機能命令を導入して実行コードとクローズイメージ取り出しコードを兼用

CHIでは、これらの方式の比較検討を行なった結果、実行時の性能が高速であるという観点から、(3)の方式について実装 / 評価を行なった。

3　ダイナミックコード

1つのクローズを翻訳して生成するダイナミックコードは、図1に示したような構造をしている。以下に、特徴的な要素について説明する。

インデキシング情報
このクローズの頭部ゴールの第1引数の情報を保持する。これによって、述語呼びだし時に成功する可能性のあるクローズを絞り込むことができる。

エントリ命令
このクローズがチェインの最初に登録されていた場合で、このクローズを含む述語が呼び出された時に最初に実行する命令である。

再実行命令
このクローズの実行に失敗したときに実行する命令である。CHIでは、動的述語のクローズCの代替クローズを、Cの実行に失敗した時にはじめて探すようにしたため、再実行命令をここにおいた。

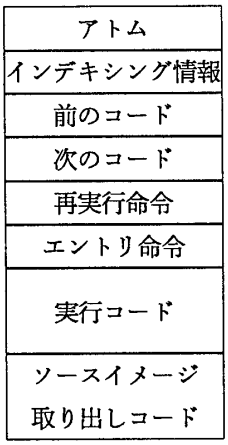

図1: ダイナミックコードの構造

4　アーキテクチャサポート

CHIでは、動的コンパイル方式を支援するために、次のようなアーキテクチャサポートを行なっている。

4.1　選択点制御命令

チェインでつながれた複数のダイナミックコードの中から、実行すべきダイナミックコードを選びだし、選択点の制御を行なう命令であり、icc_try、icc_retryの2つの命令を用意している。これらの命令は、3で述べたエントリ命令、再実行命令に各々対応し、WAMの選択点制御命令であるtry_me_else、retry_me_else、trust_me_else_failの拡張であり、次のような特徴を持つ。

モード切替え
制御レジスタの保持しているモードを判断し、実行コード / クローズイメージ取り出しコードのいずれかに分岐する。

第1引数によるインデキシング
第1引数の保持しているデータを基に、インデキシング情報がマッチするダイナミックコードを探して実行する。

4.2　クローズイメージの取り出し

assertしたクローズのイメージを取り出すための述語clauseを実現するための命令としてcall_clause、execute_clauseの2つの命令を用意している。これらの命令は、制御レジスタのモードをクローズイメージ取り出しモードに切り替え、第1引数の保持しているアトム / ファンクタに登録されているコードを実行する。

Implementation and evaluation of dynamic predicates on sequential inference machine: CHI
Atsushi Atarashi, Akihiko Konagaya, NEC Corporation

5 評価

5.1 項形式解釈法との比較

動的述語の評価して、動的コンパイル方式と、インタプリタによってグローバルスタック上の項形式を解釈する従来方式との比較を行なった。項形式を解釈する方式では、クローズを登録する際、このグローバルスタック上のクローズイメージをヒープ上のベクタ形式に変換してアトムに登録し、述語を呼び出す際にはアトムに登録されたベクタをグローバルスタック上のクローズイメージに展開して解釈して行く。

assert の性能
assert は項形式のヒープ上のベクタへの変換の 10 倍の時間を必要とし、assert の生成するコードは、項形式を変換したヒープ上のベクタの 1.7 倍のメモリを消費する。

実行性能
nrev/3 について、動的コンパイルを行なった述語の実行速度を、項形式解釈インタプリタ方式の実行速度と比べると、50 倍の速度で実行できる。また、メモリ使用量も 1/10 で済む。

これらの結果から、動的コンパイル方式では assert の速度が遅い点が問題となるが、クローズの登録の頻度はクローズの呼び出しの頻度に比べて低いことを考えると、assert の速度が実際の応用プログラムの実行に影響を与えることはないと考えられる。

5.2 静的述語との速度の比較

動的述語と静的述語との実行速度の比較について述べる。ここでは、ECRC のベンチマークプログラムの一部を用い、最適化コンパイルした場合の速度と、動的コンパイルした場合の速度とを計測した (表 1、なお表中で、S で示した欄が最適化コンパイルしたコードを実行した結果、D で示した欄が動的述語として実行した結果であり、単位は KLIPS である。マシンクロックは 200nS にて計測した)。また、汎用計算機上ではもっとも高速な処理系の 1 つである SUN-4/280 上の Quintus Prolog(v2.4)[6] において、同様の条件で計測した結果も示す。

これらの 2 つの処理系における動的述語のインプリメントには、次のような差がある。

クローズの assert 方式
Quintus の方式は不明であるが、CHI では動的コンパイル方式を用いている。

クローズの失敗時の代替クローズの探索方法
Quintus では、CHI の方法に比べて選択点の操作が少なくて済むような、述語が呼び出された時点で有効なクローズのみを代替クローズとして持つような方式を採用している。

また、どちらの処理系も第 1 引数によるインデキシングを行なっている。

これらのベンチマークのうち、実際の応用プログラムでの動的述語の使われ方に近い述語を含んだプログラム (例えば query) における静的述語に対する動的述語の速度の比率は Quintus では約 10% にとどまっている。これに対して CHI ではこの比率が約 40% であり、CHI の代替クローズの探索方法を Quintus と同じにした場合はこの比率は数 % 向上する。また、現在 CHI では assert の速度低下を避けるために行なっていない動的述語のコードに対する最適化を施すことによって実行速度向上の余地も残されており、CHI で採用した動的コンパイル方式は動的述語の高速な実行に有効であり、応用プログラムでの動的述語の使用が全体の性能に影響することは少ないことがわかる。

表 1: ECRC ベンチマークプログラムの実行結果

プログラム	CHI			Quintus(SUN/4-280)		
	S	D	D/S	S	D	D/S
boresea	2487	61	0.02	630.9	3.21	0.01
cpoint	333	61	0.18	120.5	6.86	0.06
baktrak1	156	29	0.19	150.4	6.31	0.04
baktrak2	200	105	0.52	200.0	23.09	0.12
envir	194	34	0.18	200.0	1.41	0.01
index	64	22	0.34	40.6	5.89	0.15
fibonacci	147	67	0.46	89.7	4.22	0.05
map	85	33	0.39	84.9	6.66	0.08
mham	92	29	0.32	76.3	6.43	0.08
mutest	92	57	0.62	68.0	5.73	0.08
qs	122	38	0.31	66.1	4.29	0.06
qu	153	55	0.36	99.6	5.20	0.05
query	73	27	0.37	46.3	2.65	0.06
differen	54	22	0.41	38.7	4.10	0.11
diff	108	37	0.34	54.9	2.61	0.05

6 まとめ

CHI の動的述語の実現とその評価について述べた。クローズを assert する際に動的にコンパイルすることによって、動的述語を静的述語の 30%-40% 程度の速度で実行できることがわかった。assert の速度が遅いという問題があるが、assert の頻度が動的述語の呼び出しの頻度に比べて低く、実用的な応用プログラムに影響を与えることはないと考えられる。

今後は、応用プログラムにおいて動的述語がどのように使用されるかを正確に評価した上で、理想的な動的コンパイル方式を追求して行く予定である。

謝辞 本研究に対して有益なコメントを頂いた日本電気(株)C&C システム研究所の大野部長、横田課長、幅田主任に感謝致します。

参考文献

[1] Buettner, K.A., "Fast Decompilation of Compiled Prolog Clauses", in *Proc. the Third International Conference on Logic Programming*, New York, 1987

[2] Clocksin, W.F., "Implementation Techinquies for Prolog Databases", Software — Practice and Experience, Vol. 15(7), 1985

[3] Habata, S., Nazakaki, R., Konagaya, A., Atarashi, A. and Umemura, M., "Co-operative High Performance Sequential Inference Machine: CHI", in *Proc. ICCD'87*, New York, 1987

[4] 小長谷 明彦、高速 Prolog インタプリタの構築法とその評価について、記号処理研究会 46-4, 1988

[5] Nakazaki, R., Konagaya, A., Habata, S., Shimazu, H., Umemura, M., Yamamoto, M., Yokota, M. and Chikayama, T., "Design of a High-speed Prolog Machine (HPM)", in *Proc. of the 12th International Symposium on Computer Architecture*, 1985

[6] Quintus Computer Systems, Inc., "Quintus Prolog Reference Manual (Version 10)",1987

[7] Warren, D.H.D, "AN ABSTRACT PROLOG INSTRUCTION SET", Technical Note 309, SRI International, 1983