6E−1

Common ESP 機械語生成系の自動生成
−高移植性を目指したMD方式の概要−

佐藤良治　高橋文男　実近憲昭
(株) AI言語研究所

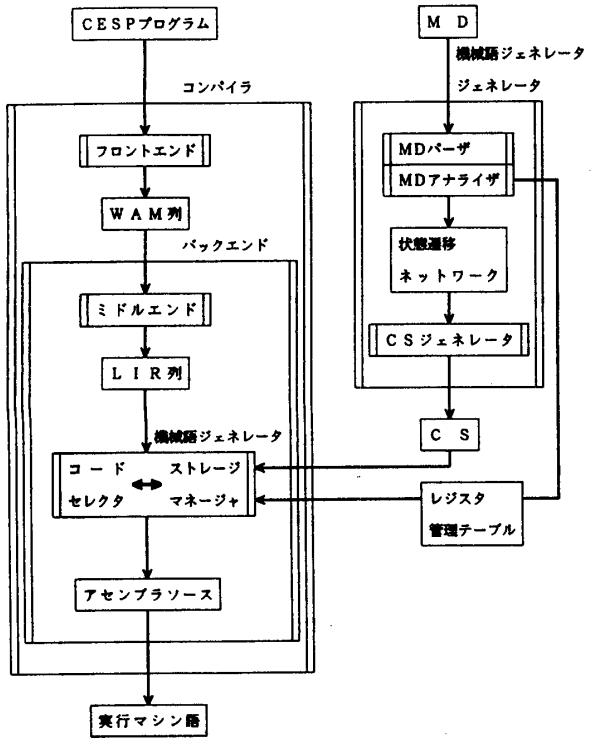
1　はじめに

Common ESP(CESP)は、ICOTで開発された言語ESPを基に、汎用マシン上での稼働を主要な目的として開発・機能拡張されたAIシステム記述言語である。

我々は、高い実行性能を実現するために、実行マシンの機械語を直接生成する「機械語生成コンパイル方式」を採用した。さらに、「多くの汎用マシンに容易に移植できるシステム」という相矛盾する目標をもクリアするために、機械語生成コンパイラシステムを自動生成する「機械語ジェネレータジェネレータ」を開発した。

本論文では、機械語ジェネレータジェネレータの入力情報となるMD(マシン記述ファイル)について報告する。

2　コンパイラの処理概要



CESPコンパイラは、CESPプログラムを解析してCESP WAMに落すフロントエンドと、CESP WAMを機械語に変換するバックエンドとからなる。バックエンドは、さらにWAMをより低レベルな中間表現であるLIR(Low-level Intermediate Representation)に展開する機種独立部(ミドルエンド)と、LIRに対応する機械語(アセンブラソース)を生成する機種依存部(機械語ジェネレータ)とからなる。

機械語ジェネレータジェネレータ(CGG:Code Generator Generator)は、MDを入力として、コードセレクタ、およびレジスタ管理テーブルを生成する。これらが、機械語ジェネレータ(CG:Code Generator)の主要要素となる。

3　高移植性のための手法

機種依存の機械語を直接生成するシステムに、移植性を持たせるために以下のような手法を用いた。

- 機種独立な低レベル中間表現の導入
- 簡便なマシン情報記述からの機械語ジェネレータの自動生成

本システムは、CESP WAMから機械語を生成する過程に、LIRという中間表現を新たに導入した。これは、機種に依存しない、言わば抽象的アセンブラ言語のようなものである。そして、できる限り、この段階で最適化等の処理を行うことにより、機種に依存する処理の極小化を図った。LIRは、20の命令、6単項演算子、11二項演算子、10関係演算子、および8種類のオペランドからなる。

CGは、LIRの列を機種依存のアセンブラ命令にマッピングするパターンマッチャである。CGは、アセンブラコードの選択マッピングを行うコードセレクタ、レジスタ・メモリの記憶管理を行うストレージマネージャ、その他のアクションルーチンからなる。このCGがアセンブラソースを生成するのに必要な情報を、できるだけ簡単に設定させるために用いたのがMD(マシン記述ファイル)である。CGの機種依存部分となるコードセレクタ、レジスタ管理テーブル等は、MD情報からCGGが自動生成する。

4　MD(マシン記述ファイル)

[MD記述項目]

%%Register	ハードウエアレジスタ情報
%Register_class	レジスタクラス，レジスタ印字表現
%Static_assignment	静的割り当て指定
%Scratch	動的割り当て指定
%Return_value	リターン値レジスタ指定
%%Mapping	アセンブラ構文にマッピングするための情報
%Asm_prologue	アセンブラソース先頭部出力テキスト
%Asm_epilogue	アセンブラソース末尾部出力テキスト
%Alignment_specification	コード、データ桁揃え指示
%Ext_name_specification	外部名参照、定義構文
%Label_specification	ラベル参照、定義構文
%Relative_address_specification	相対アドレス表現
%Constant_specification	定数表現
%Comment_specification	コメント構文
%Data_specification	データ領域宣言
%Declaration_specification	データ部、グローバル宣言
%Token_group	ユーザ定義集合トークンの指定
%Function	ユーザ定義のC関数の記述
%Transfer_rule	レジスタ・メモリ間の転記規則を記述
%Mapping_rule	LIRパターンとアセンブラ命令の対応を記述

Common ESP Machine Code Generator Generator
- An Overview of Machine Description Method for High Retargetability -
Yoshiharu Sato, Fumio Takahashi, Noriaki Sanechika, AI Language Research Institute, Ltd.

```
%%Optimization   最適化情報

%%Control        システムインタフェース指定
%Function        アセンブラ、ローダ起動関数
```

MDは、四つのセクションからなっている。(1)レジスタの情報、(2)アセンブラ構文にマッピングするための情報、(3)最適化情報、そして(4)システムが生成アセンブラソースから実行ファイルを作る時に用いる情報である。今回は(2)の主要機能である%Mapping_ruleと%Transfer_ruleを中心に説明する。

%Mapping_rule

ここには、LIRパターンに対応するアセンブラ命令を選択するためのマッピングルールを定義する。

```
[マッピングルールの例1]
:: stm_goto label_ref {Emit("jra $2");}
```

これは無条件分岐パターンの例である。stm_gotoやlabel_refは、LIRの一つであり、LIRトークンと呼ばれる。{・・・}内はアクション部と呼ばれ、パターンがマッチした時に実行される。Emitは、出力するアセンブラ命令を作成し、内部バッファに文字列を出力する関数である。文字列内の$2は、当ルールの2番目のLIRトークンlabel_refを指し、ラベル参照アセンブラ表記が自動生成されて、文字列にセットされる。

ラベルを参照する場合のアセンブラ構文については、%Label_specificationに定義する。

```
(例)  %Label_specification
          REF   "L_$.labno"
          DEF   "L_$.labno:\n"
```

このようにして、例1の場合、機種独立なLIRパターン(stm_goto label_ref)から、機種依存のアセンブラ命令(例えば、"jra L_000003")が自動生成される。

```
[マッピングルールの例2]
:: stm_simp_assign
    const^Eq(0) Oprd^OnReg(A)
                    {Emit("movl  0, $3");}
:: stm_simp_assign
    const^Eq(0) Oprd
                    {Emit("clrl  $3");}
```

これは単純代入パターンの例である。LIRのマッピング・パターンの各要素は、トークンと呼ばれ、LIRトークンとマクロの二種類がある。さらにマクロは、機能マクロと集合マクロに分かれる。機能マクロとは、下記の例における Same($4) であり、4番目のトークン Oprd と同じトークンであることを示す。

```
(例) :: stm_offset_ld
          Oprd^OnReg(D) const Oprd
        stm_offset_st
          Same($4) Oprd^OnReg(D) const
        {Emit("movl @($3, $2:1), @($8, $7:1)");}
```

また集合マクロには、システム提供のものと、ユーザ定義のものがある。システム提供のものとしては、任意の単項演算子を意味するUop、同様に、二項演算子用のBop、関係演算子用のRop、オペランド用のOprdがある。ユーザ定義のものは、%Token_groupに以下のように定義する。

```
(例)  %Token_group
          AddSub  : bop_add bop_sub
          Logical : bop_and bop_or bop_xor
```

例えば、加減・シフト・論理演算・比較等は，命令選択条件や生成パターンがそれぞれ似ていることが多い。このような場合に、ユーザ定義集合マクロを用いると、何度も同じ記述を繰り返さずに済む。抽象化して一つのパターン定義にできるわけである。

例2のトークン・パターンにあるEq(0)やOnReg(A)は、制約と呼ばれる。これを用いて、各トークンには，より厳しいマッチング条件を指定できる。この例のEq(0)はオペランドの値が0に等しい、OnReg(A)はオペランドの値が，レジスタクラスAのレジスタ上に有る、という制約である。

ルールのマッチングは、トークンの数の多いものから行われる。例2の場合、二つのルールが書かれており、両方共トークン数は3個である。この場合、MDの記載順にマッチングが行われる。まず始めのルールについて、各トークンのマッチングおよび制約のチェックが行われ、成立したらアクション部の実行が、不成立の時には次のルールのマッチングが行われる。

%Transfer_rule

アセンブラ命令のオペランドは、最新値が乗ったレジスタでなければならないことが多い。そのためにレジスタ上の最新値を保証する必要が有り、また、そのためのロード処理が多く発生する。このように類似パターンで頻繁に発生するレジスタ・メモリ間の転記処理を効率良く記述できるようにするために次のような仕組みを考えた。

まず%Transfer_ruleというサブセクションを設け、そこに転記処理タイプ毎のアセンブラ命令を定義させる。そしてマッピングルールでは、必要に応じてこれらの転記命令を自動的に挿入できるようにした。

```
[転記規則の定義例]
%Transfer_rule
::  ST    Oprd {Emit("movl $.reg, $.sym");}
::  LD    Oprd {Emit("movl $.sym, $.reg");}
::  LA    Oprd {Emit("lea  $.sym, $.reg");}
```

転記処理タイプには以下のものがある。

```
LD : メモリの内容をレジスタにロードする
LA : メモリのアドレスをレジスタにロードする
LI : 即値をレジスタにロードする
ST : レジスタの内容をメモリにストアする
MV : レジスタの内容を別のレジスタにコピーする
```

```
[転記規則の呼び出し例]
:: stm_offset_st const^Eq(0)
   Oprd^OnReg(A)?{}:{ForceReg(A,LD)} const
                {Emit("clrl $3@($4)");}
```

この例の3番目のトークンOprdについて見ると、これは、当オペランドがレジスタクラスAのレジスタ上に有るかどうかをチェックして(^OnReg(A))、有れば、転記処理としては何もせず(?{})、無ければ、強制的にレジスタクラスAのレジスタを獲得してその値を獲得したレジスタにロードする(:{ForceReg(A,LD)})という指定になっている。この処理が、アクション部の実行に先立って行われる。

このif-then-else型の導入により、制約の成立・不成立時の処理のみが異なる関連マッピングルールを一つに記述できるようになった。

以上見てきたように、(1)集合マクロ、(2)転記処理の一括定義、(3)転記処理のif-then-else型呼び出し、等を導入することにより、マッピングルールにおける、記述量の削減、関連ロジックの一括管理による可読性・保守性の向上を計ることができた。

5 今後の予定

CESPを移植するマシンが増えるに従って、本移植システムも機能アップが必要となるであろう。そのニーズに従って、MDの記述内容も改善・追加を行っていく予定である。

6 参考文献

[1] 佐藤ほか"Common ESP 機械語ジェネレータの概要"情報処理学会第40回全国大会(1990)

[2] 高橋ほか"CESP 第2.0版 マシン記述ファイル(MD)仕様書"AI言語研究所内部資料(1990)