平成 5 年度

筑波大学第三学群情報学類

卒業研究論文

題目 並列記号処理言語のデータ並列言語による実装

主 専 攻 情報科学

著 者 名 古川英司

指導教員 電子・情報工学系　田中二郎

要 旨

超並列計算機上のプログラミングは、プログラマにとって困難である場合が多く、ハードウェアの性能を活かしたソフトウェアの開発が遅れている。本論文では、プログラマにとってソフトウェアの開発が容易な並列記号処理言語を、ハードウェアの性能を活かすデータ並列言語で実装する方法について述べる。

目次

第 1 章

序論

最近、 Thinking Machies の CM-5 や 富士通の AP1000、 CRAY の T3D などの超並列計算機が、商用のレベルになっている。超並列計算機とは、各プロセッサにメモリが分散し、それぞれのプロセッサが一つの処理単位を構成するような並列計算機である。

そのような計算機上でソフトウェアを開発する場合、二つのアプローチがある。それぞれのプロセッサに別々の命令を与え自由度の高いプログラムを書く、いわゆる MIMD 的なプログラミングと、それぞれのプロセッサに一度に同じ動作をさせることによってプロセッサの稼働率が高くなるプログラムを書く、いわゆる SIMD 的なプログラミングがある。

しかし、 MIMD 的なプログラミングは、プログラマがそれぞれのプロセッサを管理しながら、プロセッサの稼働率が高くなるように意識しなければならず面倒である。一方、 SIMD 的なプログラミングは、数値計算のような特殊な用途では大変便利であるが、汎用性のあるプログラムを書くのが大変困難である。

そこで、二つのプログラミングの長所をいかした新しいプログラミングの手法が生まれる。つまり、プログラマは MIMD 的なプログラミングで自由度の高い汎用的なプログラムを作成しつつ、それを SIMD 的なプログラムで実行されることにより、プロセッサの管理をプログラマが行なわなくても稼働率の高くなるような手法である。

本論文では、 MIMD 的なプログラムを SIMD 的なプログラムでインタプリトする、その手法について述べる。

第 2 章

準備

本章では、実装の説明をする前に、それぞれの言語の説明をする。

2.1 並列記号処理言語

並列記号処理言語とは、記号処理を並列に行なうための言語である。

記号処理では、数値計算に比べ複雑にからみあうデータを取り扱う必要が多く、こうしたデータをどのようなデータ構造で表現するか、またそうしたデータ構造をどのようにメモリ上で管理するかが重要になる。このようなあまりプログラミングの本質的でないことに多くの労力を割かずに済むように、言語システムで基本的な機能を用意しよう、というのが記号処理言語の考え方である。その基本的な機能として、一意性のあるものに名前をつけると自動的に一意な表現を与えてくれるアトムの機構や、標準的なデータ構造についてそのためのメモリ割り付けや解放の労力を減らしてくれる自動メモリ管理機構などがある。この点、代表的な記号処理言語である Lisp と同様の機能を提供している。

並列処理言語は、全体の処理を複数の部分処理に分割して、必要なところでは部分処理間の同期をとりながら計算を進める必要がある。このために、逐次処理言語に並列実行を指定する機構と同期のための機構を追加し並列処理にも使えるようにする、というアプローチもあるが、並列に実行できる部分をいちいち指定したり、同期処理の面倒さがある。

2.1.1 並列記号処理言語 KL1

KL1 [1] は、逐次処理言語に並列実行のための機構を追加するという方法で設計されたのではなく、最初から並列処理に使うことを前提とした言語である。何も指定しなくてもすべてが並列実行可能で、データフロー同期機構を導入して同期が自動的に行なわれるようになっている。このおかげで、並列処理ソフトウェアの開発の労力が軽減される。

また、 KL1 は、 GHC [2, 3] を元に言語仕様をさらに簡素化した Flat GHC と呼ばれる言語を基礎として作られた言語でもある。

2.1.1.1 プログラムの形式

KL1 のプログラムは、**節**と呼ばれる基本単位の集合である。**節**は一般に以下のような形をしている。

```
述語名 (引数, ...) :- ガード | ボディ.
```

述語名 節での定義 (の一部) を与える述語の名前。

引数 述語の引数と節の中で使う変数の対応づけ。

ガード 節の適応条件。あらかじめ決められた種類の組み込み述語の呼出しの並び

ボディ 節を選んだときに実行すべき計算。ユーザが定義するものを含むユーザ定義述語と組み込み述語の呼出しの並び。

また、 KL1 の述語には二種類ある。

組み込み述語 あらかじめ言語仕様として定義された述語

ユーザ定義述語 ユーザがプログラム中に定義する述語

特殊な組み込み述語として、"`true`" がある。これは、ガードでは無条件を表し、ボディでは何もしないことを表す。

2.1.1.2 プログラムの実行

例えば、以下のようなものを用意する。

```
:- module main.

main :- not(1, X), io:outstream([print(X),nl]).

not(In, Out) :- In = 0 | Out = 1.
not(In, Out) :- In = 1 | Out = 0.
```

最初の行が、これがメインプログラムのモジュールだということを宣言している。`main` から始まる行が、メインプログラムの本体である。ここで、`not` の引数に `1` と `X` を与えた。大文字で始まる名前は変数を表す。

このプログラムで、`not` は以下のように実行される。

1. 第一引数の In が、呼び出したときの引数 1 に対応する。これは、前の節のガードの適応条件 “In = 0” を満たさないが、後の節の条件 “In = 1” を満たしている。そこで、後の節を選ぶ。

2. 選ばれた節のボディを実行する。ボディで、第二引数であるOut の値を 0 にする。この Out は、呼び出したときの引数 X に対応しているので、X の値が 0 になっている。

3. 他にすることがないので、これで実行を終わる。

この実行の結果、X の値が 0 に決まったので、io:outstream([print(X),nl]) を実行することにより、それが出力され、プログラムは終了する。

2.1.1.3 実行の機構

節はある条件を満たすゴールを、ボディのゴールに書き換える書換えルールと見ることも出来る。書き換えを繰り返していき、“true” にまで書き換えられると実行は終了する。

この書き換えは、必ずしも逐次的に行なわなくてもよく、複数のゴールについて同時に行なえば、実行は並列になる。

複数のゴールが同じ変数を共有することがある。あるゴールが変数の値を決めれば、それを共有する他のゴールはその値を使うことが出来る。これがゴール間の通信である。

節の適用条件を調べるとき、データがまだ決まっていないため、条件を試すことが出来なくなることがある。このような場合、当面の間このゴールの実行を見合わせることになり、実行を中断 (サスペンド) することになる。データが決まり、条件の真偽が出来るようになると、このゴールの実行を再開する。これがゴール間の同期の機構である。

以上が、基本的な実行機構のすべてである。

もし複数の節が適用条件を満たした場合、言語仕様ではそのような節のうちどれを選ぶかは決めていない (非決定性)。したがって、KL1 の正しいプログラムでは、一つの述語についてガード条件が排他的になるように書くか、排他的でない場合にはどれを選んでも正しく動作するように書かなければならない。

2.1.1.4 データ

KL1 には、いくつかの基本的なデータ型がある。

- **アトム** 内部構造を持たず、その値自身だけに意味があるようなデータ。識別子である記号アトムと整数値を表す整数アトムがある。

- **論理変数** 値はいったん決まったら後から変更することが出来ないという特徴を持っている。また変数には型がなく、どんな型の値でも持つことが出来る。
- **構造を持つデータ** 要素となるデータを集めて出来ているデータ。 KL1 では以下のようなものがある。

  ファンクタ 名前と任意の型のデータの並びを要素に持つ構造体

  コンス 任意の型の二つだけの要素をもつ構造体

  ベクタ 任意の型の要素を持つ一次元の並び

  ストリング 文字コードの並び

2.1.1.5 そのほかの機能

KL1 は上に述べてきた機能の他に、実行の制御 (優先度制御、負荷分散制御)、荘園機能などがある。

2.2 データ並列言語

データ並列言語とは、多くのデータに対する演算を並列に行なうための言語である。

一般的な計算機は、単一の命令によって単一のデータを処理する。しかし、流体力学的な運動のシミュレーション等、三次元空間内の連続な存在に対する計算を行なう場合、三次元空間内の格子上の運動としてとらえ、特定の時刻の特定の点についての計算を、対象とする点を変えながら繰り返すことになる。このような繰り返しは、時間に関する繰り返しはともかく、各点についての計算の繰り返しは、本来一回の計算だけですむはずの処理を、点の数に比例した時間に長引かせるものである。

このような状況に対する解として、並列処理がある。この並列処理の実現にはいくつかあり、パイプライン処理、ベクトル処理、マルチプロセッサなどが今まで使われてきた。

ここでは、超並列計算機による超並列計算を取り上げる。超並列計算とは、今までのマルチプロセッサによる計算を、より押し進めたものである。超並列計算機とは、通常のマルチプロセッサよりも多くのプロセッサが、それぞれ独立したメモリを持ち独立した処理単位を構成するような並列計算機である。

この計算機上で動く言語として、データ並列言語がある。

データ並列言語は、

- データ主導型のプログラミングで、広がりのあるデータにたいして、同時に処理を行なう
- 実プロセッサの個数、トポロジに依存せず、それぞれのプロセッサの管理の必要がない

といった特徴がある。

2.2.1 データ並列言語 C*

C*[4, 5] は、超並列計算機のために C を拡張した言語である。ここでは C に関する説明は行なわない。以下では、拡張された主な機能として、 shape、 with, where、演算子・関数の拡張について述べる。

2.2.1.1 shape

shape は、並列データを作り出すためのテンプレートを宣言する時に使用するキーワードである。したがって並列データを扱う前には shape を使用しなければならない。

例えば、次のように使用する。

```
shape [1024]goal;
```

これは、goal という名前で仮想プロセッサを 1024 個用意したことを意味する。またこの場合は仮想プロセッサが一次元的に用意されている。また、

```
shape [2][512]goal2;
```

と記述しても、用意される仮想プロセッサ数は、双方とも 1024 個で等しいが、後者の場合は仮想プロセッサが二次元的に用意されるのが相違点である。

```
int:goal t1;
```

これは shape として 宣言されている goal によって作られる、それぞれの仮想プロセッサ上に、

```
int t1;
```

を宣言していることに等しい。すなわち int 型の変数t1 を仮想プロセッサにそれぞれ割り当てる。

2.2.1.2 with, where

上の例では、並列データを作り出すことしか出来なかった。これらの作り出した並列データを、操作する前には、with を使って shape を選択したければならない。

例えば、次のように使う

```
with (goal) {
    t1 = 1;
}
```

これは、仮想プロセッサ上の t1 に、1 をそれぞれ代入している。

```
with (goal) {
    int:goal t2, t3;
    t2 = 2;
    t3 = 3;
    t1 = t3 - t1;
}
```

これは、仮想プロセッサ上に t2, t3 を宣言し、t2 には 2 を、t3 には 3 を代入し、t1 には t3 - t2 の値を計算し代入している。これらそれぞれの操作は並列的に行なわれる。

今までの例では、仮想プロセッサはすべて動いていた。いくつかの仮想プロセッサのみで動かしたいとき、つまり動かす仮想プロセッサを選択したいときに where を使う。

例えば、次のように使う。

```
with (goal) {
    where (t1 != 1)
        t1 = 1;
}
```

これは、t1 の値が 1 でない仮想プロセッサのみ、t1 に 1 を代入している。

2.2.1.3 演算子・関数の拡張

C* は、並列データを扱うために演算子・関数が拡張されている。そのうちの主なものを例をあげて説明する。

```
int s1 = 0;
with (goal) {
    s1 += t1;
}
```

これは、+= の拡張である。s1 という並列でない変数に、並列変数である t1 の値を合計して代入している。

```
with (goal) {
    if (&= (t1 == 1))
        printf("success\n");
}
```

これは、&= の例である。t1 値がすべて 1 であるとき、if 文内の printf を実行している。

```
int:goal add(int:goal a, int:goal b)
{
    return a + b;
}
```

これは、拡張された関数定義の例である。goal 上のint 型のデータを二つ引数にとり、その和を仮想プロセッサ毎それぞれ計算し、その結果を返り値にする、ということをする関数の定義である。

以上、 C* の機能をまとめると以下のようになる。

- 実プロセッサを仮想的に扱うために導入された shape によって、並列データを容易に扱える
- shape を選択する with、並列データを制限して操作する where がある
- 並列データを扱うための、演算子・関数 の拡張がある。

第 3 章

KL1 の C* による実装

KL1 は記号処理を並列に行なうための言語であり、C* はデータに対する演算を並列に行なうのための言語である。本研究では、実装の基本となる部分、つまり KL1 の並列性の基本部分である **複数のゴールを同時に書き換える** 処理を、C* の特徴の一つである**データの同時処理** によって行なう、ということに重点をおいた。

また、KL1 の言語仕様はコンパクトであるが、容易な実装を妨げる部分がいくつがあり、本研究においてはいくつかの機能 (モジュール、いくつかの組み込み述語、いくつかのデータ型、実行の制御、荘園機能) を省略することにした。

本章では、その実装方法について述べる。

3.1 概要

本節では、KL1 を C* で実装するための概要を示す。

3.1.1 C* の制限

本研究における KL1 は、C* で書かれている。つまり、KL1 で表現されるデータは、C* のデータ構造で表現されなければならない。しかし実装するにあたって、C* のデータ構造には主に以下のような制限がある。

- 並列構造体 (`struct`,`union`) は、ポインタ、並列変数を要素に持てない

  例

```
shape [1024]goal;
struct term {
    struct term *car;
    struct term *cdr;
};
struct term:goal list;
```

は、エラー

- 制御文 (if,while など) の条件式に、並列データを置けない

  例

```
int:goal p;
if (p) {
    ;
}
```

  は、エラー

- スカラ (並列でない) 配列のインデックス部に並列データを置けない。

  例

```
int:goal q;
int m[1024];
n = m[p];
```

  は、エラー

3.1.2 データの表現

本研究では、実装を容易に行なうために、 KL1 のデータ型として記号アトム、論理変数、複合項 を用意する。しかし、上のような制限があるため、 C* の並列性を生かすためにはデータ構造に工夫が必要である。本研究では、以下のような機能をデータに持たせることを目標とした。

- 複数のゴールから同時に別々のデータをアクセスできるようにする
- 条件を満たすゴールの選択を、同時に行なうようにする
- 複数のゴールの書き換えを、同時に行なうようにする

そのために、 C* でデータを以下のように表現することにした。

- データ (項) は、自然数で表す。つまり異なるデータは、それに対応する自然数も異なる。
- データの型は、そのデータに対応する自然数をインデックスとするスカラ配列で表す。

- 記号アトム、論理変数、複合項も、そのデータに対応する自然数をインデックスとするスカラ配列で表す。

それぞれのデータ型は以下のように表現することにした。

記号アトム 識別子である、文字列 (文字へのポインタ) で表現する。

論理変数 二つの状態を持つ。値が決まっていない状態と値が決まっている状態である。変数がどちらの状態でも構わないように、両方の値を構造体で持つことにする。構造体の要素として、変数の名前と、値が決まっていない時のために変数の番号、値が決まっている時のためにその値を参照するリファレンス値がある。

複合項 構造体で表す。構造体の要素として、ファンクタ値、引数の数、引数の配列がある。

3.1.3 プログラムの内部表現

ゴールの書き換えを同時に行なうようにするためには、 KL1 の節 (プログラムの単位) を計算機内に保存するための表現も、同様に工夫する必要がある。複数のゴールから、条件を満たすゴールの選択を別々に同時に行ないたいからである。そこで、以下のようにする。

- プログラムは、節の並列配列で表す。並列配列であるが、それぞれの仮想プロセッサは、すべて同じ値の配列を持つようにする。また同じ述語の節は、となり合うように、配列上に配置する。
- 節は構造体で表現し、その要素として、ヘッド (述語)、ガードの数、ガードの配列、ボディの数、ボディの配列、その節に含まれる変数の数がある。
- ゴールの選択に並列性を持たせるため、プログラムとは別に、述語の並列配列を用意する。この述語の配列は、構造体で表し、その要素として、その述語がプログラムの配列のどの位置から始まるか、その開始位置と、その述語がプログラムの配列中にいくつ並んで配置されているか、その数がある。

3.1.4 ゴール

ゴールは、並列実行の単位であるの仮想プロセッサとする。つまりゴールは、独立したメモリを持つ計算機とみなすことができる。ゴールは、主に以下のような値を持つ。

述語 ゴール本体を表す。

環境 そのゴール中で使われている変数の束縛を表す。

状態 ゴールの状態。実行中、中断、終了、の三つがある。

それ以外にも、作業用の変数がいくつかある。

3.1.5 実行機構

KL1 のプログラムは、以下のようなループで実行が進められる。

1. ゴールが組み込み述語の場合、それを処理する。
2. 組み込み述語でない場合、
   (a) 条件を満たす節が選択できるか調べる。
   (b) 節を選ぶことができれば、その節のボディのゴールをを新しく用意されるゴールに置き換える。
   (c) 節を選ふことができなければ、中断する。

この実行は、基本的にすべてのゴールで同時に行なわれる。

3.2 実装

本節では、前節で述べた実装の概要について C* で実際に実装するための手法について述べる。

3.2.1 データの表現

本研究における KL1 では、データは自然数で表すことにする。つまりあるデータ `t1` と `t2` がある時、 `t1 == t2` (等しい) ならば、同じデータを表し、 `t1 != t2` (異なる) ならば別のデータを表す。 C* では、データを次のように表す。

```
typedef int Term_i;


Term_i t1, t2;
```

これは、`t1` と `t2` の二つのデータを宣言している。また、これらの値が負数の時は、データを表さないことにする。

それで、実際のデータの実態は、データをインデックスとするデータ型の配列の要素という形で保持される。例えば、論理変数である`t1` の実体を参照する時は、次のようにする。

```
a_atom[t1]
```

a_atom とは、記号アトムの配列である。

データ型として 記号アトム以外に、論理変数、複合項 の三つがある。それらはそれぞれ、次のように表現される。

```
typedef char *Atom;
typedef struct Var_ Var;
typedef struct Comp_ Comp;

struct Var_ {
    Term_i name;
    int num;
    Term_i ref;
};

struct Comp_ {
    Term_i func;
    int arity;
    Term_i args[MAX_ARGS];
};

Atom a_atom[MAX_TERM];
Var a_var[MAX_TERM];
Comp a_comp[MAX_TERM];
```

MAX_TERM とは、データの最大値である。データは、MAX_TERM 個作ることができる。

記号アトム (Atom) は、文字列 (文字へのポインタ) で表す。

論理変数 (Var) は、構造体で表され、その要素であるname は、変数の名前で記号アトム (この場合は単なる文字列) によって表し、num は、その変数が節の中で何番めに表れたかその番号を表し、ref は、その変数がどのデータを指し示しているかを表す。論理変数は値が決まっていないと値が決まっているの二つの状態を持ち、それらは ref の値が、負数か非負数かで表現される。

複合項 (Comp) は、構造体で表現され、その要素であるfunc は、ファンクタ (複合項の名前) で記号アトムによって表し、arity は、複合項の引数の数を表し、args は、引数の配列を表している。

a_atom a_var a_comp は、実際のデータを保持するための、配列である。

また、実際のデータがどのデータ型であるか調べるために、以下のようなタグの配列を用意した。

```
enum Tag_ {
    T_ATOM, T_VAR, T_COMP
};

typedef enum Tag_ Tag;

Tag a_tag[MAX_TERM];
```

タグ (Tag) は、T_ATOM、T_VAR、T_COMP の三つの値を持つことができるデータ型である。それぞれ、記号アトム、論理変数、複合項 の三つに対応する。あるデータ t1 が与えられた時、a_tag[t1] == T_ATOM ならば、t1 は記号アトムである、というふうに使用する。

3.2.2 プログラムの内部表現

本研究における KL1 では、 KL1 のプログラムは節を表すデータ型の並列配列で表現される。

```
typedef struct Clause_ Clause;

struct Clause_ {
    Term_i head;
    int guard_num;
    Term_i guard[MAX_GUARD];
    int body_num;
    Term_i body[MAX_BODY];
    int var_num;
};

Clause:goal program[MAX_PROGRAM];
```

MAX_PROGRAM とは、 KL1 のプログラムの最大値である。プログラムは、MAX_TERM 個入力することができる。MAX_GUARD MAX_BODY も同様で、それぞれガードの最大値、ボディの最大値で、MAX_GUARD MAX_BODY 個入力することができる。

節 (Clause) は、構造体で表され、その要素であるhead は、その節の述語 (ヘッド) を表し、guard_num は、節のガードの数を表し、guard は、節のガードの配列を表し、body_num は、節のボディの数を表し、body は、ボディの配列を表し、var_num は、節に含まれる論理変数の数を表している。

goal は shape である。詳しくは次節で述べることにする。

プログラムは、同じ述語の節がとなり合うように配置する。これは述語を走査するのを簡単にするためである。

前説でも述べたように、プログラムは、節の並列配列で表現される他に、節の述語の並列配列でも表されている。

```
typedef struct Pred_ Pred;

struct Pred_ {
    int start;
    int num;
};

Pred:goal predicate[MAX_TERM];
```

述語 (Pred) は、構造体で表され、その要素であるstart は、その述語の program 中での開始位置を表し、num は、その述語がいくつ並んでいるかを表す。

predicate は、述語をインデックスとする並列配列である。述語をインデックスとして与えると、その述語がprogram 中で、どこを占めているかが分かる。いわば、ハッシュの役割もある。

例として、図 3.1に KL1 のプログラムが入力された時の内部表現を示す。

3.2.3 ゴールおよびその環境

ゴールは、仮想プロセッサとする。つまり、並列実行の処理単位として、ゴールを設定する。C* で次のように表す。

```
shape [MAX_GOAL]goal;
```

goal とは、それ自身ではなにもデータを持っていない。あくまでも、並列データをつくり出すためのテンプレートだからである。ゴール内にデータを持ちたい時は、別に定義する必要がある。

ゴール内の主なデータとして、以下のようなものを定義する。

```
enum State_ {
    S_FINISH, S_ACTIVE, S_SUSPEND
};
typedef enum State_ State;

Term_i:goal term;
Term_i:goal env[MAX_ENV];
State:goal state;
```

KL1 のプログラムとして、

```
not(0,Out) :- true | unify(Out,1).
not(1,Out) :- true | unify(Out,0).
```

を与えた時の、a_tag、a_atom、a_var、a_comp、program、predicate の内容

	a_tag	a_atom	a_var		a_comp			predicate	
			name	num	func	arity	args	start	num
0	T_ATOM	"true"							
1	T_ATOM	"unify"							
2	T_COMP				3	2	4,5	0	2
3	T_ATOM	"not"							
4	T_ATOM	"0"							
5	T_VAR		6	1					
6	T_ATOM	"Out"							
7	T_COMP				1	2	5,8	0	2
8	T_ATOM	"1"							
9	T_COMP				3	2	8,10		
10	T_VAR		6	1					
11	T_COMP				1	2	10,4		

	program					
	head	guard_num	guard	body_num	body	var_num
0	2	1	0	1	7	1
1	9	1	0	1	11	1

なお、a_atom[0] の "true" と、a_atom[1] の "unify" は、予約されているアトムである。

図 3.1: プログラムの内部表現

term とは、それぞれのゴールの本体の述語である。env は、そのゴールに含まれる論理変数の束縛 (環境) を表す。state は、ゴールの状態を状態を表す。状態としてS_FINISH S_ACTIVE S_SUSPEND があり、それぞれ、ゴールの終了、ゴールの実行中、ゴールの中断、という状態を表す。

ここで、もうすこし**環境**について説明する。env はインデックスとして、「ゴールが選択している節に含まれる論理変数の出現した順序」を与える。例を挙げると、

```
app([X0|X1],Y,Z) :- true | app(X1,Y,Z1) , unify(Z, [X0|Z1]).
```

上の四角で囲まれた部分が現在注目しているゴール (つまりterm) で、上の節はゴールが選択している節であるとする。そこでapp(X1,Y,Z1) の論理変数 Y にアクセスする場合を考える。term 中の Y は、a_var[a_comp[term].args[1]] であるが、この値はプログラム中の値であり、実際の実行中の値とは異なる。そこで、env を使う。節中で Y の出現した順序は、a_var[a_comp[term].args[1]].num で分かる (この場合は2)。この値をインデックスとしたenv[a_var[a_comp[term].args[1]].num](この場合はenv[2]) が、実行中の Y の値となる。

3.2.4 実行機構

以上のことを踏まえて、実行機構の実装手法について述べる。

本研究において、 C* のプログラムの実行の主体となる部分は

```
void solve (Clause *goal_cl)
```

である。この関数の引数 goal_cl は、節へのポインタでその節のボディ部が、初期のゴールになる。以下この関数のアルゴリズムを順をおって述べる。

3.2.4.1 初期ゴールの設定

初めに、最初のゴールを仮想プロセッサに割り当てる。

```
allocate_goal(1, allocate_place);
goal_place = [0]allocate_place[0];

where (pcoord(0) == goal_place) {
    env_num = goal_cl->var_num;
    allocate_env(env_num, env);
    term = goal_cl->body[0];
    state = S_ACTIVE;
}
```

allocate_goal はゴールの割り当てをする関数である。割り当てられたゴールを allocate_placet に、一時的に割り当ててから goal_place という変数に代入する。後にgoal_place は、すべてのゴールを解き終えた時にもう一度使用する。次に、割り当てられたゴールの位置に最初のゴールを割り当てる。”where (pcoord(0) == goal_place)” の中に表れている pcoord(0) とは、ゴール自分自身の位置を返す関数である。つまり、ゴール自分自身の位置と割り当てられたゴールの位置が等しい仮想プロセッサのみ、 where 文の中を実行する。これで、割り当てられたゴールの位置に最初のゴールを割り当てる準備ができる。そして最初のゴールに、term(述語)、env(環境)、state(状態) を割り当てる。環境はallocate_env で割り当てる。それ以外にenv_num (環境のデータの数) を割り当てておく。

そして、プログラムを解くためのループに入る。

3.2.4.2 組み込み述語の処理

ループは、組み込み述語の処理から始める。

```
where (term == true) {
    state = S_FINISH;
}
else where ((tag_term == T_COMP)
            && (comp_term.func == unify)
            && (comp_term.arity == 2)) {
    where (b_unify(comp_term.args[0], comp_term.args[1]))
        state = S_FINISH;
    else
        state = S_SUSPEND;
}
```

もしも、ゴールが true であれば、そのゴールの状態state に S_FINISH を代入し終了する。

次にゴールが unify であれば、関数 b_unify で処理され、その戻り値が !0 であれば、ユニファイに成功したということで state に S_FINISH を代入、終了する。戻り値が 0 であれば、ユニファイに失敗したということで、とりあえず state に S_SUSPENDH を代入しておく。

本研究においては、組み込み述語はこの true とunify しか用意していない。必要があれば、組み込み述語の処理を別関数として定義し、その中で複雑な処理をさせればいい。

3.2.4.3 節の選択

ゴールが組み込み述語でなければ、そのゴールはユーザ定義の述語とみなし、その述語で条件を満たすことの出来る節を選択する。

```
clause_start = predicate[term].start;
clause_num = predicate[term].num;
for (i = 0, ended = 0;; i++) {
    where (i >= clause_num) ended = 1;
    if (&= ended) break;
    where (!ended) {
        clause_temp = program[clause_start+i];
        result = commit(clause_temp);
        where (result != R_NOCOMMIT) {
            chosen_clause = clause_start+i;
            ended = 1;
        }
    }
}
```

まずその述語が、プログラムの配列である program 中のどこにあるかを調べる。述語の開始位置を clause_start に、その述語がいくつ定義されているかを clause_num に代入する。

次にそれぞれのゴールについて、clause_num 回 節選択のための関数 commit を実行する。commit は、引数として Clause:goal をとり、戻り値として Result:goal を返す関数である。Result とは、

```
enum Result_ {
    R_NOCOMMIT, R_SYSTEM, R_USER
};
typedef enum Result_ Result;
```

と定義されており、R_NOCOMMIT、R_SYSTEM、R_USER はそれぞれ、節選択に失敗した、節選択に成功し組み込み述語であった (この値は現在使っていない)、節選択に成功しユーザ定義述語であった、という意味に対応する。

繰り返しの途中で commit の戻り値が R_NOCOMMIT 以外になった時、節選択が成功したことになる。その時のprogram 中の位置を chosen_clause に保存しておき、ループから抜ける。(また、この時 result の値はR_USER になっているはずである。)

もし、clause_num 回 繰り返しても commit の戻り値が R_NOCOMMIT であるならば、何もせずに抜ける。(この時 result の値は R_NOCOMMIT になっているはずである。)

3.2.4.4 ユーザ定義述語の処理

節選択の繰り返しの終了後、result の値が R_USER であれば、ユーザ定義述語の処理に入る。

```
body_num = program[chosen_clause].body_num;
sum_body_num = 0;
sum_body_num += body_num;
allocate_goal(sum_body_num, allocate_place);
total_body_num = scan(body_num, 0, CMC_combiner_add,
    CMC_upward, CMC_none, CMC_no_field, CMC_exclusive);
for (i = 0, ended = 0;; i++) {
    where (i >= body_num) ended = 1;
    if (&= ended) break;
    where (!ended) {
        int:goal ended;
        int i;
        current_place =
            allocate_place[total_body_num + i];
        [current_place]env_num = newenv_num;
        for (i = 0, ended = 0;; i++) {
            where (i >= newenv_num) ended = 1;
            if (&= ended) break;
            where (!ended) {
                [current_place]env[i] = newenv[i];
            }
        }
        [current_place]term =
            program[chosen_clause].body[i];
        [current_place]state = S_ACTIVE;
    }
}
state = S_FINISH;
```

始めに、ボディの数だけゴールを用意する。複数のゴールで同時にゴールの割り当てを行なおうとすると、衝突が発生するおそれがあるので、それぞれのゴールで選択された節のボディの数body_num を sum_body_num に合計し、sub_body_num 個のゴールの割り当てを一括して行なう。

次に、割り当てられたゴールに、選択された節のボディの内容を書き込む。それぞれのゴールで body_num 回、term、env、state、env_num を書き込む。current_place とは、それぞれのゴールが、現在注目している割り当てられているゴールの位置である。

繰り返しが終了すると、それぞれのゴールの state にS_FINISH を代入し終了する。

3.2.4.5 節選択失敗の処理

節選択の繰り返しの終了後、result の値がR_NOCOMMIT であれば、そのゴールは、節選択のために必要なデータがまだ決まっていないために起こる中断か、単に節選択に失敗か、ということになる。とりあえず、state にS_SUSPENDH を代入しておく。

3.2.4.6 実行の終了

3.2.4.2 から始まるループは、以下のような条件になっている。

```
while (|= (state == S_ACTIVE)) {
    where (state != S_FINISH) {
```

つまり、あるゴールが実行中ならばループはまわり続け、終了していないゴールがループの中を実行する。このループから出た時、以下の状態になっている。

1. すべてのゴールが終了している。
2. いくつかのゴールが中断している。

1 の時、 3.2.4.1 でgoal_place に、初期ゴールの位置を代入しておいたので、初期ゴール中に含まれる論理変数の内容を表示して、関数solve を終了する。

2 の時、中断しているゴールの内容を表示して、関数solve を終了する。

3.2.5 ユニフィケーション

本研究では、データ並列時におけるユニフィケーションの研究を、まだ十分に行なっているわけではないが、ここで本実装におけるユニフィケーションの概略を述べる。

KL1 では、ユニフィケーションに二つの意味がある。

1. 節選択時におけるユニフィケーション。二つのデータが等しいか (等しくなれるか) どうかを試す。

2. ボディでのユニフィケーション。二つのデータを等しいものにするという操作をする。

以下に 1 に対応するヘッド・ユニフィケーションと、 2 に対応するボディ・ユニフィケーションの実装手法を述べる。

3.2.5.1 ヘッド・ユニフィケーション

ヘッド・ユニフィケーションとは、節選択時において、ボディと節の述語が等しくなれるかどうかを試すことである。このとき、節中の述語に含まれる論理変数は、このユニフィケーションが成功できるように値を決めることが出来るが、ボディ中に含まれる論理変数は、たとえ値が決まっていない状態であっても、値を決めることが出来ない。

アルゴリズムを以下に示す。

1. 与えられたデータが、変数で、 KL1 のプログラム中の変数であるならば、その変数が使われている実際のデータと置き換える。
2. データが、変数で、どこかのデータを指し示しているならば、そのデータを置き換える。
3. 二つのデータのうち、どちらかが無名変数ならば、真を返し、終了する。
4. 節の述語側が変数以外の時、
   (a) ゴール側が変数ならば、偽を返し、終了する。
   (b) どちらもアトムならば、直接比較した値を返し、終了する。
   (c) どちらも複合項で、ファンクタ、引数の数ともに等しい時、
      i. それぞれ対応する引数を、本アルゴリズムにかけてみて、すべて真が返ってきたならば、真を返し、終了する。
      ii. それ以外ならば、偽を返し、終了する。
   (d) それ以外ならば、偽を返し、終了する。
5. 節の述語側が変数の時、述語側の変数が、ゴール側のデータを参照するようにし、真を返し、終了する。

3.2.5.2 ボディ・ユニフィケーション

ボディ・ユニフィケーションとは、組み込み述語の一つで、ボディで指定することが出来る。データ中に値が決まっていない状態の論理変数があれば、積極的に値を決めて二つのデータを等しいものにしようとする。

アルゴリズムを以下に示す。

1. 与えられたデータが、変数で、 KL1 のプログラム中の変数であるならば、その変数が使われている実際のデータと置き換える。

2. データが、変数で、どこかのデータを指し示しているならば、そのデータを置き換える。

3. 二つのデータのうち、どちらかが無名変数ならば、真を返し、終了する。

4. どちらもアトムならば、直接比較した値を返し、終了する。

5. どちらも複合項で、ファンクタ、引数の数ともに等しい時、

   (a) それぞれ対応する引数を、本アルゴリズムにかけてみて、すべて真が返ってきたならば、真を返し、終了する。

   (b) それ以外ならば、偽を返し、終了する。

6. どちらかが変数ならば、その変数が、もう一方のデータを参照するようにし、真を返し、終了する。

7. それ以外ならば、偽を返し、終了する。

第 4 章

関連研究の動向

本研究の関連研究として、いくつかある。本章ではそれらについて簡単に述べることにする。

4.1 コンディショングラフによる KL1

Barklund ら [6] が、コネクションマシン上にコンディショングラフを実装し、 KL1 をその上にマッピングする方法を提案している。

たとえば、次のような KL1 のプログラムが与えられたとき、

```
:- append([1,2],[3],R).

append([A|X],Y,R) :- true | R=[A|Z], append(X,Y,Z).
append([],Y,R)    :- true | Y=R.
```

図 4.1のようなグラフになる。

このコンディショングラフを、コネクションマシン上で並列に解くことにより、並列性を出している。

4.2 Fleng による GHC の実装

Nilsson [7] が、 SIMD 計算機上に並列論理型言語の実装の研究をしている。 GHC を 中間言語である Fleng にコンパイルしたのち、それを SIMD 計算機上でインタプリトするというものである。 Fleng とは、 GHC のガードをなくしたような、見た目が Prolog に似た言語であるが、

- ゴールの順序に、依存しない。
- バックトラックしない。

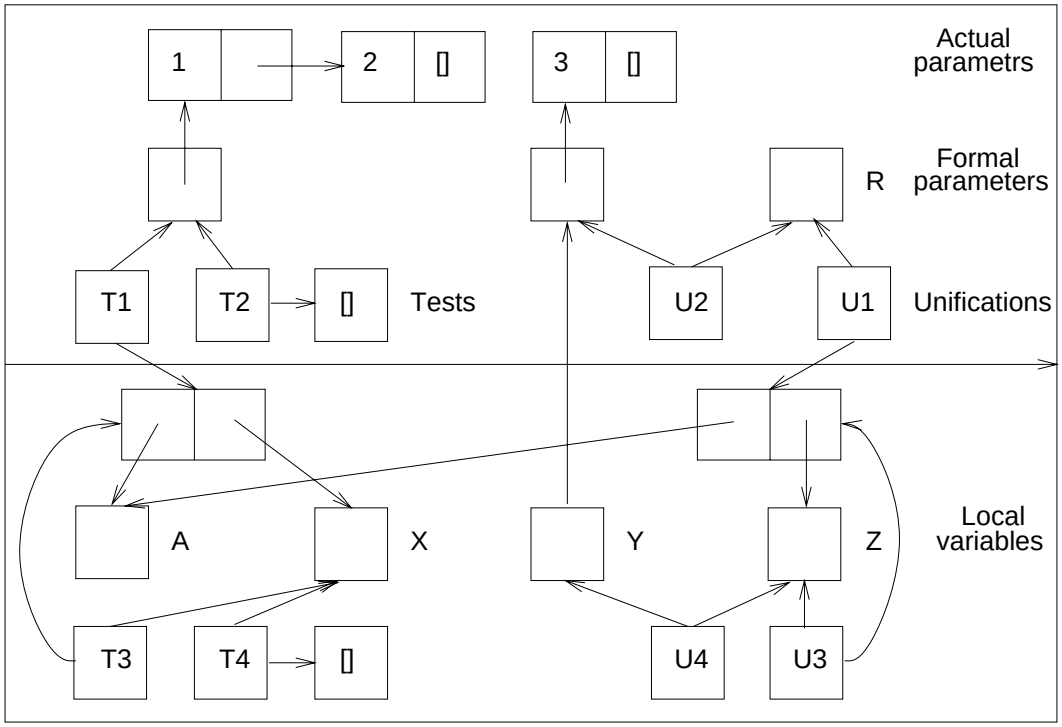

図 4.1: `append` のグラフ

といった特徴がある。

たとえば、次のような GHC のプログラムが与えられたとき、

```
p(X,Y) :- X < 0  | q(X,Y).
p(X,Y) :- X >= 0 | r(X,Y).
```

次のような Fleng のプログラムにコンパイルする。

```
p(X,Y) :- less(X,0,R), p1(R,X,Y).
p1(true,X,Y)  :- q(X,Y).
p1(false,X,Y) :- r(X,Y).
less(X,Y,R) :- X:=A, Y:=B, compute('<',A,B,R).
```

この Fleng のプログラムを SIMD 計算機で並列に解くことにより、並列性を出している。

4.3 本研究との違い

以上の研究は、どれも **並列記号処理言語を** SIMD **計算機で実装する** という点で本研究と類似性があるが、本研究では C* **という、一段水準の高いレベルから、並列記号処理言語を実装する**ということが、大きな違いである。現在 C* は 超並列計算機上に

しか処理系がないが、将来 他のアーキテクチャ上に C* の処理系が作られた時、本研究のような実装手段は有効であると思われる。

第 5 章

結論

本研究では、並列記号処理言語をデータ並列言語で実装する方法について述べた。まだ、実用的なレベルとまではいかないが、 C* で KL1 を実装したことにより、より柔軟なプログラミングが可能になった。また、この実装手法は他の並列論理型言語を C* で実装する場合にも、非常にに有意義であると思われる。

今後の課題として、より効率的なインタプリタの手法、コンパイルする手法、それに伴う言語の拡張などの研究の余地が残されている。今後は上記の点を含め、実用的な処理系実現のために、様々な理論、実装手法について研究を深めていく予定である。

謝辞

本研究を行うにあたり、指導教員である電子・情報工学系田中二郎助教授には終始、親身なご指導をいただきました。また、中原鉱一氏には、多大なるご協力をいただきました。 新情報処理開発機構の石川裕氏には、 CM-5 の使用に関する手続きを行なっていただきました。 電子・情報工学系のソフトウエア研究室の中田育男教授、加藤和彦講師には、 C* に関するアドバイスをいただきました。井田哲雄教授にはデータ並列計算に関してアドバイスをいただきました。佐々木重雄氏には、多くの助言をいただきました。中野勝次郎氏、浦賀毅氏、小森由紀子氏には、心から感謝し御礼申し上げます。最後になりましたが、ソフトウェア研究室の皆さんに感謝の言葉を送らせていただきます。

参考文献

[1] 新世代コンピュータ技術開発機構 第四研究室. KL1 プログラミング.

[2] 淵一博監修 古川康一・溝口文雄共編. 並列論理型言語 GHC とその応用. 知識情報処理シリーズ 6. 共立出版, 1987.

[3] Jiro Tanaka. Meta-interpreters and Reflective Operations in GHC. In *The International Conference on Fifth Generation Computer Systems 1988*, 1988.

[4] Thinking Machines Corporation. *Getting Started in C**.

[5] Thinking Machines Corporation. *C* Programming Guide*.

[6] Jonas Barklund, Nils Hagner, and Malik Wafin. KL1 in Condition Graphs on a Connection Machine. In *The International Conference on Fifth Generation Computer Systems 1988*, 1988.

[7] Martin Nilsson. *Parallel Logic Programming for SIMD Supercomputers and Massively Parallel Computers*. PhD thesis, The University of Tokyo, 1988.

付録 A

ソース

本研究において、 C* で実装した KL1 のインタプリタのソースをここに示す。

```
/*
 * 卒論プログラム
 * KL1 もどき
 *
 * coded by age@softlab.is.tsukuba.ac.jp
 */

#include <stdio.h>
#include <stdlib.h>
#include <cscomm.h>

#define MAX_GOAL 128
#define MAX_TERM 1024
#define MAX_PROGRAM 16
#define MAX_GUARD 8
#define MAX_BODY 8
#define MAX_ARGS 8
#define MAX_PREDICATE 16
#define MAX_ENV 16

enum Result_ {
    R_NOCOMMIT, R_SYSTEM, R_USER
};

enum State_ {
    S_FINISH, S_ACTIVE, S_SUSPEND
};

enum Tag_ {
    T_ATOM, T_VAR, T_COMP
};

typedef enum State_ State;
typedef enum Result_ Result;
typedef enum Tag_ Tag;

typedef char *Atom;
typedef struct Var_ Var;
```

```
typedef struct Comp_ Comp;
typedef struct Clause_ Clause;
typedef struct Pred_ Pred;
typedef struct Goal_ Goal;

typedef int Term_i;

struct Var_ {               /* 変数の定義         */
    Term_i name;            /* 名前               */
    int num;                /* 番号               */
    Term_i ref;             /* リファレンス       */
};

struct Comp_ {              /* 複合項の定義       */
    Term_i func;            /* ファンクタ         */
    int arity;              /* 引数の数           */
    Term_i args[MAX_ARGS];  /* 引数               */
};

struct Clause_ {             /* 節の定義          */
    Term_i head;             /* ヘッド            */
    int guard_num;           /* ガードの数        */
    Term_i guard[MAX_GUARD]; /* ガード            */
    int body_num;            /* ボディの数        */
    Term_i body[MAX_BODY];   /* ボディ            */
    int var_num;             /* 変数の数          */
};

struct Pred_ {               /* プログラム中の述語 */
    int start;               /* 開始位置          */
    int num;                 /* 数                */
};

/* shape 関係 */
shape [MAX_GOAL]goal;

Term_i:goal env[MAX_ENV];
int:goal env_num;
Term_i:goal term;
State:goal state;
Clause:goal program[MAX_PROGRAM];
Pred:goal predicate[MAX_TERM];
Clause:goal clause_temp;
int:goal clause_start, clause_num;
Result:goal result;
int:goal ended;
int:goal i;
int:goal body_num;
int:goal total_body_num;
int:goal current_place;
int:goal chosen_clause;
int:goal allocate_place[MAX_GOAL];
Term_i:goal newenv[MAX_ENV];
int:goal newenv_num;
```

```
/* global varibale */
Tag a_tag[MAX_TERM];
Atom a_atom[MAX_TERM];
Var a_var[MAX_TERM];
Comp a_comp[MAX_TERM];

int predicate_num;
int sum_body_num;
int j;
Term_i true, unify, anon;

/* local variabel */
static char *nexttoken = 0;
static char text[256];
static int current_term = 0;
static int program_start_term;

/* Input & Output */
void error(char *s)
{
    char buf[256];
    sprintf(buf, "%s\n", s);
    fputs(buf, stdout);
    exit(1);
}

char *next_token()
{
    if (nexttoken == 0) {
        if (scanf("%s", text) == 1) {
            nexttoken = (char *) malloc(strlen(text) + 1);
            strcpy(nexttoken, text);
        }
        else
            error("input error!!");
    }
    return nexttoken;
}

char *token_read()
{
    char *token;
    if (nexttoken == 0) {
        if (scanf("%s", text) == 1) {
            token = (char *) malloc(strlen(text) + 1);
            strcpy(token, text);
        }
        else
            error("input error!!");
    }
    else {
        token = nexttoken;
        nexttoken = 0;
    }
    return token;
```

```
}

Term_i term_read()
{
    int i, j;
    Term_i t;
    char *s = token_read();

    if (isupper(s[0]) || (s[0] == '_')) {
        t = -1;
        for (i = program_start_term; (a_tag[i] >= 0) && (i < MAX_TERM); i++) {
            if ((a_tag[i] == T_VAR) &&
                (strcmp(a_atom[a_var[i].name], s) == 0)) {
                t = i;
                break;
            }
        }
        if (t == -1) {
            t = allocate_term(1);
            a_tag[t] = T_VAR;
            a_var[t].name = -1;
            for (i = 0; (a_tag[i] >= 0) && (i < MAX_TERM); i++) {
                if ((a_tag[i] == T_ATOM) &&
                    (strcmp(a_atom[i], s) == 0)) {
                    a_var[t].name = i;
                    break;
                }
            }
            if (a_var[t].name == -1) {
                a_var[t].name = allocate_term(1);
                a_tag[a_var[t].name] = T_ATOM;
                a_atom[a_var[t].name] = s;
            }
            j = 0;
            for (i = program_start_term; (a_tag[i] >= 0) && (i < MAX_TERM);
                                         i++) {
                if (a_tag[i] == T_VAR)
                    j++;
            }
            a_var[t].num = j;
        }
    }
    else {
        if (strncmp(next_token(), "(", sizeof ("(")) != 0) {
            t = -1;
            for (i = 0; (a_tag[i] >= 0) && (i < MAX_TERM); i++) {
                if ((a_tag[i] == T_ATOM) &&
                    (strcmp(a_atom[i], s) == 0)) {
                    t = i;
                    break;
                }
            }
            if (t == -1) {
                t = allocate_term(1);
                a_tag[t] = T_ATOM;
```

```
                a_atom[t] = s;
            }
        }
        else {
            t = allocate_term(1);
            a_tag[t] = T_COMP;
            a_comp[t].func = -1;
            for (i = 0; (a_tag[i] >= 0) && (i < MAX_TERM) ; i++) {
                if ((a_tag[i] == T_ATOM) &&
                    (strcmp(a_atom[i], s) == 0)) {
                    a_comp[t].func = i;
                    break;
                }
            }
            if (a_comp[t].func == -1) {
                a_comp[t].func = allocate_term(1);
                a_tag[a_comp[t].func] = T_ATOM;
                a_atom[a_comp[t].func] = s;
            }

            token_read();
            i = 0;
            for (;; i++) {
                a_comp[t].args[i] = term_read();
                if (strncmp(next_token(), ",", sizeof(",")) != 0)
                    break;
                token_read();
            }
            if (strncmp(next_token(), ")", sizeof(")")) != 0)
                error("term read error!!");
            token_read();
            a_comp[t].arity = i + 1;
        }
    }
    return t;
}

void guard_read(Clause *cl)
{
    int i = 0;
    for (;; i++) {
        cl->guard[i] = term_read();
        if (strncmp(next_token(), ",", sizeof(",")) != 0)
            break;
        token_read();
    }
    cl->guard_num = i + 1;
}

void body_read(Clause *cl)
{
    int i = 0;
    for (;; i++) {
        cl->body[i] = term_read();
        if (strncmp(next_token(), ",", sizeof(",")) != 0)
```

```
            break;
        token_read();
    }
    cl->body_num = i + 1;
}

void clause_read(Clause *cl)
{
    int i, j;

    program_start_term = current_term;
    if (strncmp(next_token(), "?-", sizeof("?-")) == 0) {
        cl->head = -1;
        cl->guard_num = 0;
        token_read();
    }
    else {
        cl->head = term_read();
        if (strncmp(next_token(), ":-", sizeof(":-")) != 0)
            error("clause read error!!");
        token_read();
        guard_read(cl);
        if (strncmp(next_token(), "|", sizeof("|")) != 0)
            error("clause read error!!");
        token_read();
    }
    body_read(cl);
    if (strncmp(next_token(), ".", sizeof(".")) != 0)
        error("clause read error!!");
    token_read();

    j = 0;
    for (i = program_start_term; (a_tag[i] >= 0) && (i < MAX_TERM); i++) {
        if (a_tag[i] == T_VAR)
            j++;
    }
    cl->var_num = j;
}

void term_write(Term_i t)
{
    switch (a_tag[t]) {
    case T_ATOM:
        printf("%s", a_atom[t]);
        break;
    case T_VAR:
        printf("%s", a_atom[a_var[t].name]);
        break;
    case T_COMP:
        {
            int i;
            printf("%s(", a_atom[a_comp[t].func]);
            for (i = 0; i < a_comp[t].arity - 1; i++) {
                term_write(a_comp[t].args[i]);
                printf(", ");
```

```
            }
            term_write(a_comp[t].args[i]);
            printf(")");
        }
    }
}

void guard_write(Clause *cl)
{
    int i;
    for (i = 0; i < cl->guard_num - 1; i++) {
        term_write(cl->guard[i]);
        printf(", ");
    }
    term_write(cl->guard[i]);
}

void body_write(Clause *cl)
{
    int i;
    for (i = 0; i < cl->body_num - 1; i++) {
        term_write(cl->body[i]);
        printf(", ");
    }
    term_write(cl->body[i]);
}

void clause_write (Clause *cl)
{
    term_write(cl->head);
    printf(" :- ");
    guard_write(cl);
    printf(" | ");
    body_write(cl);
    printf(" .\n");
}

/* environment */
void allocate_env(int:goal num, int:goal *env)
{
    Term_i temp_term;
    int i;
    int sum_num = 0;
    sum_num += num;
    temp_term = allocate_term(sum_num);
    for (i = temp_term; i < sum_num; i++) {
        a_tag[i] = T_VAR;
        a_var[i].num = -1;
        a_var[i].ref = -1;
    }
    env[0] = temp_term;
    env[0] += scan(num, 0, CMC_combiner_add, CMC_upward,
                   CMC_none, CMC_no_field, CMC_exclusive);
}
```

```
Term_i allocate_term(int num)
{
    int old_term = current_term;
    current_term += num;
    return old_term;
}

void allocate_goal(int sum, int:goal *place)
{
    static int current_place = 0;
    int i;
    for (i = 0; i < sum; i++) {
        place[i] = current_place++;
    }
}

/* sub function */
int:goal h_unify(Term_i:goal t1, Term_i:goal t2)
{
    int:goal r;
    Tag:goal tag_t1, tag_t2;
    Var:goal var_t1, var_t2;
    Comp:goal comp_t1, comp_t2;
    Tag:goal p_tag;
    Var:goal p_var;
    Comp:goal p_comp;
    write_to_pvar(&p_tag, a_tag, boolsizeof(Tag:goal));
    write_to_pvar(&p_var, a_var, boolsizeof(Var:goal));
    write_to_pvar(&p_comp, a_comp, boolsizeof(Comp:goal));
    tag_t1 = [t1]p_tag; tag_t2 = [t2]p_tag;
    var_t1 = [t1]p_var; var_t2 = [t2]p_var;
    comp_t1 = [t1]p_comp; comp_t2 = [t2]p_comp;

    while (|= ((tag_t1 == T_VAR) && (var_t1.num >= 0)))
        where ((tag_t1 == T_VAR) && (var_t1.num >= 0))
            t1 = env[var_t1.num];
    while (|= ((tag_t2 == T_VAR) && (var_t2.num >= 0)))
        where ((tag_t2 == T_VAR) && (var_t2.num >= 0))
            t2 = env[var_t2.num];

    while (|= ((tag_t1 == T_VAR) && (var_t1.ref >= 0)))
        where ((tag_t1 == T_VAR) && (var_t1.ref >= 0))
            t1 = var_t1.ref;
    while (|= ((tag_t2 == T_VAR) && (var_t2.ref >= 0)))
        where ((tag_t2 == T_VAR) && (var_t2.ref >= 0))
            t2 = var_t2.ref;

    where (((tag_t1 == T_VAR) && (var_t1.name == anon))
           || ((tag_t2 == T_VAR) && (var_t2.name == anon))) {
        /* どちらかが無名変数の場合 */
        r = 1; /* TRUE */
    }
    else where (tag_t2 != T_VAR) {
        /* ヘッド側が変数以外の場合 */
        where (tag_t1 == T_VAR) {
```

```
            /* ゴール側が変数の場合 */
            r = 0; /* FALSE */
        }
        else where ((tag_t1 == T_ATOM) && (tag_t2 == T_ATOM)) {
            /* どちらもアトムの場合 */
            r = (t1 == t2);
        }
        else where ((tag_t1 == T_COMP) && (tag_t2 == T_COMP)
                    && (comp_t1.func == comp_t2.func)
                    && (comp_t1.arity == comp_t2.arity)) {
            /* どちらも複合項の場合 */
            int i;
            int:goal ended;
            int:goal arity = comp_t1.arity;
            r = -1;
            for (i = 0, ended = 0;; i++) {
                where (i >= arity) ended = 1;
                if (&= ended) break;
                where (!ended) {
                    where (!h_unify(comp_t1.args[i], comp_t2.args[i])) {
                        r = 0; /* FALSE */
                        ended = 1;
                    }
                }
            }
            where (r == -1) r = 1; /* TRUE */
        }
        else
            r = 0; /* FALSE */
    }
    else {
        /* ヘッド側が変数の場合 */
        var_t2.ref = t1;
        r = 1; /* TRUE */
    }
    return r;
}

int:goal b_unify(Term_i:goal t1, Term_i:goal t2)
{
    int:goal r;
    Tag:goal tag_t1, tag_t2;
    Var:goal var_t1, var_t2;
    Comp:goal comp_t1, comp_t2;
    Tag:goal p_tag;
    Var:goal p_var;
    Comp:goal p_comp;
    write_to_pvar(&p_tag, a_tag, boolsizeof(Tag:goal));
    write_to_pvar(&p_var, a_var, boolsizeof(Var:goal));
    write_to_pvar(&p_comp, a_comp, boolsizeof(Comp:goal));
    tag_t1 = [t1]p_tag; tag_t2 = [t2]p_tag;
    var_t1 = [t1]p_var; var_t2 = [t2]p_var;
    comp_t1 = [t1]p_comp; comp_t2 = [t2]p_comp;

    while (|= ((tag_t1 == T_VAR) && (var_t1.num >= 0)))
```

```
    where ((tag_t1 == T_VAR) && (var_t1.num >= 0))
        t1 = env[var_t1.num];
while (|= ((tag_t2 == T_VAR) && (var_t2.num >= 0)))
    where ((tag_t2 == T_VAR) && (var_t2.num >= 0))
        t2 = env[var_t2.num];

while (|= ((tag_t1 == T_VAR) && (var_t1.ref >= 0)))
    where ((tag_t1 == T_VAR) && (var_t1.ref >= 0))
        t1 = var_t1.ref;
while (|= ((tag_t2 == T_VAR) && (var_t2.ref >= 0)))
    where ((tag_t2 == T_VAR) && (var_t2.ref >= 0))
        t2 = var_t2.ref;

where (((tag_t1 == T_VAR) && (var_t1.name == anon))
       || ((tag_t2 == T_VAR) && (var_t2.name == anon))) {
    /* どちらかが無名変数の場合 */
    r = 1; /* TRUE */
}
else where ((tag_t1 == T_ATOM) && (tag_t2 == T_ATOM)) {
    /* どちらもアトムの場合 */
    r = (t1 == t2);
}
else where ((tag_t1 == T_COMP) && (tag_t2 == T_COMP)
            && (comp_t1.func == comp_t2.func)
            && (comp_t1.arity == comp_t2.arity)) {
    /* どちらも複合項の場合 */
    int i;
    int:goal ended;
    int:goal arity = comp_t1.arity;
    r = -1;
    for (i = 0, ended = 0;; i++) {
        where (i >= arity) ended = 1;
        if (&= ended) break;
        where (!ended) {
            where (!b_unify(comp_t1.args[i], comp_t2.args[i])) {
                r = 0; /* FALSE */
                ended = 1;
            }
        }
    }
    where (r == -1) r = 1; /* TRUE */
}
else where (tag_t1 == T_VAR) {
    /* t1 が変数の場合 */
    where (t1 == t2)
        r = 1; /* TRUE */
    else
        var_t1.ref = t2;
        r = 1; /* TRUE */
}
else where (tag_t2 == T_VAR) {
    /* t2 が変数の場合 */
    var_t2.ref = t1;
    r = 1; /* TRUE */
}
```

```
    else
        r = 0; /* FALSE */
    return r;
}

int:goal g_commit(Term_i:goal guard[], int:goal num) /* incomplete */
{
    int i;
    for (i = 0, ended = 0;; i++) {
        where (i >= num) ended = 1;
        if (&= ended) break;
        where (!ended) {
            where (guard[i] != true)
                return 0; /* FALSE */
        }
    }
    return 1; /* TRUE */
}

Result:goal commit(Clause:goal cl)
{
    newenv_num = cl.var_num;
    allocate_env(newenv_num, newenv);
    where (h_unify(term, cl.head) && g_commit(cl.guard, cl.guard_num)) {
        return R_USER;
    }
    return R_NOCOMMIT;
}

void solve (Clause *goal_cl)
{
    int:goal ended;
    int goal_place;
    Tag:goal tag_term;
    Comp:goal comp_term;
    Tag:goal p_tag;
    Comp:goal p_comp;

    with (goal) {
        /* 初期ゴールの設定 */
        allocate_goal(1, allocate_place);
        goal_place = [0]allocate_place[0];

        where (pcoord(0) == goal_place) {
            env_num = goal_cl->var_num;
            allocate_env(env_num, env);
            term = goal_cl->body[0];
            state = S_ACTIVE;
        }

        /* 解く */
        while (|= (state == S_ACTIVE)) {
            where (state != S_FINISH) {

                /*                      */
```

```
/* 組み込み述語かどうか */
/*                      */
write_to_pvar(&p_tag, a_tag, boolsizeof(Tag:goal));
write_to_pvar(&p_comp, a_comp, boolsizeof(Comp:goal));
tag_term = [term]p_tag;
comp_term = [term]p_comp;

where (term == true) {
    /* 組み込み述語 true/0 の処理 */
    state = S_FINISH;
}
else where ((tag_term == T_COMP)
            && (comp_term.func == unify)
            && (comp_term.arity == 2)) {
    /* 組み込み述語 unify/2 の処理 */
    where (b_unify(comp_term.args[0], comp_term.args[1]))
        state = S_FINISH;
    else
        state = S_SUSPEND;
}
else {
    /* 組み込み述語でない場合   */

    /*                          */
    /* 節を選ぶことが出来るか？ */
    /*                          */

    /* まず、プログラム中の節の位置を見つける */
    clause_start = predicate[term].start;
    clause_num = predicate[term].num;
    /* コミット出来るか試す */
    for (i = 0, ended = 0;; i++) {
        where (i >= clause_num) ended = 1;
        if (&= ended) break;
        where (!ended) {
            clause_temp = program[clause_start+i];
            result = commit(clause_temp);
            where (result != R_NOCOMMIT) {
                chosen_clause = clause_start+i;
                ended = 1;
            }
        }
    }

    /*            */
    /* その結果が */
    /*            */

    where (result == R_USER) {
        /* ユーザ定義述語の処理 */
        body_num = program[chosen_clause].body_num;
        sum_body_num = 0;
        sum_body_num += body_num;
        allocate_goal(sum_body_num, allocate_place);
        total_body_num = scan(body_num, 0, CMC_combiner_add,
```

```
                    CMC_upward, CMC_none, CMC_no_field, CMC_exclusive);
                for (i = 0, ended = 0;; i++) {
                    where (i >= body_num) ended = 1;
                    if (&= ended) break;
                    where (!ended) {
                        int:goal ended;
                        int i;
                        current_place = allocate_place[total_body_num + i];
                        [current_place]env_num = newenv_num;
                        for (i = 0, ended = 0;; i++) {
                            where (i >= newenv_num) ended = 1;
                            if (&= ended) break;
                            where (!ended) {
                                [current_place]env[i] = newenv[i];
                            }
                        }
                        [current_place]term = program[chosen_clause].body[i];
                        [current_place]state = S_ACTIVE;
                    }
                }
                state = S_FINISH;
            }
            else where (result == R_NOCOMMIT) {
                state = S_SUSPEND;
            }
            else {
                if (|= result)
                    error("internal error!!");
            }
        }
    }
}

/*                              */
/* もう active なものがないとき */
/*                              */

if (&= (state != S_SUSPEND)) {
    /* suspend なものがないとき */
    fputs("success!!\n", stdout);
    for (j = 0; j < [goal_place]env_num; j++) {
        printf("_%d : ", j);
        term_write([goal_place]env[j]);
        printf("\n");
    }
}
else {
    /* suspend なものがあるとき */
    fputs("deadlock!!\n", stdout);
    for (j = 0; j < MAX_GOAL; j++) {
        if ([j]state == S_SUSPEND) {
            term_write([j]term);
            printf("\n");
        }
    }
```

```
        }
    }
}

/* MAIN */
int main()
{
    Clause cl;
    int place = 0;
    int i, j;

    with(goal) {

        /* init */
        for (i = 0; i < MAX_TERM; i++) {
            a_tag[i] = -1;
            a_atom[i] = 0;
            a_var[i].name = -1;
            a_var[i].num = -1;
            a_var[i].ref = -1;
            a_comp[i].func = -1;
            a_comp[i].arity = 0;
            for (j = 0; j < MAX_ARGS; j ++) {
                a_comp[i].args[j] = -1;
            }
            predicate[i].start = -1;
            predicate[i].num = 0;
        }
        for (i = 0; i < MAX_PROGRAM; i++) {
            program[i].head = -1;
            program[i].guard_num = -1;
            for (j = 0; j < MAX_GUARD; j++) {
                program[i].guard[j] = -1;
            }
            program[i].body_num = -1;
            for (j = 0; j < MAX_BODY; j++) {
                program[i].body[j] = -1;
            }
            program[i].var_num = -1;
        }

        true = allocate_term(1);
        a_tag[true] = T_ATOM;
        a_atom[true] = "true";
        unify = allocate_term(1);
        a_tag[unify] = T_ATOM;
        a_atom[unify] = "unify";
        anon = allocate_term(2);
        a_tag[anon] = T_ATOM;
        a_atom[anon] = "_";

        for (;;) {
            clause_read(&cl);

            if (cl.head == -1) {
```

```
                printf("?- ");
                body_write(&cl);
                printf("\n");

                if (cl.body_num != 1) {
                    error("goal must be only one!!");
                }
                solve(&cl);
                break;
            }
            else {
                clause_write(&cl);

                if ((place > 0)
                    && ([0]program[place-1].head != cl.head)
                    && ([0]predicate[cl.head].num != 0)) {
                    error("program error!!");
                }

                program[place].head = cl.head;
                program[place].guard_num = cl.guard_num;
                for (i = 0; i < cl.guard_num; i++) {
                    program[place].guard[i] = cl.guard[i];
                }
                program[place].body_num = cl.body_num;
                for (i = 0; i < cl.body_num; i++) {
                    program[place].body[i] = cl.body[i];
                }
                program[place].var_num = cl.var_num;

                if ([0]predicate[cl.head].num == 0) {
                    predicate[cl.head].start = place;
                }
                predicate[cl.head].num++;

                place++;
            }
        }
    }
    return 0;
}
```