プログラミング言語 10－4
（1987. 2. 13）

C－Prologコンパイラの最適化

小林 茂，松本憲幸，落合正雄，本位田真一
(株) 東芝 府中工場

スーパミニコンピュータG8050上に、C－Prolog用コンパイラを開発した。本コンパイラはC－Prologインタプリタのサブシステムとして機能し、ソースファイルより、オブジェクトプログラムをメモリ上に生成する。中間コードにはWarrenによって提案された命令セットをベースとして用いた。
オブジェクトコードの実行速度を最適化するために、Warrenの命令セットに対し、細分化や新命令の追加などの改良を加えた。クローズ内における最適化だけでなく、引数レジスタセーブの遅延やグローバル変数の採用など、クローズ間あるいは述語間にまたがる最適化も試みた。

"The Optimization of C-Prolog Compiler" (in Japanese)
WGPL 10-4

by S. Kobayashi, N. Matsumoto, M. Ochiai, S. Honiden
Fuchu Works, TOSHIBA Corporation, 1, Toshiba-cho, Fuchu City

We developed a C-Prolog compiler on G8050, a super-mini computer. This compiler works as a subsystem of the C-Prolog interpreter. It generates object programs on main memory from source files. It uses instruction set suggested by D.H. Warren as the base of intermediate code.
To optimize the efficiency of the object code, we broke down and added new instructions to Warren's instruction set. Not only clause-local optimizations, we also examined inter-clause or inter-predicate wided optimizations such as delaying of saving argument registers or adopting global variables.

1. はじめに

スーパミニコンピュータG8050上にC-Prolog用コンパイラを開発した。本コンパイラは最終的にはマシンコードによるオブジェクトを生成するが、中間コードとして、Warrenによって提案された抽象Prolog命令セット（An Abstract Prolog Instruction Set：以下、APISと略す。）を用いている。最適化の過程では、この命令セットに対し、細分化・新命令の追加等の改良を加えた。

以下に、本コンパイラの概要と、我々が採用した最適化の手法について述べる。

2. コンパイラの概要

コンパイラはC-Prologインタプリタのサブシステムとして組込まれている。その操作性やコンパイルされた述語（オブジェクト述語と呼ぶ。これに対し、インタプリタにより実行される述語をソース述語と呼ぶ。）の動作については、インタプリタとの整合性を保つよう考慮した。

コンパイラの起動は、組込述語"compile"により行なう。通常、コンパイル対象はファイルである（述語を指定することもある）。

compile（ファイル名）

compile述語は指定されたソースファイルの内容をコンパイルし、オブジェクト述語定義をインタプリタ内に生成する。

コンパイラは次の4つのフェーズより成る。

（1）指定されたソースファイルをその時点における インタプリタ環境に従って解析する。
（2）解析結果ファイルより、中間コードファイルを生成する。
（3）中間コードより、マシンコードのオブジェクトファイルを生成する。
（4）オブジェクトファイルを読み、オブジェクト述語を定義する。

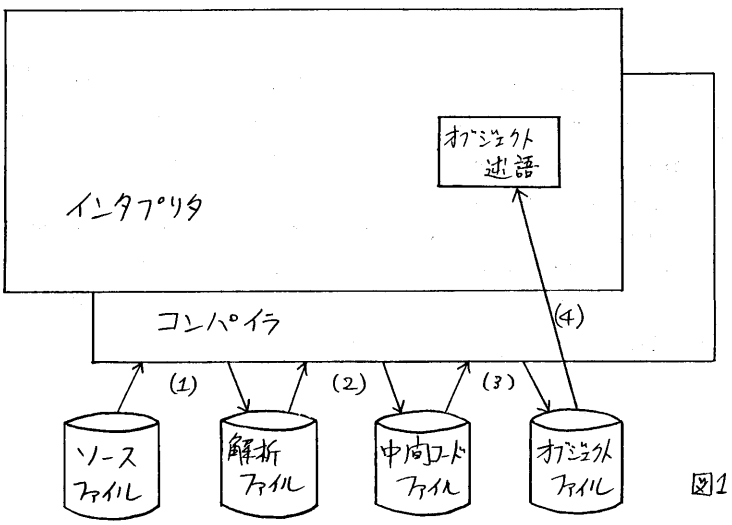


図1. コンパイラの処理の流れ

述語定義は、内部的にはファンクタと呼ぶ構造体により表わされる。その構造を図2に示す。

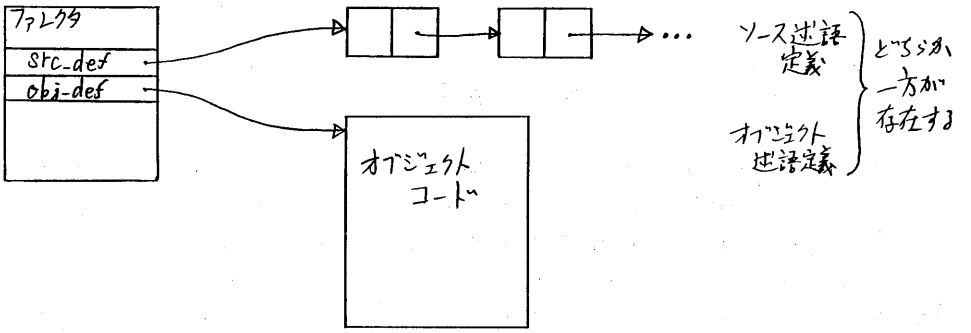


図2、ファンクタの内部構造

オブジェクト述語間での呼出しは、ファンクタの obj-def フィールドを通しての間接ジャンプにより行なわれる。呼ばれた述語だった場合、そのファンクタの obj-def フィールドにはオブジェクト述語からソース述語を呼び出すためのインタフェースルーチンがリンクされているので、このルーチンによりインタプリタに制御が移される。また、インタプリタがソース述語を実行している途中で src-def フィールドに定義を持たない述語が呼ばれた場合、その逆の働きをするルーチンがインタプリタから呼ばれる。このようにして、オブジェクト述語とソース述語の相互呼出しが可能になる。また、オブジェクト述語が再定義された場合も、ソース述語の場合と同様に、他からの呼出しは新しい定義を参照できる。

3、中間コード

コンパイラが使用する中間コードはAPISをベースとしている。APISは次の6種類の命令より成る。

(1) INDEXING系命令
引数の型による候補クローズの選択，代替クローズの実行等を行なう。

(2) CONTROL系命令
述語の呼出し・復帰，変数領域の割当て・解放を行なう。

(3) GET 系命令
クローズのヘッドにおいて、引数レジスタより引数を取出す処理を行なう。このとき同時に、ユニフィケーションも行なう。

(4) PUT 系命令
クローズのボディにおいて、引数レジスタに引数を渡す処理を行なう。

(5) UNIFY系命令

GET・PUT系命令において引数が構造体・リストであった場合、その読出し・生成を行なう。

(6) その他

バックトラック等の処理を行なう。

APISはprologプログラムの構造と動作を内状を反映しており、多くのprologコンパイラ開発において採用されている。しかし、この特長は、インタプリタ的な実行のしかたを(機能毎と各命令に分割してはいるが)そのまま残していると見ることもでき、実際のインプリメントでは対象マシンに応じて改善すべき余地が残されている。今回の開発でも、APISに対してかなり自由な追加・変更を行なった。

4. 最適化

4.1 最適化の方法

prologにおける最適化の方法としては、次のようなものが挙げられる。

(1) バックトラック発生の最小化

prologプログラム実行の高速化のためには、バックトラックの発生を少なくすることの効果が大きい。そのためと、試される代替クローズを限定し、もう1つは、失敗するクローズについて、早い時期にこれを検出する。代替クローズの限定については、APISでは既にINDEXING系の命令によりある程度行なっている。

(2) データ構造操作の簡略化

prologではユニフィケーションやバックトラックという特有な操作のため、変数値の多段階のインタによる間接参照など、操作負荷の大きな内部データを用いている。また、引数渡しにおいて、構造体・リストの読み出しや生成を要する操作が多い。これらについて、不要な処理の省略や効率化を行なう。

(3) 一般的最適化

冗長な処理の除去、共通な処理のくくり出しなど、一般的に用いられる最適化を行なう。

今回は主に(3)に属する最適化を行なった。(1),(2)の最適化は、例えば代替クローズの限定のためにはインデキシングに最適な引数、というように、ユーザにより補助的な宣言を与えられた場合に効果が大きいと思われるが、どのような宣言を用いるのが繁雑でなくかつ有効か、今後の検討を要する。

次節以下では、今回行なった最適化の手法について個々に説明する。

4.2 述語のローカル呼出し

先述のように、オブジェクト述語は、通常はファンクタを介して間接呼出しされる。ファンクタはさらに、オブジェクトコードからリテラルテーブルを通して参照されるので、2重の間接参照になる。これを効率化するために、"local" 述語によってそのファイル内からのみ呼ばれることを宣言された述語は、直接に呼出すようにした。ローカル述語については呼出し側では再定義について考慮する必要がないので、これが可能になる。

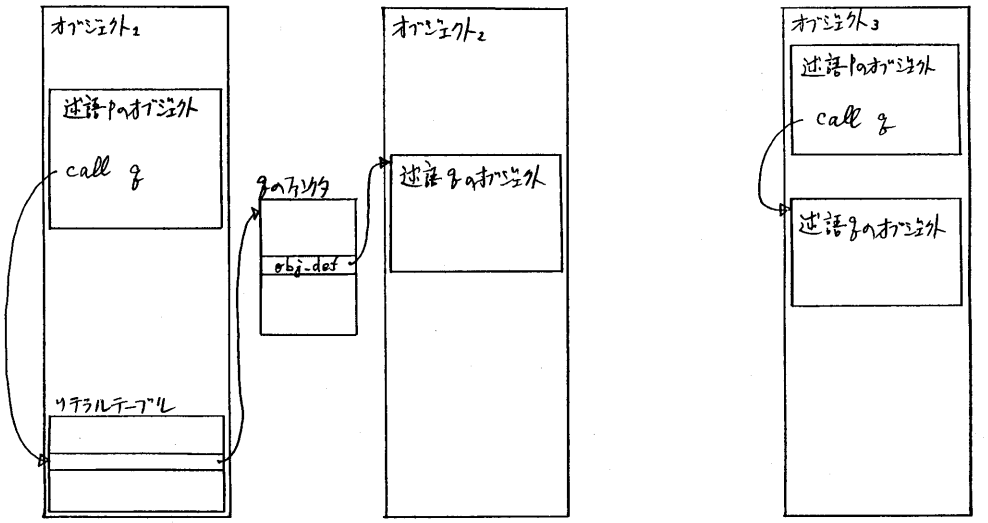


図3、ローカル述語呼出し

4.3 引数のグローバル変数化

pro/og には述語間で共通に参照されるような "グローバル変数" はない。そのため、直接の呼出し関係にない述語間でデータを受渡すために、引数を次のように用いる場合がある。

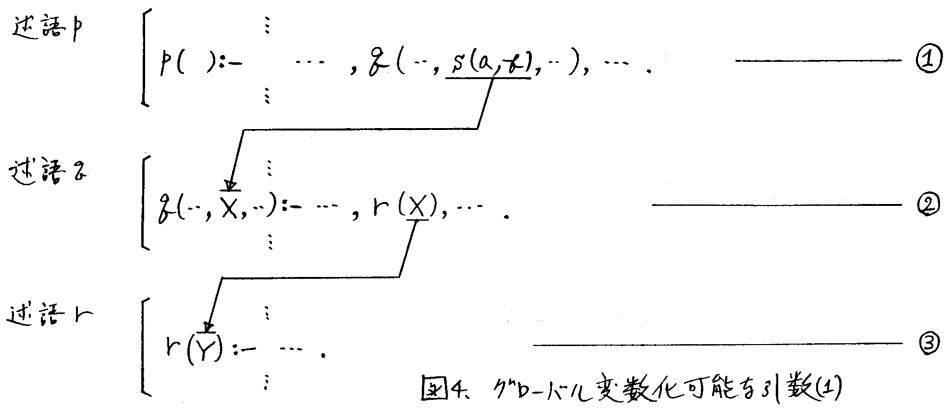


図4、グローバル変数化可能な引数(1)

この場合、クローズ②の変数Xは、述語pからrへ、引数 s(a,x) を渡すためだけに使われている。そこで、クローズ①において s(a,x) をグローバル変数にPUTし、クローズ③では引数のかわりにこのグローバル変数をGETするようにすれば、クローズ②における GET・PUT を省略できる。(図5)

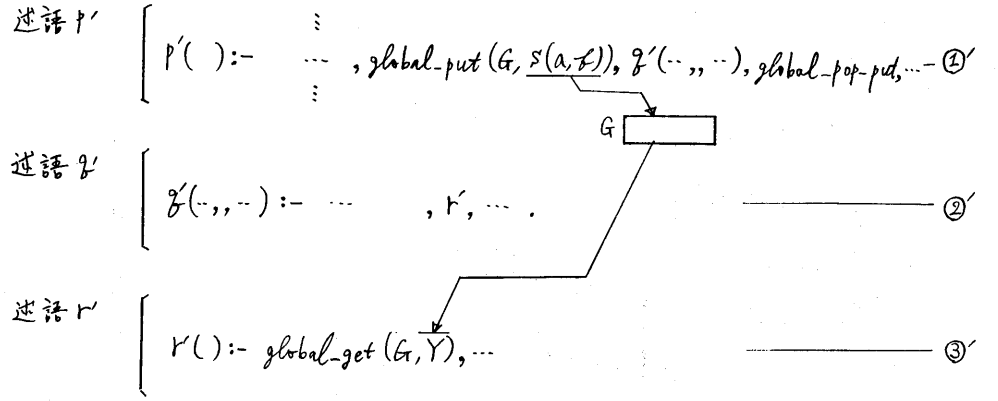


図5. グローバル変数化された引数(1)

ただし、このような中介となる述語の引数を省略すると呼ばれる述語(この例ではqとr)の引数の個数が変わるわけであるから、これらはローカル述語でなければならない。

ところで引数をグローバル変数化するとき、中介となっている述語の数が多いほど、その効果が大きい。しかし一般には、このような呼出しが深くなることはあまり期待できない。そのため、現在のところ、本手法の適用は受けとった引数をそのまま同一引数に渡すような、引数の単純受渡しを含む直接的な再帰呼出しを行なっている述語(図6)に限っている。

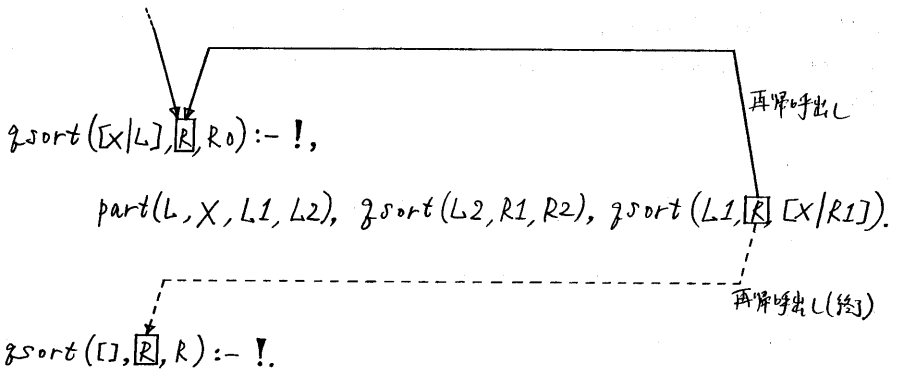


図6. グローバル変数化可能な引数(2)

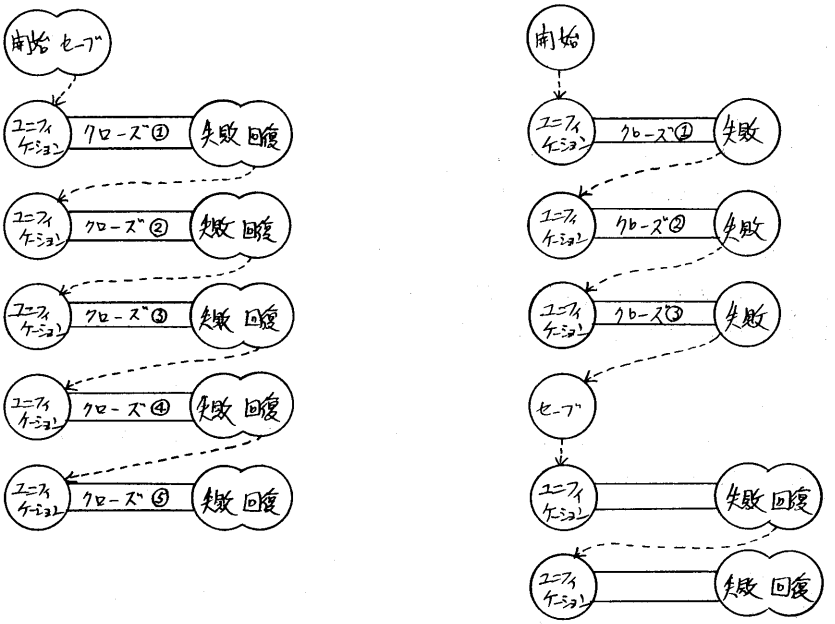
4.4 引数レジスタセーブの遅延

述語の中には、次の例のように、始めにいくつかの、ヘッダのみから成るクローズを持つものがある。

```
age(taro, 29).                                  ——①
age(jiro, 21).                                  ——②
age(saburo, 18).                                ——③
age(X, AGE) :- age2(X, AGE).                    ——④
age(X, AGE) :- write('? '), read(AGE), dep_age2(X, AGE). -⑤
```

図7. ヘッダのみのクローズを持つ述語の例

この場合、クローズ①～③の間、ボディの実行による破壊が生じないので、クローズ①～④では、受取った引数が保存されている。しかしAPIでは先頭クローズの実行開始時に引数レジスタをセーブし、また、ユニフィケーションの失敗毎にセーブした値をレジスタに回復する。そこで、レジスタのセーブを先頭クローズの開始命令から、一方、レジスタの回復を失敗処理命令から独立させて1つの命令とすることにより、余分なレジスタの回復処理を省略できる。



APIによる処理の流れ　　セーブ遅延時の処理の流れ

図8. 引数レジスタセーブの遅延によるレジスタ回復処理の効率化

4.5 変数を含まない構造体のリテラル化

APISでは、構造体（およびリスト）の受渡しに、コピー方式を用いている。従って、構造体型の引数については必ず、そのデータ型と引数が1つずつ、スタックに書込みまたは、スタックから読出しされる。しかし、PUTの場合および出力用としてモード宣言された引数のGETの場合は、構造体中に変数が含まれなければ予めその構造体を作っておき、これを定数的にPUTまたはGETしてよい。

この処理を行なうために、専用のPUTおよびGET命令を追加した。

5. おわりに

C-Prolog用コンパイラの開発に関し、処理系の概要と最適化手法について述べた。今回行なった最適化は、APISに対する専用命令の追加や機能による組合化等の修正を加え、本来のAPISコードに含まれる冗長な処理を除去するというのが主なものであった。この種の最適化の可能性は、今後も継続して検討して行く予定である。一方、述語の特性に関する宣言としては、今回は"local", "mode", "dynamic"（述語の内容が実行時に変更されることを宣言する）のみを実現した。プログラムの実行に関する特性がユーザによって補足されれば、コンパイラにとって、最適化のうえで大きな助けとなる。しかしあまりに多様な宣言が存在するのは操作性の面から問題があり、その仕様に関しても、今後の検討が必要である。

参考文献

D.H.D. Warren
"An Abstract Prolog Instruction Set"
SRI International Technical Note 1983

小林 他
C-prologコンパイラの開発（1），（2）
情報処理学会第33回全国大会予稿集 1986

浅川 他
複数のアーキテクチャをターゲットとした高速Prologコンパイラ
記号処理 37-3 1986.6

花崎 他
C-Prolog上でのインタプリタとコンパイラの共存法について
知識工学と人工知能 45-7 1986.3