

Inside KLIC
Version 1.0

KLIC Task Group/AITEC/JIPDEC
関田 大吾 (sekita@mri.co.jp)

1998 年 5 月

(C) 1998 Japan Information Processing Development Center

本文書は JIPDEC が著作権を有します。利用/配布条件は、以下に述べられている通りです。ただし、ICOT(財団法人新世代コンピュータ技術開発機構)と書かれている箇所はすべて JIPDEC/AITEC(財団法人日本情報処理開発協会/ 先端情報技術研究所)と読みかえてください。

ICOT 無償公開ソフトウェアの利用条件

1. ICOT 無償公開ソフトウェアの目的

財団法人新世代コンピュータ技術開発機構(以下、ICOT という)は、日本国通商産業省より委託され、第五世代コンピュータ・プロジェクトを推進してきた。また、平成5年度からは、このプロジェクトの後継プロジェクトとして、第五世代コンピュータの研究基盤化プロジェクトを推進している。第五世代コンピュータ・プロジェクトおよびその後継プロジェクト(以下、これらの一連のプロジェクトを本プロジェクトという)は、並列推論処理を中核メカニズムとする新しいコンピュータの基礎技術を創出し、その知見と技術を世界の研究者と共有することによって、コンピュータ科学の発展に貢献することを目的としている。

本プロジェクトによって、並列推論マシン、並列推論ソフトウェア技術といった新しい技術が開発され、また、こうした技術開発に伴い、多くの先進的なソフトウェアが試作されている。これらのソフトウェアは、基礎的な研究開発段階にあるため、多くの研究者に広め発展させていくべきものである。

そこで、ICOT は、本プロジェクトの国際貢献の目的に鑑み、著作権が国ではなく ICOT に帰属することとなるこれらの研究開発段階のソフトウェアを、「ICOT 無償公開ソフトウェア」として公開してきた。これらのソフトウェアについては、研究開発のための障害となるいっさいの制約をはずすことによって、多くの研究者の方々に自由に利用してもらい、新しいコンピュータ科学への貢献を实践したいと考えている。

本プログラム及びドキュメント(以下、本プログラムという)は、「ICOT 無償公開ソフトウェア」の一つとして、ICOT において無償で配布しているものである。

2. 使用、変更、複製、配布の自由

本プログラムの利用者は、その使用、変更、複製を自由に行うことができる。ここでいう変更には、本プログラムの機能、性能、品質を向上させるために改良、拡張を行うこと、もしくは自ら開発したプログラムやドキュメントを本プログラムに追加することが含まれるが、それだけには限定されない。

本プログラムの利用者は、本「ICOT 無償公開ソフトウェアの利用条件」第3項(「無保証」)が記されていることを条件として、関連法令に違反しない限り、本プログラムそのもの、または本プログラムの変更版を第三者へ自由に配布することができる。

3. 無保証

本プログラムは、本プロジェクトの研究開発の試作物を『あるがまま』の状態を提供するものである。このため、明示的であるか黙示的であるか、または法令の規定により生ずるものであるか否かを問わず、一切の保証をつけないで提供されるものである。ここでいう保証とは、プログラムの品質、性能、市場性、特定目的適合性、および他の第三者の権利への無侵害についての保証を含むが、それに限定されるものではない。

本プログラムの利用者は、本プログラムが無保証であることを承諾し、本プログラムが無保証であることによるすべてのリスクを利用者自身で負うものとする。

従って、利用者が本プログラムを利用したこと、または利用できないこと、もしくは本プログラムを利用して得られた結果に起因する一切の損害について、著作権者である ICOT および本プログラムの開発に関与した関連機関並びにそれらの役職員及び従業員は、そのような損害の発生する可能性について、知っていたか否かにかかわらず、何らの責任も負わない。本プログラムの利用者は、本プログラムの利用を開始したことによりこれを承諾しているものとみなされる。ここでいう利用とは、本プログラムの使用、変更、複製、配布、二次的著作物の作成を含むがこれらに限定されない。

利用者が本プログラムそのもの、または本プログラムの変更版を、ICOT 以外の第三者から配布を受けた場合においても、配布を行った第三者が独自に特別な保証を文書で行わない限り、配布を行った第三者は、その利用者に対して、本プログラムに関係する限りにおいて同様に何らの責任を負わないものとする。

目次

第 1 章	はじめに	4
第 I 部	逐次処理系	5
第 2 章	データ構造	6
2.1	KL1 レベルのデータ構造	6
2.2	タグ関連のマクロ	10
2.3	ジェネリック・オブジェクト	12
2.4	実装レベルのデータ	16
第 3 章	KLIC の基本的な動作	26
3.1	動作モデル	26
3.2	トップレベルループ	27
3.3	コンパイルドコード	28
3.4	例外的な処理	32
3.5	例	37
第 4 章	割り込み/中断/失敗	53
4.1	例外処理	53
4.2	中断、失敗処理: <code>interrupt_goal</code>	56
4.3	Timer	58
第 5 章	単一化器	61
5.1	Passive unification	61
5.2	Active unification	62
5.3	単一化器: <code>do_unify()</code>	62
第 6 章	ジェネリック・オブジェクト	70
6.1	ジェネリック・オブジェクトの概略	70

第 7 章	GC	77
7.1	基本的な GC のアルゴリズム	77
7.2	GC の対象領域	78
7.3	Copy されたかどうかの判定	78
7.4	GC のきっかけ	79
7.5	GC のためのデータ構造	79
7.6	GC のアルゴリズム	79
7.7	copy_one_queue()	81
7.8	copy_terms()	81
第 8 章	トレーサ	84
8.1	機能概要	84
8.2	提供機能の概要	84
8.3	リンク時のトレース指定	85
8.4	名前情報の管理	86
8.5	トレースの制御と情報入手	87
第 II 部	並列処理系	95
第 9 章	概要	96
9.1	両版の違い	96
9.2	ノードの表現	96
9.3	逐次版とのインターフェース	96
9.4	両者の切りわけ	97
第 10 章	メッセージ通信版	98
10.1	処理系の構成	99
10.2	基本方式	99
10.3	詳細な実装	114
第 11 章	共有メモリ並列実装	120
11.1	概要	120
11.2	処理系の構成	120
11.3	共有ヒープのガーベジコレクション	124
索引		130

第 1 章

はじめに

この文書は、並列論理型言語 KL1 の一実装である KLIC の内部の実装について記述した文書である。基本的には KLIC-3.002 版に沿って記述されている。

この文書は、(KLIC 上の)KL1 の言語仕様を熟知していることを前提に記述されており、言語仕様については記述されていない。よって、KL1 の言語仕様については以下のような文書であらかじめ知っておくことが望まれる。

- KLICj.info (KLIC と共に配布されるマニュアル)

また、KLIC の利用法については、上記の文書や AITEC が発行している KLIC 講習会資料等を参照することが望まれる。

本文書は大きく以下の 2 つの部分に分かれている。

逐次処理系: KLIC の実装のうち、逐次処理系にのみ関わる部分。

並列処理系: KLIC の実装で、並列処理系を実装するに必要な、上記に追加されている部分。

KLIC の並列処理系の実装は、殆どの部分は逐次処理系に負っており、ほぼ、ノード間通信に必要な部分のみを追加したものである。よって、並列処理系について知りたい場合でも、逐次処理系より順に読みすすむことが望ましい。特に、第 2 章 (6 ページ)、第 3 章 (26 ページ) については、ごく基本的事項について書かれているため、まず最初に読むことを推奨する。

なお、文中のファイル名は、全て KLIC ソース配布のルートディレクトリからの相対指定で記述されている。

なお、逐次版の記述の殆んどは関田 (三菱総合研究所: sekita@mri.co.jp) が行ない、AITEC/JIPDEC の KLIC TG メンバー、拡大 KLIC TG メンバーの手で修正、追加等が行なわれた。並列版の記述については KLIC TG メンバーの過去の記述を流用させていただき、今回、KLIC TG メンバー、拡大 KLIC TG メンバーの手で修正、追加等が行なわれた。基本的に文責は関田にあり、問い合わせなどについては sekita@mri.or.jp あてに願います。

第 I 部

逐次処理系

第 2 章

データ構造

KLIC で用いられるデータ構造の実装について述べる。

KLIC で用いられるデータ構造は以下の 2 つに大別される。

- KL1 レベルに現れるデータ構造。KL1 のいわゆる「項」であり、シンボル、整数、リスト、構造体、ベクタ、文字列などがある。
- 実装レベルで用いられ、KL1 レベルでは出現しないデータ構造。ゴール、ゴールスタック、など。

またこれらの中間として、KL1 レベルでの構造を実装するための枠組として「ジェネリック・オブジェクト」という機構が KLIC では導入されている (12 ページ、第 2.3 章参照)。この機構を用いると、ユーザはカーネルに手を入れずに新しいデータ型を導入することができる (但し、KL1 のパーザに手を入れることはできず、KLIC 附属の標準のパーザではこのジェネリック・オブジェクトで新規に定義したデータ構造を文字列表現からメモリ表現に変換することはできない)。実際、KL1 レベルで現れるデータ構造のうち、ベクタ、文字列などはこのジェネリック・オブジェクトの機構を用いて実装されている。

本章ではこれらのデータ構造について説明を行う。KL1 のデータ構造の殆どは、*include/klic/struct.h* 内で定義されている。

2.1 KL1 レベルのデータ構造

KL1 という言語はいわゆる「動的型付」の言語であり、動的にデータの型をプログラム中で判定することができる。このようなデータを実現するためには、個々のデータに「データ型」を表現する印を付けておく、ということが一般的に行われる。この「印」のことを「タグ」と呼ぶ。

KL1 のデータ構造は 1 ワードが単位であり、この 1 ワード毎にすべてタグが付加されている。図 2.1 に、`foo(bar, [0,1])` のメモリ中での表現を図示する。図中 “:” で区切られている文字列はそのワードの“タグ:値”であることを表す。“:” で区切られていないものはポインタであり、タグだけが書かれている。

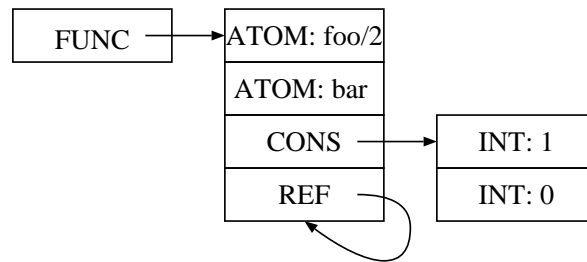


図 2.1 KL1 の項の例

表 2.1 タグとデータ型

タグ (2 進)	型
00	CONS, ファンクタ以外への参照 (REF)
01	CONS への参照
10	アトミックデータ
11	ファンクタへの参照

2.1.1 ワード

KL1 の 1 ワードは基本的に 32bit であり、全てのヒープに置かれるデータはみなこの整数倍で表現される。64bit 計算機では 1 ワードは 64bit 確保され、データを格納するためには 60bit が利用されている (ポインタの表現のためには当然 64bit 全てを利用している)。

この 1 ワードは *include/klic/struct.h* 内で、`q` という名称の型として定義されており、これは架空の構造 `gazonk` を指すポインタとして定義されている (この架空の構造を指すポインタ型としている理由は、ポインタを手廻りすぎた場合、コンパイラが知らない型を検出してエラーとすることを利用した、バグ除けである)。この `q` 型のデータを今後 “セル” と呼ぶことにする

おのこのセルはワード境界に置かれるようにしているため、これらのセルへのポインタは下位 2bit は必ず 0 になっている。この下位 2bit をタグとして用いている。タグとデータ型との関係を表 2.1 に記す。

さらに、アトミックなデータ、すなわち、記号アトム (以下アトム)、整数についてはもう 2bit (つまり下位 4bit) をタグとして用いている (表 2.2 参照)

よって、KLIC での整数値は 32bit 計算機では 28 ビットしかなく、通常の 32bit 整数値を格納すると溢れる。64bit 計算機では 60bit まで用いることができる。

なお、これまでの解説で明かなように、アトムは、実装レベルでは、28 ビット (タグ付き 32bit) のデータにエンコードされて実装されており、そのコードと実際の印字表現との関係は、別途表 (*include/klic/atomstuffs.h* 中の配列 `atomname`) にて管理される。

表 2.2 アトム型のタグ

タグ (下位 4bit2 進)	型
0010	整数
0110	アトム
1010	予約
1110	予約

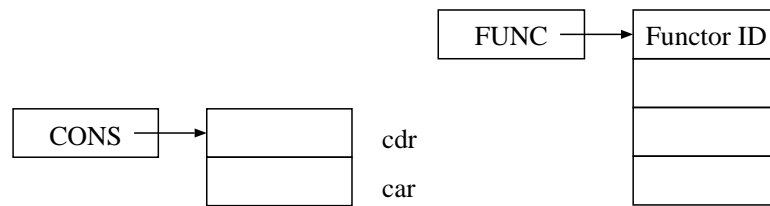


図 2.2 CONS とファンクタ

2.1.2 CONS とファンクタ

整数、アトムの場合にはそのタグ付きの値が 1 つのセルにおさまるため、そのまま格納する。CONS、ファンクタでは、複数セルが必要になるため、実体は別に領域を確保し、その実体へのポインタを各々 CONS、FUNC タグ付きのポインタをセル内に格納する。つまり、ポインタに「この先の連続領域は CONS/ファンクタである」と記述してある (図 2.2 参照)。

CONS は連続した 2 ワードの領域であり、CDR が下位ワード CAR が上位ワードにある^{*1}。C のレベルでは以下のように表現されている。

```
struct cons {
    q cdr, car;
};
```

ファンクタは (引数個数 +1) ワードの領域を確保する。

```
struct functor {
    q functor;                /* principal functor as atomic q object */
    q args[1];                /* arguments */
};
```

^{*1} これは、一般的に CAR より CDR に純粋未定義変数 (9 ページ、第 2.1.3 章参照) が割り付けられる可能性が高いこと、KLIC では純粋未定義変数は自己参照の形式で表現されるため、CDR を未定義参照にする処理に対して offset が加わらない方が高速な実現が期待できることに依る。

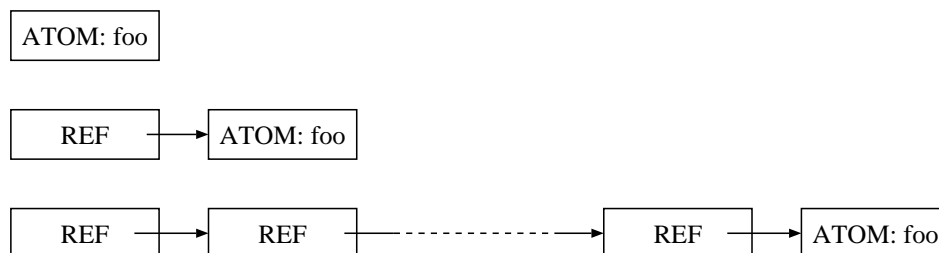


図 2.3 参照セルの例

ポインタが指す最初のセルには、「ファンクタ ID」と呼ばれるデータが記述されている。このファンクタ ID により、当該ファンクタの主ファンクタと引数個数がわかるようになっている（これらの 2 つデータを 1 ワードにエンコードしてある、と見なすことができる）。より具体的には、このファンクタ ID は、線型配列である「ファンクタ ID 表」（*include/klic/functorstuffs.h* 内の変数 **functors** により表現される）への配列インデックスであると見なすことができる。正確にはファンクタ ID にはアトムの 4bit タグが付加されており、上位 28bit が上記の表のインデックスである。このファンクタ ID の 4bit タグは重要である。というのは、ここが REF タグになってっている場合にはジェネリック・オブジェクト（の一種のデータオブジェクト（14 ページ、第 2.3.2 章参照））であることを表わしているからである。

2.1.3 参照セルと純粋未定義変数セル

論理型言語で特徴的な実装レベルのデータとして、「参照セル」（以下 REF セル）がある。これは、言語仕様上は全く現われない、「不可視の」データである。例えば、図 2.3 に示されるデータ構造は全て共に ATOM: foo として扱われる。このデータには REF なるタグが付加される。図にも現われる連続する REF のつながり（言語仕様上は全く意味のないデータ群）を REF 鎖と呼ぶ。

このようなデータが必要な理由は、論理型言語特有な操作である「単一化」、および「論理変数」と「具体化」を実装するためである。まず、論理変数を実装するためには、まだいかなる項にも具体化していない項、「未定義変数」を実装する必要がある。この未定義変数の実装のために独立したタグを用いたりする流儀もあるが、タグ節約も兼ね、KLIC では REF タグで、「自分自身」を参照した構造を用いる（図 2.4(1) 参照）。この構造を「純粋未定義変数」と以下で呼ぶ。

今、 $X = [a]_-$ で、 Y は純粋未定義変数であることが保証されている時に、 $X=Y$ なる操作を行ったとする。すると、純粋未定義変数 Y は $[a]_-$ に具体化されるが、実装レベルでは、これは純粋未定義変数 Y のポインタ部分が X を実現するデータを指すように書き換えられる、ということが行われる（図 2.4(2) 参照）。ここで挙げたような具体化操作がなんども行われると、先に示した REF の連鎖ができることもある。

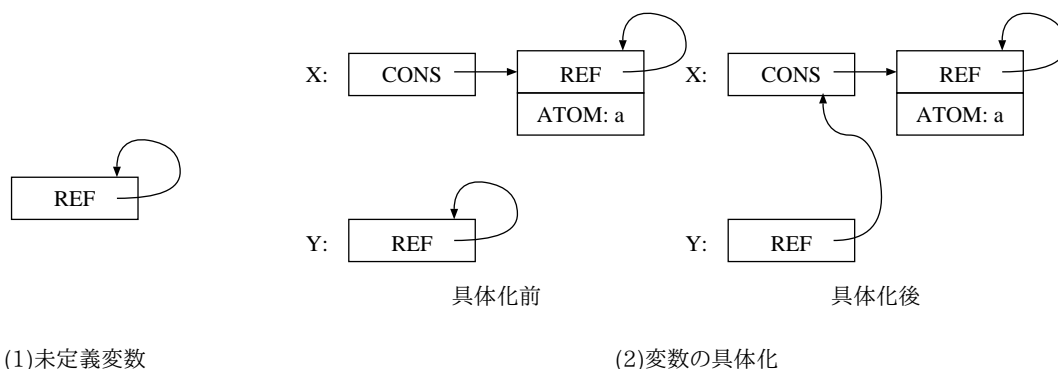


図 2.4 純粹未定義変数の実装と単一化

2.1.4 中断ゴールを現わす構造

KL1 では、具体化されていない変数の値をコミット時に読み出しをしたゴールは中断をすることになっている。中断したゴールに対しては、読み出しを試みた未定義が具体化した時には実行を再開される必要がある。中断要因の変数に対して、再開されるべきゴール群 (1 つの変数が複数のゴールにより読み出され、結果的に複数のゴールを中断させることはありえる) がすぐに判明することが望ましい。

以上の要件を実現するため、KLIC では、中断要因変数から中断しているゴールレコードを容易に手繰ることができるようになっており、言語上は判別できない純粹未定義変数と、すでにあるゴールの中断の原因になっている未定義変数とは実装上は区別される。

中断ゴールを現わす具体的な構造については、第 2.4.5 章 (22 ページ) を参照のこと。このように、中断対象となった変数に中断するゴールレコードを関連付けることを「ゴールをフックする」と称する。

2.2 タグ関連のマクロ

KLIC のごく低レベルの処理で、これまでに説明してきた、タグ自身、また、タグを扱う操作に関しては、マクロ化されており、記述性、可読性の向上、操作の統一を計っている。

Inline 文、ジェネリック・オブジェクトの記述など、C レベルで KL1 のデータ構造をアクセスする際には、表 2.3, 2.4 に挙げるマクロを利用して記述することが推奨される。

`include/klic/struct.h` で記述されている主な関連マクロを表 2.3 に纏める。

また、CONS、ファンクタの要素をアクセスするためのマクロを表 2.4 に纏める。

表 2.3 タグ関連のマクロ一覧

マクロ	意味
PTAGBITS	ポインタデータに付加されているタグの bit 長
PTAGMASK	ポインタデータに付加されているタグ部のマスク
VARREF	REF のタグ値
CONS	CONS への参照のタグ値
ATOMIC	アトミックデータのタグ値
FUNCTOR	ファンクタへの参照のタグ値
ptagof(x)	x のポインタタグ部分を得る
isatomic(x)	x がアトミックデータであることの判定
iscons(x)	x が CONS への参照であることの判定
isfunctor(x)	x がファンクタへの参照であることの判定
isref(x)	x が参照であることの判定
isstruct(x)	x が CONS またはファンクタへの参照であることの判定
functnotcons(x)	x が構造であるときに CONS でないことの判定
atomicnotref(x)	x が構造でないときにアトミックデータであることの判定
ATAGBITS	アトムデータに付加されているタグの bit 長
ATAGMASK	アトムデータに付加されているタグのマスク
INT	整数タグ
SYM	アトムタグ
atagof(x)	x のアトミックタグを得る
isint(x)	x が整数データかどうか検査する
issym(x)	x のアトムデータかどうか検査する
makeint(n)	整数 n を KL1 整数データにする
makesym(x)	x のアトム番号により KL1 アトムデータを生成する
makeatomic(x)	x にアトミックタグ (つまり 2bit のみ) を付加する
makeref(x)	x のポインタにより KL1 REF データを生成する
makecons(x)	x のポインタにより KL1 CONS データを生成する
makefunctor(x)	x のポインタにより KL1 functor データを生成する
intval(x)	KL1 整数データ x から整数値を得る
symval(x)	KL1 アトムデータ x からアトム番号を得る
derefone(x)	KL1 REF データ x の先を一段手繰る

表 2.4 CONS, ファンクタの要素をアクセスするためのマクロ

マクロ	意味
<code>functor_of(s)</code>	s の指すファンクタの主ファンクタ
<code>arg(s, k)</code>	s の指すファンクタの第 k 引数
<code>car_of(x)</code>	x の指す CONS の CAR
<code>cdr_of(x)</code>	x の指す CONS の CDR
<code>pcar_of(x)</code>	x の指す CONS の CAR を指す REF セルの内容
<code>pcdr_of(x)</code>	x の指す CONS の CDR を指す REF セルの内容

2.3 ジェネリック・オブジェクト

KLIC では組み込みデータ型である基本データ型を、ポインタの下位数ビットを使って表現する方式にしたため、その組み込みデータ型の種類に限りがあり、一般的なユーザが必要とするデータ型を全てそれらでサポートしているわけではない。また、ユーザが必要とするデータ型を自分で定義してシステムに組み込んだり、他言語とのインターフェイスを確保したいという要求に答える機能は、KLIC の目指す汎用的な言語システムにとって必要であろう。これらの問題を解決し、言語システムの容易な拡張を実現するものとして、ジェネリック・オブジェクトなる機構を導入した。

ジェネリック・オブジェクトとして実現されたデータは、単一化や GC など様々な局面での自分自身の処理の仕方を記述したメソッドを集めたメソッド表、及びデータ領域から成る。データの領域のサイズや構成は、オブジェクトクラスにより定義され、KLIC システムの実行時カーネルは感知しない。このメソッド表がオブジェクトのクラス定義に当たるものである。なお、メソッドは実際には C の関数により実装されている。

ユーザはメソッド表に含めるべきメソッド、オブジェクト生成のためのルーチン、データ領域の構成などを、ジェネリック・オブジェクト定義用に作成したマクロや関数を使用しつつ、C 言語で記述することで、ジェネリック・オブジェクトを定義する。ジェネリック・オブジェクトのデータ領域は KL1 の組み込みデータ型や C で定義可能なデータ型から構成することが可能であり、様々なデータをジェネリック・オブジェクトに持たせることが出来る。

ジェネリック・オブジェクトの定義に当たっては、KLIC システムの実行時カーネルに一切手を入れる必要はない。KLIC の組み込みデータ型のデータが KLIC システムの実行時カーネルによって処理されるのに対し、ジェネリック・オブジェクト型のデータでは、実行時カーネル及び KL1 プログラムからの要求に応じ、オブジェクトが自分で処理を行う。実行時カーネルは、ジェネリック・オブジェクトのメソッドの中身やデータ領域の構成については何も知らず、従って扱い方も知らない

現在、KLIC に組み込みとして備えられているデータ型のうち、以下のようなものはこのジェネ

リック・オブジェクトの機構により実装されている。

- 浮動小数点数
- コード、モジュール
- ベクタ
- 文字列

また、以下の、プロセスのインターフェースを持つオブジェクトも、ジェネリック・オブジェクトにより実装されている。

- マージャ オブジェクト
- ファイル I/O オブジェクト
- 乱数発生器

さらに、並列化のために導入されている、KL1 レベルのユーザには全く見えないようなデータ構造などについてもジェネリック・オブジェクトにより実装されている。

さらに、この枠組を利用することにより、ユーザは新しい機能を KLIC に導入することも可能である。

2.3.1 ジェネリック・オブジェクトの種類

ジェネリック・オブジェクトは実行時カーネルから見た場合、(少なくとも top level について)、以下のように分類できる。

- 具体化されているデータ
- 値が決定すれば、行われる計算が決められている変数 (ゴールをフックしている変数に類似)
- 値が必要ななら、値を求める方法が定められている変数

分類の各々に応じて、以下の呼ばれるオブジェクトの種別が用意されている。

Data object 具体化されているデータと同様の扱いを行うべき オブジェクト である。

Consumer object 値が決められればそれに応じて計算が行われるオブジェクト である。

- 書き込みを行うことにより、なにか処理が行われる。
- 読み出すことによってはなにも処理は行われない。つまり、純粹未定義変数と同じ動作をする。

Generator object 今までの KL1 処理系には (少なくとも言語表面には) なかったデータ型。

- 書き込みを行うことにより、なにか処理が行われる (Consumer object と共通)。
- 読み出すことによって、何か値を決めるための処理が行われる。

つまり、generator は cosumer を包含した機能を持つものである。

メソッド表の構成、他の (つまり、通常の KL1 の) データ構造との関係は、これら 3 種類で各々

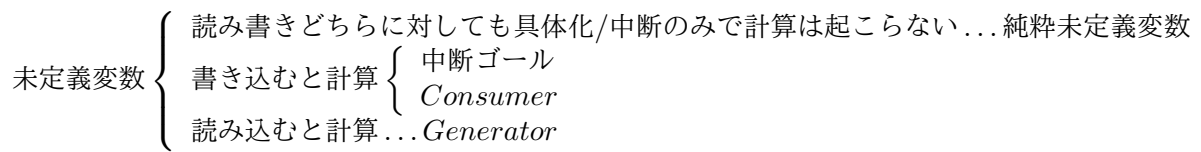


図 2.5 未定義変数の分類

異なる。

なお、consumer, generator は、言語表層上では、あくまで「未定義変数」であり、具体値ではない。よって、「未定義変数」には、純粋未定義変数以外に、これらのオブジェクトも含まれる。さらに、次章で述べる「中断要因となっている未定義変数」も未定義変数である。言語上は「未定義変数」となるデータ構造についての実装の一覧を図 2.5 に挙げる。

2.3.2 ジェネリック・オブジェクトのデータ構造

Data object

Data object を表現するデータ構造を、図 2.6 に示す。

ファンクタタグのポインタの先にオブジェクトの本体がある。第 1 ワードはメソッド表へのポインタで、第 2 ワードからオブジェクトのボディ（データ領域）が始まる。同じファンクタタグで指されるファンクタ型では、第 1 ワードはアトミックタグになっている。一方、data object では、REF タグになっており、両者の区別を行うことができる。

この data object のメソッド表の構造は、`include/klic/g-methtab.h` 内で、`data_object_method_table` なる名称で定義されている。

Consumer

Consumer object を表現するデータ構造を、図 2.7 に示す。

consumer object の本体は、中断構造 (22 ページ、第 2.4.5 章参照) の中に、その変数に suspend

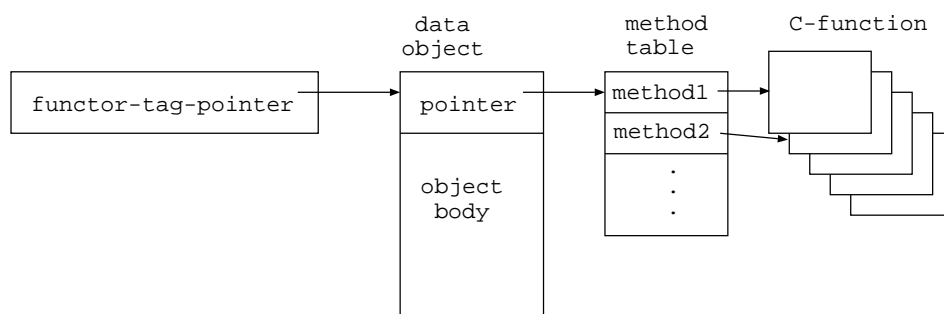


図 2.6 data object の実装

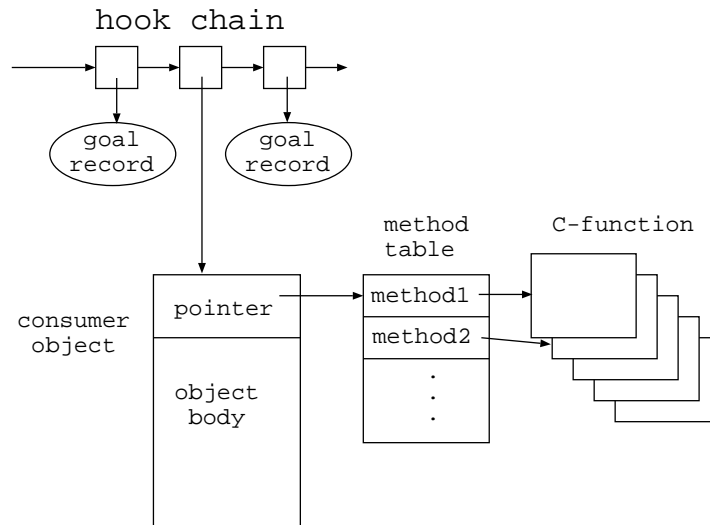


図 2.7 consumer object の実装

しているゴールに混ざって並んでいる。つまり、consumer object はフックされた変数の顔をしている。この変数に対する active unification が起き、中断構造 (22 ページ、第 2.4.5 章参照) 中のゴールがエンキューされていく中で、consumer object に対して active unify メソッドが発行される。この局面、及び GC 局面以外の局面では、consumer object は hook したゴールと類似の扱いを実行時カーネルから受ける。

この構造については第 2.4.5 章 (22 ページ) でさらに詳細に説明するが、ここであげた中断構造 (22 ページ、第 2.4.5 章参照) は、中断原因である未定義変数より特殊な構造 (susprec) を経由して指されている。この「中断原因の変数セル」と susprec とは互いに参照しあう「二重ループ構造」になっている。

Consumer object のメソッド表の構造は、*include/klic/g_methstab.h* にて `consumer_object_method_table` として定義されている。

Generator

Generator のデータ構造を図 2.8 に記す。Generator は未定義変数より、特別な構造を介して参照されている。

Generator も、consumer や中断ゴールと同様、「二重 REF ループ」によりまず検出される。Consumer, 中断ゴールとの最大の相違は、「1 つの変数について複数の generator, consumer, 中断ゴールなどがぶらさがることではない」ということである。つまり、generator は他のものと単一化が行われる場合、一度「具体化を試み」てから、その結果を単一化する (試みた結果、具体化されない、つまり、「未定義変数」のままである、ということはある)。よって、Generator がそのままの形で他のものと変数を共有することはない。

Generator のメソッド表の構造は、*include/klic/g_methstab.h* にて `generator_object_method_table`

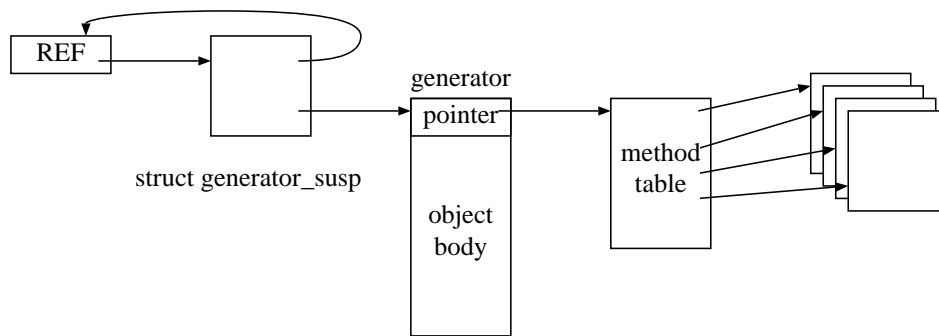


図 2.8 generator object の実装

として定義されている。

2.4 実装レベルのデータ

これまで述べたものは、KL1 のレベルでも「データ」として現れるものを表現する構造が主であった。以下では、KL1 のレベルでは陽にデータとしては出現しないようなものについて解説する。

すなわち、以下のような構造である。

- 大域データ構造体
- ヒープ
- モジュール
- 述語
- ゴールレコード
- 優先度ゴールスタック

2.4.1 大域データ構造体

`include/klic/struct.h` 中で、`global_variables` として定義されている構造体である。この構造体の実体は、各プロセスに 1 つ (よって、逐次版であれば、システム内に 1 つ) だけ存在する。

KLIC では、大域的に用いられるデータの殆どはこの構造体に格納されている。一般的に大域な変数をアクセスするためには、そのアドレス、すなわち、32bit 計算機では 32bit 長即値を扱う必要があり、この処理は通常効率があまり良くない (例えば、SPARC などの RISC プロセッサ、つまり、命令がすべて 32bit 長である場合には、即値のレジスタへのロードだけで 2 命令になってしまう)。そこで、大域データを構造体にいれておき、その構造体へのポインタをレジスタ上に持ち、個々の構造体要素データに関しては、offset で (つまり、32bit よりも短いデータで) アドレス表現できるようにすることにより、大域データへのアクセスコストを減らすことを念頭に置いている。

以下でこの構造体の説明を行う (各々の詳細は、個別におこなう)。

```
struct global_variables {
    q *heapp0; /* ヒープ割付点 */
    q * Volatile heaplimit0; /* heaplimit 変数 (詳細後述) */
    struct goalrec *current_queue0; /* 実行中優先度のゴールスタック */
    struct goalrec *resumed_goals0; /* リダクション中に生じた再開ゴール群 */
    unsigned long current_prio0; /* 現在の優先度 */
    unsigned long top_prio0; /* 実行可能ゴールでの最高優先度 (未使用) */
    struct prioqrec prioq0; /* 「優先度ゴールスタック」のリストの先頭 */
    q *heaptop0; /* 現在のヒープの先頭 */
    q *heapbottom0; /* 現在のヒープ面中のシステムヒープの割付点 */
    q *real_heaplimit0; /* heaplimit のマスター (詳細後述) */
    unsigned long heapsize0, maxheapsize0, incrementsize0;
    /* 現在の片面ヒープサイズ、最大ヒープサイズ、ヒープマージンサイズ (in word) */
    unsigned long real_heapbytesize0; /* ヒープサイズ (byte) */
    double maxactiveratio0; /* ヒープ中の最大アクティブセル比 */
    unsigned long this_more_space0; /* 最小の空領域 */
    q *new_space_top0, *old_space_top0; /* ヒープ新領域、旧領域の先頭 */
    unsigned long new_space_size0, old_space_size0;
    /* ヒープ新領域、旧領域のサイズ */
    q **gcstack0; /* GC で用いるスタックの底 */
    q **gcsp0; /* GC で用いるスタックの先頭 */
    q **gcmax0; /* GC で用いるスタックの上限 */
    unsigned long gcstack_size0; /* GC で用いるスタックのサイズ */
    Volatile long interrupt_off0; /* 割り込みフラグ */
    struct goalrec *interrupt_qp0; /* 未仕様 */

    struct { /* 並列実装用の構造 */
        /* parallel comm Imp. */
        long my_num0;
        long num_pes0;
        union {
            /* shared-memory Imp. */
            struct {
                long queued0;
                struct ex_goalrec* ex_qp0;
                long currid0;
                long oldid0;
                long shm_htop0;
                long shm_hbyte0;
                long dummy[10];
            } shm;
            /* dist-memory Imp. */
        };
    };
};
```

```

    } aux;
} par;

char *program_name0; /* コマンド名称 */
int command_argc0; /* コマンド引数個数 */
char **command_argv0; /* コマンド引数 */
q* (**gc_hook_table0)(); /* GC 時のフックの表 */
int gc_hooktab_size0;
int num_gc_hooks0;
q* (**after_gc_hook_table0)(); /* GC 後のフックの表 */
int after_gc_hooktab_size0;
int num_after_gc_hooks0;
unsigned long /* 計測用のカウンタ */
    suspensions0, resumes0, copied_susp0, cum_susps0, cum_resumps0;
struct suspended_goal_rec *suspended_goal_list0; /* デッドロック検出用の
                                                    中断ゴールリスト */
Const struct predicate *postmortem_pred0; /* post mortem 用の述語 */
q postmortem_args0;
long generic_argc0; /* ジェネリック・オブジェクトのインターフェース用の引数個数 */
q generic_arg0[MAXGENERICARGS]; /* ジェネリック・オブジェクトのインターフェース用
の引数領域 */
q reasons0[MAXSUSPENSION]; /* サスペンションスタック領域 */
}

```

この構造体は各プロセッサ内では、globals なる名称の変数として保持されている。構造体のこれらの要素は、以下のとりきめにより、*include/klic/struct.h* 中でマクロ化されている。

- 上記の globals なる変数へのポインタを glbl なる名称の局所変数として確保する。
これを行うマクロが `declare_globals` である。つまり、大域データにアクセスする関数に置いては、`declare_globals` をまず記述することが推奨される。
- 「実際の要素名の末尾の 0 をとったもの」で参照する。例えば、`globals.heapp0` は、`heapp` というマクロを利用することにより、`glbl->heapp0` として参照することができる。

2.4.2 ヒープ

KL1 は自動メモリ管理を前提とした言語であり、KLIC では、“stop and copy” 方式の GC を採用している。つまり通常の KL1 のデータはヒープに置かれ、適当な時に KL1 プログラムの実行を中断し一括的に回収される。特筆すべきことは、通常の KL1 データだけではなく、ゴールレコード、ゴールの中断に必要な付加的なデータなど、動的に増減するものの殆どはヒープ中に置かれており、通常の GC の一環で管理されていることである。これは処理系製作が楽になること、将来世代 GC などより効率的なメモリ管理方式を KLIC が採用するときに整合性が良いことが期待できること、といったことを念頭に置いての設計である。

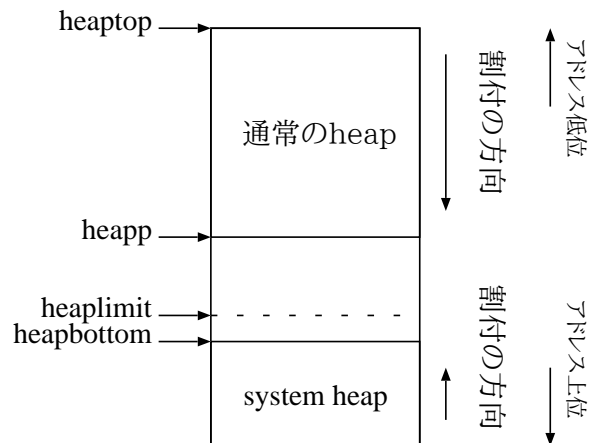


図 2.9 ヒープの利用

Copying GC であることからわかるように、ヒープは同じサイズのものが 2 面存在する。片方が一杯になったらもう一方にコピーをして利用面を切りかえる。

各面のヒープの利用状況を図 2.9 に記述する。

「通常のヒープ」はアドレス下位側から連続的に割付を行う。heap_limit に達するとヒープは一杯になったこととする。

「ヒープ溢れ」の検査を割付の度に毎回行うことは、頻繁にヒープ割付が行われる KL1 の実装としては速度、命令サイズの上で不利である。そこで、この検査は KL1 のリダクションの際に行うことにしている。このため、heap_limit の先に常に一定語数の「余裕」がヒープにあるように管理し、検査と検査の間にこの語数以下の割付しか行われないことを原則とすることにより、チェックの手間の軽減を計っている。

一方、通常のヒープに割付を行うためには、ヒープ割付点の値を知り、変更できる必要がある。このヒープ先頭のアクセスを高速にするために、この変数はレジスタに載せられることを期待して局所変数として持つこととし、できるだけ大域変数にアクセスすることを避けたい。よって、naive にはこの局所変数にのっているヒープ割付点の値をヒープ割付を行う可能性のある全関数に引数として渡し、戻り値として返させる必要がある。しかしながら、C 言語の仕様として戻り値として返すことができる値は 1 つであること、ヒープ割付をする「可能性」はあっても、実際に割付を行う頻度は低い関数が多いことより、例外的に割付を行うような関数にはこのヒープ割付点を渡さず、アドレス上位にある「システムヒープ」からメモリ割付を行うようにしている。このシステムヒープはアドレス上位より順に割り付けられる。システムヒープの割付を行ったら、それと同等の分だけ heap_limit をアドレス低位側に変更し、システムヒープとしては「常に余裕」があるようにして利用する。

GC を行うと、すべてのアクティブセル (その時点で有効であるデータセル) はシステムヒープにあったものも含め新面の通常のヒープ側にコピーされ、GC 終了時には新面のシステムヒープは空の状況になっている。

また、KLIC ではヒープのサイズは動的に変更されるようになっている。すなわち、GC をしても、ある一定の比率以上アクティブセルが存在するような場合には、ヒープを確保しなおし、ヒープを拡大する、という機構を備えている (この機構が動作することを禁止することも、実行時ファイルの起動時に指定できる)。

各々の heap 面についての値は、前述の `global_variables` に設定されている。すなわち、以下の通りである。

`heap`: ヒープへの新規割り付け点 (以下では「ヒープ割付点」と記述)

`heap_top`: 面の最低位アドレス

`heap_limit`: 通常ヒープの上限

`heap_bottom`: ヒープ全体の最上位アドレス

`real_heap_limit`: `heap_limit` のコピー

`heap_size`: 片面通常ヒープのサイズ (ワード数)

`max_heap_size`: 最大片面通常ヒープサイズ (ワード数)。これで示されるサイズを越えてヒープ自動拡張は行わない。

`increment_size`: 片面システムヒープのサイズ (ワード数)

`new_space_top`, `old_space_top`: GC 時の新/旧面の先頭 (通常実行時には、`new_space_top == heap_top` になっている)。

`new_space_size`, `old_space_size`: GC 時の新/旧面全体 (通常 + システムヒープ) のサイズ (バイト数)

2.4.3 モジュール

KL1 のモジュールは、KLIC 中では C 関数として実装されている。よって、C レベルでは、その関数へのポインタが「モジュール」を表現するものとして使われている。

この関数は関数型 `module` として定義されている。この `module` 型は、「`module` 型関数へのポインタを返す関数」として `include/klic/struct.h` 中で定義されている。よって、本来は、`typedef module *(module)()` のように定義することが望まれるが、このような自己再帰的な関数記述は C 言語では許されず、`typedef char *(module)()` と定義されている。なお、KL1 言語レベルで利用されるモジュール型データはこれをさらにジェネリック・オブジェクトで隠蔽したものである。

実際のモジュールを表現する関数は通常 KLIC のコンパイラが生成し、リンクする。また、関数のアドレスは、`long int` と同じ `alignment` となることが期待されている。つまり、下位 2bit は共にゼロであることが期待されている。

2.4.4 述語

KL1 の述語は、KLIC 中では *include/klic/struct.h* 内で定義されている、`predicate` 型として表現されており、これを「述語記述子」と呼ぶ。KL1 言語レベルで用いられる述語型データ型データは、これをさらに generic object で隠蔽したものである。

基本的には、「どのモジュールの、どの述語か」ということが記述されている構造である。

```
struct predicate {  
    module (*func)();           module 関数  
    unsigned short int pred;    述語 ID  
    unsigned short int arity;   引数個数  
};
```

なお、述語 ID とは、述語に対してモジュール内でユニークに付けられる番号である。よって、引数個数がなくとも func, pred なるペアで述語の特定は可能であるが、内部処理 (たとえば GC) によってはゴールレコードのサイズを知るため引数個数が必要になることがあり、ここに記録されている。

この構造体は静的、つまり、KLIC のコンパイラにより生成される。

2.4.5 ゴールレコード

述語に対し、適用される環境 (引数列) を持っているのがゴールレコードである。KLIC ではゴールレコードは、(通常の KL1 データを置くのと同じ) ヒープ領域に置かれており、GC の対象になる (回収は GC によってのみ行われる)。

この構造は *include/klic/struct.h* 内の、`goalrec` 構造により定義されている。^{*2}

```
struct goalrec {  
    struct goalrec *next;       次のゴールを指すためのリンク領域  
    Const struct predicate *pred; 述語記述子へのポインタ  
    q args[6];                 引数領域。  
};
```

ゴールスタックは上記構造の線型リストにより表現される。線型リストにせずにスタック状にゴールレコードを隣接して置くことは可能ではあるが、優先度別に複数のゴールスタックを管理する必要があり、各々を個別にメモリ管理することは繁雑であるということ、KL1 では中断処理があ

^{*2} 引数領域で [6] とあるのは、処理系デバッグの際に、GDB などの C レベルのデバッグ、中身を見るときにこの `goalrec` 構造体を印字することにより 6 引数程度は印字される、ということのためである。よって、プログラム実行上は意味がない。

るため、必ずしもゴール生成順と消滅順とは逆順 (LIFO の関係) にならないことに起因している。

上記の定義からは明らかではないが、ゴールレコードは、不定長の構造であり、引数の数に応じて末尾が伸縮する。つまり、args の領域は、pred で示される述語記述子内にある引数個数分だけ確保される。

このような実現をとらずにゴールレコードを原則固定長メモリ塊の組合せにより表現し、その固定長メモリ塊をヒープとは別にフリーリスト管理する、という実装も可能である (実際、初期の KLIC ではそのような実現をしていた)。しかしながら、この方法は、ゴールを組合せる処理が繁雑になること、純粋未定義変数をゴールレコード中に置くことが難しくなり、純粋未定義変数をゴールレコードに置かないよう工夫をすることが繁雑になることなどの理由により、現在の方式に改められた。

結果的に、ゴールレコードをヒープに置くことにより、不定長の構造の管理が大きく簡略化され、また、純粋未定義変数は直接ゴールレコード内に置いても何の配慮もいらなくなった。

このゴールレコードは以下の 2 つの状態を持つ。

- 実行可能状態
- 中断状態

これらの状態にあるゴールについて以下に記述する。

実行可能状態

`next` フィールドで、同じ優先度を持つゴール同士で線型リストを作る。よって、実行中には、一般的に複数のゴールのリストが作成される。そのうち一本は、大域情報の `current_queue` に置かれ、「実行対象」になる。実行時には、このリストは LIFO 的に扱われる。つまり、先頭より順に実行対象となり、エンキューも先頭になされる。

中断状態

中断状態にあるときのゴールは、第 2.10 章 (23 ページ) に示すような構造になっている。この図に表われる構造、およびそれらにアクセスするためのマクロ群は `susp.h` に記述されている。

まず、中断要因になっている純粋未定義変数を、`susprec` なる構造を指すようにセットする。この `susprec` の先頭は、この未定義変数を指させるようにする。つまり、中断要因変数と、`susprec` 先頭とは二重ループになっている。この `susprec` の 2 ワード目以降は、`hook` なる構造になっている。この `hook` 構造に中断中の `goalrec` 構造体がぶらさがる。この構造を以下では中断構造と呼ぶ。

また、中断時のゴールレコード (`goalrec`) の先頭 (`next` フィールド) には、そのゴールの優先度が `INT` タグ付きで格納されている。ゴールが再開可能になった時には、この `field` で示される優先度のゴールキューに格納される。

```
struct hook {  
    struct hook *next;
```

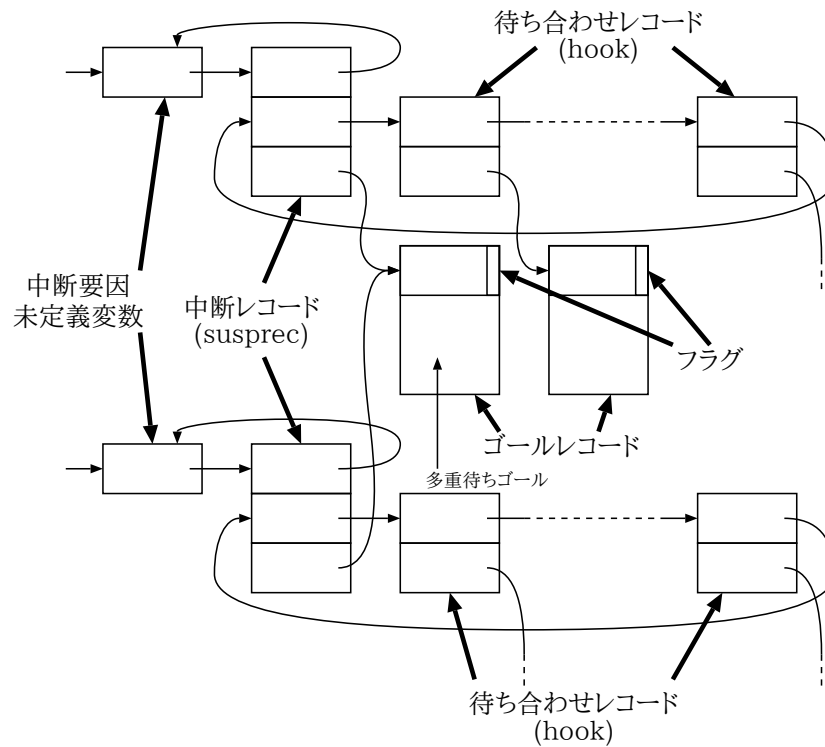


図 2.10 中断を表現するデータ構造

```
union goal_or_consumer {
    long l;
    struct goalrec *g;
    struct consumer_object *o;
} u;
};
```

```
struct susprec {
    q backpt;
    union {
        struct hook first_hook;
        long l;
    } u;
};
```

ひとつの未定義変数について、複数のゴールが中断する場合には、この hook の next フィールドに hook 構造体が線型リストになる。susprec 構造体に含まれる hook 構造体も含めた hook 構造体のリストは、中断原因を同じくする hook 構造体同士で環状のリストになる。つまり、末尾の

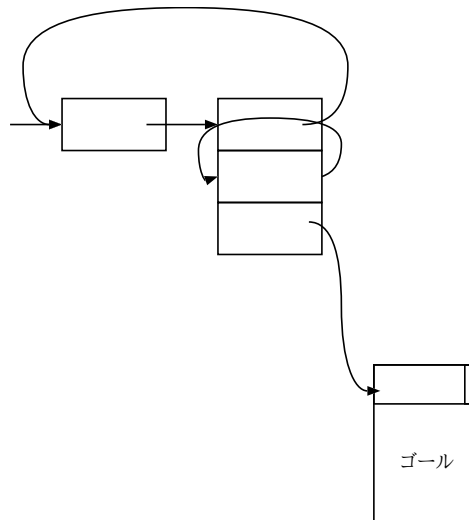


図 2.11 単一ゴールの際の中断構造

hook 構造体の next は、susprec に含まれている hook 構造体の hook 部分を指す。典型的には、1 つのゴールしか含まれない場合には図 2.11 で示すような形状になる。

これまでの図で「ゴール」がぶら下っている部分には、consumer object が来ることがある。この consumer object の構造は、data object の構造と類似であり、「通常の KL1 のデータ」部分に出現するか、「中断ゴール群」のなかに出現するか、だけが違っている。

ぶらさがっているのがゴールなのか、consumer なのかは、実行時に、hook 構造体の goal/consumer へのポインタを含む 2 ワード目 (つまり、hook.u.l) のタグ部で判定される。

また、1 つのゴールが複数の中断要因を OR 待ちする、いわゆる「多重待ち」の場合も基本的に構造は同じである。すなわち、ゴールは一般的に複数の hook 構造体より指されることがある。図 2.10 中の「多重待ちゴール」に多重待ちをしているゴールの例を挙げる。

ただし、多重待ちしているゴールがそのうちのつの要因が具体化されたため、再開可能になることはあり、その場合には他の要因をしめす変数からは hook 構造体を経由して指されたままになっている。このような状況の時に、再度、他の要因の具体化のため、1 つのゴールに対して、複数回「再開」処理がなされることを防ぐためには goalrec の next フィールドのタグ部分を参照すれば良い。すなわち、中断中はこのタグは INT タグが付けられているが、再開後には、通常の (goalrec を指す) ポインタとされ、INT タグは解除される。

これらの構造をアクセスするために `include/klic/susp.h` 内で定義されているマクロについて以下で概説する。

`is_consumer_hook(hook)`: hook が consumer であれば真、goal ならば偽。

`tag_consumer_hook(hook)`: hook に consumer タグを付ける。

`untag_consumer_hook(hook)`: hook の consumer タグを取りさる。

`suspp(x)`: x を (struct susprec *) に cast する。

`allocnewsusp(var, srec)`: 1 ワードセル 1 つ、および `susprec` を割りつけ、中断構造を作り、`var` に指させる。新しい `susprec` は `srec` より指すようにする。

`makenewsusp(var,srec,allocp)`: 1 ワードセル 1 つ、および `susprec` を割りつけ、中断構造を作り、`var` に指させる。新しい `susprec` は `srec` より指すようにする。`allocp` はヒープトップである。

`allochook(oldhook,newhook)`: `newhook` として新しい `hook` 構造体を割りつけ、`oldhook` の直後に指しこむ。

`addhook(oldhook,newhook,allocp)`: `newhook` として新しい `hook` 構造体を割りつけ、`oldhook` の直後に指しこむ。

第 3 章

KLIC の基本的な動作

本章では、KLIC での、KL1 プログラムの基本的な動作について説明をする。

3.1 動作モデル

KLIC での KL1 プログラムの動作モデルは以下の通りである。

1. 処理系は実行可能ゴールスタックから、1 つゴールを選択し、ガード実行を試みる (try 操作)。
2. ガード実行を試みた結果により、通常以下のどちらかの動作をする。
 - (a) ガード実行を試みた結果、そのゴールが中断することが判明した場合には、中断の原因となった未定義変数とゴールとを「ペア」にして記憶しておく (中断操作)。
 - (b) ガード実行できた場合には、ボディ実行を行う。その結果、以下のようなことが行われる可能性がある
 - 単一化 (単一化操作)。その結果、未定義変数の具体化により、中断ゴールが再開可能になる可能性がある。その場合には中断ゴールを実行可能ゴールスタックに入れる。
 - あらたなゴールのエンキュー (エンキュー操作)。
 - 実行の失敗 (失敗操作)。
- さらに、ガード実行を試みた結果、「失敗」する (どの候補節にもコミットしない) 可能性がある (ガード失敗操作)。
3. 以上を繰り返し、実行可能スタックにゴールがなくなれば、実行は終了 (top-level succeed 操作)。

以上のような処理の各々は、KLIC のコンパイラにより KL1 プログラムから生成されたコンパイルコード (すなわち module 型関数) と、KLIC の実行時ライブラリとが協調することにより実現されている。その概略を以下で説明する。

3.2 トップレベルループ

先に説明した動作モデルをごく単純化すると、「ゴールの選択」⇒「実行」⇒「ゴールの選択」...という「ループ」であることがわかる。これを KLIC のトップレベルループと呼ぶ。このループは *runtime/kmain.c* 内の *toploop* 関数により実現されている。

もう少し正確、詳細に解説すると、モジュールは *module* 型関数で実装されるが、この関数は以下のように動作する。このモジュール型の関数は KLIC のコンパイラで出力されるものであり、ユーザが定義したモジュール、システムで定義されているモジュール各々に一関数ずつある。

- この関数は、当該モジュール内で定義されているモジュール内の述語群の実行の連鎖が続く限り、*return* しない。
- 外部モジュールで定義されている実行の呼び出しが起きたり、割り込み処理が起きたりしたことにより、KL1 述語の実行以外の「例外的な処理」を行う必要が生じたときには *return* し、トップレベルループに戻る。

この関数が終了した後は、大域変数 *current_queue* にその時の実行可能 *queue* の先頭がセットされているので、それを *qp* に戻し、次のモジュールの実行を行う。

以下が *module* 型関数を呼び出す *toploop* 関数の全てである。

```
static void toploop()
{
    declare_globals;
    struct goalrec *qp = current_queue;
    Const struct predicate *toppred = qp->pred;
    module func = (module)toppred->func;
    while (1) {
        func = ((module (*)())func)(gbl, qp, heapp, toppred);
        qp = current_queue;
        toppred = qp->pred;
    }
}
```

current_queue とあるのが (現在実行中の優先度の) 実行可能ゴールスタックであり、*struct goalrec* の線型リストとなっている。先頭を *qp* にとりだし、述語レコード (*struct predicate*) をとりだし、モジュールを実現する関数 (*func*) をさらに取りだしている。そして、その関数を呼びだすことを繰り返す *while* ループに突入する。

ここで、*module* 型関数の引数は以下のようにになっている。

gbl: 大域データ構造へのポインタ (第 2.4.1 章 (18 ページ) 参照)

qp: 現在の実行優先度の実行可能キュー

heapp: ヒープ割付点

toppred: (この文脈で判明するように) これから実行する述語のレコード

3.3 コンパイルドコード

3.3.1 モジュールの原則

出力コードの原則として、モジュール全体の構造について解説する。

- 1 モジュール、1 関数。
- アトム ID、ファンクタについては別ファイル (各々 `atom.h`, `funct.h`) に (少なくとも) リンク対象となるモジュールに含まれるもの、KLIC の実行可能ライブラリで利用されているものが全て記述され、それを `include` する。
- コードは以下の様な順で記述されている。
 1. ヘッダの `include`。常に固定して、`runtime/klic/klicdr.h`, `runtime/atom.h`, `runtime/funct.h` が `include` される。
 2. 出力対象となるモジュール関数の `forward` 宣言がなされる。これはこの直後にて、必要なデータを出力するための措置である。モジュール関数の C レベルの名称は規則的にエンコードされるが、その規則については、第 3.4.2 章 (36 ページ) を参照のこと。
 3. 当該モジュールで定義されている述語の全述語構造体が大域変数の形で出力される。述語構造体の C レベルの名称は規則的にエンコードされるが、その規則については、第 3.4.2 章 (36 ページ) を参照のこと。
 4. この後、モジュール関数が出力される。
 - (a) モジュール関数の関数宣言がなされる。
 - (b) 最初に、「定数の構造体」のデータ構造が出力される。詳細は第 3.4.2 章 (36 ページ) 参照のこと。
 - (c) その直後に、KL1 で記述された内容を実行する C のコードが出力される。このコードは概要としては以下のような構造になっている。
 - まず、引数レジスタを模す、`a0`, `a1` といった変数が確保される。この変数は必要な分 (すなわち、当該モジュール中の述語の最大引数数) だけ確保される。
 - `switch_on_pred` なるマクロにより、述語間の `dispatch` を行うコードが出力される。
 - 以降に各述語を実現するコードが出力される。
 - 末尾 (ラベル `interrupt_N` 以降) に、モジュール関数を抜ける時の処理が出力される。「モジュール関数を抜けるとき」原因には、他のモジュールの述語の呼び出しの発生、割り込みや、メモリ不足 (を解消のため GC を行う) などがある。

```

<インクルード部分>
<モジュール関数のフォワード宣言>
<述語の構造体の定義>
module module_XXXX(glbl, qp, allocp, topred) /* モジュール関数 */
    ....
{
    <定数構造体、局所変数定義>
    module_top:
        switch_on_pred() {
            case_pred(0, main_0_top);
            case_pred(1, nrev_2_top);
            last_case_pred(2, append_3_top);
        }

    main_0_top: {
        <main/0 のコード>
    }

    nrev_2_top: {
        <nrev/2 のコード>
    }

    append_3_top: {
        <append/3 のコード>
    }
}

```

図 3.1 「モジュール」の構成

以上の説明の内容を、後述する例の出力コード図 3.5 全体を簡略化したコードにより図 3.1 に示す。

3.3.2 述語呼び出しの実現

述語の呼び出しは、大別して、以下の 2 通りの実装がある。

通常呼び出し: KLIC では述語を呼び出すためには、基本的に、

1. ゴールレコードを作成し、そこに述語情報、引数情報を書きこみ、ゴールスタックにエンキューする。これを必要な回数繰り返す。
2. 実行している述語が終了したならば、ゴールスタックよりゴールをデキューして、その述語を次の実行対象にする。

という手順が行われる。

直接呼び出し： 上記のような呼び出し手続を行う場合、基本的には、直前にエンキューした述語をデキューし実行することになる。よって、ゴールレコードのエンキューとデキューとが連続する場合には、ゴールレコード作成、エンキュー、デキューという処理を介さずに直接実行対象とするという最適化が考えられる。これを直接呼び出しと呼ぶ。

KLIC では、中断がなければユーザが記述した順にゴールを実行するようにコードを出力する。そのために、ユーザが記述した順と逆の順でゴールを準備し、エンキューしていくようなコードが出力される。よって、上記の直接呼び出しの最適化が行われる対象となるゴールは最後にエンキューされたゴール、すなわち、最初に記述されたゴールであり、この最適化を FGO(First Goal Optimization) と呼ぶ。

この FGO は、呼び出し側と同じ述語と同じモジュールに対しての呼び出しに対してのみ行っており、この場合には、C の goto 文によるジャンプが出力されるようになっている。C 言語では他の関数の中間に飛び込む、という機能はないため、「最初に記述されたゴール」が他モジュールの場合には FGO 最適化を行わず通常呼び出しを行う。

また、以上のような最適化を行っているため、例外的な場合 (32 ページ、第 3.4 章参照) を除き、当該モジュールだけで実行されている間はこの関数から return はされない。

なお、通常呼び出しのデキュー処理は、preceed() なるマクロで実現されている。また、直接呼び出しは execute なるマクロで実現されている。

3.3.3 述語の原則

次にある述語に着目した出力コードの原則について解説する。つまり、第 3.1 章 (29 ページ) にて `<nrev/2 のコード>` と記述した部分について解説する。図 3.5 の `nrev/2` の出力部分、すなわち、84 – 125 行目部分を模式化した図を図 3.2 に示す。

図 3.2 内に記述された各説明についての概説を以下に記述する。

通常呼び出しの入口： 述語が通常呼び出しされた場合には、モジュール関数の入口を経由し、`switch_on_pred` マクロを経由してこの部分より実行される。このあち、ゴールレコード内に格納されている引数を `a0, a1 ... an` といった変数に読みこむ。

特殊な直接呼び出しの入口： 通常、直接呼び出しの際には、この箇所に goto 文により飛び込む。直接呼び出しでは、ゴールレコードを用いずに、引数 (`a0...n`) は、すでに呼び出し側で用意しているため、引数の読み出し処理は必要ない。

この入口の直後では、`reasonp = reasons;` なる操作が行われる。`reasonp` は中断スタックの先頭を指すことになっている変数である。中断スタックとは、KL1 での「中断操作」を実装するために用いられているスタックである。KLIC でのガード実行中に、具体化されていることが必要であるデータがまだ具体化されていない場合には、この中断スタックにそのデータへのポインタを積む。当該述語の全ての節の実行可能性を確認した後、どの述語も実行することができない場合には、後述する「例外処理入口」に飛ぶ (これは、上記での

```

nrev_2_top: {                                <通常呼び出しの入口>
    q x0, x1, x2;
    a0 = qp->args[0];                        <ゴールレコード内の引数の読み出し>
    a1 = qp->args[1];
    qp = qp->next;
    nrev_2_clear_reason:                    <特殊の直接呼び出しの入口>
    reasonp = reasons;
    nrev_2_0:                               <通常な直接呼び出しの入口>
    nrev_2_1:
    switch (ptagof(a0)) {
        <nrev/2 のコード部分>
        .....
        .....
    }
    nrev_2_ext_interrupt:                   <直接呼び出し時の例外処理入口>
    reasonp = 01;
    nrev_2_interrupt:                       <例外処理入口>
    goto interrupt_2;
}

```

図 3.2 述語の構造

interrupt_2 に goto した後に行われる)。その結果、例外処理中でこの中断スタックが調査され、空かどうか検査を行う。空でなければ、積まれているデータが中断した原因であり、中断操作を行う。中断スタックが空であれば、中断する要因はなかったことになり、「fail 操作」が行われることになる。

この「特殊な直接呼び出しの入口」の直後ではこの中断スタックを空にする操作が行われる。つまり、直接呼び出しを行う時点で中断スタックが空ではない可能性がある場合にはこの部分に飛びこむようなコードが出力される。

通常の直接呼び出しの入口: ここに飛び込むのは、直前の述語 (直接呼び出しを行う述語) で直接呼び出しを行った時点で中断スタックが空であるような場合にここに飛び込むコードが出力される。KL1 のかなり多くの述語では、「中断要因が積まれた (=中断スタックが空ではない)」にも関わらず述語の実行ができることはなく、多くの場合には直接呼び出し時にはこちらの方を呼び出すことができる。これができない (つまり、中断スタックが空ではない状態で次の述語を直接呼び出しする可能性がある) のは多重待ちをしている場合である。

nrev/2 のコード部分: 実際の nrev/2 のコードが出力される。これらのについては、第 3.5 章 (38 ページ) を参照のこと。

直接呼び出し時の例外処理入口: この部分は、当該述語 (nrev/2) が他の述語を直接呼び出しする際に、例外が発見された (例えば、ヒープが溢れた場合。詳細は第 3.4 章 (32 ページ) を参照) 場合にはこの部分に飛びこまれる。その結果 reasonp が 0 に初期化され、例外処理で判

定され、当該処理が行われる。

例外処理入口: 述語のガード部で失敗または中断したことが発見された場合には、この部分に飛び込まれる。その結果、`reasonp` は中断スタックの底、またはいくつかデータが積まれた状態の先頭を指しており、例外処理 (32 ページ、第 3.4 章参照) で判定され、当該処理が行われる。

3.3.4 データのキャッシュ

プログラム実行上、アクセス頻度の高い大域変数は、できるだけローカル変数にキャッシュしてからアクセスし、大域変数へのアクセスはできるだけ少なくなるようにしている。アクセス頻度の高いローカル変数は最近のプロセッサ、コンパイラではレジスタに置かれる可能性が高く、その結果、出力される native な命令が単純になり、アクセスも高速化する。

特に、`alloca`(ヒープ割付点)、`qp` (ゴールスタック先頭) については留意されており、通常にモジュール関数を実行している間には殆んど大域変数への書き戻しなく実行される。

また、コンパイルコードより、実行時ライブラリを呼び出す場合があり、その実行時ライブラリ内でヒープ割付を行ったり、ゴールをエンキューしたりする時がある。そのような実行時ライブラリについては、naive には上記のキャッシュされている値を大域変数に書きもどすか、引数により値を渡し、戻り値、または大域変数経由で戻すようにする必要がある。KLIC では、`qp`、`alloca` 各々について以下のように対処している。

- `qp` に対しては、例えば、単一化を実行時ライブラリで行った結果中断中のゴールが実行可能になった際にエンキューされるなどによりアクセスされる可能性があるが、基本的には実行時ライブラリでアクセスされる可能性は低い。そのため、`qp` は実行時ライブラリには渡さないこととした。その代りに、「例外的なエンキュー用のゴールスタック」を設け、実行時ライブラリ内で、エンキュー、デキューされたゴールを保持するようにした。この例外的なゴールスタックはリダクションの切れ目で検査され、空でない場合にはその保持しているゴール群を通常のゴールスタックに再度エンキューすることとした (55 ページ、第 1 章参照)。
- `alloca` に対しては、比較的アクセス頻度が高いと思われる実行時ライブラリには引数で渡し、戻り値、または大域変数 `heapp(glbl->heapp0)` を経由して戻すこととした。
また、アクセス頻度が低いと思われる実行時ライブラリでは、`alloca` を渡さずに、システムヒープ (19 ページ、第 2.4.2 章参照) に割り付けを行うこととした。

3.4 例外的な処理

KLIC では、基本的に、先 (26 ページ、第 3.1 章参照) に述べた動作モデルのように、KL1 で書かれたコードを実行しつつける。しかしながら、以下に記述するような理由により、「例外的な処

理」を行う必要が生じるときがある。

- ヒープ溢れによる GC 処理
- 現在よりも、より高い優先度を持ったゴールの再開可能化
- 割り込み (タイマ、I/O など)

これらの「例外的処理」は必要に応じて、「述語実行の切れ目」で行う。つまり、1つの述語が終了し、次の述語を実行する間に例外的処理を行う必要があるかどうかを判断し、もし必要であれば、必要な処理 (例えば、GC) を行う。

KLIC では、1つのゴールのリダクション途中では、様々な変数 (一時変数、引数変数、ヒープ割付点など) が更新された状態にあり、GC 等を含み得るような処理をするために必要なこうした変数の退避は難しく、効率も悪いため、述語の実行中は「他の処理」(27 ページ、第 3.2 章参照) を行わず、述語実行に専念するようにしている。

よって、述語実行中に「例外的処理」が必要になった場合には (例えば、割り込みハンドラで) 「次の述語実行の切れ目でわかるように」覚えておくだけで、すぐに通常の述語実行を継続する。そして述語実行の切れ目では、「例外的処理」を行う必要があるかどうか検出し、必要あれば、他の処理を行う。

この「例外的処理」要求の検出が、`allocp` と `heaplimit` の比較である。`allocp` はヒープ割付点であり、`heaplimit` は、ヒープの上限である。つまり、通常のヒープ溢れはこの比較で検出され、GC が起動される。他の「割り込み」については、人為的 (一時的) にこの `heaplimit` を 0 にすることにより割込まれたことを記憶しておく、という方式をとっている。したがって、この大小比較のみで、「例外的処理」要求有無の検出ができる。

この例外処理は、モジュール関数の末尾で呼び出される実行時ライブラリ、`interrupt_goal` で行われる。この内部では `reasonp` の値により、どのような例外が起きたのか判定され、適切な処理が行われる。これらの処理をひきおこすコード箇所については、第 3.2 章 (31 ページ) を参照のこと。

他にすべき処理が生じた時: この場合には、`reasonp` には 0 がセットされている。この場合には、`resumed_goals` に `goal` を格納し、他の処理をすべく `return` する。

述語が「失敗」したとき: この場合には中断スタックポインタは中断スタックの底を指している。失敗の処理 (関数 `do_fail()`) が呼び出される。

述語が「中断」しそうなとき: これまでの説明で判明したように、KLIC では、3 段以上 REF の連鎖がある場合にはコンパイルコード中では追跡をあきらめ、`interrupt_goal` の呼び出しを行っている。これは、通常それほど長い REF 鎖はあまりないこと。その結果、コンパイルコード量が増加するわりには、実行速度向上がないことに起因している。

この場合には、「中断原因の候補」は中断スタックに積まれているので、それを調べ、候補が全て真に未定義変数 (含む中断構造) であれば中断処理を行う。候補のいずれかが具体値であることが判明すれば、`resume_same_prio()` 関数により、当該ゴールを「再開」する (つ

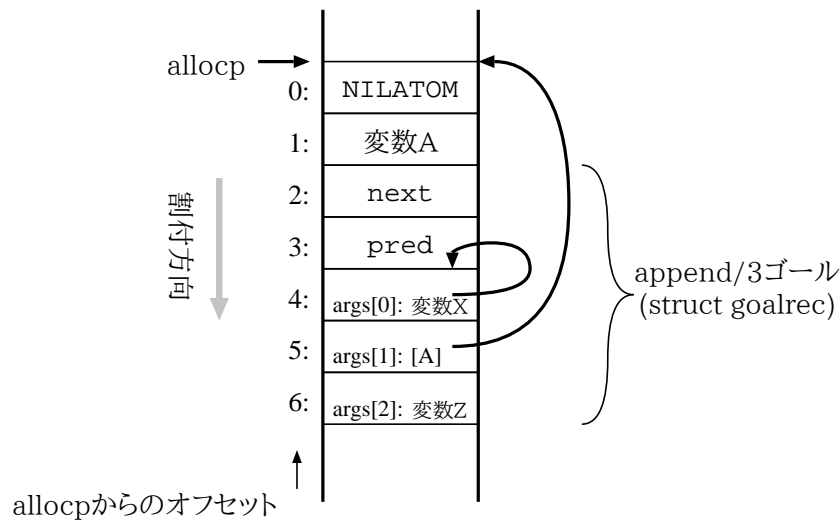


図 3.3 ヒープ割付の説明

まり、エンキューしてしまう)。

3.4.1 ヒープの使い方

KLIC のコードでは、以下のようにヒープを利用する。

- ゴールレコードは、ヒープ上に確保する。
- KL1 データはヒープ上に確保する。

通常の述語実行中は、`allocp` なるポインタを介してそこからの相対アドレスをもってヒープアクセスをする。述語開始時には `allocp` はヒープ割付点を指しており、そこから相対アドレスで正の方向は空領域である。具体的には、コード中では `allocp[x]` ($x \geq 0$) なる記述でヒープアクセスが行われる。

例えば、以下の、`nrev/2` の 1 節の出力コードを例にとりヒープの割付の解説を行う。

```
nrev([A|X], Y) :- nrev(X, X1), append(X1, [A], Y).
```

この節のコードは、前述の 84 行目より 109 行目に出力されているが、ここでは以下のようにヒープを利用している (図 3.3 参照)。以下でヒープ割付に関連する部分のみコンパイルコードを掲載するとともに解説する (全体のコードは第 3.5 章 (37 ページ) に再掲する)。

- まず `[A]` を作成する。このためには `allocp[0 - 1]` が用いられる (95 - 97 行目)。

```
95     allocp[0] = NILATOM;
96     x1 = car_of(a0);
```

表 3.1 変換の例

KL1 での名前	エンコーディング後
foo	foo
foo_bar	foo__bar
'foo<>bar'	foo_3C_3Ebar

```
97    allocp[1] = x1;
```

- 次に、allocp[2 – 6] を用いて、append/3 のゴールレコードが作成され、引数が準備される (99 – 103 行目)。

```
99    allocp[2] = (q)qp;
100   allocp[3] = (q)(&predicate_main_xappend_3);
101   allocp[4] = x2 = makeref(&allocp[4]);
102   allocp[5] = x0;
103   allocp[6] = a1;
```

- 最後に、これまで利用されてきたヒープ領域を確保するため、allocp を 7 ワード足しこむ (107 行目)。

```
107   allocp += 7;
```

3.4.2 エンコーディング規則

モジュール関数の名称、述語構造体の名称は、基本的に KL1 レベルの各々の定義に用いられた名称等を C で用いられる名前にエンコードすることにより実装している。そのエンコード規則についてここで記述する。

基本規則

基本的に、KL1 言語レベルのモジュール名、述語名 (これは KL1 のアトムである) は、以下のようにエンコードされる。

- KL1 レベルの名前で、[A-z0-9] についてはそのまま。
- KL1 レベルの '_' は、 '__' に変換される。
- 上記にあてはまらない文字は、'_HH' と 3 文字に変換される。HH の 2 文字は元の文字の文字コードをあらわす 2 桁の 16 進文字列で、0-9,A-F の 16 文字のうち何れのうち 2 文字。

表 3.1 にいくつか例をあげる。

モジュール関数名エンコード規則

モジュール関数名は、以下のようにエンコードされる。

`module_ + KL1` モジュール名を基本規則によりエンコードした文字列

たとえば、`foo` モジュールは、“`module_foo`” となり、`foo_bar` モジュールは、“`module_foo__bar`” となり、`'foo<>bar'` モジュールは “`module_foo_3C_3Ebar`” となる。

述語名エンコード規則

述語名エンコード規則は以下のようにになっている。

`predicate_ + KL1` モジュール名を基本規則によりエンコードした文字列
+ `_x + KL1` 述語名を基本規則によりエンコードした文字列
+ `_ + 引数個数`

定数構造体のコンパイル

ソースコード中に、全ての要素が定数であるような構造体が出現することがある。これを定数構造体と呼ぶ。例を以下に示す。

```
[1, a, 10, [2, x]]
foo([1,2], bar, 3)
```

プログラムを実行する際には、最終的にこれらの構造をメモリ中に置くこととなる。Naive には、この構造の組み立ては動的に行う（つまり、定数でなかった場合と同様に行う）が、すべてが定数である場合には、あらかじめ静的に構造を作成しておくことが可能であり、これは、以下の意味での最適化になる。

- コード実行時に動的に構造を構築する時間を節約できること。
- 幾度も同じ構造を必要とするようなプログラムの場合（例えば、`foo(X) :- X=[1,2,3]` なる述語で、`foo(X)` が幾度も呼びだされる場合）、静的に 1 つ確保することによりメモリを節約できることがあることがある（静的に確保してしまうため、GC の対象にはならず、領域が不要となっても領域の回収はできないため、必ずしも常に節約になるとは限らない）。
- さらにコンパイラ/アセンブラ/ローダが（KL1 では、一度具体化されたデータは書きかわることがないため、この領域は変更されることがないため）この領域をプロセス間で共有できる領域に置く可能性があり、同時に同じプログラムが動作する（たとえば共有メモリ実装での KLIC の実行）場合、メモリの節約になること。

例えば、`foo([1,2,3])` という構造は以下のようにメモリ中に置かれる。

```
static Const q cons_const_0[] = { /* (1) */
    NILATOM,
```

```

    makeint(3),
};
static Const q cons_const_1[] = { /* (2) */
    makecons(cons_const_0),
    makeint(2),
};
static Const q cons_const_2[] = { /* (3) */
    makecons(cons_const_1),
    makeint(1),
};
static Const q funct_const_3[] = { /* (4) */
    makesym(funcutor_foo_1),
    makecons(cons_const_2),
};

```

C では、参照される変数を参照する側よりも先に定義する必要があるため、ボトムアップに出力される。

まず (1) で、[3] が q 型の配列 cons_const_0 として出力されている。makeint によりタグを付けていること、先に CDR が出ていることに注意。

次に (2) で、その [3] へのポインタに cons タグを付加し (makecons)、CDR とし、さらに makeint(2) を CAR として、[2,3] が作成される。同様に、(3) で、[1,2,3] が作成される。

最後に (4) では、foo に対してのファンクタ ID と、[1,2,3] へのポインタを並べることにより、foo([1,2,3]) が生成される。実際にこの構造を利用する時には、makefunc(funct_const_3) としてコード中で参照される。

以上で判明するように、定数構造体については、その実体はヒープ中にとられない。KLIC の GC では、ヒープ外へのポインタ (REF, CONS, FUNC タグ付きのもの) を発見したときには、そのポインタをコピーするだけで、その先はコピーの対象としない。よって、定数構造体については GC 時にコピーがなされない。つまり、長期間参照されるような定数構造体については、GC 時にコピーをしない、という最適化も行われていることになる。

3.5 例

以下のような KL1 プログラム “naive reverse” を例として説明をする。

```

:- module main.

main :-
    nrev([1,2,3], X),
    klicio:klicio([stdout(normal([putt(X),nl]))]).

```

```

nrev([], Y) :- Y=[].
nrev([A|X], Y) :- nrev(X, X1), append(X1, [A], Y).

append([], Y, Z) :- Y=Z.
append([A|X], Y, Z) :- Z=[A|Z1], append(X, Y, Z1).

```

このコードを KLIC コンパイラでコンパイルすると以下のような C コードになる。

```

1  /* Compiled by KLIC compiler version 3.002 (Wed Sep 17 15:37:13 JST 1997) */
2  #include <klic/klichdr.h>
3  #include "atom.h"
4  #include "funct.h"
5
6  module module_main();
7  Const struct predicate predicate_main_xmain_0 =
8      { module_main, 0, 0 };
9  Const struct predicate predicate_main_xnrev_2 =
10     { module_main, 1, 2 };
11  Const struct predicate predicate_main_xappend_3 =
12     { module_main, 2, 3 };
13  extern Const struct predicate predicate_klicio_xklicio_1;
14
15  module module_main(glbl, qp, allocp, toppred)
16     struct global_variables *glbl;
17     struct goalrec *qp;
18     register q *allocp;
19     Const struct predicate *toppred;
20  {
21     static Const q cons_const_0[] = {
22         NILATOM,
23         makeint(3),
24     };
25     static Const q cons_const_1[] = {
26         makecons(cons_const_0),
27         makeint(2),
28     };
29     static Const q cons_const_2[] = {
30         makecons(cons_const_1),
31         makeint(1),
32     };
33     static Const q cons_const_3[] = {
34         NILATOM,
35         makesym(atom_nl),
36     };

```

```

37     q a0, a1, a2;
38
39     q *reasonp;
40 module_top:
41     switch_on_pred() {
42         case_pred(0, main_0_top);
43         case_pred(1, nrev_2_top);
44         last_case_pred(2, append_3_top);
45     }
46
47 main_0_top: {
48     q x0, x1, x2, x3, x4, x5;
49     qp = qp->next;
50 main_0_clear_reason:
51     reasonp = reasons;
52 main_0_0:
53     allocp[0] = (q)qp;
54     allocp[1] = (q)(&predicate_main_xnrev_2);
55     allocp[2] = makecons(cons_const_2);
56     allocp[3] = x0 = makeref(&allocp[3]);
57     allocp[4] = makesym(functor_putt_1);
58     allocp[5] = x0;
59     x1 = makefunctor(&allocp[4]);
60     allocp[6] = makecons(cons_const_3);
61     allocp[7] = x1;
62     x2 = makecons(&allocp[6]);
63     allocp[8] = makesym(functor_normal_1);
64     allocp[9] = x2;
65     x3 = makefunctor(&allocp[8]);
66     allocp[10] = makesym(functor_stdout_1);
67     allocp[11] = x3;
68     x4 = makefunctor(&allocp[10]);
69     allocp[12] = NILATOM;
70     allocp[13] = x4;
71     x5 = makecons(&allocp[12]);
72     allocp[14] = (q)(struct goalrec*)&allocp[0];
73     allocp[15] = (q)(&predicate_klicio_xklicio_1);
74     allocp[16] = x5;
75     qp = (struct goalrec*)&allocp[14];
76     allocp += 17;
77     proceed();
78 main_0_ext_interrupt:
79     reasonp = 0l;
80 main_0_interrupt:
81     goto interrupt_0;

```



```

82  }
83
84  nrev_2_top: {
85      q x0, x1, x2;
86      a0 = qp->args[0];
87      a1 = qp->args[1];
88      qp = qp->next;
89  nrev_2_clear_reason:
90      reasonp = reasons;
91  nrev_2_0:
92  nrev_2_1:
93      switch (ptagof(a0)) {
94  case CONS:
95      allocp[0] = NILATOM;
96      x1 = car_of(a0);
97      allocp[1] = x1;
98      x0 = makecons(&allocp[0]);
99      allocp[2] = (q)qp;
100     allocp[3] = (q)(&predicate_main_xappend_3);
101     allocp[4] = x2 = makeref(&allocp[4]);
102     allocp[5] = x0;
103     allocp[6] = a1;
104     a0 = cdr_of(a0);
105     a1 = x2;
106     qp = (struct goalrec*)&allocp[2];
107     allocp += 7;
108     execute(nrev_2_0);
109     goto nrev_2_ext_interrupt;
110  case ATOMIC:
111     if (a0 != NILATOM) goto nrev_2_interrupt;
112     x0 = NILATOM;
113     unify_value(a1, x0);
114     proceed();
115  case VARREF:
116     deref_and_jump(a0, nrev_2_1);
117     *reasonp++ = a0;
118     goto nrev_2_interrupt;
119     };
120     goto nrev_2_interrupt;
121  nrev_2_ext_interrupt:
122     reasonp = 0l;
123  nrev_2_interrupt:
124     goto interrupt_2;
125  }
126

```

```

127  append_3_top: {
128      q x0, x1, x2;
129      a0 = qp->args[0];
130      a1 = qp->args[1];
131      a2 = qp->args[2];
132      qp = qp->next;
133  append_3_clear_reason:
134      reasonp = reasons;
135  append_3_0:
136  append_3_1:
137      switch (ptagof(a0)) {
138  case CONS:
139          allocp[0] = x1 = makeref(&allocp[0]);
140          x2 = car_of(a0);
141          allocp[1] = x2;
142          x0 = makecons(&allocp[0]);
143          allocp += 2;
144          unify_value(a2, x0);
145          a0 = cdr_of(a0);
146          a2 = x1;
147          execute(append_3_0);
148          goto append_3_ext_interrupt;
149  case ATOMIC:
150          if (a0 != NILATOM) goto append_3_interrupt;
151          unify(a1, a2);
152          proceed();
153  case VARREF:
154          deref_and_jump(a0, append_3_1);
155          *reasonp++ = a0;
156          goto append_3_interrupt;
157      };
158      goto append_3_interrupt;
159  append_3_ext_interrupt:
160      reasonp = 0l;
161  append_3_interrupt:
162      goto interrupt_3;
163  }
164  interrupt_3:
165      allocp[4] = a2;
166  interrupt_2:
167      allocp[3] = a1;
168  interrupt_1:
169      allocp[2] = a0;
170  interrupt_0:
171      allocp = interrupt_goal(allocp, toppred, reasonp);

```

```

172  proceed_label:
173      loop_within_module(module_main);
174  }

```

3.5.1 コード説明

このコードを先頭から順に説明する。

- 2-4 行目には、コンパイルするに必要なファイルの include リストがならぶ。*include/klic/atom.h*, *include/klic/funct.h* は各々、アトム番号、ファンクタ番号が記録されたファイルで、マクロの羅列になっている。つまり、アトム番号、ファンクタ番号は C のコンパイルを行うときにはすでに確定されている。

```

2  #include <klic/klichdr.h>
3  #include "atom.h"
4  #include "funct.h"

```

- 6 行目はこのモジュールで定義される module 型関数の module_main の宣言。

```

6  module module_main();

```

- 7 - 12 行目は、このモジュール内で定義される述語、main/0, nrev/2, append/3 の述語構造体 (struct predicate) の定義。このように述語構造体は定数としてコンパイル時に生成される。

```

7  Const struct predicate predicate_main_xmain_0 =
8      { module_main, 0, 0 };
9  Const struct predicate predicate_main_xnrev_2 =
10     { module_main, 1, 2 };
11  Const struct predicate predicate_main_xappend_3 =
12     { module_main, 2, 3 };

```

- 13 行目は、参照している外部モジュールである、klicio:klicio/1 の述語構造体の宣言。当然、klicio モジュールを実装する C ファイル中に実体は定義されている筈である。

```

13  extern Const struct predicate predicate_klicio_xklicio_1;

```

- 15 行目から module_main の定義がはじまる。引数はすでに説明をした通り。

```

15  module module_main(global, qp, allocp, toppred)
16      struct global_variables *global;
17      struct goalrec *qp;

```

```

18     register q *allocp;
19     Const struct predicate *toppred;
20 {

```

- 21 – 36 行目には [1,2,3] を実現するリスト構造が定義されている。このように、コンパイル時に確定している構造は、コンパイル時に静的に生成される (36 ページ、第 3.4.2 章参照)。

```

21     static Const q cons_const_0[] = {
22         NILATOM,
23         makeint(3),
24     };
25     static Const q cons_const_1[] = {
26         makecons(cons_const_0),
27         makeint(2),
28     };
29     static Const q cons_const_2[] = {
30         makecons(cons_const_1),
31         makeint(1),
32     };
33     static Const q cons_const_3[] = {
34         NILATOM,
35         makesym(atom_n1),
36     };

```

- 37 行目に、述語間引数用変数 (引数レジスタの気分) の宣言。これは必要な数だけコンパイラが生成する。

```

37     q a0, a1, a2;

```

- 39 行目にあるのは、中断スタック (詳細後述) の先頭ポインタの宣言。

```

39     q *reasonp;

```

- 40 行目に、`module_top` なるラベルが定義されている。一般的に “モジュール名” + “_top” なる名称のラベルが実行行の先頭に定義される。
- 41–45 行目以降にあるのは、`switch` 文となるマクロ。KLIC の制御構造は、`include/klic/control.h` 内でマクロ化されている。この `switch` は、`toppred->pred` を参照し、それが、0, 1, 2 のいずれであるかにより各々、ラベル `main_0_top`, `nrev_2_top`, `append_3_top` に飛ぶ。predicate 型の `pred` field はこのように `switch` するため、モジュール内で unique

な番号になるようにコンパイラが定めている。

```
40  module_top:
41  switch_on_pred() {
42      case_pred(0, main_0_top);
43      case_pred(1, nrev_2_top);
44      last_case_pred(2, append_3_top);
45  }
```

- 47 行目にはラベル `main_0_top` の定義がなされている。各述語の先頭には、“述語名_” + 引数個数 + “_top” なる名称のラベルが生成され、`switch_on_pred` からジャンプされる。

```
47  main_0_top: {
```

- 48 行目にあるのは述語内で使われる一時変数（一時レジスタの気分）。

```
48      q x0, x1, x2, x3, x4, x5;
```

- `qp` の先頭を 49 行目でずらす。

```
49      qp = qp->next;
```

- 50 行目のラベルは 51 行目のサスペンションスタックを初期化する操作に対してのラベルである。述語 `main` の実行にあたり、`reasonp` の設定が済んでいることがわかっている場合にはこではなく、52 行目に入る。

```
50  main_0_clear_reason:
51      reasonp = reasons;
```

- 52 行目から述語メインの実際の動作が開始される。53-56 で、`nrev([1,2,3],Y)` に対応するゴールレコードが生成される。`alloca` とあるのはヒープ割り付けである。このように KLIC ではゴールレコードもヒープ中に生成される。ゴールレコードの `next` に相当する先頭には `qp` が格納されている。

また、基本的に KLIC プログラム中で記述した順とは逆にゴールが生成され、エンキューされていく。56 行目で純粋未定義変数を生成しているのに注意。

```
52  main_0_0:
53      alloca[0] = (q)qp;
54      alloca[1] = (q)(&predicate_main_xnrev_2);
55      alloca[2] = makecons(cons_const_2);
56      alloca[3] = x0 = makeref(&alloca[3]);
```

- 57 - 71 では、`[stdout(normal([putt(X),nl]))]` に対応した構造を組み立てている。構造

の末端からくみたてていることに注意。

```
57  allocp[4] = makesym(functor_putt_1);
58  allocp[5] = x0;
59  x1 = makefunctor(&allocp[4]);
60  allocp[6] = makecons(cons_const_3);
61  allocp[7] = x1;
62  x2 = makecons(&allocp[6]);
63  allocp[8] = makesym(functor_normal_1);
64  allocp[9] = x2;
65  x3 = makefunctor(&allocp[8]);
66  allocp[10] = makesym(functor_stdout_1);
67  allocp[11] = x3;
68  x4 = makefunctor(&allocp[10]);
69  allocp[12] = NILATOM;
70  allocp[13] = x4;
71  x5 = makecons(&allocp[12]);
```

- 72 – 74 では `klic:klicio(...)` に対応したゴールレコードが生成される。next field には先に作成した `nrev(...)` のゴールレコードを格納することに注意。

```
72  allocp[14] = (q)(struct goalrec*)&allocp[0];
73  allocp[15] = (q)(&predicate_klicio_xklicio_1);
74  allocp[16] = x5;
```

- 75 行目では、今組み立てた `klic:klicio(...)` のゴールを `qp` に格納している。つまり、通常呼び出しのためのゴールスタックへのエンキューを行っている。

```
75  qp = (struct goalrec*)&allocp[14];
```

- 76 行目でヒープ割付点を更新している。

```
76  allocp += 17;
```

- 77 行目の `proceed()` は `include/klic/control.h` 内で定義されている macro で、module 型関数内で必ず定義されている `proceed_label` なるラベルに飛ぶ。

```
77  proceed();
```

ここでは、さらに実行するゴールがないため、ゴールスタックのデキュー操作を行うため `proceed()` が出力されている (30 ページ、第 3.3.2 章参照)。

- 78–80 行目は中断、失敗するためのコードである。引数がない (= 中断も失敗もしな

い)main/0 では意味がない。^{*1}

```
78  main_0_ext_interrupt:
79      reasonp = 01;
80  main_0_interrupt:
```

- 84 行目は nrev/2 の入口である。外部モジュールから main:nrev/2 が呼び出された時や、モジュール内でもゴースタックより nrev/2 を呼ぶコードが取り出された時にはここに飛ぶ。

```
84  nrev_2_top: {
```

- 85 行目は一時変数の宣言。

```
85      q x0, x1, x2;
```

- 86 – 87 行目で引数を引数レジスタにコピーしている。

```
86      a0 = qp->args[0];
87      a1 = qp->args[1];
```

- 88 行目 で実行可能キューをデキュー。

```
88      qp = qp->next;
```

- 89 – 92 行目に述語の先頭の処理が記述される。

```
89  nrev_2_clear_reason:
90      reasonp = reasons;
91  nrev_2_0:
92  nrev_2_1:
```

- 93 行目 から始まる switch 文で、a0 のタグにより分岐する。

```
93      switch (ptagof(a0)) {
```

- 94 – 109 行目は CONS タグの場合。

```
94      case CONS:
95          allocp[0] = NILATOM;
96          x1 = car_of(a0);
```

^{*1} KLIC ではこのような無駄なコードを生成する場合がある。しかしながら、さらに、C コンパイラでの最適化が行われ、通常は除去されるため、最適化済コードの大きさには影響しないことが期待され、実際、GCC では大きさには関係がない。

```

97     allocp[1] = x1;
98     x0 = makecons(&allocp[0]);
99     allocp[2] = (q)qp;
100    allocp[3] = (q)(&predicate_main_xappend_3);
101    allocp[4] = x2 = makeref(&allocp[4]);
102    allocp[5] = x0;
103    allocp[6] = a1;
104    a0 = cdr_of(a0);
105    a1 = x2;
106    qp = (struct goalrec*)&allocp[2];
107    allocp += 7;
108    execute(nrev_2_0);
109    goto nrev_2_ext_interrupt;

```

- 95 - 98 で [A] を組み立て、x0 にセットする。
- 99 - 103 で、append/3 のゴールを作成し、エンキューしている。
- 104 - 108 で、nrev/2 の呼び出しを行っている。

ここでは、nrev/2 が「直接呼び出し」されている。

この時には、引数レジスタ群 (a0, a1, ...) は、呼び出された先でもそのまま (ゴールからひきあげることなく) 利用されるため、execute までの間に引数レジスタ群に適当な引数を用意しておく必要がある。

- * まず、104 行目で X を第 0 引数ににセット。
- * 105 行目では、純粋未定義変数である X1(x2 に載っている) を第一引数にセット。
- * 106 行目 で、append/3 を qp にセット (つまり、append/3 のエンキュー)
- * 107 行目 でヒープ割付点の更新。
- * 108 行目で execute で直接呼びだしする。これは、*include/klic/control.h* で以下のように定義されたマクロである。

```

#define execute(label)\
{\
    if (allop < heaplimit) goto label;\
}

```

ここで、allop と heaplimit を比較している。これは第 3.2 章 (27 ページ) で述べた、「例外的処理の有無」の検出である。ここで「例外的処理処理」要求が検出できなかった場合には指定されたラベル (ここでは、nrev_2_0 に飛ぶ)。検出された場合には 109 行以降の処理を行う。

nrev_2_0 とはすでに経過した部分であり、91 行目にある。これは、「ゴールから引数が引き上げ」られたのと「引数の検査」の切れ目に存在するラベルである。これ

は、execute 実行の場合には前述のように引数 (a_x レジスタ) はすでに正しい内容を持っているため「ゴールからの引数の引き上げ」は不要であり、省略するためにこのようなラベルに飛ぶようになっている。つまり、述語が開始されるには、一般的には以下の 2 通りがある (29 ページ、第 3.3.2 章参照)。

通常呼び出しによる実行: “述語_引数個数” + “_top”。直後にゴール引数の引き上げがある。

直接呼び出しによる実行: “述語_引数個数” + “_0”。ゴール引数の引き上げは省略。

- 直前の execute で、「他の処理」に対する要求があった場合には、109 行目を実行し、その後 121 に飛ぶ。その結果、中断スタックポインタを 0 にしてから `interrupt_2` に飛ぶ。中断スタックには、通常はゴール実行の原因となった未定義変数をプッシュする。しかしながら、中断スタックポインタが 0 である場合には、ヒープ溢れの検出による中断であることを示す (57 ページ、第 4.2.2 章参照)。`interrupt_2` とは、「2 引数用の割出し処理ラベル」であり、引数レジスタ上の 2 つの引数をゴール引数部に格納してから、「他の処理」の実行のため、関数 `interrupt_goal` に制御を移す。
- 110 行目からは、`a0` がアトミックな場合の処理である。

```
111    if (a0 != NILATOM) goto nrev_2_interrupt;  
112    x0 = NILATOM;  
113    unify_value(a1, x0);  
114    proceed();
```

- 111 行目は `a0` が `NILATOM` かどうかを検査し、`NILATOM` でなければ `nrev_2_interrupt` に飛ぶ。`nrev_2_interrupt` では、中断スタックポインタを 0 にすることなく、中断スタックの底を指す (つまりスタックを空にする) 状態にする。これは、リダクション実行の失敗を示す (57 ページ、第 4.2.2 章参照)。その後引数を退避し、関数 `interrupt_goal` に制御を移す。
- 112 では、`x0` に `NILATOM` を代入し、`a1` と `unify_value()` により `Y=[]` が実行される。`unify_value` は、第 2 引数が具体値であることが判明しているときに呼びだされる macro で、一般的には `do_unify()` が呼びだされる。これら 2 つはマクロであり、`include/klic/unify.h` に定義されている。`unify_value` の場合には第 0 引数が純粋未定義変数である場合には、実行時ルーチンと呼ばずに、その場で具体化を行う。この「単純な場合」以外は、`runtime/unify.c` で定義されている C 関数、`do_unify`, `do_unify_value` が呼びだされる。
- 114 行目では `proceed` する。
- 115–118 行目では `a0` が `REF` タグの場合処理が記されている。

```
115    case VARREF:
```

```

116     deref_and_jump(a0,nrev_2_1);
117     *reasonp++ = a0;
118     goto nrev_2_interrupt;

```

116 行目の `deref_and_jump` は *include/klic/index.h* に定義されるマクロで、`deref_and_jump(レジスタ、ラベル)` という引数をとる。具体的には以下のようなマクロである。

```

#define deref_and_jump(x, loop) \
{ \
    q temp0 = derefone(x); \
    if(!isref(temp0)) { \
        (x) = temp0; \
        goto loop; \
    } \
}

```

これを以下で解説する。

- レジスタを 1 段手繰り (マクロ `derefone()`)、
 - * 具体値であれば書き戻し、ラベルに飛ぶ。その結果第 0 引数 (a0) の検査を最初から行うことを示す。
 - * 具体値でなければ、次の行に進む。
- 117 行目では、(少なくとも 2 段以上)REF であった a0 を中断スタックに詰み、`nrev_2_interrupt` に飛ぶ。その結果、最終的には引数を退避し、関数 `interrupt_goal()` が呼びだされる。
- 127 行目からは `append/3` のコードである。128 – 134 行目 までで引数の引き上げ、中断スタックポインタの初期化が行われる。

```

127     append_3_top: {
128         q x0, x1, x2;
129         a0 = qp->args[0];
130         a1 = qp->args[1];
131         a2 = qp->args[2];
132         qp = qp->next;
133     append_3_clear_reason:
134         reasonp = reasons;

```

- 137 行目 で、a0 の型検査が行われる。CONS であれば、139 – 144 行目で 単一化がなされ、147 行目で `append/3` が直接呼び出しされる。

```

137     switch (ptagof(a0)) {

```

```

138  case CONS:
139      allocp[0] = x1 = makeref(&allocp[0]);
140      x2 = car_of(a0);
141      allocp[1] = x2;
142      x0 = makecons(&allocp[0]);
143      allocp += 2;
144      unify_value(a2, x0);
145      a0 = cdr_of(a0);
146      a2 = x1;
147      execute(append_3_0);
148      goto append_3_ext_interrupt;

```

- ATOMIC であれば、150 行目で NILATOM かどうか検査され、NILATOM であれば、2 つの引数を単一化し、proceed() する。NILATOM でなければ、append_3_interrupt に飛び、引数が書き戻され、interrupt_goals が呼びだされる。

```

149  case ATOMIC:
150      if (a0 != NILATOM) goto append_3_interrupt;
151      unify(a1, a2);
152      proceed();

```

- 153 – 156 行目は REF の時のコードであり、nrev と同様に手繰りの後に検査され、再度 137 より検査されるか、interrupt_goal が呼びだされるかする。

```

153  case VARREF:
154      deref_and_jump(a0, append_3_1);
155      *reasonp++ = a0;
156      goto append_3_interrupt;

```

- 164 – 169 行目 は引数をゴールレコードに書きもどす処理である。

```

164  interrupt_3:
165      allocp[4] = a2;
166  interrupt_2:
167      allocp[3] = a1;
168  interrupt_1:
169      allocp[2] = a0;

```

- 171 行目で interrupt_goal を呼び出している。この関数呼びだしが起きる時は、以下いずれかであり、おのおの中断スタックの状態により interrupt_goal 関数内で適切な処理が行われ

る。interrupt_goal 関数内の処理の概要は第 3.4 章 (33 ページ) を参照のこと。

```
170  interrupt_0:
171  allocp = interrupt_goal(allocp, toppred, reasonp);
```

なお、interrupt_goals(allocp, toppred, reasonp) の関数 spec は以下の通り。

allocp: ヒープ割付点。

toppred: 失敗/中断を起した述語の述語構造体。この例ではなかったが、各述語より interrupt_X に飛ぶコードが出る直前で、toppred に当該述語の構造体がセットされる。この例では出現しなかった理由は、以下である。

- モジュール内での execute によるジャンプを考えなければ、この toppred は module 型関数を呼び出すときに設定されているので、内部であらたに設定する必要はない。つまり、execute される場合に考慮する必要がある。
- この例で、execute されるものは、nrev, append が共に自分自身を呼び出すものしかなかった。よって、この例では、execute により toppred と実際に実行している述語とがずれる可能性はないため、このような処理は必要がない。

reasonp: 中断スタックポインタ。

また、戻り値としては更新されたヒープ割付点を返す。

どちらにしても、この interrupt_goal が呼び出された後には、レジスタなどは「中途半端な状態」ではなく、すべて goal record に収められている。

- 最終的に、include/klic/control.h で定義されている、loop_within_module なるマクロに制御が移される。

```
172  proceed_label:
173  loop_within_module(module_main);
```

具体的には以下のような定義である。

```
#define loop_within_module(f) \
{ \
  module (*func)(); \
  if (allocp >= heapl原因) { \
    allocp = klic_interrupt(allocp, qp); \
    qp = current_queue; \
  } \
  if ((func = (toppred = qp->pred)->func) == (f)) \
    goto module_top; \
  heapp = allocp; \
  current_queue = qp; \
}
```

```
    return (module) func; \
}
```

ここでは、

1. `allocp` と `heaplimit` を比較する。つまり、「例外的な処理 (GC 要求を含む)」があるかどうか検査する。「例外的な処理が」ある場合には、`runtime/intrrpt.c` で定義されている関数 `klic_interrupt` が呼びだされ、「例外的な処理」が処理される。
2. 必要ならば「例外的な処理」を終了したあとに、「次に実行される述語の属するモジュール関数」が、現在のモジュール関数であるかどうか調べられる。現在のモジュール関数であれば、`return` はせずに、関数の実行部分の先頭である `module_top` に飛ぶ。現在のモジュールでなければ、`allocp` を `heapp` に、`qp` を `current_queue` に退避し、次に実行すべき `module` 型関数を返し、トップレベルループに戻る。

先に「例外的な処理」の検査、措置を行う理由は、以下の通りである。

- 「例外的な処理」を行った結果、ゴールがエンキューされる場合がある。
- そのゴールは、最高の優先である可能性があり、その場合には、そのゴールの実行を例外的処理の措置の前のキュー先頭のゴールよりも先に行う必要がある。

第 4 章

割り込み/中断/失敗

KLIC では、KL1 のコードをコンパイルした部分が実行されている時の大部分は、割り込み処理を行わず、KLIC にとって「都合の良い」タイミング、すなわち、リダクション処理の「切れ目」で「割り込みがあった」ことを検出し、割り込みが起きていた場合については、「割り出し」が行われ、実行時ライブラリにて行われることはすでに第 3 章 (26 ページ) で述べた。また、中断の処理、失敗の処理についても、コンパイルしたコード中では行われず、実行時ライブラリに割り出されて行われることも述べた。

本章では、これらの「割り出し処理」についてさらに詳細に解説を行う。

具体的には、以下についての説明を行う。

- 割り込み処理について、*runtime/interrupt.c* 中で定義されている、`klic_interrupt`、および、例外が起きたときに起動されるコード群について解説を行う。
- 中断、失敗の処理 *runtime/faisus.c* 中で定義されている、`interrupt_goal` 中の処理について解説を行う。

4.1 例外処理

4.1.1 割り込み時の処理

割り込みが発生した場合、先に述べたように、KLIC では実行中には一般的には割り込み処理を行わず、割り込みがあったことのみを記録しておき、後に `klic_interrupt` にて処理を行う。

割り込みがあった「その時」に実行されるコードについての記述は、*runtime/signal.c* 中で記述されている。

KLIC での割り込み処理は、基本的に以下のように 2 段がまえになっている。

- 各割り込み種別 (UNIX の SIGXXX) 毎に、UNIX レベルの割り込みハンドラが存在し、また、各割り込み毎に「割り込みフラグ」(`signal.flags`) がある。このハンドラは、割り込みが起きたときには、当該「割り込みフラグ」を on にすることにより「割り込みがあった」と

いうことを記録し、`heaplimit` を 0 とするだけである。このハンドラは、`handle_signal()` なる既定の関数で、*runtime/signal.c* で定義されている。

- リダクションの切れ目にて、ヒープ検査の際に割り込みがあったことが確認された場合には「割り込みフラグ」を検査し、割り込み種類別の処理を実行する。各割り込み毎の処理は、割り込み処理表 (`signal_handlers`) に記録されている。この処理は後述する、`klic_interrupt()` 内で行われる。

つまり、割り込みハンドラには 2 通りがある。以下の説明で、特に断ることなく「割り込みハンドラ」と称した場合には、後者、つまり、リダクションの切れ目で行われる処理を記述したハンドラを指す。

この「割り込みハンドラ」の関数仕様は以下のようにになっている。

`handler(allocp, signal)`

`q *allocp`: ヒープトップ。つまり、割り込みハンドラ中では、ヒープ割付を行うことが可能で、したがって、KL1 のゴールをわりつけることもできる。

`int signal`: 割り込み種別 (`SIGXXX`)

`SIGINT`, `SIGALRM` については、以下のように、特殊な割り込みハンドラをシステム側で設定している。

- `SIGINT` に関しては、`sigint_interrupt` をセットするだけの処理を行うハンドラ (`default_sigint_handler`) が設定される。
- `SIGALRM` に関しては、そもそも、UNIX レベルのハンドラが異なっており、`timer_expiration_handler` がセットされる (タイマについての処理は *runtime/timer.c* にて記述される)。タイマ割り込みは、システムで利用している一方、ユーザも利用する可能性があるため、仮想化の処理を行い、実効上、複数のタイマを同時に設定できるようになっている。

これまでの説明で登場しなかった割り込み処理に関して定義されている主な関数群の説明を以下に記述する。

`klic_signal_handler()`: 割り込みフラグを参照し、割り込みフラグが on である場合には当該処理を割り込み処理表を参照することにより実行する。後述される `klic_interrupt` 内で呼びだされる。

`add_signal_handler()`: 割り込み処理を登録する。つまり、

- 割り込み処理表に指定された関数を登録する (`add_slit_check_signal_handler()`)。
- 指定された割り込み種別に対して `sigaction()` にて UNIX レベルの割り込み処理を設定する。

`init_general_signal_handling()`: 割り込み処理表、割り込みフラグを初期化する。

4.1.2 KL1 レベルでの割り込み処理

KLIC の内部で行われている割り込み処理はこれまでに述べてきたようなことを行っている。一方で、KL1 ユーザレベルで割り込み処理を記述、登録することも可能になっており、そのための処理を「割り込みハンドラ」として実装している。すなわち、シグナルが発生するたびに、CONS でユーザが指定した未定義変数を具体化していく、という処理を行っている。

KL1 レベルでのインターフェースは `unix` モジュール (`gunix.kl1`) で定義されており、`signal_stream(Signal, Result, Variable)` なるインターフェースである。

このような処理に関しての関数群も `signal.c` で定義されている。

`streamed_signal_handler()`: 割り込みがおきたたびに CONS を割りつけ、あらかじめ登録された変数に単一化を行う関数。

`register_streamed_signal()`: KL1 述語 `signal_stream` より呼びだされる登録用関数。
`streamed_signal_handler()` を割り込みハンドラとして登録する。複数回登録されたならば、シグナルストリームになる変数同士を単一化する。

4.1.3 割り込み処理: `klic_interrupt`

`klic_interrupt` は、リダクションの切れ目にて、ヒープ溢れが起きたかのように見えた時に呼び出される処理であり、「実際にヒープが溢れた場合」、および「例外処理がおきたため、擬似的にヒープが溢れたかのように見えた場合 (`heaplimit` が 0 の場合) に呼び出される処理であった。

実際の処理は以下のように行われている。

1. まず、`resume` されているかもしれないゴールを通常のスタックに書き戻す。これは、実行時ライブラリ中で行われた処理によりエンキューされたようなゴールは大域変数 `resumed_goals` にエンキューされている。この二次的なゴールスタックにエンキューされているゴールを本来のゴールスタックにエンキューし直す (32 ページ、第 3.3.4 章参照)。この「エンキューし直し」処理を行う関数が `runtime/intrpt.c` で定義されている `enqueue_resumed_goals()` である。
ここでは、まず、ここで述べたようなエンキューが完了していないようなゴールがあることを考慮し、まず、`enqueue_resumed_goals()` を呼びだしている。
2. トレーサ対応のコードの場合、ここで、トレーサの処理をおこなう。具体的には、`stepping_flag`、`trace_flag` を参照し、を参照し、ON ならば各々 `step_after()`、`trace_after` 処理 (共に `runtime/trace.c` 内で定義されている) を行う。詳細は、第 8.3 章 (85 ページ) を参照のこと。
3. 割り込みの原因を調査する。
 - まず、以下で説明する「特殊な割り込み」以外の「本来の割り込み処理」が必要かどうか

か検査する。「本来の割り込み処理」がおきたかどうかは、`signal_done` を参照することにより判明する。割り込みがあった場合には、`klic_signal_handler` を呼びだし、必要な処理を行う (詳細後述)。その結果、`heap` に不足ができていないか判断する。

また、割り込み処理も KL1 のゴールにて実現することができるようになっている。この場合、ゴールのエンキューは前述の「例外的ゴールスタック」に対して行われるため、`enqueue_resumed_goals` を呼び出し、処理ゴールがもしあればスケジュールされるようにしておく (先に `heap` に不足ができていないか判断している一因は、このエンキューのためにヒープが消費されている可能性のあることである)。

- 割り込み原因が SIGINT によるものであれば、大域変数 `sigint_interrupt` が on になっている。これについてはその場で KLIC の処理全体を終了させる。
- より高い優先度のゴールがエンキューされたのであれば、大域変数 `higher_priority_goal` が on になっている。その場合には、次に実行しようとしていたゴールをエンキューし、同変数を off にし、最も高い優先度のキューに現在実行中のキューを置き換える。

この「割り込み」は、通常のエンキュー時に起こる可能性があり、汎用の (つまり、エンキューすべきゴールが、現在実行中の優先度と同じであることが確定できない場合に呼びだされる) ゴールエンキュー関数 `enqueue_goal()` 中で優先度を比較した上でセットする。このエンキューはいわば同期的に行われるため、前述の「割り込み用ゴールスタック」を用いずに、直接、通常の優先度付きゴールキューにエンキューされる。しかしながら、実行ゴールスタックの置き換えはやはりここで行われる。

- 真にヒープに溢れがおきたかどうか、検査する。この検査は以下のように行われる。

```
if (allocp + this_more_space >= real_heaplimit)
```

この `real_heaplimit` は、割り込み発生があっても、強制的に 0 に置き換えられることのない `heaplimit` のコピーである。`this_more_space` は、ヒープ不足で中断した計算を実行するために最低限確保しなければならない空きヒープ領域の容量を示している。

上記の if 文が true になった場合には GC を起動する (`klic_gc` の呼びだし)。GC 呼び出し前後に `GC_ON()`, `GC_OFF` とあるのはマクロであり、GC 処理についての統計情報を確保するためのものである。

4.2 中断、失敗処理: `interrupt_goal`

中断、失敗については、コンパイルコードより、必要な場合に適宜関数 `interrupt_goal()` が呼び出されることにより処理される。

4.2.1 関数型仕様

`interrupt_goal` は、以下のような関数型になっている。

```
q *interrupt_goal(allocp, pred, reasonp)
    q *allocp;
    struct predicate *pred;
    q *reasonp;
```

引数は各々、ヒープトップ、最後に実行しようとしていた述語の述語構造体、中断スタック先頭を表わす。戻り値は、更新されたヒープトップである。

4.2.2 動作

この関数に制御が移された時の状況、および、対応する処理については、第 3.4 章 (33 ページ) に記述した通りである。繰り返すと以下のようになる。

他にすべき処理が生じた時: この場合には、reasonp には 0 がセットされている。この場合には、resumed_goals に goal を格納し、他の処理をすべく return する。

述語が「失敗」したとき: この場合には reasonp は中断スタックの底を指している。失敗の処理 (関数 do_fail()) が呼び出される。

述語が「中断」しそうなとき: これまでの説明で判明したように、KLIC では、3 段以上 REF の連鎖がある場合にはコンパイルコード中では追跡をあきらめ、interrupt_goal の呼び出しを行っている。

この場合には、「中断原因の候補」は中断スタックに積まれているので、それを調べ、候補が全て真に未定義変数であれば中断処理を行う。候補のいずれかが具体値であることが判明すれば、resume_same_prio() 関数により、当該ゴールを「再開」する (つまり、エンキューしてしまう)。

これらに相当する処理を、interrupt_goal 内で行っており、その内容を処理順に従い詳説する。

1. まず、中断スタックポインタ reasonp が 0 であるかどうか検査する。0 の場合には、(擬似的に)heap 溢れがおきており、GC 要求または例外要求が起きている。その場合には、現在実行中のゴールを、resumed_goals に接続する。
2. 中断スタックが空であれば失敗したことを表わしているので、終了する。中断スタックが空でない場合には、内容にある中断要因を調査する。中断要因を手繰った結果、実は具体値であったことが判明した場合には、ゴール引数を手繰った結果に置換し、現在実行中のゴールを再度実行する (resume_same_prio())。全ての内容が未定義変数であることが分れば、真に中断すべきゴールであるので、次の処理に進む。
3. トレースサポートのシステムであれば、stepping_flag, trace_flag が ON の場合、各々、step_susp(), trace_susp() を呼び出す (90 ページ、第 8.5.3 章参照)。
4. ゴールを中断させる処理を行う。
 - (a) ゴールレコードに現在の優先度を記録する。

- (b) 再度中断スタックの内容を 1 つずつ手繰る。一重ループの変数 (つまり、純粹未定義変数) であることがわかれば、中断レコードを作り、中断要因となった未定義変数とポインタを張り合う。
- (c) 二重ループの変数であれば、そこには、中断ゴール/consumer 群、または、generator がぶらさがっている。この判断はマクロ `is_generator_susp()` で行うことができる。
- 中断ゴール/consumer がぶらさがっていることがわかれば、その中断レコードの先に現在実行中のゴールもぶらさげる。
 - Generator の場合には、object に対して、generate メソッドを発行する。これはマクロ `generic_generate()` で行うことができる。
 - － 結果が `makecons(0)` (つまり、cons タグで VALUE が 0) であれば、これは「generate 遅延」を意味するので、suspend メソッド (`generic_suspend()`) を発行する。その結果が `makeref(0)` 以外であれば、このジェネレータにより具体化が行われたことになる。そこで再実行するため、flag `redo_request` を on にする。
`makeref(0)` 以外であれば、なにもしない。
 - － 結果が `makecons(0)` 以外であれば、「その値で変数は具体化される」ことを意味するので、変数を戻り値で具体化し、再実行をするため、`redo_request` を on にする。
- (d) 以上の処理を中断スタック中の変数すべてに行ったら `redo_request` を参照し、on であれば `resume_same_prio()` により再実行する。

4.3 Timer

Timer の実装について解説を行う。

前述のように、timer はシステムで利用している一方、ユーザでも利用される可能性が高いため、仮想化したものになっている。つまり、以下のように、通常の割り込みの実装とは異なっており、すでに特殊なハンドラが用意されている。

- UNIX レベルの割り込み関数としては、`timer_expiration_handler()` を利用する。
- 割り込みの登録は、C レベルでは `call_at_specified_interval()`、`call_at_specified_time()`、`call_at_time_intervals()` を用いる。

さらに、KL1 レベルで利用できるタイマも用意されており、このタイマは、上記のタイマの仕掛けを利用して実装されている。つまり、上記のタイマのイベントとして KL1 レベルのタイマが動作する。

4.3.1 基本的な方式

基本的な方式を以下で述べる。

- KLIC で独自に、タイマを必要とする処理群のリスト (timer_wait_root) を持つ。このリストは処理すべき時間の順に時刻と処理すべき内容 (関数ポインタ) より成る。ちなみに、このリストはフリーリスト管理しており、通常のヒープの管理とは独立している。
- KLIC は上記のリストの先頭を参照し、次にアラームを受けるべき時刻を知り、現在の時刻と比較してその時刻に SIGALRM を受けるよう、タイマにセットをする (このために、UNIX の setitimer() を利用している)。
- SIGALRM を受けたら、その直後に再度リストを参照し指定された処理を行い、「タイマ要求リスト」の先頭を参照し、前段に戻る。なお、実際には、リストの先頭以降が要求する時刻をすでに過ぎていたり、同時刻に複数の処理が指定されている可能性があるため、現在の時刻と比較し、「未来のもの」以外はみな処理されるようにしている。

KL1 レベルのタイマの実装もほぼ同様の方式であるが、時刻が来たことの検出は、直接 UNIX レベルに依頼するのではなく、上記のタイマ管理の一環として行っている。また、KL1 レベルのタイマでは、上記のタイマが「SIGALRM を受けた直後」に動作するのに対して、他の例外処理同様、「SIGALRM を受けた直後」のリダクションの切れ目で実際には行われる。

4.3.2 実装詳細

Timer の実装はほぼ以下のつのファイルよりなる。より低レベルなものから、KL1 のインターフェースを実現したものの順に概説する。

timer.c: 再低位の関数群。

init_virtualized_timer(): タイマ関連の初期化。

call_at_specified_interval(), call_at_specified_time(), call_at_time_intervals(): 各々、「特定の時間後」、「特定の時刻」、「特定の間隔で」タイマをセットするための関数。処理については引数 func により関数として設定する。通常は klic_timer_interrupt_handler() がセットされる (ktimer.c)。

schedule_timer_interrupt(): 前期 3 つの関数の下請けであり、タイマ要求リストに要求された時刻と処理関数をセットする。

process_timer_expired_events(), timer_expiration_handler(): 実際に SIGALRM を受けたときに実行される UNIX レベルの例外ハンドラ。リストを先頭から嘗め、「現在の時刻」以前の処理を次々で行う。処理の結果、タイマ要求が残っているならば、restart_timer() を呼び出すことにより、タイマ処理を再開する。

restart_timer(): タイマ要求の先頭を参照し、次の時刻をセットする。

`insert_timer_queue()`: タイマ要求を新たに登録する。登録時にタイマ要求リストは時刻の昇順にソートされている状態にする。

`ktimer.c`: より高水準のタイマの関数群。

低水準のタイマとの最大の差は、低水準タイマは `SIGALRM` が発生した直後に処理を行うのに対して、高水準なものは、直後のリダクションの切れ目に処理をおこなうことにある。よって、少なくとも `KL1` の実行に関わるような事柄はこの高水準タイマで設定される必要がある。

高水準のタイマ要求リストには、要求時刻と、ハンドラ、およびデータを 1 つ渡すことができるようになっている。

`reserve_klic_timer_interrupt()`: 高水準タイマの C レベルの API。指定された時刻と具体化すべき変数とを高水準タイマ要求として登録する。仕様としては、タイマ設定時に変数が渡され、その変数を具体化することにより時刻が来たことを `KL1` に知らせる、となっている。つまり、`KL1` で利用されることを考慮した API である。

指定された時刻、処理をタイマ要求リストに登録し、その後、`process_timer_interrupt()` を呼び出す。

タイマ要求リストには、具体化すべき変数と、それに「`NIL` を具体化する」という関数 (`process_timed_instantiation()`) が登録される。

`set_simple_interval_timer_handler()`: 高水準のタイマであるが、設定する処理の内容は、C の関数を指定する。よって、「リダクションの切れ目で呼びだされる C レベルのタイマハンドラ」である。

指定された時刻、処理をタイマ要求リストに登録し、その後、`process_timer_interrupt()` を呼び出す。

タイマ要求リストには、具体化すべき関数ポインタ (をアトムタグにしたもの)、と、その「関数を実行する」関数 (`process_simple_timer_handler()`) が登録される。アトムタグが付加されているのは、後述の `gc_timer_data` で単純にコピーされることを狙っていることである。

`process_timer_interrupt()`: 上記 2 つの関数の下請けであり、かつ、リダクションの切れ目で実行される例外処理ハンドラである。

この関数では、高水準タイマ要求リストを嘗め、すでに過去のものになっている要求について、その指定されているハンドラ (つまり、`process_timed_instantiation()`/`process_simple_timer_handler()`) を実行する。実行した後は `klic_timer_interrupt_handler()` を例外処理関数として `call_at_specified_time` により低レベルタイマに登録する。

この `klic_timer_interrupt_handler()` は、一般的な例外処理と同様、単に「例外があった」ことを記録し、`klic_signal_handler()` 内で `process_timer_interrupt()` も実行される。

`gc_timer_data()`: 高水準タイマ要求リストには変数が含まれている。その変数のメンテナンスを行うための関数。GC hook の機構を用いて登録されている (80 ページ、第 3 章参照)。

第 5 章

単一化器

本章では、KLIC の単一化器の説明をおこなう。

KL1 では単一化は `passive unification` と `active unification` の 2 つが存在するが、両処理は全く別の実装をされている。

両者に共通している実装のポリシーは、

- すくなくとも片方がごく単純な項であることが静的に判明するような場合には、コンパイル時に (部分的に) インライン展開してしまう。
- 動的にしか判明しないような場合には実行時コードを利用する。ただし、片方の項が手練りが不要ですぐに純粋未定義変数と判明するようなごく単純な場合は、動的にしか判断できないがインライン展開されている。

ここでは、`passive/active` ともに、実行時ライブラリとして用意されている部分についてのみ解説を行う。

5.1 Passsive unification

Passive unification は `unify2.c` で定義されている。`eq_terms.body(x, y)` なる関数で、`x, y` は比較対象の項であり、両者が同一かどうかを検査している。

内部で行っていることはごく単純であり、単純比較し、同一でない場合でかつ構造の場合には再帰的に比較を行っている。また、末尾再帰最適化をかけたコードになっている。

特記すべきなのは以下の 2 点である。

- 比較は際限なく行う。よって、循環構造体同士の比較を行うと比較は終了しなくなる。本来はこれは、バグである。
- とともに `data object` の場合には、`passive_unify` メソッドにより比較を行う。

5.2 Active unification

Active unification は unify.c、unify2.c に定義されている。passive unification と比較すると、明かに複雑である。

単一化器のトップレベルは基本的に、do_unify(), do_unify_value() の 2 つである。両者の違いは、前者が第一、二引数共にその構造がはっきりしない場合に呼びだされるのに対して、後者は第二引数 (x, y とあった場合の y) は具体値であることが保証されているため処理が単純であることである。

また、do_unify では、基本的に 1 レベルずつしか単一化をせず、構造体の単一化については、1 レベルずつ、単一化を行うゴールをエンキューすることにより実現している。単一化を行う際には、ゴールがフックされている変数を具体化することにより、中断ゴールを再開したり、consumer/generator object の unify メソッドの起動が行われることがある。通常の do_unify では連続して、それらの処理を行うが、「中断構造」を発見したときに、それらの再開や unify method の起動を行わずに単一化を行うゴールをエンキューしてしまう版の単一化器も存在する (do_shallow_unify())。*1

ここでは、もっとも複雑な版である、do_unify() についての説明を行い、他の版は特に説明しないこととする。

5.3 単一化器: do_unify()

単一化器 do_unify() のアルゴリズムについて解説する。

do_unify() のインターフェースは以下のようになっている。

```
q *do_unify(allocp, x, y)
    q * allocp;
    q x, y;
```

x, y は単一化対象である項であり、allocp はヒープ割付点のアドレスである。単一化後、更新されたヒープ割付点のアドレスが返される。

以下でアルゴリズムの説明を行う。

1. 片方の項 x を手繰りつつ型検査を行う。

- x が二重ループであることがわかれば、もう一方の項 y を手繰り、一重ループかどうか検査する。一重ループであれば、y を具体化して終了。

*1 このような版の単一化器が存在する理由は、特にジェネリックオブジェクトのメソッド起動については、このメソッドを起動した結果、再帰的に連続してメソッドが起動され、巨大なヒープを消費するなどにより処理が複雑になることがあり、そのような状況に対処するためである。例えば、merger のコードを参照のこと。

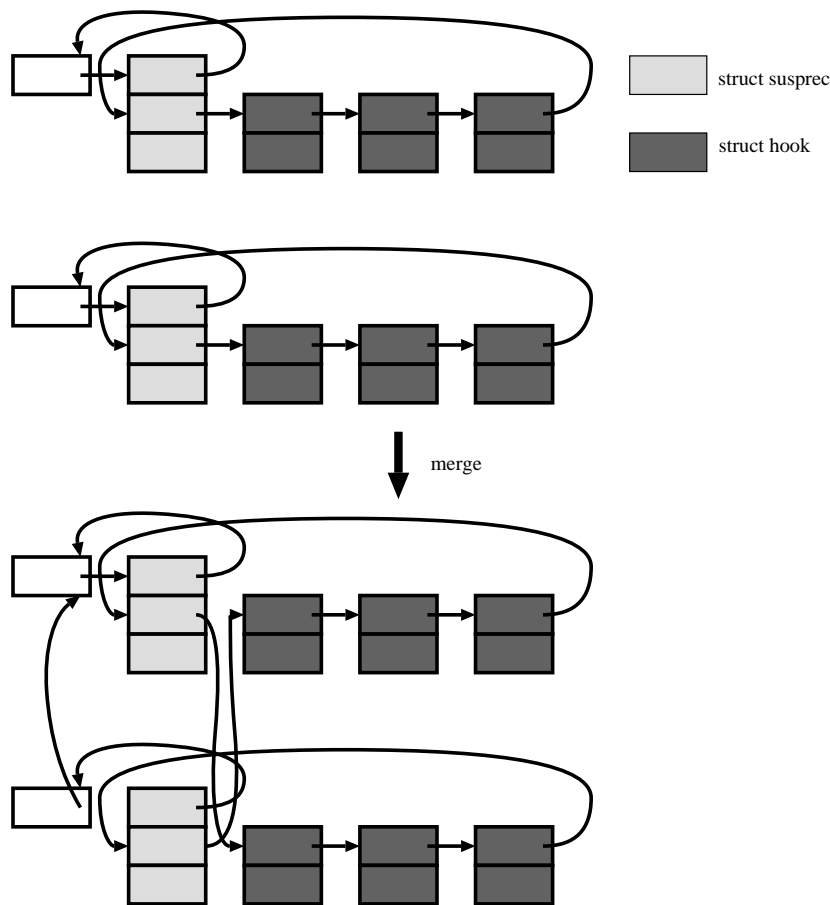


図 5.1 ゴールチェーンの merge

- y も二重ループであれば、
 - (a) x と y が同一でなければ、x が generator であるかどうか検査する (is_generator_susp で検査する)。generator であれば、generator_unify() を実行する。y が generator であれば、その逆。
 - (b) x, y が共に generator でなければ、共に goal/consumer ゴールであることが判明するので双方を merge し終了。このマージについては、suspend しているゴール群の数に関係なく、一定コストで行われるように工夫がなされている (63 ページ、第 5.1 章参照)。
- y が二重ループではない場合には、具体値であることが判明したので x を待つゴールを再開する (resume_goals())。
- x が一重ループであることがわかれば、y が具体値、または一/二重ループであることがわかるまで手繰り x を具体化する。y を手繰るのは、x と y とが実は同じ変数である場合に、REF のループができてしまうことを防ぐため。
- x が具体値であることがわかった場合には、y を手繰る。

- y が一重ループであれば、y を x で具体化して終了。
- y が二重ループであることがわかれば、y を待つゴールを再開 (resume_goals(generator) の場合もこれで良い)。
- y が具体値であることがわかれば、x とを比較する。同一でなければ、両者を単一化するようなゴールを割りつけ、エンキューする (enqueue_unify_goal())。単一化を行っている間は割り込み関連の処理がまったく行われない。何故なら、割り込み処理はゴールのリダクションの合間に行なわれるが、単一化器のような実行時ライブラリが動作している間にはリダクション操作が行われないためである。これは、あまりに複雑な項同士を単一化する場合 (特に並列実装時に) システムのレスポンスがあまりに悪くなる可能性が高いため、ある程度しか連続して処理が行われないようにするための配慮である。

以上で述べたように、active unifier の内部ではさらに副次的な処理が内部で行われる。つまり、ゴールの再開、generator の起動、さらに複雑な項同士の単一化である。それらの説明を以下で行う。

5.3.1 ゴールの再開: resume_goals()

中断構造に対して単一化を行い、ゴールの再開、consumer/generator object の unify メソッドを起動するための処理は、resume_goals() で行われる。

中断構造に対しての処理の概略

Generator の処理に比較してより複雑であるゴールレコード、consumer に対しての処理については、概略の解説を以下で行う。図 2.10 として挙げた中断構造の説明の図を図 5.2 として再掲する。

この図でわかるように、中断構造の内部は、ゴールレコードと consumer が混在して環状のリスト構造になっている。この「ゴールの再開」処理では、このリスト構造をたどり、以下のような処理を行う。

- ゴールを発見したならば、そのゴールを再開する (つまり、実行可能ゴールスタックにエンキューする) 処理を行う
- consumer を発見したならば、その consumer の unify メソッドを起動する。

ような処理を行う。

この consumer に対しての unify メソッドの発行の結果、また再度同種の構造の作成が必要になるような場合がある。例えば、マージャの処理は、

1. 入力に CONS があれば、出力用に CONS を 1 つ割付け、入力の CAR をその CONS にコピーし、出力側と単一化する。

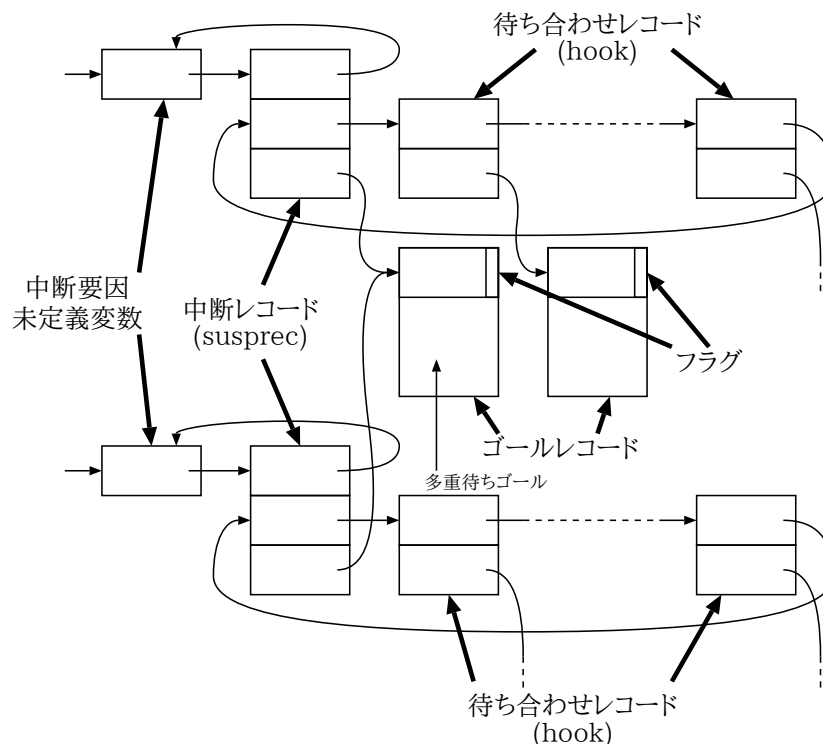


図 5.2 中断を表現するデータ構造

2. マージャオブジェクト自身を入力の CDR にフックしなおす。

という処理が行われる。このような処理は、入力が“ストリーム”であるような consumer で広く用いられるため、効率良く処理できることが望ましい。そのため、unify メソッドの結果 (戻り値) により、これまでの中断レコード、待合せレコードなどを再利用し、すぐにまた中断構造を作成できるような処理も実装している。

resume_goals() の詳細

resule_goals() の仕様は以下のようになっている。

```

Inline q *resume_goals(allocp, x, y)
    q * allocp;
    q x;
    q y;

```

x は中断レコード、または、generator 用の構造 (struct generator_susp) を直接差すポインタであり、y は具体化対象となる項である。allocp はヒープ割付点のアドレスであり、戻り値として、更新されたヒープ割付点アドレスが返される。

- まず、x が generator かどうか検査する (is_generator_susp)。Generator であれば、generator の active_unify を呼ぶ。その結果が成功すれば、終了、成功しなければ、generate method を呼ぶ。
 - makeref(0) が返れば、メモリ不足であるので、x と y とを単一化するゴールをエンキューしひとまず終了。
 - makecons(0) が返ると内部エラーである。
 - それ以外のものであれば、generator により生成された具体値なので、それを y と単一化する。このとき、x がぶらさがっていた変数のエントリも具体化する必要がある。
 - x が goal/consumer であることがわかれば、
 1. まず、エントリの変数を y で具体化しておく (共有ヒープ並列化の対応)
 2. チェーンを 1 つづつたどっていく。
 - Consumer であることがわかれば (is_consumer_hook()), active_unify method を発行し、失敗すればそこで異常終了、成功すれば、大域変数 rest_of_stream を参照し、非ゼロであれば中断構造鎖の構造を保存し、rest_of_stream にフックする。ゼロならば、鎖の構造を保存しない。
 - メモリ不足のために consumer が失敗することがあるが、この場合には、「現在行おうとしている単一化の処理はキャンセル」する。つまり、フック構造は残しておき、再度当該具体値 (y) と単一化を行うようなゴールをエンキューして終了。
 - それ以外の結果であれば、method_result に返された値を新しい値とし、再度 object を含めた鎖構造を再構成し、単一化を行う (enqueue_unify_goal())。
 - Consumer ではない、ゴールである場合には、ゴールレコードの next field のタグを検査する。
 - * Next のタグが INT であれば、このゴールはまだ起動されていないことを意味する。この場合 Next 部には優先度がタグ付きで記述されているので、その優先度を現在の優先度とチェックして、一般的なエンキュー (enqueue_goal()), または、現在実行中の queue に enqueue する (resume_same_prio()) を行い、終了 (ここで、現在の優先度より高いゴールをエンキューしたならば、enqueue_goal() 中で、heaplimit が 0 になり、次のリダクションとの間でチェックを行い、切り替わる。現在の優先度の queue に接続する場合も「再開ゴールスタック」に接続されるため、heaplimit は 0 となり、次のリダクションとの間で、実際のゴールスタックにエンキューされる。
 - どちらにしても、エンキューを行った結果、next field は他のゴールへのポインタが記録され、下位 2bit は 00、つまり、タグ部は REF になる。
 - * Next のタグが REF の場合には、すでに前述の再開処理が行われたゴールであるので、なにもしない。
- このように、1 つのゴールに対して重複して再開処理が行われるのは、multiple wait, いわゆる “OR 待ち”(図 5.3 参照) があるためである。

```

mwait(A, _) :- wait(A) | ... % (1)
mwait(_, B) :- wait(B) | ... % (2)

```

図 5.3 Multiple wait を起すコードの例

つまり、goal record はその中断原因となった変数にフックするが、複数の中断原因がある場合にはそのすべてにフックするが、再開は 1 度しか行ってはならない。例えば、図 5.3 で、変数 A,B 共に未定義であると、これは multiple wait になる。その後、A が先に具体化されたならば、(1) の節が選択され、(2) の節の実行は廃棄され、後に B が具体化されたとして実行されてはならない。この言語仕様を満足させるために、これまで説明したような実装となっている。

5.3.2 Generator の起動: generator_unify()

この関数は、generator の active_unify メソッドを起動し結果に応じて、単一化の処理を行う。片方は generator であることが確定し、もう一方は、hook 構造体である (つまり、goal/consumer/generator のいずれかである) ことが確定している時に呼びだされる。

関数仕様は以下である。

```

static inline q*
generator_unify(gsx, sy, allocp)
    struct generator_susp *gsx;
    struct susprec *sy;
    q *allocp;

```

引数は、gsx は、generator_susp 型構造へのポインタ、sy は susprec へのポインタを持つ。

1. まず、gsx の先の generator に対して、sy を引数として active_unify() メソッドを起動する。
 - 結果が 0 以外のものでなければ、単一化処理は、当該メソッド中で行われたので、終了する (このとき、戻り値は heap top を示している)。
 - 結果が 0 であれば、当該メソッドの実行は拒否されたことになるので、以下の処理を続ける。
2. sy の先を調べる。
 - sy の先も generator であることが判明したならば、gsx を引数とし、sy の先のオブジェクトに active_unify() を起動する。
 - － 戻り値が 0 以外であれば、それを heap top と解釈し終了。
 - － 戻り値が 0 であれば、sy 側も unify が失敗したことを示すので、基本的には gsx, sy 共に generate method を発行し、単一化する。すなわち、gsx 側の object に generate メソッドを発行する。その結果、戻り値が

- * `makeref(0)` であれば、メモリ不足であったことを意味する。よって、`sy` の先の `generator` に対しても `generate` を発行する。その結果、
 - ・ `makeref(0)` がもどれば、後に再度単一化を行うため、`enqueue_unify_goal()` により、単一化ゴールをエンキューする。
 - ・ `makecons(0)` であれば、異常終了。
 - ・ 上記以外であれば、`sy` はその値に変身したことを示す。`sy` のエントリをまずその値で具体化し、`do_unify()` により `gsx` と再度単一化を試みる。
- * `makecons(0)` であれば、異常終了。
- * 上記以外であれば、`gsx` はその値に変身したことを示すので、`gsx` のエントリをその値に具体化し、`sy` と再度 `do_unify()` により単一化を行う。
- `sy` の先は `generator` ではない (`consumer/goal` である) ことが判明した場合には、`gsx` に対して、`generate` の処理を行い、戻り値に応じて、`enqueue_unify` をかけるなり生成された値と `sy` との単一化を行うなりする。

5.3.3 単一化ゴールのエンキュー: `enqueue_unify_goal()`

`enqueue_unify_goal()` は、メモリ不足、単一化処理の時間がかかるなどの理由により、一時的に単一化処理を中断し、後に再度行うためエンキューするための API であり、マクロとして定義されている。

これは、結局、ゴールレコードを 1 つ割り付け、*runtime/utermns.kl1* で定義されている `unify_terms_dcode:unify_goal` を述語として指定し、エンキューするだけである。ちなみに、`unify_goal` のコードは以下。

```
unify_goal(X,Y):- true | X = Y.
```

5.3.4 複雑なゴールのエンキュー: `enqueue_unify_terms()`

これは、構造体同士の単一化を行うゴールを KL1 レベルで実装したものをエンキューするのみである。

エンキューされるゴールの述語は *runtime/utermns.kl1* で定義されている `unify_terms_dcode:unify` であり、以下のようなものである。

```
unify(X,Y) :-
    functor(X,PX,AX),
    functor(Y,PY,AY),
    AX == AY |
    unify_pf(PX,PY),
```

```

unify_args(AX,X,Y).

unify_pf(PX,PY) :-
    inline:"
    if (isatomic(%0)) {
        if (%0 != %1) goto %f;
    } else if (isatomic(%1)) {
        goto %f;
    } else {
        generic_active_unify(data_objectp(%0),
                               data_objectp(%1),allocp);
    }":[PX+bound,PY+bound] | true.

unify_args(0,_,_):- true | true.
unify_args(N,X,Y):- N>0,
    arg(N,X,EX), arg(N,Y,EY) |
    EX=EY, N1:=N-1, unify_args(N1,X,Y).

```

unify_pf の一部で inline が利用されている以外は全く簡単な述語である。inline 部は、双方が data object であるときの処理を行っている。Data object はまず functor として見えるため、このような処理が妥当である。

第 6 章

ジェネリック・オブジェクト

6.1 ジェネリック・オブジェクトの概略

ジェネリック・オブジェクトとして実現されたデータは、単一化 や GC など様々な局面での自分自身の処理の仕方を記述したメソッドを集めたメソッド表、及びデータ領域から成る。このメソッド表がオブジェクトのクラス定義に当たるものである。

ユーザはメソッド表に含めるべきメソッド、オブジェクト生成のためのルーチン、データ領域の構成などを、ジェネリック・オブジェクト定義用に作成したマクロや関数を使用しつつ、C 言語で記述することで、ジェネリック・オブジェクトを定義する。ジェネリック・オブジェクトのデータ領域は KL1 の組み込みデータ型や C で定義可能なデータ型から構成することが可能であり、様々なデータをジェネリック・オブジェクトに持たせることが出来る。

ジェネリック・オブジェクトの定義に当たっては、KLIC システムの実行時カーネルに関する完全な知識は必要とせず、ジェネリック・オブジェクト定義時には、実行時カーネルには一切手を入れる必要はないように設計がなされている。KLIC の組み込みデータ型のデータが KLIC システムの実行時カーネルによって処理されるのに対し、ジェネリック・オブジェクト型のデータに対しては、実行時カーネルがそのメソッド表を参照して適切な処理を呼び出す。実行時カーネルは、ジェネリック・オブジェクトのメソッドの中身やデータ領域の構成については特に仮定を置いておらず、これらの点についてユーザはかなり自由に KLIC を拡張することができる^{*1}。

ジェネリック・オブジェクトはこのように定義の記述に大きな自由度があり、また、処理実行の面や定義記述の面で実行時カーネルから独立しているので、言語システムの高い拡張性を実現するものになっている。

^{*1} もちろん、あるデータがジェネリック・オブジェクトかどうかの判定は出来る。また、そのジェネリック・オブジェクトが、後述する 3 種類の内のどの種類のジェネリック・オブジェクトかの判定も出来る。

6.1.1 ジェネリック・オブジェクトの種類

ジェネリック・オブジェクトは実行時カーネルから見た場合、(少なくとも top level について)、以下のように分類できる。

- 具体化されているデータ
- 値が決定すれば、行われる計算が決められている変数 (ゴールをフックしている変数に類似)
- 値が必要なら、値を求める方法が定められている変数

分類の各々に応じて、以下の呼ばれるオブジェクトの種別が用意されている。

Data object 具体化されているデータと同様の扱いを行うべき オブジェクト である。

Consumer object 値が決められればそれに応じて計算が行われるオブジェクト である。

- 書き込みを行うことにより、なにか処理が行われる。
- 読み出すことによってはなにも処理は行われない。純粹未定義変数と等価の動作をする変数。

Generator object 今までの KL1 処理系には (少なくとも言語表面には) なかったデータ型。

- 書き込みを行うことにより、なにか処理が行われる (Consumer object と共通)。
- 読み出すことによって、何か値を決めるための処理が行われる。

つまり、generator は consumer を包含した機能を持つものである。

メソッド表の構成、他の (つまり、通常の KL1 の) データ構造との関係は、これら 3 種類で各々異なる。

6.1.2 Data object

Data object については浮動小数点型を実装するために用いられている float クラスのオブジェクトを例として説明する (*runtime/gfloat.c*)

Data object に関しては *include/klic/gdobject.h*, *include/klic/gd_macro.h* 中に定義されたマクロを用いて記述する。

データ構造: Float 型は、基本的に、

```
struct {
    struct data_object_method_table *method_table;
    double value;
};
```

という型である。これを定義するのに、


```
GD_OBJ_TYPE {
    struct data_object_method_table *method_table;
    double value;
};
```

とする。各々、メソッド表ポインタ、浮動小数点の値、である。

以下でメソッドの説明をする。データオブジェクトのメソッドは“GDDEF_”で開始されるマクロにより行われている。

単一化: KL1 でデータに対して行う単一化操作は、汎用的なものとして、以下の 2 つがある。

GUNIFY メソッド: 引数である他の data object と自分の passive unification を試みる。

メソッド実行の結果としては、成功、失敗、中断の 3 種類である。それぞれの状態に対応する値*2をメソッドの返り値としている。

Float 型では、まず、メソッドテーブル同士を同じかどうか比較し、その後、value 部分を比較している。

UNIFY メソッド: 引数である他の object と自分の active unification を試みる。メソッド実行の結果としては、成功、中断の 2 種類である。失敗ならば GD_GUNIFY_FAIL により、abort する。

印字: 実行時カーネルのプリントルーチンが Data object を発見した場合に呼びだされる。次のメソッドを呼ぶ。

PRINT メソッド: 引数は、UNIX の FILE 構造体へのポインタ、オブジェクトのプリントの深さや幅を示す値である。その FILE 構造体へのポインタが示す UNIX のストリームに自分の印字内容を流す。

PRINT メソッドは GD_RETURN_FROM_PRINT にて return する必要がある。

GC: GC 時に生きているオブジェクトの (新領域への) コピーの仕方を記述したメソッドである。実行時カーネルの GC ルーチンがデータ追跡中にジェネリック・オブジェクトを発見したときに利用される。このメソッドがあるため、タグを持たない要素 (例えば、float の value 部) を持ったオブジェクトでも安心して作成することができる。通常は、自分自身をコピーする。内部に KL1 の項を持つ場合などは再帰的にコピーさせる必要がある。

なお、KLIC の GC は幅優先ではなく、深さ優先の順でデータのコピーを行う。よっていわゆる“scan”は行わず、ジェネリック・オブジェクト内部で保持しているタグのないデータも上手く扱うことができる。

メソッドでは、コピー先ヒープのヒープ割付点へのポインタを引数に持つ。自分のサイズに従って、コピー先ヒープ上に領域を確保し (ヒープ割付点を進める)、その自分をそこへコピーする。メソッドの返り値はコピー後のヒープ割付点へのポインタを返す。

gfloat では、value 部が 8byte align されている必要がある場合が多いため、align を合せる

*2 中断の場合は中断原因の変数への参照

ためヒープを進めている。

ジェネリック・メソッド: 今までのメソッドは、基本的には実行時カーネルから直接呼び出され、実行時カーネルが行うべき仕事を肩代りして行うために記述されたメソッドであった。これらのオブジェクトは実装されていることが必須である。

これらのメソッドの他に、ユーザが自由に定義し、ジェネリック・オブジェクトを任意の時点で任意に操作するための様々なメソッドが定義できる。例えば、float では、各種の演算が行えなければ、float を実装する意味はあまりない。このような、オブジェクトに依存するようなメソッドをジェネリック・メソッドと呼ぶ。

ジェネリック・メソッドにはガード用とボディ用があり、2 つ (1 つのジェネリック・オブジェクトクラス当たり 2 つ) のメソッドとして実装される。ジェネリック・メソッドを呼ぶ時には、そのメソッド群の中の 1 メソッドを指定するメソッド記述子を引数として呼ぶことで、実行したいメソッドを指定する。

body generic メソッド: ボディ用のジェネリック・メソッドである。引数は、メソッド記述子およびジェネリック・メソッド引数、そしてヒープ割付点へのポインタである。

Body generic メソッドは、以下の 2 種より構成される。

- 様々な独立した処理を記述したメソッド。float では、print_1 と int_1 が定義されている。
- 上記も含めた様々な処理に dispatch するためのメソッド。つまり、コンパイルコードからはこのメソッドが直接的に呼びだされ、dispatch を行い、必要であれば、上記個別処理に飛ぶようになっている。

前者は GDDEF_METHOD(メソッド名_アリティ) として定義される。後者は、GDDEF_GENERIC により定義される。float の場合を例にして基本的な処理を以下で説明する。

1. GD_SWITCH_ON_METHOD なるマクロで、method 名/アリティにより dispatch される。個別の処理 (GDDEF_METHOD により定義される処理) に飛ぶ為には、この SWITCH の中で GD_METHOD_CASE(method 名/アリティ) として定義をしておく。
2. 残りについて default: で受け、アリティで SWITCH するため、GD_SWITCH_ON_ARITY を用いる。

その結果、case 1(アリティが 1)、case 2(アリティが 2) の場合で各々 GD_SWITCH_ON_METHOD により再度 dispatch している。その結果行うべき処理を GD_METHOD_CASE_DIRECT で記述する。別に GDDEF_METHOD により定義せずに、直接記述する (単順な処理の場合にはそのような記述が適当であろう) 場合にはこの記述が便利である。

guard generic メソッド: ガード用のジェネリック・メソッドである。この場合も、以下を除き body とほぼ同様である。

- 個々の処理のためには GDDEF_GMETHOD マクロを利用する。

- Dispatch のためには GDDEF_GGENERIC マクロを利用する。その他、GD_SWITCH_ON_GMETHOD, GD_GMETHOD_CASE などと記述する。

引数は、メソッド記述子およびジェネリック・メソッド引数である。このメソッドはガードゴールに相当する。メソッドを実行した結果は成功/失敗/中断のいずれかになるが、それは戻り値により表現する (GENERIC_SUCCEEDED 成功、GENERIC_FAILED: 失敗、それ以外: 中断の原因となった変数への参照)。詳細は *include/klic/generic.h* の *guard_generic* マクロを参照のこと。

new ルーチン: メソッドは、既にヒープ上に生成されたジェネリック・オブジェクトのメソッド表に登録されたものである。メソッド表には、ジェネリック・オブジェクトを生成するためのメソッドを含めず、new ルーチンとして、メソッド表とは別に、各ジェネリック・オブジェクトに対して定義される。何故なら、生成の時にはオブジェクトは存在せず、結果、メソッド表へのポインタも存在しないためである。

new ルーチンは、ジェネリック・メソッドと同様に、コンパイルされた実行時ルーチンから呼ばれ得るルーチンである。また、new ルーチンに引数としてデータを渡すことも可能である。これを new ルーチン引数と呼ぶことにする。

このルーチンは引数として、new ルーチン引数とそのサイズおよびヒープ割付点へのポインタを取る。ヒープ上に領域を取り、そこにジェネリック・オブジェクトを生成する。新たなヒープ割付点をルーチンの戻り値として返す。

6.1.3 Consumer object

Consumer object については、マージャを例として説明する (*runtime/gmerge.c*)

Consumer オブジェクトを記述するに必要なマクロ群は *include/klic/gc_macro.h* に記述されている。また、マクロの名称は一般的には GC_, または GCDEF_ で開始されている。

単一化: consumer object は文字通り、データ消費者で、他からデータが送られてきたら動き出す。よって、自発的には実行を開始しようとしないので、consumer object に対しては passive unification は無意味である。よって、active unify に対応する UNIFY メソッドしか存在しない。

consumer object は変数に フックしたゴール列である中断構造中のフック鎖の中に入っている。通常の KL1 項とフックされた変数の active unify 時に、鎖をたどっていった 実行時カーネルが consumer object を見つけた場合に、このメソッドを呼び出す。

引数として、単一化対象の他のデータ、そしてヒープ割付点へのポインタを取る。例えば、merger の入力ストリームに vector を 単一化する時の処理から分かるように、このメソッドの中身は通常かなり複雑なものになる。メソッド実行の結果としては、成功、中断、

GC 要求の 3 種類である。それぞれの状態に対応する値^{*3}を大域変数を通して返す。active unification には goal の エンキューを行うなど、ヒープを消費する場合があるので、メソッド引数として、ヒープ割付点へのポインタを持ち、メソッドの返り値として、メソッド実行後の新たなヒープ割付点ポインタを返す。

GCDEF_UNIFY() により定義される。ここで利用されている主なマクロ群の解説を行う。

GC_SWITCH_ON_TERM: 単一化対象項の型により dispatch する。引数はラベル名称。

GC_TRY_TO_ALLOC: 指定された分のデータをヒープに割りつける。割付に失敗したら
ならば、そこで gc_request に飛ぶ。

GC_UNIFY: 2 つの引数の項をその場で単一化する。

GC_KL1_UNIFY: 2 つの引数の項を単一化するゴールをエンキューする。メモリが不足した時などに利用する。

GC_SELF: オブジェクト自身

GC_RETURN_WITH_HOOK: 再度フックされるべき項を返す。

印字 実行時カーネルのプリントルーチンが consumer object を印字する。次のメソッドを呼ぶ。
print メソッド: data object の print メソッドと同様である。

GC: GC に関しての処理は data object の時と同様である。Hook 関連の処理は kernel がやるため、自オブジェクト、および、自分が参照している分のコピーだけを行えばよい。

new ルーチン: new ルーチンに関しても、data object と同様である。

6.1.4 Generator object

Generator については、乱数発生器 (*runtime/random.c*) を例として記述する。

単一化: 単一化については、generator は active unification 用のものしか持たない。しかしながら、以下の 2 つのメソッドを定義することが可能であり、特定のクラスとの単一化について、幅広い処理を行うことができるようになっている。

UNIFY メソッド: 単一化の対象となるオブジェクトが渡される。対象オブジェクト種別に対して処理を変更したり、対象のオブジェクトに対しての操作を行ったりすることができる。また、「遅延」を返すことが可能であり、遅延の時には次の generate メソッドが呼びだされる。

「遅延」かどうかは、戻り値が 0 であるかどうかによる。0 の場合は「遅延」である。非ゼロの場合には、新しいヒープ割付点を示す。

GENERATE: 文字通り、generator オブジェクトを他のデータ型に「変身」させるためのオブジェクト。オブジェクトが関連付けられている変数の値が必要がゴールがあるときや、active_unify が失敗したときなどに呼びだされる。

^{*3} 中断の場合は中断原因

戻り値は 3 通りである。

`makeref(0)`: メモリ不足等の理由により生成を中断した。再度、`generate` メソッドが呼び出されることが期待される。

`makecons(0)`: 内部エラー。

上記以外: 生成された項を示す。

乱数発生器の実装では、特に単一化の相手による処理の変更などはないため、`UNIFY` に対しては常に失敗を返し、`GENERATE` に対しては乱数を `CAR` にもち、`CDR` に生成器 (つまり自分自身) を持つ新しい `hook` を入れた `CONS` を返す。

印字: `PRINT` メソッドはこれまでと同様。

`GC`: `GC` についてはこれまでと同様。

`NEW` ルーチン: `new` ルーチンに関しては、これまでと同様である。

第 7 章

GC

KL1 は自動メモリ管理を前提とした言語であり、KLIC は GC を備える。KLIC の GC のアルゴリズムは大まかには以下のような特徴がある (殆どはこれまでに既に述べてきたことである)。

- “stop and copy” GC。よって、ヒープは 2 面を持つ。
- コピーの順は、深さ優先であり、新領域の scan は行わない。これは、ジェネリックオブジェクトのように、GC が理解できないようなデータ (端的にはタグがないようなデータ) について、不用意に GC がアクセスし誤った処理をしないようにするための配慮である。
- ヒープのサイズは可変であり、GC 後に free な領域がすくなくなるとヒープのサイズを拡張する。

GC の処理は、*runtime/gc.c* に記述されている。

7.1 基本的な GC のアルゴリズム

基本的、一般的な KLIC の GC は、以下のように行われる (`copy_one_term(runtime/gc.c)`)。

1. コピー対象の項のアドレスを GC スタックに積む (`push_gc_stack()`)(図 7.1(1))。
`copy_one_term` では 1 つの項しか積まないが、一般的には任意個の項を積んで良い。
2. GC スタック先頭中のアドレスの示す項に対して、
 - 一重ループ未定義変数 (= 純粹未定義変数) であれば、新領域に未定義変数を生成し、旧領域からその変数にポインタを張る。(純粹未定義変数である、ということは、その GC ルート自身もヒープに存在する場合に限られる)。
 - さもなくば、とりあえず新領域にコピーし (図 7.1(2))、そのデータが
 - アトミックならばそれで終了
 - 旧領域を指すポインタであれば、新領域のコピー先を指すように旧領域のコピー対象を書きかえ、その旧領域データを GC スタックに積む (図 gc-stat(3))。構造であれば、基本的には構造の内容同様に検査しスタックに積む。

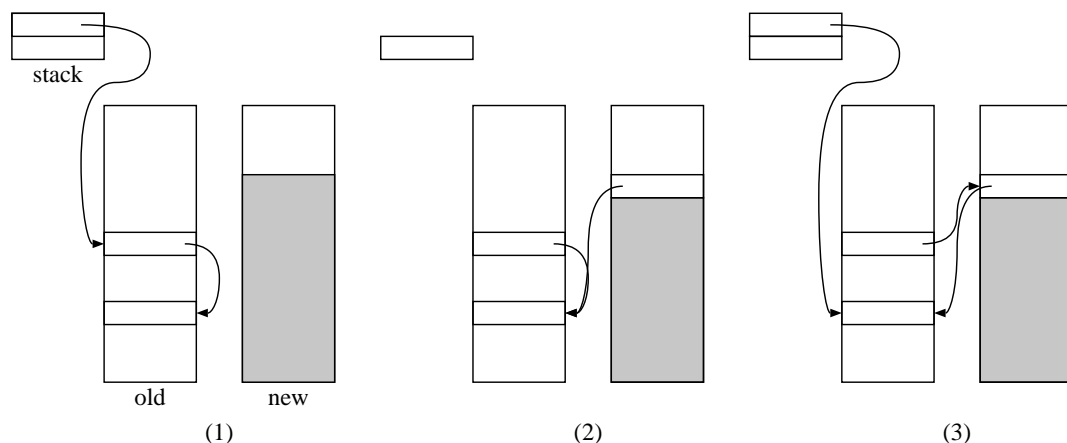


図 7.1 KLIC の GC の基本原理

3. 2 にジャンプし、GC スタックが空になるまで続ける。

7.2 GC の対象領域

KLIC では、ヒープ内のデータ、ヒープ外のデータ (主に定数構造体) が存在するが、GC の対象となるのはヒープ内のデータである。よって、一般的には「ヒープ旧領域」にはないデータはヒープ新領域か、ヒープ外に存在することになり、コピーの対象にはならない。

7.3 Copy されたかどうかの判定

KLIC では構造体の内部を直接指すポインタを許しているため、構造体全体がコピーされているのか、構造体の一部がコピーされているのかを判断する方法は必ずしも自明ではない。

重複してコピーを行わないため、コピーされたかどうかを判定する必要がある。

- ATOMIC データの場合には「旧領域にある ATOMIC データはコピーされていない」という基準で判断する。
- 通常の REF は「旧領域を指している限りその先にはコピーされていないデータがある」という判断で良い。
- CONS の場合、その CDR 部が CONS タグで新領域を指している場合には、CAR/CDR 共に新領域にコピー済みであるとする。CONS のコピーは、CONS タグを持ったセルと 2 ワードセル同時に行うため、CONS タグのセルと 2 ワードセルがこのような新旧領域に跨がることはないので特殊なマークとして利用できる。
- FUNCTOR の場合、1 ワード目がさらに FUNCTOR タグであれば、それは新領域へのコピー先であることを表わす。FUNCTOR タグの先は、ATOM である (通常の関数

FUNCTOR) であるか、REF である (data object) 以外には通常ありえないため、特殊なマークとして利用できる。

- ゴールレコードの場合には、
 - エンキュー済みのゴールレコードの場合には、ゴールスタックをコピーすることにより行われ、ゴールスタック同士でゴールレコードを共用することはないため、重複コピーはなされない (中断構造中で発見しても、タグが REF であればすでにエンキューされているので、コピーしない (66 ページ、第 2 章参照))。
 - 中断ゴールは、コピーの後に、next field にコピー先へのポインタを入れる。さらに、pred 部に 0 を入れる。よって、中断ゴールで、pred 部に 0 が入っているものはコピー済みである。さらに、ゼロが入っておらず、かつ、next field が REF であるものはすでに schedule されたものであるので、中断ゴールとしては copy する必要がない (goal queue をコピーするときにコピー対象になる)。
- Consumer, generator object では、先頭の field が新領域へのポインタになっていればそこがコピー先である (コピーされる前はこの field はメソッド表を指しており、これはヒープの外である)。

7.4 GC のきっかけ

GC のきっかけは、リダクションの切れ目にて、割り込み検査が行われ、真に heap が溢れそうなことが判明することによる (*runtime/intrpt.c*, *klic_interrupt()*)。 (56 ページ、第 3 章参照)。

7.5 GC のためのデータ構造

GC のために付加的に用いられるデータ構造は (2 面あるヒープを除けば) *gcstack* である。これは GC 途中のデータを置くスタックである。つまり、C のレベルのスタックを用いることなく、自前で用意したスタックを用いて GC の再帰処理を行っている。

7.6 GC のアルゴリズム

GC のアルゴリズムを、実際のソースコードを参照しつつ説明する。GC 処理の top level は *runtime/gc.c* の *klic_gc* である。

7.6.1 *klic_gc*

この *klic_gc()* 中では、heap 拡張などを含む、GC についての全体制御を行っている。

1. *make_heap_larger* が真かどうか検査する (初期的には偽)。
 - *make_heap_larger* が真であれば、

- (a) heapsize を倍にするため、heapsize, bytesize などの大域変数を変更する。(この時、maxheapsize が有効であればその検査をしていることに注意)。
- (b) new_old_space_top に bytesize 分のメモリを確保する。
 - メモリが確保できたならば、old_space_top を free し、old_space_top, old_space_size を更新する。
 - メモリが確保できず、heap 拡張不能なことが致命的である (lastgc_dangerous が真) 場合には fatal error とする。

この lastgc_dangerous は、heap 拡張が不能な場合、そのまま続行すると、無限ループを起こす可能性があるため、複数回 heap 拡張が不能になったかどうか検出するために設けられている。
- make_heap_larger が偽でかつ、heap 拡張不能なことが致命的である (lastgc_dangerous が真、詳細後述) 場合には fatal error とする。
- 2. (上記の処理で fatal error にならなかった場合) flip_space() にて、xxx_space_top, xxx_space_size を old/new について swap する。
- 3. copied_susp を 0 に初期化する。これは「永久中断の検出」のために用いる、GC により発見された中断ゴールの数を意味する。
- 4. collect_garbage を呼び出す。この関数は現在実行中のゴールスタック (qp)、および、他の優先度のゴールスタック、さらに GC フックされた他のデータ構造を GC ルートとしてコピーを行う。
- 5. コピーをした結果、空領域が「近々に必要なヒープサイズ (this_more_space) を考慮して不足しそうな場合 (heapsize 中、maxactivation で指定される比率を上まわるかどうか検査し、必要に応じて make_heap_larger を ON にする。
- 6. コピーをした結果、「近々に必要なヒープサイズよりも空領域が少ない」ということがわかれば、lastgc_dangerous を ON にし、もう一度 GC を行う。

7.6.2 collect_garbage()

実際に copy を行う部分。

1. GC を行うのが最初であれば、gcstack の実体を割りつける。
2. allocp, ntop, heaptop を news_space_top とする。otop を old_space_top とする。osize を、old_space_size とする。
3. gc_hook_table に登録されている関数群を起動する。このテーブルには、処理の際に、ゴールレコード以外に、GC ルートとなるような関数群を register_gc_hook()(runtime/alloc.c) なる関数により予め登録されている。現在の KLIC では以下のような名称の関数および処理が登録されている。

gc_asyncio_stream: 非同期 I/O を行う file descriptor に毎にある、I/O ストリームの登

録先。

gc_timer_data: タイマ用 stream I/O のための変数の登録先 'REFERtimer'。

gc_signal_stream: 割り込み酔うの変数の登録先 (53 ページ、第 4.1.1 章参照)。

gc_exp_table: 輸出表に登録されているデータ群 (並列実装)。

gc_decode_stack: メッセージデコード中のデータ群 (並列実装)。

4. 優先度毎に goal queue をコピーする (copy_one_queue()) (81 ページ、第 7.7 章参照)。

7.7 copy_one_queue()

goal queue を 1 つコピーする。

1. コピーする時には、ゴールの末尾よりコピーする。これは、以下のような理由による。
 - 一般的に 1 つの変数を共有する「書き込みゴール」と「読み出しゴール」があるときに、読み出しゴール内に純粋未定義変数が置かれることが実行効率上有利である (手繰りの段数が少なくても良い)。
 - 一方、GC によりゴールレコードをコピーする際には、先にコピーを行った方に純粋未定義変数セルは確保される。よって、読み出しゴールを先にコピーしておきたい。
 - 一方、通常の記述では、「読み出しゴール」は「書き込みゴール」よりも後に実行される (ゴールスタックの後方に位置する)。

よって、ゴールスタックの順にコピーを行うよりも後からコピーを行う方が有利である。

2. ゴールを 1 つコピーする (マクロ copy_one_goal())。
 - (a) アリティよりゴールレコードサイズを知り、新領域にゴールレコードを確保する。
 - (b) next, pred field をコピーする。旧領域のゴールの next 部にはコピー先、pred 部には 0 を代入する。
 - (c) 引数の末尾より GC スタックに積むための処理を行う (マクロ reserve_copy())。直接的に純粋未定義変数、ATOMIC、コピー済み参照でなければ新領域に旧領域の内容をコピー、旧領域に新領域への参照を書き、新領域へのポインタをスタックに積む。
3. GC スタックを top から順に再帰的にコピーする (copy_terms())。

7.8 copy_terms()

GC スタックの内容をコピーする。GC の心臓部分。

1. GC スタックトップを取りだす (addr)。この addr は今後書き換えられるが、常に「書き戻すべきアドレス」を指している一段 deref したものを obj とする。
2. obj のタグに応じて処理する。
 - ATOMIC の場合には、addr の先に obj を書き、1 へ。

- REF の場合には、obj の先を deref し、value に入れる。value のタグに対応し、以下の処理を行う。
 - － REF の場合:
 - * 一重ループであり、かつ、addr が新領域にあれば、addr 自身を一重ループにし、1 にもどる。
 - * 一重ループであり、かつ、addr が新領域になければ、新領域に一重ループを生成し、それへの REF を addr に入れ、1 にもどる。
 - * 二重ループであれば、その先は generator か hook 構造体であるのでコピーする。
 - ・ Generator であれば、新領域に 1 ワードわりつけ、addr に書きこむ。このワードは、二重ループのエントリになる。object の先頭を調べ、コピー済みでなければ、そっくりそのままコピーする (Generator 自身のコピーは、GC method を呼び出す)。先頭をコピー先に書きかえる。コピー済ならば、suspend 構造までを作成し、generator のコピー先のアドレスをコピーする。
 - ・ 中断ゴールまたは consumer であるならば、不要となった goal(すでに scheduling されたゴール) を避けつつ、コピーする。
 - * 一/二重ループでなく、かつ、value が旧領域を指しているならば、obj を value とし、2 へ (つまり、dereference loop を成す)。
 - * value が旧領域以外 (つまり、新領域 or ヒープ外) であれば、この先はコピー済であることがわかったので、addr の先に value を書きこみ 1 へ。
 - － CONS の場合には、value が新領域であれば、これは CONS2 ワードともすでにコピーされていることを意味する (78 ページ、第 7.3 章参照) ので、それを addr の先に書きこむ。新領域でなければ、obj に value をコピーし、一ループレベル上の CONS の項 2 に飛ぶ。
 - － ATOMIC であれば、addr の先に ATOM 値を書き 1 に。
 - － FUNCTOR であれば、obj に value をコピーし、一ループレベル上の FUNCTOR の項 (83 ページ、第 2 章参照) に飛ぶ。
- obj が CONS であることが判明すれば、それが新領域を指しているか、旧領域を指しているか判断する。新領域であれば、それは CONS2 ワードともすでにコピーされていることを意味する (78 ページ、第 7.3 章参照) ので、それを addr の先に書きこむ。旧領域であれば、CDR を読み出す。
 - － CDR が構造体 (実際に意味を持つのは CONS) タグを持ち、新領域を指していれば、それは、CDR の示す先に CONS 全体がコピーされていることを意味するので、addr の先に CDR をコピーする。
 - － 上記以外であれば、まだその CONS はコピーされていない。
 - (a) 新領域ヒープ上に 2 ワードセルを確保する。
 - (b) CAR 部分をコピーするために、reserve_copy によりスタックに積む。

- (c) 新しい 2 ワードセルへ CONS ポインタを、旧領域のコピー元の 2 ワードセルの CDR 部分、および、addr の先 (追跡が開始されたセルで新領域にある) にコピーする。
 - (d) CDR 部分が ATOMIC であれば、そのままコピーし、2 にジャンプする。ATOMIC でなければ、新しい 2 ワードセルの CDR 部分のアドレスを addr に、obj に CDR をコピーし、スタックに積まずに 2 にジャンプする。
- つまり、この CDR 処理部分は、末尾再帰処理の最適化をしている。
- Obj が FUNCTOR タグであることがわかれば、それが旧領域を指すかどうか検査する。旧領域でなければ addr の先にコピーしてそのまま終了。旧領域であれば、第一ファンクタを読み出し、そのタグを調べる。
 - 第一ファンクタが REF であれば、これはデータオブジェクトであることを意味するので、GC method を呼びだし、それへのポインタを FUNCTOR タグ付きでコピーする。旧領域の第一ファンクタ部も新領域の object への FUNCTOR 付きポインタとし、addr の先に書きこむ。
 - 第一ファンクタが REF でなく、FUNCTOR でもなければ、これは FUNCTOR であることがわかるので、新領域にファンクタの領域を確保し、第一ファンクタへのポインタを FUNCTOR タグ付きで addr の先、旧領域のオブジェクトの第一ファンクタ位置に書きこむ。さらに全ての要素についてのコピーをスタックに積む (reserve_copy())。
 - 第一ファンクタがファンクタであれば、これはコピーされた FUNCTOR またはオブジェクトであるので、その第一ファンクタの内容を addr の先に書く。

以上の処理のあと、1 にジャンプする。

3. スタックが空になれば、処理は終了する。

第 8 章

トレーサ

トレーサは、プログラムの動作状況をユーザに提示することによって、プログラムがユーザの想定したと同じ動作をしているかどうかをユーザに確かさせることによって誤りを発見するための、基本的なデバッグ・ツールである。

8.1 機能概要

KLIC のトレーサは、概略以下の機能を持つ。

- プログラムのステップ実行
- トレース不要プログラム部分のトレースなし実行
- 注目する述語にいたるまでのトレースなし実行
- その他、付随する諸機能

提供機能の一覧を表 8.1 に示す。

8.2 提供機能の概要

KLIC では、以下の操作の繰り返しによりリダクションを進める。

1. ゴールプールからゴールを 1 つ取ってくる。(ゴール呼びだし操作)
2. ゴール引数の値を、KL1 節のガード部の規則にしたがってチェックし、チェックに適合した節を 1 つ選んでそのボディ部を実行し、生成されたサブゴールをゴールプールに入れる。(リダクション操作)
3. 2 で呼び出されたゴールの引数に変数等になっており、チェックが即時にできない時、ゴールをサスペンションプールに入れる。(サスペンション操作)
4. 2 で呼び出されたゴールが失敗した場合、プログラムを呼び出したプロセスに制御が戻される。逐次版の KLIC では、プログラムの実行を終了し shell に戻る。(失敗操作)

トレーサでは、各操作をモニターする部分を「ポート」と呼ぶ。ゴール呼びだし操作、リダクション操作、中断操作、失敗操作についてのポートはそれぞれ、「CALL ポート」、「REDU ポート」、「SUSP ポート」、「FAIL ポート」と呼ばれる。

図 8.1 に、KL1 の実行モデルと各ポートの図を示す。

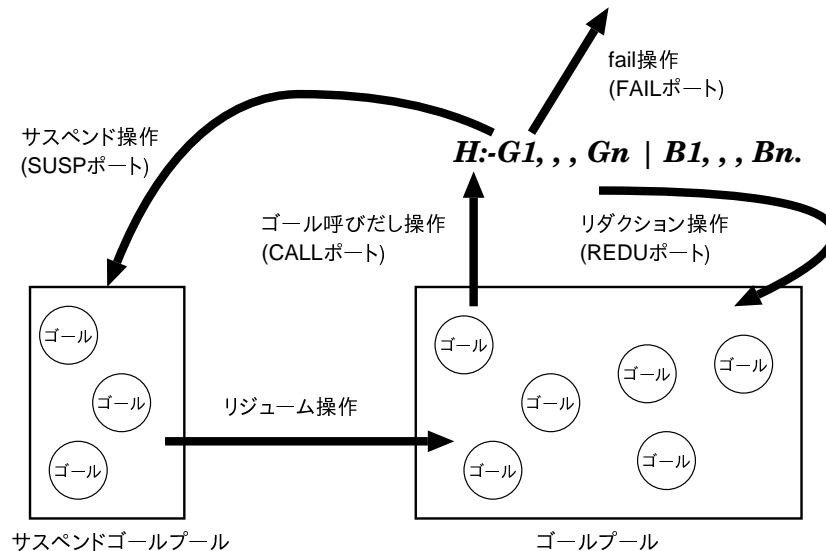


図 8.1 KL1 の実行モデルと各ポート

トレースモードでコンパイルされたプログラムは、ゴールが上記の 4 つのポートを通過する時に、実行を停止してユーザからの入力待ちとなったり、そのゴールの表示を行ったりする。

以下で、実現方式について解説を行う。

8.3 リンク時のトレース指定

多くのトレーサを持つ言語処理系では、解釈実行型のシステムを別途用意する、または、トレースを行うか否かによってユーザ・プログラム自身をコンパイルし直すといった方法により、トレースの有無に応じて意味は同じでも実現の詳細がかなり異なるプログラム・コードを動かすことによって、トレースができる柔軟性と、トレースしない場合の効率性を両立させている。

KLIC のトレーサではこのような処理をせずに、ユーザ・プログラムについてはまったく同じオブジェクト・コードを用い、リンクする実行時システム・ライブラリを通常実行用とトレース用の二種用意し、リンクをし直すだけでトレースの有無を切替えられるような方式をとった。このために、実際にゴールのリダクションを行うユーザ・プログラムを 1 リダクションずつ進め、リダクションの前後の状況の差を検出することによってトレースを行っている。

具体的には、ゴール・リダクションごとに必要なヒープ溢れの検査を、トレースのきっかけとして利用している。つまり、「一例外処理」としてトレースのきっかけを実装している (55 ページ、第 2 章参照)。

トレース用のライブラリで、トレース対象となるゴールを実行する直前にヒープあふれの比較対象であるヒープの上限値をダミーの値に設定することによって、1 リダクションごとに強制的にヒープあふれ処理ルーチンに割出させるようにする。このあふれ処理ルーチンの中でトレース用の割出しか、本当のヒープあふれかを检查し、必要ならトレースを行う。なお、KLIC でデフォルトで利用されるのはトレース用のライブラリである。

8.4 名前情報の管理

KLIC では一般的には実行コードにしてしまえば、その述語名、モジュール名はもはや必要はない。しかしながら、KLIC のトレースでは、それらの情報をユーザに提示するし、モジュール、述語を特定させる必要があるため、名前情報を保持している必要がある。

KLIC では、トレース時のみリンクされるようなデータベースを持つことにより管理を行っている。

- 名前管理のため、以下の表が生成される。

述語表: すべての `pte(pred_table_entry)` の配列を保持する、変数 `pred_table`。

述語ハッシュ表: `pte` の配列を検索するための述語構造体、`pte` へのポインタの表で、変数 `pred_hash`。

モジュール表: モジュール情報 (`mte: mod_table_entry`) を持つ表、変数 `mod_table`。

モジュールエントリ表: モジュール情報へのポインタの表。変数 `mod_index`。

なお、構造体 `pred_table_entry`、`module_table_entry` は以下に示す構造を持つ。

```
struct pred_table_entry {
    Const struct predicate *pred; /* 述語情報 */
    Const struct mod_table_entry *mte; /* 所属するモジュールの mte */
    Const unsigned char *name; /* 述語名 */
    char spied; /* 述語としてスパイされているかどうか */
    char default_trace; /* デフォルトのトレースフラグ */
};

struct mod_table_entry {
    module (*func)(); /* そのモジュールを実現する関数へのポインタ */
    Const unsigned char *name; /* モジュール名 */
};
```

- これら名前表は、`make_name_tables()` (`runtime/trace.c`) により以下のように生成される。
 1. 配列 `defined_modules` を営め、全モジュールの数を数える。さらに、`defined_modules` より述語表を営め、全述語数を数える。この `defined_modules` は、KLIC コンパイラに

より生成されるデータであり、predicates.c に定義されている。

2. 得られた述語数を用い、述語エントリを割り付け、得られた述語数より少々大め (述語数の 1.5 倍を越える 2 冪程度) にハッシュエントリ表を割り付け、hash_mask を決め、すべてのエントリを 0 で初期化する。
3. 全てのモジュール表を嘗め、さらに、それを經由して全ての述語を嘗め、述語エントリ (pte) を初期化する。pte は以下の形式を持つ。

エントリ名	説明
Const struct predicate *pred	述語構造体
Const struct mod_table_entry *mte	所属するモジュールの mte
Const unsigned char *name	述語名
char spied	spy 中かどうかのフラグ。初期的には off にする。
char default_trace	デフォルトのトレースフラグ。初期的には on にする。

4. 調べたモジュール数より、モジュール表、モジュールエントリ表を割り付ける。
5. 全てのモジュールを嘗め、モジュール表を初期化する。出現順にモジュールエントリ表も初期化する。
6. モジュール表は名前順にソートし、モジュールエントリ表はモジュール関数順にソートする。
7. モジュール表を検索しながら、述語表の mte を初期化する。
8. 述語表を述語名順にソートする。
9. 全述語を、述語構造体アドレスで検索できるようにハッシュテーブルに登録する。

また、pte, mte を、pred 構造体から検索する関数として、get_pte(), get_mte() が各々定義されている。

8.5 トレースの制御と情報入手

トレースを行うためには:

- 適切なトレースのタイミングで、トレースのためのルーチンが呼ばれること
- トレース・ルーチンからトレースに必要な情報を得ることができること

が必要である。KLIC のトレーサでは、以下のような方法を用いてトレースのタイミングをつかみ、必要な情報を得ている。全体として、トレース不要部分の実行効率が、まったくトレースしない場合とほとんど変わりなくなるように留意している。

KLIC では、種々のポートでユーザが実行に介入できるように主に以下のような処理を行っている。

トレースのきっかけ: トレース対象となるゴールの実行直前に、「擬似的なゴール」を実行することによりトレースのきっかけとしている。これは、擬似的なゴール内に本来のゴールを持つようなゴールを作成しておき、それを実行する。この「擬似的ゴール」の解説については第 8.5.1 章 (88 ページ) を参照のこと。この擬似的にゴールがスケジュールされた際には必要な処理 (例えば、CALL ポートでのユーザコマンドの受付) を行った後に、本来のゴールをエンキューしなおすことにより実現されている。

子ゴールの認識: リダクション処理が行われた結果生成されたゴールについては、以下のように処理されている。

- リダクション直前 (つまり、「トレースのきっかけ」の時) に、ゴールスタックの先頭を記録しておく。
- リダクションが終了した際に、再度ゴールスタックを検査する。原則的には、このリダクション前後のゴールスタックの差分が生成されたゴールである。

ただし、他の優先度のゴールスタックにゴールが置かれた場合、また、再実行可能になった場合などがあるため、それらのゴールについては特別な配慮が必要になる (詳細後述)。

このリダクションが終了した際に、子ゴールについてもトレースすることが適当な場合には、前述の「擬似的なゴール」により、本来のゴールを保持させるようにゴールを変更しておく。

8.5.1 擬似ゴールの構造

KLIC の実装では、実行する各ゴールは対応するプログラム・コードへのポインタを保持している。

トレース対象のゴールについては、本来実行すべきコードのかわりに、トレーサのコードへのポインタを格納しておくことによって、実行しようとした時に自然にトレーサのコードが呼び出されてしまうようにする。これによって上述の CALL ポートに対応するトレースが可能になる。本来実行すべきコードへのポインタはゴールの追加引数として保持する。CALL ポートで実行を継続する場合は、このポインタを用いて本来のコードを呼び出せば良い (図 8.2 参照)。

具体的には、トレース対象ゴールに対して、トレースがなされるように処理される。具体的にはこの処理は `trace_goal()` で以下のように行われる。

1. 本来のゴールよりも 3 ワードサイズの大きなゴールを用意する。
2. 対応する引数の、`trace_trigger_preds` を述語として指定する。この述語内では、関数 `trace_trigger_routine` があたかも `module` 関数であるかのように実行される。
3. 引数を本来のゴールからコピーする。
4. 追加されたゴールスロットに対して、以下のデータを格納する。
 - 元のゴールの述語構造体

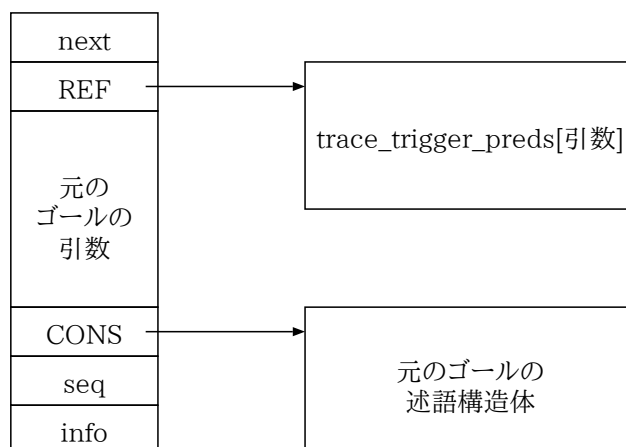


図 8.2 トレース用擬似的ゴールの構造

- トレースゴール ID。PE 毎にトレース対象ゴールに付加される整数値。
- トレース情報 (multi window でデバッグしたときに追加するパラメータ群。
詳細割愛)

なお、`untrace_goal()` なる関数は上記の逆をおこなう。ただし、ゴールレコードの再確保は行わず、単に述語構造体の付替のみを行い、トレースゴール ID を戻り値として返す。

8.5.2 CALL ポートのトレースのきっかけ

先にも延べたように、トレースのきっかけは擬似ゴールで関数として指定されている module 関数 `trace_trigger_routine` が実行されることにより行なわれる。この関数の起動は、通常の KL1 がコンパイルされて生成された module 関数と同じ方法にて行なわれる。よって、この方式では、トレース対象かどうかの条件判断が不要で、通常の実行の効率に影響を与えることなく、必要な時にだけトレースできる。

トレース用の実行時ライブラリでは、トレース用のフラグ (-t) が指定された時には、実行の開始 (`runtime/kmain.c`) にあたって初期ゴール (`main:main`) を作る際に、トレースを行うように加工している。すべてのトレースはこれをきっかけに始まる。

このようにトレース化されたゴールがスケジュールされると、`trace_trigger_routine()` が本来のモジュール関数と同様に実行される。その結果:

- `info` にそのゴールに指定されたトレース情報を持つ (このトレース情報は `trace_trigger_routine()` 終了時に大域変数 `parent_info` に退避され、`trace_after()` にて用いられる)。
- `untrace_goal()` により非トレース化する。
- 指定されているモジュール関数を調べ、ノード番号、優先度番号を wait するような関数が指定されている場合には、本来の述語をシステムヒープ部に割りつけ、以下の処理を行う。

- 述語構造体より、`get_pte()` により `pte` を得る。
- `pte` を参照し、述語スパイフラグをコピーする。`trace_flag` を立てる。
- `qp_before` に、リダクション前のゴールスタックの先頭 (`qp->next`) を記録しておく (図 8.3 での「記憶」に相当)。
- Leap 中、Spy 中であるか、などによりゴールを印字する (`print_goal()`)。さらに、Leash 中であればコマンドをユーザから読む (`call_port_command()`)。
- トレース中 (`trace_flag` が ON) であれば (コマンドを実行した結果トレース中ではなくなる可能性がある)、`heaplimit` を 0 にし、真のモジュール関数を実行する。

その結果、次のリダクションが終了したときに、割り込みが検出され (`klic_interrupt()` *runtime/intrpt.c*)、`trace_flag` が ON であることより、`trace_after()` が呼びだされる。

8.5.3 REDUCE/SUSPEND ポートのトレースのきっかけ

上述のように CALL ポートで実行を継続する場合、ヒープの上限値をダミー値に設定した後、実行を継続する。これによって、1 リダクションを経た後にヒープあふれの割出しが起きる。別途用意したトレース・フラグを用いれば、本当のヒープあふれなのか、トレースのためのダミー処理なのか (あるいは両方か) を区別することができる。これによって、ヒープあふれ処理ルーチンの入口から、上述の REDUCE ポートをトレースするルーチンを呼び出すことができる (`klic_interrupt()`, *runtime/intrpt.c*)。

変数が未定義であったためにゴールが実行を中断することもある。この場合には中断処理のためのルーチンが呼ばれる (56 ページ、第 4.2 章参照)。トレース用のライブラリ中では、トレース・フラグを調べて、必要なら SUSPEND ポートのトレースを行うルーチン、`trace_susp()` または `step_susp()` を呼び出す。中断処理自体かなりのコストを伴うので、フラグの検査のオーバーヘッドは相対的に小さい。したがって、トレース用ライブラリを用いても、実際にトレースを行わない場合の効率はほとんど落ちない。

CALL ポートでそのゴールのトレースを不要と指示されれば、このようなヒープ上限値とフラグの設定を行わなければならない。

8.5.4 リダクション結果の子ゴールの認識

プログラムの実行はゴール・スタック中のゴールをプログラムにしたがって次々にリダクションしていくことによって行われる。このゴール・スタックは優先度ごとに別のものがある。

リダクションはその時点で空でない最高優先度のゴール・スタックの先頭のゴールについて行われる。このゴールをリデュースした結果の子ゴールは、それぞれの優先度に対応するゴール・スタックの先頭に入れられる。

REDUCE ポートを処理するルーチン、`trace_after()` には、親ゴールの優先度に対応するゴール・スタックの先頭へのポインタが渡される。CALL ポートでその時点のゴール・スタックの先頭

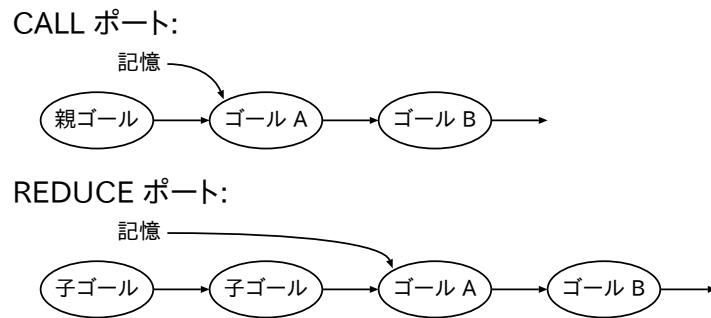


図 8.3 子ゴールの認識

(リデュースされるゴール自身の次のゴール) は `qp_before` として記録されているので、リデュース結果として生成された子ゴールはどこまでかを判別できる (図 8.3。 `qp_before` は図中「記憶」に相当する)。

この方法だけでは、親ゴールと異なる優先度の子ゴールを把握できない。異なる優先度の子ゴールができるのは、優先度指定があるボディ・ゴールについてだけである。このような優先度指定付きのゴールについては若干面倒な処理が必要なので、コンパイルしたオブジェクト・コード中で直接行うのではなく、実行時サブルーチンと呼ぶようになっている。そこで、トレース用の実行時ライブラリでは、このサブルーチンをトレース・フラグが立っている場合にはトレース用情報を保存するようにしている。

多くのプログラムでは子ゴールの優先度が親ゴールと同じである場合が支配的である。したがって、トレース用の実行時ライブラリを使う場合に生じる、優先度指定付きゴール処理ルーチン内のフラグ判定オーバーヘッドは、通常ほとんど無視できる程度である。

8.5.5 リデュースにともなう再開ゴールの把握

ゴールのリデュース時には、リデュースにともなうユニフィケーションによって中断していたゴールが再開することがある。この処理は実行時ライブラリ `resume_goals()(runtime/unify.c)` 中のサブルーチン中で、`trace_resumption()` を呼び出すことにより行われている。

トレース用の実行時ルーチンでは、トレース・フラグが立っている間に再開したゴールを記憶して、後の REDUCE ポートでのトレースの際に報告するようにしている。再開処理もコストの高い処理なので、フラグ判定コストは無視できる。

8.5.6 リデュース結果のゴールのトレース

トレースされたゴールのリデュースの結果できた子ゴールや、リデュースにともなって実行を再開できたゴールは、REDUCE ポートでトレースを指定できる。トレースのためには、上述のプログラム・コードへのポインタを、本来のコードからトレース・ルーチンのコードへ差し替え

ておけばよい。この処理は、`trace_after` の内部で行われる。

8.5.7 トレースされた中断ゴールの再開の把握

トレース対象のゴールが変数値の待ち合わせのために中断し、トレース対象外のゴールのリデュースによってそれが再開される、という場合もある。これは上述の方法だけでは把握できない。

トレース対象のゴールが中断する場合には、プログラム・コードへのポインタの差し替え処理を行った状態で中断させておく。これは前述の `trace_susp` の内部で行われている。これによって、実行を再開し、そのゴールのリダクションを始めようとする時には、自然にトレース・ルーチンが呼ばれるようになる。

8.5.8 `trace_after()` の処理

これまでの説明の各所で出現した、`trace_after()` で行われている処理について、以下に纏める。

1. トレース状態を、親ゴールがスケジュールされたときの状態 (`parent_info`) にする。
2. ゴールスタックの先頭から、親ゴールがスケジュールされたときの先頭ゴール (`qp_before == before`) までのゴールを調べる。
 - (a) 再開によりエンキューされたゴールかどうか調べる。

再開によりエンキューされたゴールは、「再開ゴールリスト (`different_prio_resume`)」に記録されているので、ゴールスタック上のゴールがこのリストに記録されているかどうか調べれば判明する。

 - 再開によりエンキューされたゴールであれば、そのゴールが再開によるものであることを記録する。
 - 再開によるものでなければ、そのゴールが再開によるものであることを記録しない。
3. 現在の優先度ではないゴールがエンキューされた時に記録されるリスト (`trace_enqueued_goal`) を調べる。
 - (a) 再開によりエンキューされたゴールかどうか調べる。

再開によりエンキューされたゴールは、「再開ゴールリスト (`different_prio_resume`)」に記録されているので、ゴールスタック上のゴールがこのリストに記録されているかどうか調べれば判明する。

 - 再開によりエンキューされたゴールであれば、そのゴールが再開によるものであることを記録する。
 - 再開によるものでなければ、そのゴールが再開によるものであることを記録しない。
4. `REDUCE port` が `enable` になっており、Leap 状態ではないか、そのゴールが Spy 状態であれば、以下の処理を行う。
5. 親ゴールの処理を以下のように行う。

- (a) まず、ゴールが印字対象かどうか検査する。以下の何れかであれば印字対象となる。
 - 「部分項モード」ではない。
 - 「部分項モード」で対象ゴールである。
 - (b) 上記の検査で印字対象であれば、REDU として印字する。
6. Subgoal について以下の処理を行い、必要あればゴールを印字する。また、印字するか否かに関わらず、subgoal を記録しておく (current_subgoals)。
- (a) まず、ゴールスタック中にあるゴールを調べる。親ゴールの時と同様の条件で判断し、必要あればゴールを印字する。
 - (b) 次に別優先度のスタックにエンキューされたゴールを調べる。親ゴールの時と同様の条件で判断し、必要あればゴールを印字する。
7. REDUCE ポートが leash 状態であるか、当該ゴールが spy 状態であるか y どちらかならば、REDUCE ポートのコマンドの入力を則す。
8. ゴールスタック中のゴール、別優先度のゴールリストを調べ、trace 状態にすべき subgoal については、trace_goal 処理を行う。さもなくば、untrace_goal 処理を行う。
9. 別優先度のゴールについては、ゴールをエンキューする (トレース中は、直接エンキューされず、このリストに記録されるので、処理が終了した後にエンキューする)。
10. trace.enqueued_goal を空にし、割り込みを終了状態にする。

コマンド名	コマンド入力形式	入力可能ポート	コマンドの意味
Continue	<cr>,c	全てのポート	対象となるゴールのトレースを続ける
Abort	a	全てのポート	対象となるプログラムのトレースを止める
Skip	s	全てのポート	対象となるゴールのトレースを止める
Leap	l	全てのポート	spy されたサブゴールが現れるまで、 トレースの表示・入力を抑制する
Enable port	E ポート名	全てのポート	そのポートを観測する
Disable port	D ポート名	全てのポート	そのポートは観測しない
Leash port	L ポート名	全てのポート	そのポートでユーザからの入力待ちにする
Unleash port	U ポート名	全てのポート	そのポートでゴールの表示だけ行う
Trace	+ サブゴール番号	REDU ポート	そのサブゴールをトレース対象とする
Notrace	-サブゴール番号	REDU ポート	そのサブゴールをトレース対象外とする
Toggle trace	サブゴール番号	REDU ポート	そのサブゴールをトレースフラグを反転する
Notrace default	n 述語	全てのポート	その述語をトレース対象外とする
Trace default	t 述語	全てのポート	その述語をトレース対象とする
Spy	S 述語	全てのポート	その述語にスパイをかける
Nospy	N 述語	全てのポート	その述語のスパイを外す
Set Print Depth	pd:引数	全てのポート	表示する構造の深さの変更
Set Print Length	pl:引数	全てのポート	表示する構造の長さの変更
Toggle Verbose print	pv	全てのポート	verbose モードへの切替え
Status query	=	全てのポート	現在のパラメータの状況を表示
List Modules	lm	全てのポート	全モジュールの表示
List Predicates	lp	全てのポート	全述語の表示
Help	h,?	全てのポート	全モジュールの表示
Queue	Q	全てのポート	ゴールプール内のゴールの表示

表 8.1 トレーサのコマンド一覧

第 II 部

並列処理系

第 9 章

概要

本章では、メッセージ通信版、共有データ通信版双方について概要を述べる。

9.1 両版の違い

メッセージ通信版、共有データ通信版の最大の違いは、「ノード間で物理的にデータを共有しているかどうか」である。メッセージ通信版では、すべてのメモリ空間はノード毎に分割されており、データの共有はない。一方、共有データ通信版では、全てのノードで共有しているようなメモリ空間が存在し、そこに置かれた KL1 データをノード間で読み書きすることによりノード間通信を行う (KLIC の共有データ通信版では、ノード間で共有されないようなメモリ空間も存在し、そこに置かれたデータは、単一のノードでのみ利用可能となる)。

9.2 ノードの表現

共有メモリ版では、ノードは、UNIX のプロセスとして実装される。メッセージ通信版では、PVM を利用する場合にはノードは PVM のタスクとして実装され、PVM を利用しない場合には UNIX のプロセスとして実装される。

9.3 逐次版とのインターフェース

並列版は、原則、「ノード間の参照」を表わす参照セルをジェネリックオブジェクトにより導入することにより実現されている。「参照」は基本的には KL1 レベルでは未定義変数、または、透過的に見えるべき参照セルである。

メッセージ通信版: 基本的に、外部への参照を表現するセルを generator、外部から参照されているセル、または、外部へ参照をしているが、すでに dereference を開始しているセルを consumer により表現している。

これらへの unify, dereference などの kernel からの操作をこれらの object がメッセージ通信

に翻訳することにより分散処理が行われる。

共有メモリ通信版: 基本的に、共有メモリ中に置かれる変数セルは `generator` として実装される。

基本的には (まさに通信されている間を除くと)、他のデータ構造については逐次処理系と全く同じものを用い、それに対しての処理も基本的に同じものとする。

9.4 両者の切りわけ

各々の切り分けはソースコード上は以下のようになっている。

メッセージ通信版でのみ必要なファイル群: `runtime/dist.c`, `runtime/lock.s`, `runtime/config` ディレクトリ以下。

共有メモリ版でのみ必要なファイル群: `runtime/shm_*.c`。

その他、(逐次版とも) 共用されているファイル群: 内部にて、マクロで切りかえられている。

DIST: メッセージ通信版としてコンパイルするときのマクロ。

SHM: 共有メモリ版としてコンパイルするときのマクロ。

第 10 章

メッセージ通信版

メッセージ通信版には、以下の 3 種類が存在する。

PVM 版: ノードプロセスの生成、通信全てに関して PVM を利用した版。

PVM-TCP 版: ノードプロセスの生成に関しては PVM を、通信に関しては TCP socket を用いた版。

共有メモリ版: メッセージの通信路として共有メモリを用い、PVM を全く利用しない版 (当然、分散メモリ計算機上では動作しない)。

上記は基本的には、ごく低レベル (バイト列) のメッセージの送受信が異なるのみで、より上位のレベルでは同じ操作を行っている。実装も、上位レベルの通信と、下位レベルの通信とは明確に切り分けられており、上位レベルの通信に関する処理は通信方式に関わらず同じである (図 10.1 参照)。

他の版と共用されるソースコードについては、先に記述したように、メッセージ通信版では、マクロ DIST を define する。「共有メモリによるメッセージ通信版」ではさらに、マクロ SHM_DIST を define する。

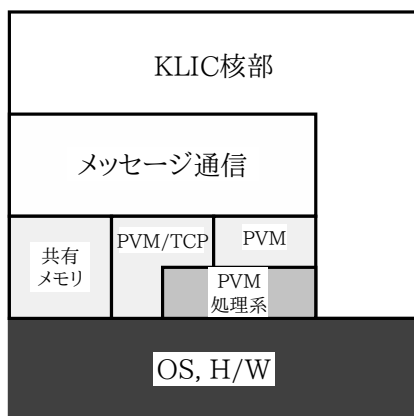


図 10.1 メッセージ通信版 KLIC の構成

この文書では、すべての通信方式に共通な部分についての説明を主に行う。

10.1 処理系の構成

メッセージ通信実装では以下の2つの基本方針を設定した。

- メッセージ通信実装のための逐次核の変更は最小限に押え、逐次実行時の性能を落さない。
- 各種の並列マシンへの移植性を高めるため、マシン依存の処理部分を独立させる。

上記の方針に基づくメッセージ通信実装の構成を、図 10.2 に示す。

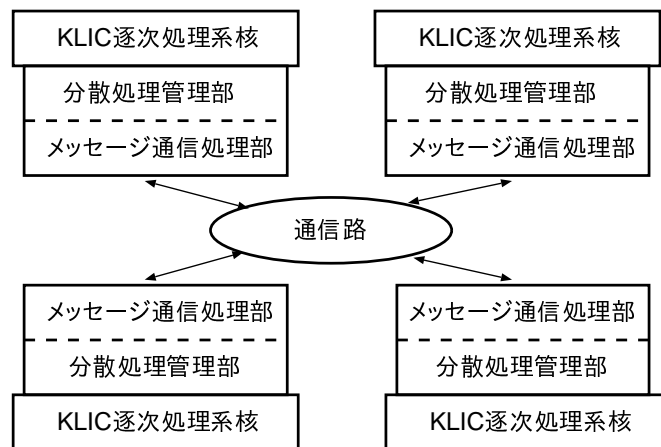


図 10.2 メッセージ通信実装の構成

メッセージ通信実装では、KLIC の逐次核に分散処理管理部とメッセージ通信処理部を付加したプロセスが複数実行される。これらのプロセス間では、各プロセスの分散処理管理部とメッセージ通信処理部を経由して通信が行われる。

分散処理管理部分は、並列推論マシン PIM で用いた分散処理方式を KLIC 用に再構成した方式を用いた。分散処理管理部分の処理方式の詳細は第 10.2 章 (99 ページ) に述べられる。

分散処理用のデータは、KLIC のジェネリックオブジェクトを使って実装されており、また、メッセージ受信処理用のメッセージハンドルーチンは、逐次核の割り込みハンドラテーブルにエントリを追加する形で実装されているので、逐次核の主処理部分を変更せずに分散処理を実現できた。

10.2 基本方式

ここでは、基本的な実装の方式について述べる。

10.2.1 莊園

分散メモリ環境において、莊園はひとつの「莊園プロセス」と(一般に複数の)「里親プロセス」からなる。莊園プロセスとすべての里親プロセスは、莊園の生成時に作られ、莊園の消滅時に消滅する。動的な生成/消滅は行わない。莊園プロセスと里親プロセス群はメッセージで通信する。ゴールはいずれかの里親プロセスで実行され、データは里親プロセス内に存在する。ここで、里親プロセスは「ノード」と呼ぶことにする。

終了検出

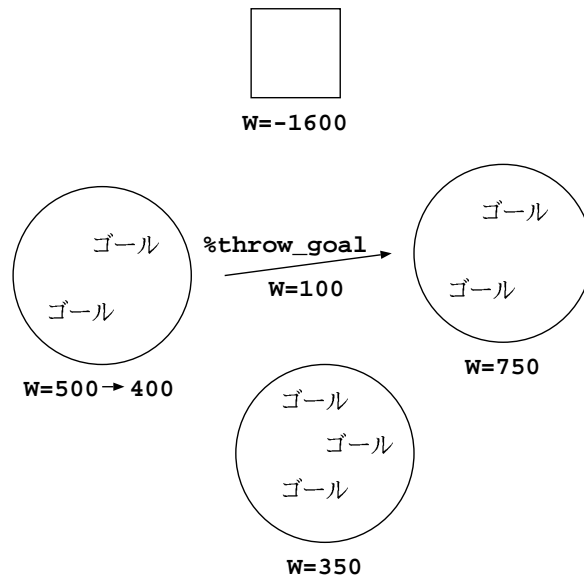
莊園の終了検出は「WTC 方式」によって行う。これは、莊園プロセスに負の重みを、里親プロセスとメッセージには正の重みを付け、重みの合計をゼロに保つことによって終了検出を行うものである。

莊園プロセス及び里親プロセスがメッセージを送信する時はメッセージに重みを付ける。自分の重みはメッセージにつけた分だけ減らす。実行を終了した(実行可能及び中断しているゴールが存在しない)里親プロセスは自分の重みを `%terminated` で莊園プロセスへ返却する。重みの返却を受けたら莊園プロセスは自分の重みに加える。その結果ゼロになったならば莊園の終了が検出される。図 10.3 にゴール (`%throw_goal` メッセージ) の送受信と里親終了時の処理を示す。

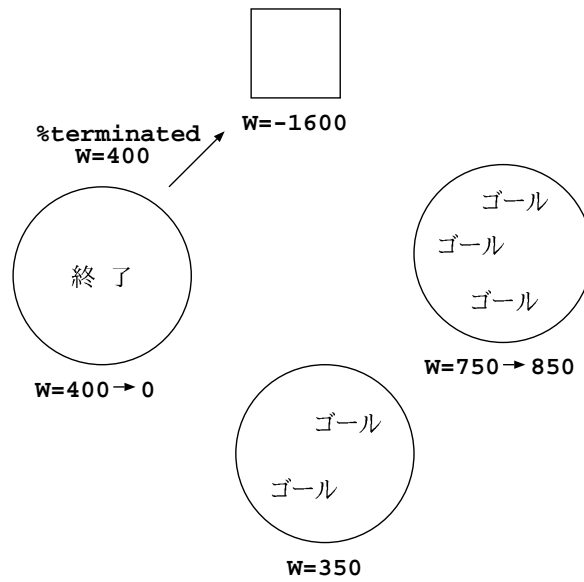
中断検出

莊園内に、実行を中断しているゴールは存在するが、実行可能なゴールはなく(少なくともひとつの里親プロセスは終了していない)、メッセージも存在しない状態を中断状態と呼ぶ。この莊園の中断状態を以下の処理によって検出する(図 10.4 参照)。

1. 莊園プロセスが `%check` をすべての里親プロセスへ送信する。`%check` には重みは付けない。
2. `%check` を受信すると里親プロセスは中断している(ゴール群の)重みを `%suspended` で報告する(返却ではない。里親プロセス自身の重みはそのままである)。中断している(中断しているゴールはあるが実行可能なゴールはない)ならば「中断している重み = 自分の重み」を報告し、「実行中」実行可能なゴールがある)、あるいは「終了」している(中断しているゴールも実行可能なゴールもない)ならば「中断している重み = ゼロ」を報告する。
3. `%suspended` を受信すると莊園プロセスは報告された「中断している重み」を加算して行く。すべての里親プロセスから `%suspended` を受信した時、加算してきた重みの合計と莊園の重みとの和がゼロになるならば「莊園が中断している」ことが検出される。合計がゼロにはならない場合や途中で `%terminated` を受信した場合は「中断していない」ことがわかる。

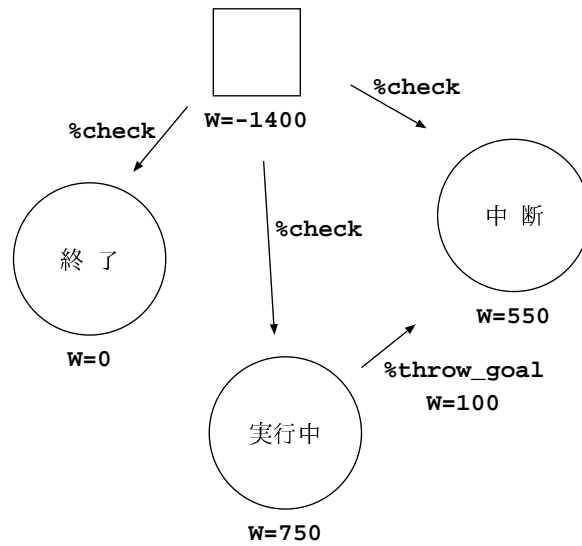


(1)ゴールの送信

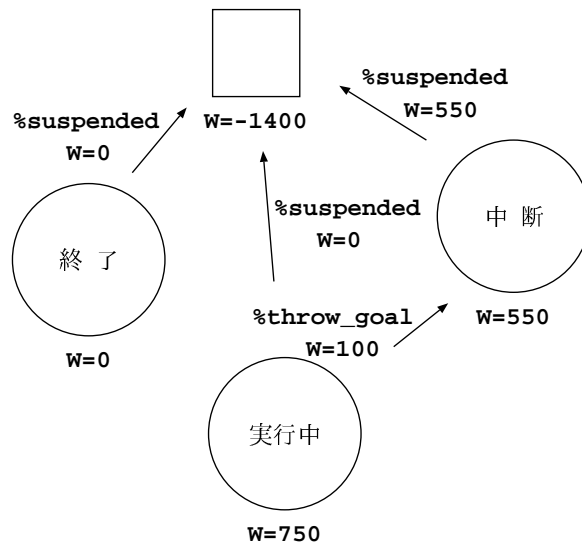


(2)ゴールの受信と重みの返却

図 10.3 WTC 方式：ゴールの送受信と里親の終了

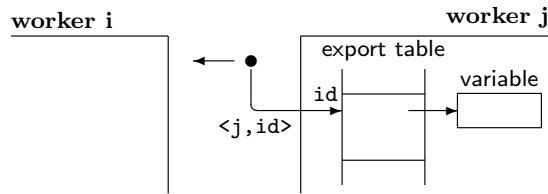


(1) %check をすべての里親プロセスへ送信する。

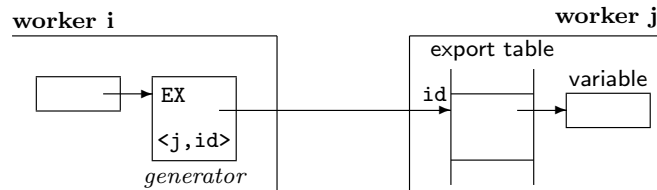


(2) 状態に応じて %suspended を送信する。

図 10.4 荘園の中断検出



(1) 変数への参照を輸出表に登録して送出する.



(2) ジェネレータ外部参照オブジェクト の生成.

図 10.5 外部参照ポインタの生成.

10.2.2 外部参照ポインタ

この章では、まず、ジェネリック・オブジェクト を用いた外部参照ポインタの実現方法を述べる。次に、外部参照ポインタに係わる二つの処理、分散ユニフィケーションと参照値の読み出し (dereference) を述べる。これらの処理における具体的な操作は、オブジェクトのメソッドとして定義され、処理系核によって呼び出される。KLIC 分散メモリ処理系ではワーカ間に渡るループの生成を避けることはできない。最後に、このループに対処する方式を述べる。

表現形式

メッセージを他のワーカに送出し、そのメッセージの引数に変数であると「他のワーカのメモリを指すポインタ」が生まれる。このポインタが**外部参照ポインタ (external reference)**である。

各ノードは**輸出表 (export table)**と呼ぶアドレス変換テーブルを持ち、各外部参照ポインタは輸出表を介してデータセルを指す。外部参照ポインタはノード番号と輸出表のエントリ番号の対で表現する (図 10.5 参照)。

ジェネリック・オブジェクトを用いた実現

外部参照ポインタはジェネリック・オブジェクトを用いて実現した。ポインタに対しては「参照値の読み出し (dereference)」と「ユニフィケーション」を行うことがある。前者の処理は値の生成要求と考えられるので *generate* メソッドで実現し、後者は *unify* メソッドで実現した。

外部参照ポインタの生成時は二つの処理とも行いするのでジェネレータ・オブジェクトで実現する。ゴールの実行において、あるポインタの指す値が必要になると、処理系核はそのポインタを実現

しているジェネレータ・オブジェクトの *generate* メソッドを呼び出す。 *generate* メソッドは読出し処理を行い、オブジェクトはジェネレータからコンシューマへ変化する。もはや読出し処理を行う必要はないからである。コンシューマに変わることにより冗長な読出し処理を防ぐことができる。

コンシューマ (群) がフックしている変数に対してユニフィケーションが行われると、処理系核は各コンシューマの *unify* メソッドを呼び出す。このメソッドの実行により分散ユニフィケーションが行われる。

外部参照ポインタを表わすオブジェクト (ジェネレータ、コンシューマ) を以下ではそれぞれジェネレータ外部参照オブジェクト、コンシューマ外部参照オブジェクトと呼ぶ。

また単に外部参照オブジェクトと呼んだ時はジェネレータ外部参照オブジェクトあるいはコンシューマ外部参照オブジェクトであることを示す。

ジェネレータ外部参照オブジェクトは *exref* なるクラス名称の *generator* として、*runtime/ge_exref.c* にて定義されている。コンシューマ外部参照オブジェクトは *read_hook* なる名称の *consumer* として *runtime/ge_readhook* として定義されている。

輸出表

まず、外部から参照されている (外部参照ポインタより指されている) オブジェクトの存在があると、以下のような問題が生じる。

- GC は各ノードで全く独立に行われる。あるデータはノードの内部では参照されていないが、他のノードからは参照されているかもしれず、そのため、正しく GC を行うためには「他のノードから参照されているデータ」を記録しておくことが必要になる。
- データのアドレスは GC により変更される。GC は各ノードで全く独立に行われるため、他ノードについてデータアドレスを直接記録しておく、GC の度にその記録を更新する必要が生じる。よって、外部から参照されているデータについては GC で移動しない、アドレス以外の ID が必要になる。

そこで、「外部から参照されているデータ群」が記録されている表を作り、その表の *index* をもって外部からのデータの ID とすることにする。この表を輸出表と呼ぶ。よって、外部データを参照する場合、「ノード番号」と「輸出表インデックス」のペアをもって特定する。

輸出表の構造は、*runtime/include/klic/interpe.h* 内で、*struct exp_entry* として以下のように線型リストとして定義されている。

```
struct exp_entry{
    long index;
    long wec;
    q data;
    struct exp_entry *next;
};
```

輸出表の根は `exp_table` である。輸出表は KLIC のヒープとは別に free list 管理されている (`runtime/export_table.c` で定義されている関数群を参照)。

外部参照ポインタの生成

リストなどの構造体や変数を引数に持つメッセージを他のノードへ送る時に、送付先で外部参照ポインタは生まれる。送信元ノードは変数などを輸出表に登録しアドレスを「ノード番号&輸出表エントリの ID」(これを「外部参照アドレス」と呼ぶ)に変換して送る。これを「輸出」処理と呼ぶ。この処理は、`runtime/datamsg.c` 中の `encode_data()` 内で行われており、この処理はデータのエンコード時、つまりメッセージ送信時に行われる。

一方、そのメッセージを受信した側では、受信データ内に「外部参照」を意味するものが含まれている場合には、(そのデータを始めて受信したならば) ジェネレータ外部参照オブジェクトを生成する。この処理は `runtime/datamsg.c` 内の `decode_exref()` 内で行われている。

外部参照ポインタの解放

外部参照ポインタの GC (分散 GC) は、重み付け参照カウント (WRC = Weighted Reference Counting) を用いたアルゴリズムを適用する。この、WRC アルゴリズムを外部参照ポインタ GC に適用したものを特に WEC (Weighted Export Counter) と読んでいる。このアルゴリズムは、輸出表エントリと外部参照ポインタが正の任意の重み (WEC) を持ち、「ある輸出表エントリの重み」と「そのエントリを指す外部参照ポインタの重みの合計」が等しくなるように制御するものである。ある輸出表エントリの重みがゼロならば、そのエントリを指す外部参照ポインタは存在しない (そのエントリがゴミになった) ことが保証される。この WEC は「輸出表」(104 ページ、第 10.2.2 章参照) と `exref class`、`read.hook class` のオブジェクトの内部に記録されている。通常のリファレンスカウントであると、参照側の重みはつねに 1 として、参照数を正確に数えることにより参照管理を行うが、WEC (= WRC) 方式では、重みが 1 にかぎらない。その結果、参照を他のノードにコピーする際に、参照元と通信を行うなどして参照数の管理を管理する必要なく、自分の「重み」を他のノードに「分割する」ことにより参照をコピーすることができ、効率的な分散参照管理が可能となる。

すなわち、ジェネレータ外部参照ポインタを「分割」して他のノードへ渡す時は、重みを分割し、一方を他ノードへ渡すポインタに付け、もう一方はノード内に残す。これを「分割輸出」と呼ぶ (図 10.6 (1))。重みが分割できない時は「間接輸出」を行う。これは変数と同じように外部参照ポインタを輸出表に登録して輸出するものである (図 10.7)。コンシューマ外部参照ポインタはいつも間接輸出を行う。

さて、外部参照ポインタは以下の時に「解放」する。

1. `%answer_value` の受信、あるいは具体化によって参照値がわかった時。どちらの場合も、最終的には `runtime/ge_exref.c`、または `runtime/ge_readhook.c` 内の `unify` メソッドでの処理される。

2. ノード内一括 GC によってどこからも参照されていないことがわかった時。

この処理のために、そのノード内に存在する外部参照をすべて記録している。この記録されている表を「輸入表」と呼ぶ。この輸入表は、*runtime/import_table.c* 内で定義されている各種関数でメンテナンスされている。GC が終了したときには、*scan_imp_table()* が呼びだされ、この輸入表を各オブジェクト毎に営めコピーされたかどうかを判断する。

外部参照ポインタを解放した時はポインタが指すノードへ **%release** を送信しポインタが持っていた重みを返却する (図 10.6 (2))。 **%release** を受信すると、対応する輸出表エントリの重みから返却された重みを引く。引いた結果ゼロになったならば輸出表エントリを解放する。

%release は *send_release()* 関数 (*runtime/cntlmsg.c*) を呼び出すことにより行われる。このメッセージを受けた場合には、対象オブジェクトを輸出表から外す。その結果、もしそのデータがそのノード内でも参照されていなければ、次の GC ではゴミになる。

これらの処理は、データを encode する関数 *encode_data()* (*runtime/datamsg.c*) *exref class* については、データ送信時に呼びだされる *encode method* により以下のように実装されている。

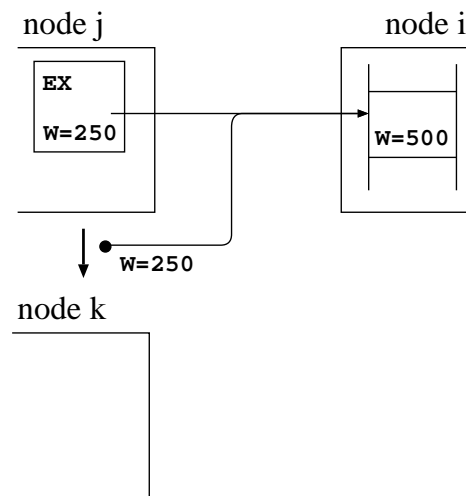
1. 送出対象が *generator* である場合、
 - *encode method* を呼び出す。
 - *encode method* 内では *WEC* を調べ、一定未満 (*MIN_WEC* の倍) 以下であれば、*encode* をやめる。さもなければ、*encode* を行い、*WEC* を分割する。
 - *encode_data()* では、*encode method* が終了し、成功していたならばそのまま処理を続ける。失敗していたならば間接輸出処理 (*label Make_exref* 以降) を行う。
2. 送出対象が *consumer* である場合には、間接輸出処理をして終了。

10.2.3 分散ユニフィケーション

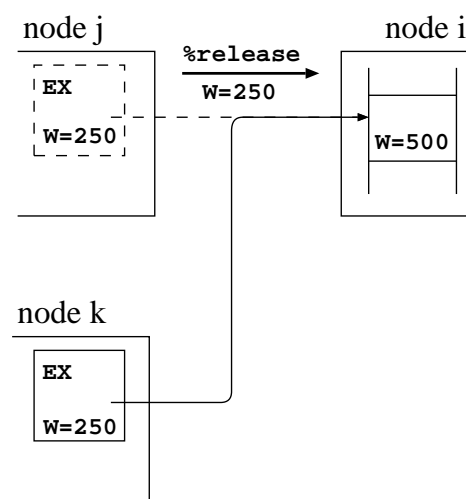
外部参照オブジェクトに係わるユニフィケーションを 分散ユニフィケーションと呼ぶ。分散ユニフィケーションは、外部参照オブジェクトがフックしている変数に何かのデータがユニファイされると起きる。このユニフィケーションを実行するため処理系核は以下の処理を行う。

1. 変数をデータで書換える。
2. フックしている外部参照オブジェクトについて *unify* メソッド を呼び出す (複数のコンシューマ外部参照オブジェクトがフックしている時は、それぞれのコンシューマについて *unify* メソッド を呼び出す)。
3. *unify* メソッド を呼び出した結果 “失敗” したならば、*generate* メソッド を呼びだし、再度単一化処理を行う。

これらの処理の結果として分散ユニフィケーションが行われる。分散ユニフィケーションの詳細を以下に示す。



(1) 重みを分割して輸出する。



(2) 解放により重みを返却する。

図 10.6 外部参照ポインタの分割輸出と解放

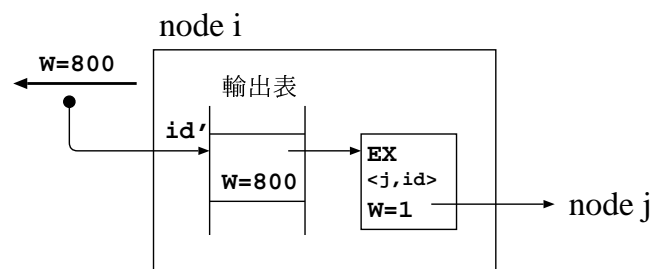


図 10.7 外部参照ポインタの間接輸出

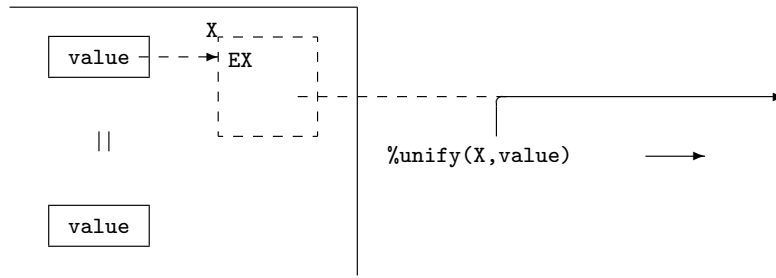


図 10.8 外部参照オブジェクト X と具体値のユニフィケーション.

外部参照オブジェクト X と具体値 value

処理系核は, X がフックしている変数に具体値 value を書き込み, X の *unify* メソッドを呼び出す. メソッドの実行により, X の指すワーカへ「具体値とのユニフィケーションを依頼するメッセージ (%unify(X,value))」が送られ, X は解放される (図 10.8). このメッセージを受信したワーカでは, X の指すデータと具体値 value のユニフィケーションが行われる.

変数に複数のコンシューマ外部参照オブジェクト がフックしていた時は, それぞれのコンシューマについてメッセージが送信される.

ジェネレータ外部参照オブジェクト X と, コンシューマ (群) あるいは中断ゴール (群) がフックした変数 Y

処理系核は以下の処理を行う.

1. X の *generate* メソッドを呼び出す. メソッドの実行により, 参照値の読出しを依頼するメッセージ (%read) が送信され, X はコンシューマに変わる.
2. コンシューマ となった X のフックしている変数と Y のユニフィケーションを行う. この結果, コンシューマ (群) と中断ゴール (群) はひとつにまとまり, 二つの変数はポインタでつながる (図 10.9).

ジェネレータ外部参照オブジェクト 同士 (X, Y)

一方のジェネレータ (X とする) の *unify* メソッドを呼び出す. メソッドの実行により, X が指すワーカへ %unify(X,Y) が送られ, 「X がフックしている変数」に「Y がフックしている変数へのポインタ」が書込まれ, X は解放される.

コンシューマ外部参照オブジェクト (群) がフックしている変数同士

二つの変数をユニファイし, 二つの コンシューマ (群) をひとつにまとめる. このユニフィケーションではメソッドの呼び出しは行われない. 処理系核の処理だけで完結する.

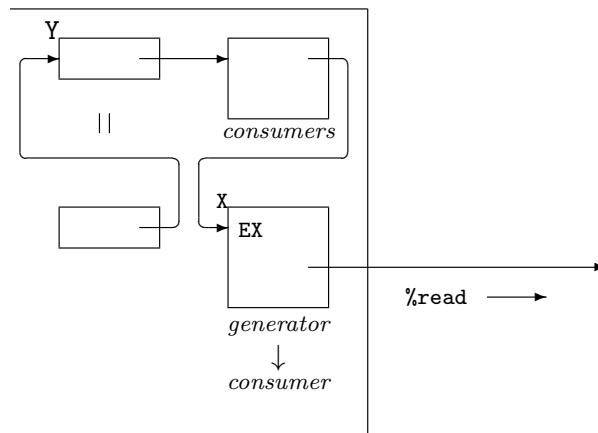


図 10.9 ジェネレータ外部参照オブジェクト X と コンシューマ (群) のユニフィケーション.

外部参照オブジェクト X と変数 V

処理系核は, X がフックしている変数へのポインタを V に書込む. この処理も処理系核の処理だけで完結する.

具体的な実装

以上で挙げた処理の具体的な実装は、概略、通常のユニフィケーションの処理で行われる (`unify()`, `unify_value()`, `runtime/unify.c`). つまり、以下の処理である。

- どちらかがジェネレータ外部参照オブジェクトであること確認する。
- ジェネレータ外部参照オブジェクトであることがわかれば、もういっぽうの項を引数にして、`unify method` を発行する。
- `Unify` に失敗したならば、`generate` を行い、それともう一方の項とのユニフィケーションをする。

これより先の処理 (もう一方の項の判定、および、メッセージの送信) は、ジェネレータ外部参照オブジェクトの `method` により実装されている。

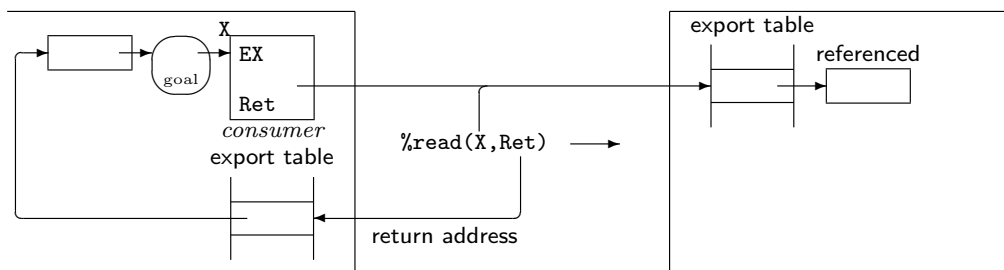
10.2.4 参照値の読出し (dereference)

ゴールの実行において、ある外部参照ポインタの指す値が必要になると、処理系核はそのポインタの `generate` メソッドを呼び出す。 `generate` メソッドの実行により 参照値の読出し処理が行われる。

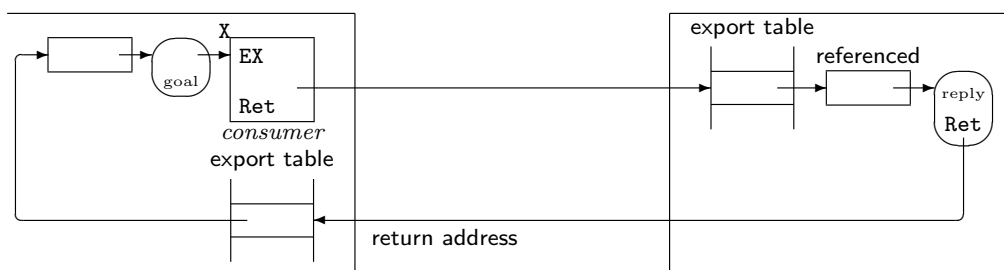
ジェネレータ X の `generate` メソッドが呼ばれると、X が指すワーカへ `%read(X,Ret)` が送られ、X はコンシューマに変わる。 `Ret` は参照値の返信先であり、X がフックしている変数を指す外



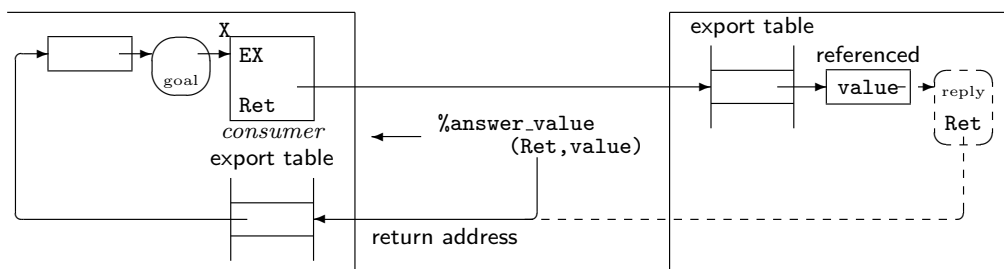
(1) 読出し開始前. 外部参照ポインタはジェネレータ外部参照オブジェクトで実現されている.



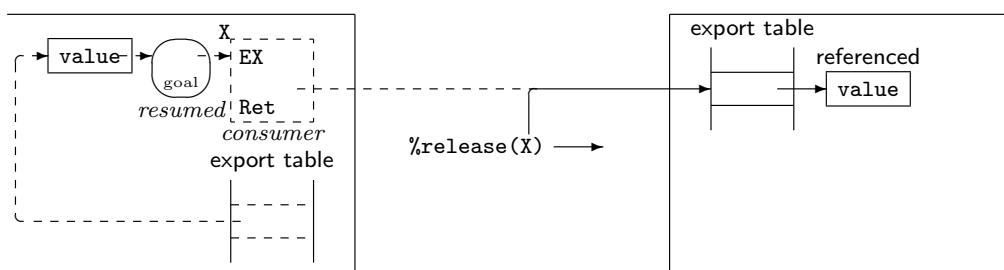
(2) 返信先を設け, `%read` を送信する. ジェネレータはコンシューマに変化する.



(3) 参照先が変数なのでリプライ・オブジェクトを生成しフックする.



(4) `%answer_value` を返信し参照値を送る.



(5) 具体値を書き込み, コンシューマ外部参照オブジェクトを解放する.

図 10.10 外部参照ポインタが指す参照値の読出し.

部参照ポインタである (図 10.10 (1), (2)).

`%read(X,Ret)` を受信したワーカは, `X` の参照先が具体化されていれば「参照値を運ぶメッセージ (`%answer_value`)」を直ちに返信して参照値を送る.

参照先が変数^{*1}の場合は, 返信先を記録したコンシューマ・オブジェクト (リプライ・オブジェクトと呼ぶ) を生成し変数にフックする (図 10.10 (3)). 変数が具体化されるとリプライ・オブジェクトの `unify` メソッドが呼ばれ, `%answer_value` が返信される (図 10.10 (4)).

参照先がジェネレータ外部参照オブジェクト `X'` の場合は, `X'` が指すワーカへ `%read(X',Ret)` を転送する. 返信先はそのままであり, 最初に `%read` を送信したワーカへ参照値が直接送られる. 外部参照ポインタがチェーンしている (外部参照ポインタの指す先が外部参照ポインタである) 場合, `%read` の転送が具体値を持つワーカに到着するまで繰り返されるが, 参照値の返信は 1 回のメッセージ送受信で済む.

`%answer_value(Ret,value)` を受信すると, 返信先 `Ret` が指す変数を具体値 `value` で書換え, 変数にフックしていた中断ゴール (群) は実行を再開し, コンシューマ外部参照オブジェクトは解放する (図 10.10 (5)).

以上の処理で, `%read` メッセージ送信側 (読み出し側) の処理は, ジェネレータ外部参照オブジェクトに対して, `generate` メソッドを発行することにより行われる (`suspend_goal()` *runtime/faisus.c*). これは 4.2 で記述した一般的な中断処理の枠組みで行われる. 一方, `%read` メッセージ受信側の処理は, メッセージの転送処理も含め, `decode_read()` (*runtime/cntlmsg.c*) で行われる. リプライ・コンシューマ・オブジェクトは, *runtime/ge_replyhook.c* で定義されている, `class reply_hook` により実現されている. また, `answer_message` の送信もこの関数内で直接行われている.

10.2.5 ループへの対処

ユニフィケーションによってワーカ間に渡るループが生じることがある. また外部参照ポインタに対する参照値の読出し処理の過程でもループが作られる. 以下ではそれぞれのループについて生成, 問題点, 解決方法を述べる.

ユニフィケーションによるループ

生成

変数と KL1 データのユニフィケーションは, KL1 データへのポインタを変数に書き込むことによって行う. このため変数と外部参照ポインタのユニフィケーションによってワーカ間に渡るループが生じることがある. 図 10.11 にその例を示す.

問題点

ジェネレータ外部参照オブジェクトのみからなるループが存在する可能性がある. このため, 受

^{*1} コンシューマ (群) や中断ゴールがフックしている変数も含む.

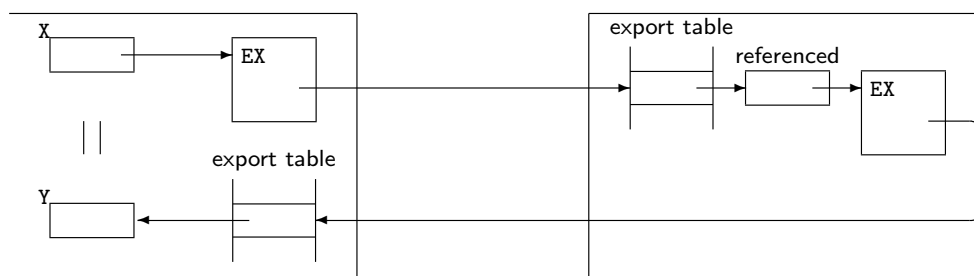


図 10.11 ユニフィケーションによるループの生成.

信した `%read` の参照先がジェネレータ外部参照オブジェクトであった場合, 10.2.4 で述べたように単純に `%read` を転送すると, 転送がいつまでも終わらなくなってしまうおそれがある.

解決方法

`%read` に 転送可能カウンタを持たせることで解決した. `%read` の参照先がジェネレータ外部参照オブジェクトの場合, カウンタ値が 1 以上ならば `-1` して転送する. 0 ならば転送は行わず, リプライ・オブジェクトを生成してフックし `generate` メソッドを呼び出す.

これらの処理により転送の終了が保証できる. ループに `%read` が入り込んだ場合, ループを構成するジェネレータ (群) の少なくともひとつはコンシューマに変わるので, いつかコンシューマのフックする変数に出会い転送は終了するからである. カウンタの初期値を適当な値, 例えば, ノード台数に設定することによって, `%read` を最初に送信したノードに `%answer_value` がいつでも直接返ることが期待できる.

読出し処理によるループ

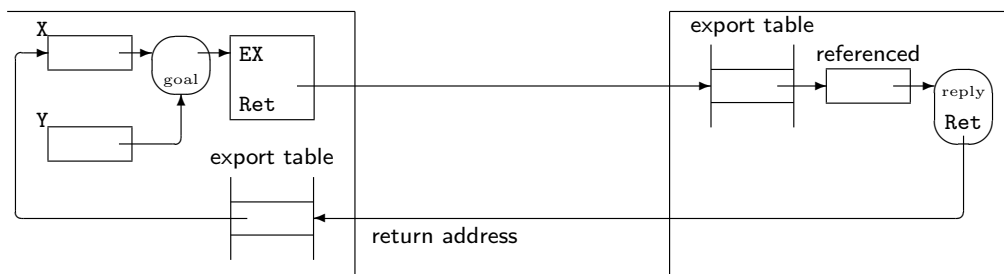
生成

10.2.4 で述べたように, 受信した `%read` の参照先が変数であった場合はリプライ・オブジェクトを生成し変数にフックする. この時, 元の外部参照オブジェクトと, フックしたリプライ・オブジェクトはループを構成する. (図 10.10 (3)).

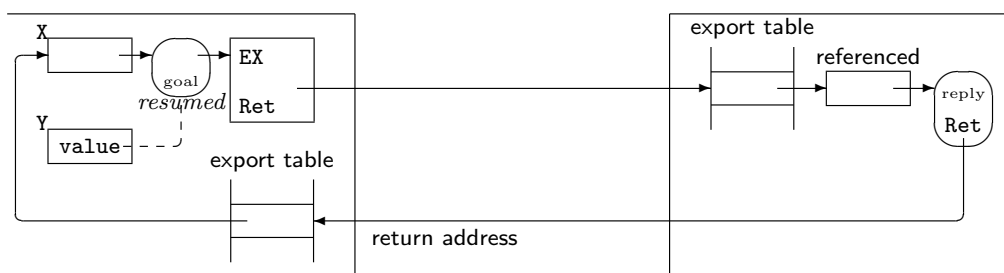
問題点

元の外部参照オブジェクト側に中断ゴールがフックしている間は, リプライ・オブジェクトからのポインタは意味がある. 変数が具体化され, `%answer_value` が送信され, ゴールは実行を再開し, ループの回収されることが期待される (図 10.10 (4), (5)).

しかし中断ゴールが多重待ちの場合は `%answer_value` を受信することなくゴールが実行を再開するかもしれない. そして変数は具体化されないかもしれない. この場合, ループは「(局所処理では) 回収できないゴミ」となってしまう (図 10.12). 「変数が具体化されない」という状況は, 非決定的処理あるいは見込み計算を行うプログラムの実行ではいくらでも起こりうることである. 特



(1) 多重待ちのゴールがフックしている。



(2) `%answer_value` を受信することなくゴールは実行を再開した。

図 10.12 ゴミのループが生まれてしまったかもしれない。

別のことではない。

解決方法

`%read` は送信したが中断ゴールがフックしていないような場合、既に送信した `%read` は取り消してもよい。その `%read` の応答である `%answer_value` を待つゴールはないからである。

局所 GC で、中断ゴールがフックしていないコンシューマ外部参照オブジェクト `X` が見つかったならば以下の処理を行う。

1. コンシューマ外部参照オブジェクト `X` をジェネレータ外部参照オブジェクトに変える (戻す)。
2. `X` が指していたワーカへ以下の「読出し要求取り消しメッセージ」を送る。

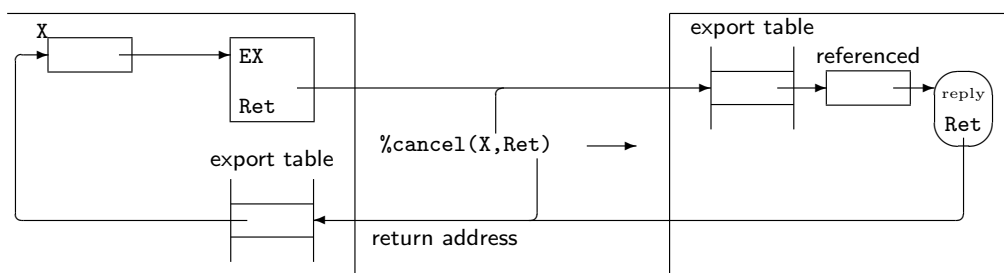
`%cancel(X,Ret)`

メッセージの引数 `Ret` は、取り消す読出し要求を特定するためのものである。

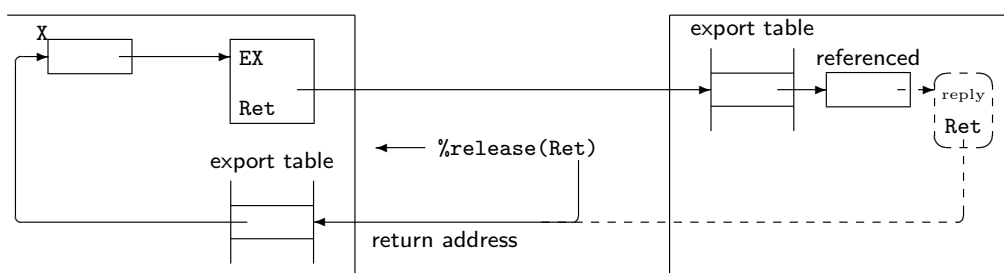
`%cancel(X,Ret)` を受信したワーカは以下の処理を行う。

参照先が変数ならば フックしているコンシューマ (群) を検索し、`Ret` と同じ返信先を記憶しているリプライ・オブジェクトを解放し `%release` を送信する。

参照先が具体値ならば 何も行わない。これは既に `%answer_value` が送信されている場合である。



(1) 中断ゴールがないので %cancel を送信する.



(2) リプライ・オブジェクトを解放し %release を送信した.

図 10.13 読出し処理によって生まれたループの回収.

以上の処理により (図 10.13 参照), ゴミとなったループを, すべてではないが, 局所処理によって解放することができる.

10.3 詳細な実装

以下の点についてより詳細に実装方式を説明する。

- 分散プロセス立ち上げの方式
- メッセージ送受信の方式
- 分散処理用ジェネリックオブジェクトの詳細

10.3.1 分散プロセス立ち上げ方式

具体的なワーカ群立ち上げ方式はマシン依存であるが、ワーカ群を立ち上げた結果、図 10.14 に示されるように 1 つの荘園プロセスと n 個のワーカが通信路によって結合されている状態になることが要求される。

入出力用に他のプロセスが必要な場合は、I/O 用のプロセスが起動され、各ワーカ及び荘園プロセスとの間で通信路が設定される。

また、ワーカ群立ち上げ後の状態では、各ワーカに以下の情報が設定されていなければならない。

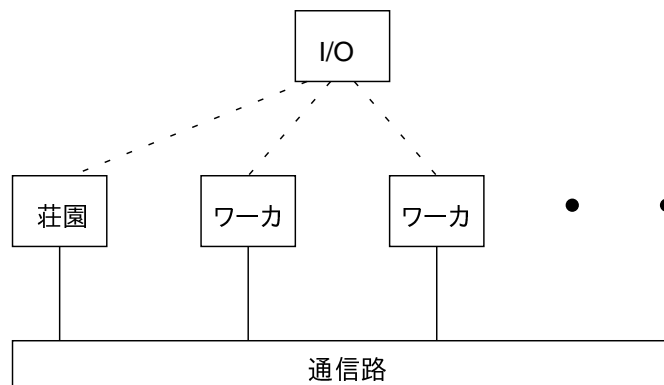


図 10.14 ワーカ群立ち上げ後の状態

- プログラム起動時に指定されたパラメータ
- 自ワーカ番号
- 他のワーカへの通信方法
- I/O 用のプロセスとの通信方法

n 個のワーカを立ち上げた場合、各ワーカには 0 から $n-1$ までのいずれかのワーカ番号が設定される。荘園はワーカの仕事は行わないが、ワーカ番号 n が設定される。ワーカ番号 0 のワーカはマスターワーカとなり、KL1 プログラムの `main:main` 述語はワーカ番号 0 のワーカで実行される。

10.3.2 メッセージの送信/受信

メッセージの送信、受信についての方式を述べる。

メッセージの送信

基本的に、各メッセージの送出は以下のように行われる。

1. ノードより通信バッファをとりだす。
2. その通信バッファに対して、エンコード (`encode_XXXX`) を試みる。XXXX はメッセージの名称 (`read`, `release` など) である。これらの処理は、`runtime/cntlmsg.c` に記述されている。
3. `encode` の結果により、以下の処理を行う。
 - WTC 不足のときにはエンコードが失敗するので、`message_suspend()(runtime/sendrecv.c)` を呼びだし、WTC を要求し、かつメッセージの内容を `susp_msg` 内に保持し、`susp_msg_list` に接続しておく。
 - エンコードに成功した場合には、エンコード結果を `send_message()` (`run-`

time/sendrecv.) により送信する。

メッセージの受信

KLIC では、送信するメッセージは `encode_XXX` により生成されるが、基本的に以下のようになっている (詳細な構造は実装に依存する)。

- 受信時に起動すべき処理関数のアドレス
- WTC
- その他必要なデータ (KL1 のデータなど)

受信時には、このメッセージ内の「処理関数のアドレス」を参照し、その関数に残りのデータを渡すことにより `decode` 処理を行うことになっている。つまり、`decode` の `dispatch` はメッセージ中に記述されている関数を直接呼び出すことにより行われている。^{*2}

なお、各種 `decode` 処理は、各々のメッセージに応じて、`decode_XXX` なる名称で *runtime/cntlmsg.c* 内で定義されている。

この受信に関する処理は、*runtime/sendrecv.c* 内で `recv_message()` により定義されている。

なお、受信処理のきっかけは、以下の 2 つを併用している。

送信時に送信元から送られる USR1 シグナル: 送信者は、送信後に USR1 シグナルを送信先に送ることになっている。よって、基本的には受信者は USR1 シグナルを受信したらメッセージが到着していると判断し、メッセージ受信を試みる。

interval タイマによる一定間隔のポーリング: PVM を用いた処理系では、現実的には、シグナルがメッセージを追い越し、USR1 シグナルを受信したにも関わらず、メッセージが未到着な場合がある。このようなケースを補完するため、一定時間毎にメッセージ受信を試みている。

割り込み、タイマとも、第 4.1.1 章 (53 ページ)、第 4.3 章 (58 ページ) で記述された方法を用い、実際の処理は、リダクションの切れ目で行われるように設定されている。

`receive_message()` の下位では、実装依存の `receive_packe()` 関数を呼びだしており、これが実装間のインターフェースになっている。

10.3.3 分散処理用ジェネリック・オブジェクトの詳細

ここではジェネレータ外部参照オブジェクトとコンシューマ外部参照オブジェクト及びブライ・コンシューマ・オブジェクトの詳細を述べる。おのおの、`exref class`、`read_hook class`、`reply_hook class` のジェネリックオブジェクトとして実装されている。

^{*2} よって、KLIC では、関数のアドレスがずれるような状況、例えば、ノード間で `loadmodule` のリンクの順などが異なるため関数アドレスがずれている場合、また、ノード間で命令コードアーキテクチャが異なる場合などは、このままでは対応できない)。

ジェネレータ外部参照オブジェクトの詳細

このオブジェクトの保持するデータとメソッドの処理内容を以下に示す。exref class として、*runtime/ge_exref.c* で定義されている。

1. 保持するデータ

(輸出元) ノード番号、輸出表エントリの ID、重み (WEC)。

より詳細には、*runtime/include/klic/ge_exref.h* に以下のように定義されている。

```
struct exref_object{
    struct generator_object_method_table *method_table;
    long node;          /* ノード番号 */
    long index;         /* 輸出表エントリ */
    long wec;           /* 重み */
    q    to_exref;      /* generator 二重ループの入口 */
    long gc_flag;       /* GC メソッドがよびだされたかどうかのフラグ */
};
```

2. デレファレンスメソッド (引数: なし)

返信先を用意して **%read** を送信しコンシューマ外部参照オブジェクトを作成する。コンシューマ外部参照オブジェクトの作成時に与える引数群のうち、ノード番号、輸出表エントリの ID 及び WEC は、自分が保持しているデータをそのまま渡す。作成したオブジェクトへのポインタをデレファレンスメソッドの返回值とする。

3. active unify メソッド (引数: 『対象データ』)

(メソッドの引数の) 『対象データ』に従って以下のいずれかの処理を行う。

『対象データ』が**具体値** 自分に具体値を書込み、以下のメッセージを自分が指していたノードへ送る。

%unify(Tn= 初期値, 自分の参照先, 具体値)

『対象データ』が**コンシューマ・オブジェクト** 自分のデレファレンスメソッドを呼出し、その返回值と『対象データ』の具体化を行う。

『対象データ』が**ジェネレータ外部参照オブジェクト** 『対象データ』を指すポインタを自分に書込み、以下のメッセージを自分が指していたノードへ送る。

%unify(Tn= 初期値, 自分の参照先, 『対象データ』)

『対象データ』が**(ジェネレータ外部参照オブジェクト以外の) ジェネレータ・オブジェクト** 『対象データ』のデレファレンスメソッドを呼出し、その返回值と自分の具体化を行う。

4. %read 受信メソッド (引数: %read の引数群)

%read の転送上限引数 **Tn** の値に従い以下のいずれかの処理を行う。

Tn>1 自分が指すノードへ以下のメッセージを送信する。

`%read(Tn-1`, 自分の参照先,(前のままの) 返信先)
Tn=1 リプライ・コンシューマ・オブジェクトを作成し、自分のデレファレンスメソッドを呼出す。そしてデレファレンスメソッド呼出しの返回值と、作成したリプライ・コンシューマ・オブジェクトとのボディユニフィケーションを行う。

コンシューマ外部参照オブジェクトの詳細

このオブジェクトの保持するデータとメソッドの処理内容を以下に示す。read_hook class として、ge_readhook.c で定義されている。

1. 保持するデータ

(輸出元) ノード番号、輸出表エントリの ID、重み (WEC)、送信した `%read` の返信先より詳細には以下である。

```
struct consumer_object_method_table *method_table;
    long node;           /* ノード番号 */
    long index;          /* 輸出表エントリ ID 1*/
    long wec;            /* 重み */
    q to_read_hook;      /* 二重ループエントリ */
    long return_index; /* 返信先 */
};
```

2. active unify メソッド (引数:『対象データ』)

以下のメッセージを自分が指していたノードへ送る。

`%unify(Tn= 初期値, 自分の参照先, 『対象データ』)`

3. %answer_value 受信メソッド (引数:『対象データ』、『返信先』)

自分が持つ返信先と (メソッド引数の)『返信先』が一致するならば以下のメッセージを、

`%release(自分の参照先)`

返信先が一致しない場合は以下のメッセージを自分が指すノードへ送信する。

`%unify(Tn= 初期値, 自分の参照先, 『対象データ』)`

10.3.4 リプライ・コンシューマ・オブジェクトの詳細

reply_hook クラスとして、runtime/ge_replyhook.c で定義されている。

1. オブジェクトの保持するデータ

返信先、重み (WEC)

より詳細には、以下。

```

struct reply_hook_object{
    struct consumer_object_method_table *method_table;
    long node;      /* 返信先のノード */
    long index;     /* 返信先の輸出表インデックス */
    long wec;       /* 返却すべき WEC */
};

```

2. active unify メソッド (引数:『対象データ』)

以下のメッセージを (自分の保持している) 返信先が指すノードへ送信する。

```
%answer_value(返信先,『対象データ』)
```


第 11 章

共有メモリ並列実装

11.1 概要

共有メモリ結合並列計算機上で KLIC の並列実装を設計・開発した。この拡張での設計目標は、逐次核をそのままにして、共有メモリ並列処理を付加することであった。

従来の共有メモリ向け実装は、排他制御コードの挿入が、ベースとなる逐次性能を損なっていたが、提案した混合方式では、局所データ領域と共有データ領域を区別することで、そのようなオーバヘッドが小さくなることを狙った。

また、高速な計算機は局所メモリと共有メモリとのアクセスコストの落差が大きく、並列に一括 GC を行くと共有バスがボトルネックになる。それを避けるために各ノードが独立に共有メモリを GC する非同期 GC を採用した。

11.2 処理系の構成

処理系は、共有メモリ変数の扱いをジェネリックオブジェクトで実現し、データ領域を局所領域と共有領域とに分割された構成となっている。

11.2.1 共有変数オブジェクト

KLIC 逐次処理系は、ユーザが独自に追加したい機能を拡張できる枠組み (ジェネリック・オブジェクト・インタフェース) を持っており、共有メモリ拡張でもジェネリック・オブジェクトの枠組みにより拡張した。

逐次実装と共有メモリ実装とのもっとも大きな違いは、共有メモリ変数の存在である。共有メモリ変数はその書換えの時に特別な扱いを要する。具体化する場合にはロックが必要であるし、フックする時にもロックが必要である。共有メモリ上の共有メモリ変数以外のデータ (具体値) へのロックは不要なので、共有メモリ変数はこれらと異なった特別なデータタイプにする。

共有メモリ変数は SHVAR というジェネレータオブジェクトで実現される (このオブジェクトは、`runtime/gg-shvar.c` なるファイルで定義されている)。共有メモリ上のそれ以外の基本データ

タイプは逐次処理の対象データと同じであり、それらのデータに対しては、ロックのオーバーヘッドは加わらず、局所メモリのデータにアクセスする場合と何等変わらない。

各ノードには固有の局所メモリを持ち、ノード間で共有する1つのメモリ領域を持つ。局所メモリはその持ち主のみがアクセスする。共有メモリ変数を局所データで具体化するときには、共有メモリ変数に局所データへポインタ付けするのではなく、局所データを共有メモリにコピーした上で、変数にポインタ付けする。

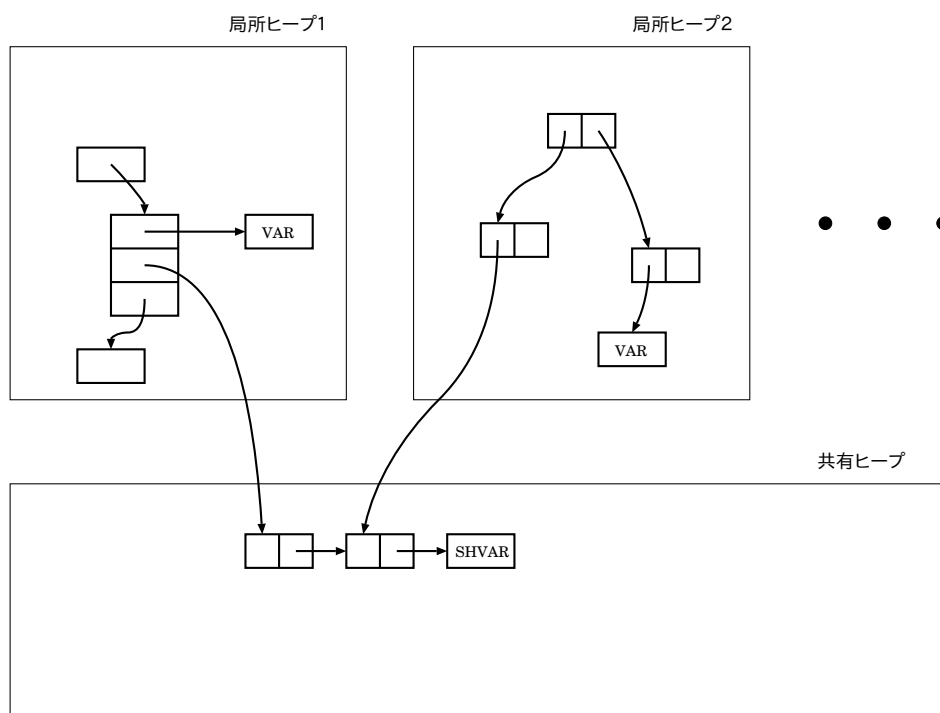


図 11.1 局所ヒープと共有ヒープ

11.2.2 局所領域と共有領域

局所領域には、局所処理を行うための管理領域と動的に局所ゴールや KL1 データを配置する局所ヒープとがある。管理データには局所ヒープの使用アドレスや実行ゴールへのポインタなどの情報を持っている。共有領域には共有領域を管理するデータ領域とゴールや KL1 データを配置する共有ヒープ領域がある。共有管理データには、外部ゴールへのポインタや各ノードの割り込みフラグなどがある。付加分散するときには、他のノードからアクセスできるようにゴールを共有メモリに配置する。

各局所ヒープは GC のために 2 つの空間に分割されており、その GC はすでに述べた逐次核の GC にほぼ等しい。

共有ヒープは 3 つの空間に分割されている。新・旧・未使用とに分け、これらはサイクリックに

切り替わる。

一般には、共有ヒープから局所ヒープへポインタ付けすることはない。しかし、いくつかの場合にそのようなポインタが必要となる。そのような場合には間接テーブルにより、ノード番号とその間接テーブルのインデックスと局所ヒープのあるアドレスを指定する。局所 GC 時には、そのテーブルも GC ルートとして扱い、また適正に更新する。

各ノードは動的にデータを配置するために共有ヒープ面を持ち、各面は現在アクセス中のノード表を持っている。同期のためのオーバーヘッドを小さくするために各ノードはゴールやデータを配置するページを確保する。そのページを使いきった時に、各ノードは現在のヒープ面から新たにページを確保する。

ノードは GC 時にアクセス面を旧面から新面に移動する。最後に旧面を参照していたノードが新面にアクセス面を移動した時に旧面を参照するポインタがなくなったことが保証され、その旧面は次のサイクルで未使用面に切り替わる。

11.2.3 ゴールの投げだし

KLIC の共有メモリ並列拡張では、ゴールの分配をプラグマとして記述し、それにより指定されたノードにゴールの投げる方式をとっている。

下記のようなコードを例とし、実際にどのようにゴールとして投げだされるかを説明する。

```
p(X) :- ... | ... , q(X,foo(Y))@node(4), ... .
```

最初にゴール *q* を局所ヒープに生成する。実行時ルーチンによりそれは共有ヒープにコピーされる。これは、逐次版にも存在する `enqueue_throw_goal()` 関数の中で、行われ、共有ヒープにコピーされる以降、これから述べる処理は共有メモリ版特有の `throw_goal_routine()` 関数 (*runtime/shm_throw.c*) により実装されている。

コピーは、(1) CONS や FUNCTOR 構造はその構成要素を再帰的にコピーしていき、(2) 変数は共有メモリ上に (ジェネリックオブジェクトである) 共有メモリ変数を生成し、局所メモリの変数はそれへのポインタに置き換える。(3) 共有メモリへのポインタであった場合にはそれ以上のコピーは必要ない。

ある場合にはもう少し処理しなければならない。(A) フックされた変数を共有メモリに移す場合には、中断情報も共有メモリに移さなければならない。コピーされた中断情報は間接テーブルを通じて局所ヒープ内のゴールへポインタ付けされることになる。(B) 現在のアクセス面より古い面のデータであった場合には現在の面にコピーしなければならない。また (C) ゴールを投げようとするノードが新しい面を参照していた場合には、新しい面にコピーして渡さなければならない。このようにしないと、新面に移ったノードが旧面を参照してしまうことになる。

ゴールをコピーし終わったら、投げようとするノードの外部受け入れキューに登録する。投げようとするノードの実行中の優先度を検査して、投げるゴールの優先度の方が高ければ、割り込みフラッグを立てて、直ちに処理されることを要求する。それ以外の場合には投げだし先のノードには

割り込まない。各ノードは低い優先度処理に移るときに外部受け入れキューを調べ、それらをスケジューリングして処理を再開する (この外部ノードへの外部受入キューに登録以降の処理は、`throw_goal_routine()` 内の、`shm_goal_stack()` で行われる)。

11.2.4 共有変数の具体化

共有メモリ変数は、ジェネレータオブジェクト SHVAR により実現している (*runtime/gg-shvar.c*)。SHVAR にはメソッド表フィールドと中断チェーンがあり、メソッド表フィールドはロック用フィールドとしても利用している。

```
typedef struct Shvar {  
    struct generator_object_method_table *method_table;  
    Sinfo chain;  
} Shm_var;
```

各ノードはロックして変数の状態を変える。通常ノードはフックした後、ロックをはずさなければならない。しかし具体化した後にはアンロックは必要ない。

これは、ロック時に、単にロック用フィールドをスピンするだけでなく、変数の値の設定の有無を監視しながら、ロック用フィールドを監視する。ゆえに、ロックが解除されればロックは成功するし、具体化されても、スピンから抜け、該当する処理に移れる。これらの処理は、shvar オブジェクトの UNIFY メソッドにて行われている。

構造体と共有変数とが具体化される時には以下で述べるような最適化も行っており、これも UNIFY メソッド中で実装されている。

11.2.5 共有ヒープへのコピーの遅延

他のプロセスに参照されるストリームを生成する場合、最初に生成側と参照側との間に変数 (S) を共有する。生成側はストリーム要素 X1 を生成し、最初に [X1|L1] を局所ヒープに生成し、そして変数 S (ジェネリックオブジェクト SHVAR) に単一化する。

SHVAR のユニファイ・メソッドは局所ヒープ上のリストを共有メモリにコピーする。そして変数 (S) にリスト・ポインタを書き込む。このような操作が 1 要素を具体化する毎に繰り返される。

連続した共有メモリ変数の生成およびその具体化は大きなオーバーヘッドを伴う。特にスループット優先のプログラムの場合には、この頻度が大きい。

この問題を回避するために、具体化遅延 (ジェネレータフック) という技法を用いる。ジェネレータフックはフックしていない共有メモリ変数を局所ヒープのデータ構造で具体化するとき、その時点での具体化は行わずに共有メモリ変数にジェネレータフック情報を記録する。あるゴールがその変数の読み出し要求をしたときに、そのオブジェクトはその変数を具体化しているノードに対して具体化要求をだすこの方式は要求時点で全てもしくはほとんどのデータ構造が局所ヒープ上で具

体化されていることを期待している。このような場合には生成側は共有メモリ変数を一度だけジェネレータフックにし、データを一気に共有メモリへコピーすることになる。変数具体化時にフックしているゴールがあった場合には、ジェネレータフックにしないで、具体化して待ち合わせているゴールを起こす。要約すると、具体値を待つものがいた場合は、局所データはコピーされ、そうでない場合は、遅延してコピーする。前者はデータ駆動またはスループット優先プログラムであり、後者は要求駆動またはレスポンス指向のプログラムである。

11.3 共有ヒープのガーベジコレクション

11.3.1 局所 GC

局所ヒープと共有ヒープの分割は独立な局所 GC を可能にしている。局所 GC は、KLIC の逐次核実装の GC に共有メモリポインタをコンスタントデータのように扱い (すなわち、そこから先はコピーしない)、また、共有ヒープから局所ヒープを参照しているデータについての間接テーブルを GC ルートに加えることで、独立動作でき、GC ルーティンの変更もごくわずかなもので済んでいる。

11.3.2 共有ヒープ GC

多くの処理系の共有メモリ実装では、共有ヒープの GC は同期して行っていた。全てのノードが通常処理を中断して、GC を始める。しかし、単一プロセッサの性能は共有バスのスループットよりも飛躍的に高くなってきている。GC は局所性が低く、並列処理では共有バスがボトルネックになってくる。

この問題を避けるために、KLIC の共有メモリ実装では、各ノードが独立に GC する非同期 GC を採用した。同時に複数のノードが GC する代りに、ノードが通常処理している中、あるノードが非同期に GC を始めるのである。

この方式は共有バスアクセスが時間軸に一定になることを期待している。その結果バスを待ち合わせするアイドル時間を軽減させることにある。

非同期 GC では、古い面を参照して通常処理していたノードが古い面の領域の不足を検知して GC 処理に切り替わる。ノードは参照している古い面のデータを新しい面にコピーする。それが終わると、また新しい面を参照面として通常処理に移る。もし、2つの面しかなければ、新面があふれた場合にコピーする面がなくなってしまう。それゆえ、もう一つの未使用面を用意している。未使用面の確保は実データ使用量が一面のサイズを越えない限り、共有ヒープ領域のオーバーフローを起こさないことを保証している。

図 11.2 に非同期 GC の状態を示した。この例では、最初に2つのノードが旧面を参照している。そして、1つのノードが旧面の領域の不足を検知して、参照データを新面にコピーする。その間、もう一方のノードは通常処理を行っている。一方のノードが GC を終え、その後にもう一方のノードが旧面の領域の不足を検知して、参照データを新面にコピーする。

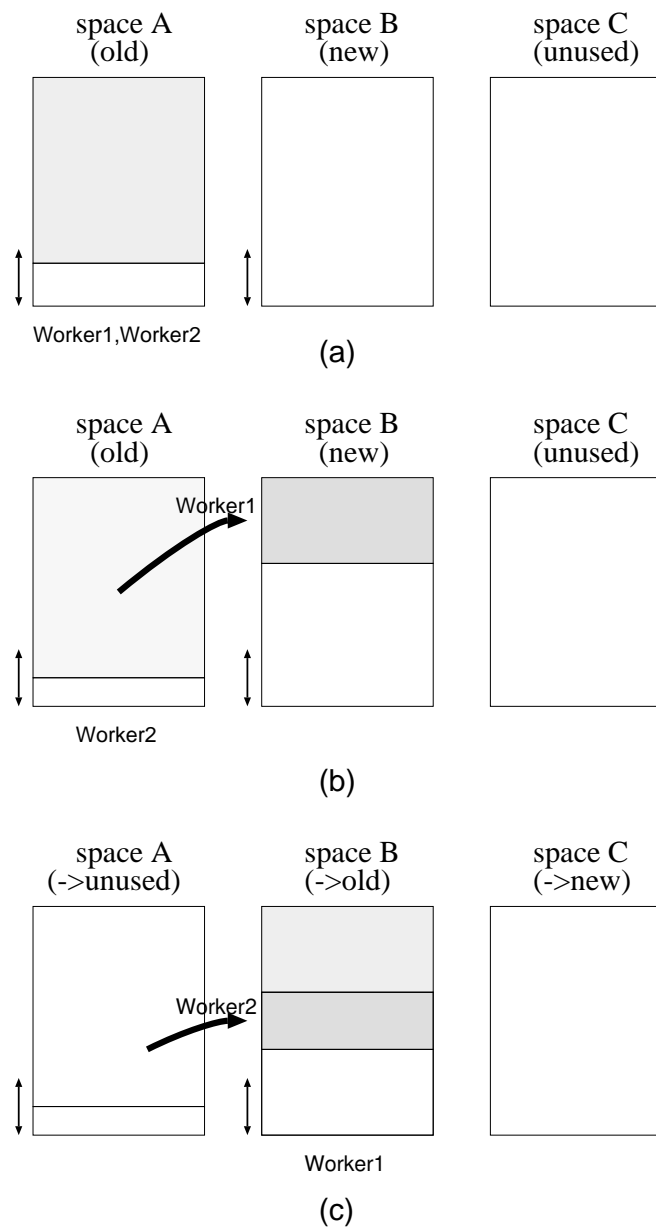


図 11.2 共有ヒープの非同期 GC

新面を使いきった場合に、未だ旧面を参照しているノードがいた場合には、それらのノードに対して、割り込みにより強制 GC を通知する。それらのノードが新面に参照データをコピーし終われば、旧面は未使用面に切り替わる。このように面は回転して使用され、新面は次の旧面、未使用面は新面へと切り替わる。3 面を使用する非同期 GC では、以下のような問題を解決しなければならない。

GC ルートの決定: 共有ヒープを参照しているルートは何か?

並列の読み書き: 通常処理と GC の同時処理方法、同じ場所をアクセスする可能性

旧面の解放の保証: 旧面の解放は新面からポインタ付けされないかまたは、そのようなポインタを識別できることであるが、どのようにするか?

これらの問題の解決方法を以下に述べる。

ルートの決定

参照データのルートの決定は on-the-fly GC の主要な問題であるが、この非同期 GC はそれほど重要でない。ノードが通常処理から GC 処理に移る際、実使用データをそのまま引き継いでいる。これをするには局所 GC の際に局所ヒープから共有ヒープを指している箇所を記録すればよい。on-the-fly GC のような通常処理と GC 処理するノード間の調整は必要ない。

並列読み出しと書き込み

データをコピーする際に通常処理ノードが同じデータをアクセスしているかもしれないということに配慮しなければならない。幅優先のコピーの場合には、旧面にある構造体データを新面にコピーする時旧面データに前方参照ポインタを書き込む。前方参照ポインタは通常ポインタと見分けがつかないので通常処理ノードにとっては異常データを参照してしまうことになる。

この問題を解決する方式はいくつか考えられるが、KLIC では以下に述べるボトムアップコピー方式と暗黙前方参照方式を実装した。(それらは条件コンパイルにより切り替えるようにしている)

旧面読み込み専用 GC ノードは旧面を読み込み専用とする。通常処理ノードは悪影響を及ぼさない。このことは、GC ノードは前方参照ポインタを一切作らないことを意味する。

しかしこの方式で共有メモリ変数は例外にしなければならない。GC ノードは、旧面の変数を新面にコピーすることで1つの変数を2つの変数に分割できない。一つの変数を2つの変数に分割することは論理変数の意味を壊してしまう。

そこで、共有メモリ領域の新しい変数を生成し、それと旧面の変数を単一化を許し、旧面の変数の値は新しい変数へのポインタにしてしまう(このポインタは実質的に前方参照ポインタとなる)。

前方参照ポインタが使用できないために、GC ノードは古い面のコピーデータが新しい面のどこにあるのかわからない。(古い面のデータが新しい面にコピーされたかどうかさえわからない) ゆえに、旧面データ生きているパスの数だけのコピーが行われる。その数は非常に大きな数になる恐れがある。別の前方参照テーブルを設ける方法も考えられるが、余分な空間を必要とすることや実行時のオーバーヘッドを考えると非現実的である。

ボトムアップコピー方式 旧面のデータ構造を新面へボトムアップにコピーする。旧面を指している構造体への旧面にあるポインタは新面にコピーした構造体へのポインタに置き換えられる。リスト [a,b](図 11.3) のコピーは、図 11.4(a) から図 11.4(b) へと進む。新面にコピーされた構造体は旧面にあるものと論理的に等価である。このような置き換えはいつでも可能である(ポインタの書き込みが不可分な処理であるとすれば、通常の計算機はこのことを

保証している)。たとえば、この場合通常処理ノードからみて常に [a,b] と見える。

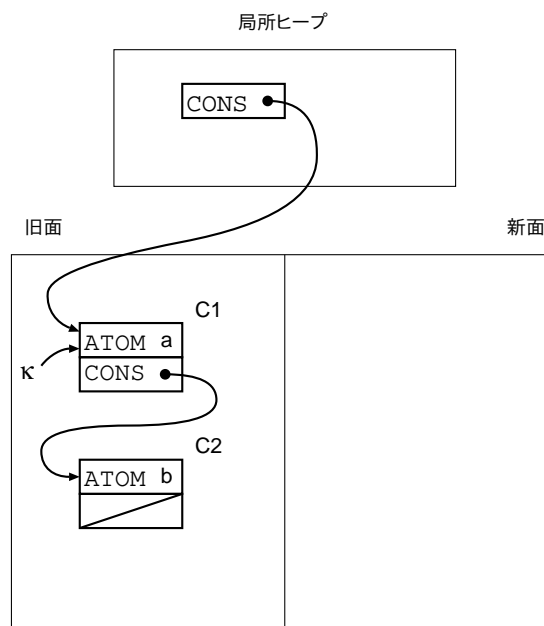


図 11.3 元の状態 (コピー前)

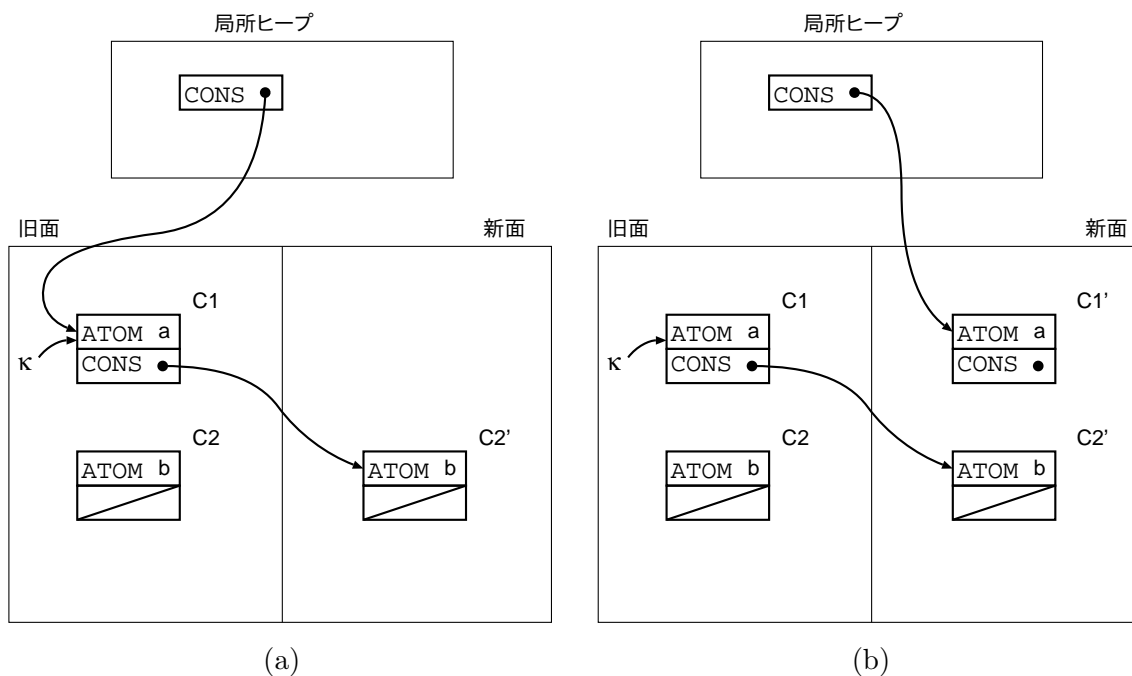


図 11.4 ボトムアップ コピー

この方式でも、重複コピーの問題をはらんでいる。特にループしているデータに出会うと無限ループに陥る。しかし、重複コピーする度合はそれ程多くない。旧面読み出し専用の場合

と違って、参照中のポインタの数だけの重複コピーしか起こり得ない。(生きているパスの数と比べれば、はるかに小さい数である。)

幸い多くの KL1 プログラムはループ構造を作っていない。また複数のポインタから参照されるデータ構造の割合も大きくない。

暗黙前方参照方式 暗黙前方参照ポインタは前方参照情報を間接ポインタの出現で代替させる。データ中の 1 語、言うならば先頭に暗黙の前方参照ポインタを配置する。このポインタは新面のコピーしたレコードを指す。たとえば、リスト [a,b] をコピーする場合、最初の CONS レコード (C1) を新面にコピーする。局所メモリから指されるポインタは直ちに新面にコピーされた CONS レコード (C1') を指す。そして、C1 レコードの最初のフィールド (CAR 部) を C1' へのポインタに書き換える。(図 11.5(a)) この参照ポインタは前方参照ポインタとして扱う。そして同じように CONS の 2 番目のフィールド (C2) をコピーし、前方参照ポインタを設定する。(図 11.5 (b)) 参照点 k から参照されているデータ構造は旧面から新面への参照に変わる。これらのポインタの付け直しの前後および途中でも論理的にリスト [a,b] と一定している。

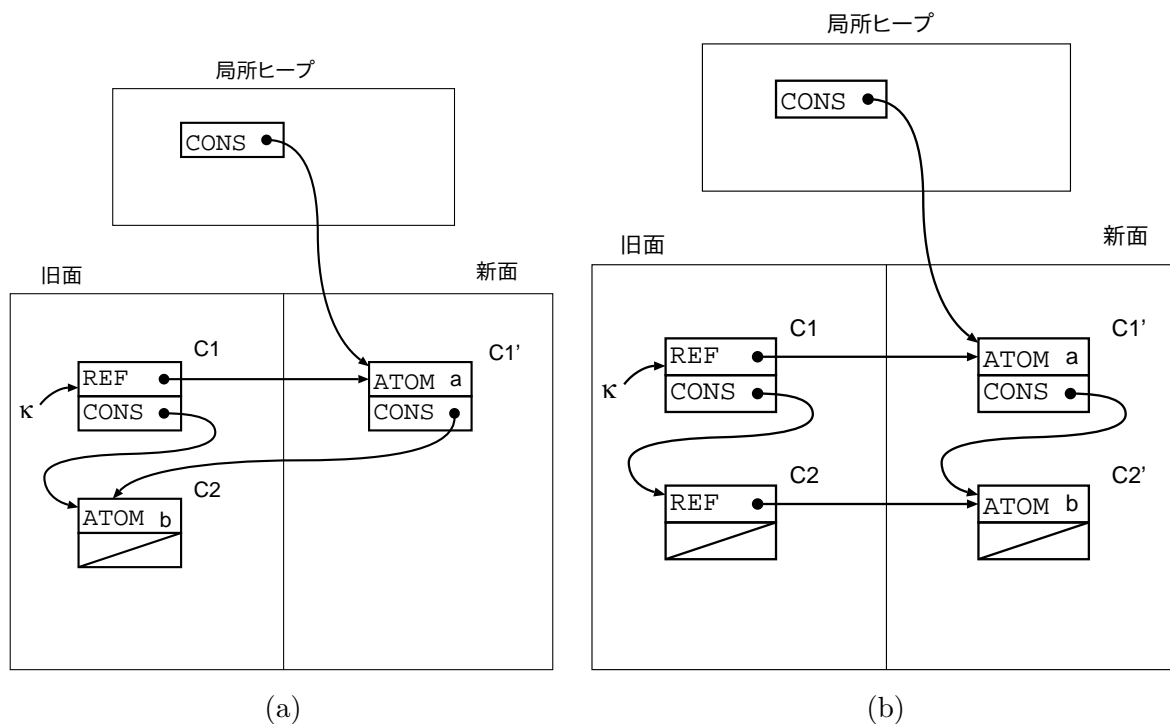


図 11.5 トップダウン コピー

この方式の問題は、(前方参照タグ方式と比較して) 間接ポインタと前方参照ポインタ見分けが付かないことである。これら 2 者を識別しなければならない。我々の実装では共有ヒープ上にポインタを作る場合は以下のような場合に限られている。

1. 構造体要素から共有変数を指すポインタ
2. 共有変数同士の単一化でできるポインタ

(2) では GC ノードによって作られたもの (共有変数を旧面から新面にコピーした時のポインタ) か通常処理ノードによって作られたものかは識別できない。しかしこれらは区別する必要がない。

(1) の場合、旧面から新面の共有変数へのポインタなら前方参照ポインタと区別ができなくなる。そのようなポインタは通常処理ノードが局所メモリ上のネストした構造体をコピーするときに生まれる。ゆえに実行時のコピールーチンでそのようなポインタができないように間接ポインタを差し入れるように修正した (この措置により通常処理に若干のオーバーヘッドが加わっている)。

この方式で更なる問題がもう一つある。新面のレコードから旧面を指すポインタの存在である。新面に移った通常処理ノードは、直接 GC 中のそのようなポインタを参照してしまうことはないのだが、旧面を参照中の別の通常処理ノードからそのようなポインタを受け取ってしまうことがある。ゆえに、この方式では面を再利用する時、再利用する面を参照しているポインタを新面へのポインタにするための局所 GC が必要である。

コンパイル時のオプション

GC の処理は、*runtime/shm_gc.c* にて行われているが、コンパイル時のオプションにより行われる。このオプションについての説明を行う。

これらのオプションは、幾つかが関連があり、各々を独立に set/unset してはならない。

TOPDOWN/BOTTOMUP: どちらか 1 つを ON にしても良い。TOPDOWN を ON にした時のみ、IAFWD/ISFWD に意味がある。

ASYNC_GC/SYNC_GC: 通常実行に対して非同期に GC を行うか、行わないかを指定する。どちらか一方のみを指定する。SYNC_GC とは、あるノードで GC を行うときには他の全てのノードはも通常実行をやめ、GC を行うか、なにもしないか、どちらかとなる。

PAR_GC/SEQ_GC: 同時に複数のノードで GC を行うことを禁止するかどうか。SEQ_GC では単一のノードでのみ GC を行う。

IAFWD/ISFWD: 暗黙前方参照方式の、さらに詳細な処理分化^{*1}

^{*1} 詳細は現在不明。

索引

active unification, 61, 62

CONS, 8

encoding, 35

execute, 29

functor, 8

GC, 77

アルゴリズム, 79

きっかけ, 79

共有ヒープ, 124

対象領域, 78

判定, 78

GC stack, 77

GC スタック, 77

Generator の起動, 67

generic object, 12

goal, 21

goal record, 21

goal stack, 21

heap, 18

module, 28

naive reverse, 37

passive unification, 61

PVM-TCP 版, 98

PVM 版, 98

ref, 9

tag, 6

tracer, 84

WEC, 105

word, 7

WRC, 105

WTC, 100

エンコーディング規則, 35

外部参照ポインタ, 103

解放, 105

生成, 105

共有変数オブジェクト, 120

共有メモリ版, 98, 120

共有領域, 121

局所領域, 121

具体化, 9, 61

ゴール, 21

エンキュー, 29

実行可能, 22

多重待ち, 24

中断, 22

直接呼び出し, 29

通常呼び出し, 29

デキュー, 29

ゴールスタック, 21, 27

ゴールレコード, 21

再開処理, 64, 65

参照, 9

ジェネリック・オブジェクト, 12, 70

consumer, 13, 71, 74

data, 13, 71

generator, 13, 71, 75

Generator の起動, 67

構造, 14

consumer, 14

data, 14

generator, 15

メソッド表, 12

失敗, 53, 56

述語, 21

出力形式, 30

述語呼び出し, 29

純粋未定義変数, 9

荘園, 100

終了, 100

中断, 100

大域データ構造体, 16

タグ, 6

単一化, 61

active unification, 61, 62

passive unification, 61

単一化ゴール, 68

中断, 53, 56

中断構造, 64

中断ゴール, 10

定数構造体, 28, 36

動作モデル, 26
トップレベルループ, 27
トレーサ, 84
 CALL ポート, 85, 89
 FAIL ポート, 85
 REDU ポート, 85, 90
 SUSP ポート, 85, 90
 擬似ゴール, 88
 機能一覧, 84
 名前管理, 86

ヒープ, 18, 34
ヒープ割付点, 20

ファンクタ, 8
分散プロセス立ち上げ方式, 114
分散ユニフィケーション, 106

未定義変数, 9
 分類, 14

メッセージ通信版, 98
メッセージの送信/受信, 115

モジュール, 20, 28

輸出オブジェクト, 104, 117, 118
輸出表, 103, 104, 117

例外処理, 32, 53

論理変数, 9

ワード, 7
割り込み, 53