逐次型パーソナル推論マシンPの
設計思想とそのアーキテクチャ

西川　宏　横田　実　山本　明
瀧　和夫　内田俊一

逐次型パーソナル推論マシンPの設計思想とそのアーキテクチャ

渕一博，横田実，山本明，瀧和男，内田俊一（ICOT）

1．はじめに

第5世代コンピュータシステム研究開発プロジェクトでは人間のように考えることのできる、推論機構を有する計算機システムの実現を目指して研究開発が進められている。

そのためには論理型プログラミングが研究開発の環境を早期に確立することが必要であり、本プロジェクトの一環としてICOTが中心になりパーソナル逐次型推論マシン（Personal Sequential Inference Machine：PSI）の研究開発を行なっている。

PSIの開発目的は本プロジェクトの研究開発を進めるのに必要なソフトウェア開発用ツールとなること、また論理型言語向きの新しいマシンアーキテクチャ研究開発のテストベッドとなることであり、実行速度20～30 KLIPS*、最大実装記憶容量16M語の高性能パーソナル推論マシンとして開発される。更に新しい試みとしてPSIの上に実現されるオペレーティングシステムを論理型言語によって記述することを目標にして、PSIの機械語である従言語KL0をProlog-likeな言語仕様とした。したがってPSIは述語論理型言語を機械語として直接実行できるマシンであると共に、オペレーティングシステムのためのハードウェアサポート機能を備えていることが最大の特徴である。

本論文ではPSIの研究開発に要求されている幾々の目標を述べると共に、PSIのシステム概要、アーキテクチャを決定するまでの設計方針、並びにアーキテクチャの特徴について述べる。

2．背景と開発目標

本プロジェクトの基盤となる論理型言語の例としては、一階述語論理に基づくプログラミング言語PROLOGが最も良く知られており、中でもDEC10 PROLOGコンパイラ版[1]は最も実行効率が良いとされている。

しかしながら既存汎用マシン上で実現されている処理系は従来型言語を実行する場合に比べて大きなメモリ空間を必要とするうえ、実行効率もあまり高くない。従って論理型言語の持つ高い記述性、非決定性といった特徴を活かすためには専用マシンアーキテクチャによるハードウェアサポートが不可欠である。

更にコンピュータシステムを一つの概念で統一できればシステムを単純化でき、理解の容易さ、開発の容易さにつながるため従来のオペレーティングシステム等をも論理型言語で記述できるであろう。

ところが、論理型言語を用いる場合の最大の利点の一つである非決定性は、オペレーティングシステムのような決定的処理が主体のシステム記述に用いるのは得策ではないとされている。このために、従しべんのシステム記述を論理型言語の枠組の中で行なえるような機能拡張が必要である。

これらの背景から本プロジェクトでは逐次型推論マシンの研究開発を進めているが、その開発目標はオーに新しい論理型プログラミング言語の研究、試作／評価のため、及び論理型言語による種々の応用システム（例えば自然言語処理、専門家システム等）の研究、開発のためのソフトウェア開発用ツールとなることである。オニには論理型言語の実行に最適な新しいマシンアーキテクチャを開発するための基礎となることである。

開発用ツールとしての目標を達成するためには以下のような要求事項を満たすこと

---

*) Logical Inference Per Second

が必要であろう。

(1) 大規模な応用プログラムを実行可能とするだけの充分な実行速度，メモリ空間を備えていること。

(2) ツールとしての高い操作性，作業性を実現するための良好なマンマシンインターフェースを備えていること。

(3) ソフトウェア開発作業の分散化，あるいは既存計算機システムとの間での機能分担，ソフトウェアの流通を図るためのネットワークとの結合機能を備えていること。

(1)の性能については最も普及している DEC-10 PROLOG を基準とするのが妥当であろう。これまでの PROLOG プログラム開発経験より実行速度，メモリ空間ともに DEC-10 PROLOG の1桁以上の性能が必要と思われる。特にメモリ空間の拡張は不可欠である。何故ならば実行速度が遅い場合にはその分だけ長時間計算機を動かすことによりとにかく解を得ることが可能であるのに対し，メモリ空間が狭い場合にはそれを使い切ってしまうとそれ以上実行不可能となることである。この意味で仮想記憶システムの導入が望ましい。

(2)の良好なマンマシンインターフェースを実現するためにはビットマップディスプレイ装置，ポインティングデバイス(マウス etc.)を用いた多重ウィンドウシステムを備えることや，日本語入/出力装置をサポートすることが必要である。またシステムの利用形態としてはパーソナルマシンとして個人で専有して利用できる方が高い操作性が期待できる。そのためにはいろいろのネットワークインターフェースをサポートすることにより共有データベースへのアクセス等のシステム拡張が不可欠となる。

オペレーティングシステムの論理型言語による記述をはじめとして，本プロジェクトで開発予定の各種ソフトウェアは実際に逐次型推論マシンを用いて開発する必要があるため，それと極力早期に完成させることが要求されている。

論理型言語向きマシンアーキテクチャの研究開発のオーダとしては次のような事項が要求されるであろう。

(1) 各種評価データの収集機能を備えること。

(2) 言語仕様の変更に対処できるだけの，また新しい解釈/実行方式の実験，評価ができるだけの柔軟性を備えること。

(3) 次の研究ステップにつながる新しい試みを積極的に導入すること。

(1)の評価項目としては論理型プログラムのプログラム特性，実行特性の抽出から，ハードウェアレベルでの設計パラメータの有効性の評価まで含まれる。後者ではキャッシュメモリのヒット率をはじめとするメモリアクセス特性が重要な課題となる。

(2)の柔軟性についてはマイクロプログラム方式や機械語の設計方法により達成することができる。論理型言語として最も普及している PROLOG に関しても今なお種々の機能拡張が研究・提案されている。また並列実行へ向かっての並列 PROLOG (Concurrent Prolog) が最近話題されており言語仕様のレベルでの試行錯誤が今後とも必要である。従って機械語命令設計時には充分な柔軟性を考慮すること，おおいにマイクロプログラム制御の適用等が要求される。

(3)の実験的試みについては前述の開発ツールとしての性格と相反するものであり，どの程度新しい試みを取り入れるかは難かしいが(1)の評価機能を有効に利用して積極的な実験，研究をすべきであろう。

本プロジェクトでは，まず当初のソフトウェア開発に用いるためのパーソナルツールとしての要件を満たすパーソナル逐次型推論マシン(Personal Sequential Inference Machine : PSI)をまず開発し，その後機能拡張した高機能版を開発する計画である。

3. ψシステムのアウトライン

ψはできる限り早期に，しかも実際にツールとして使えるマシンとして開発されることが要求されている。そのためには現状で最も実行効率が良いと考えられているスタックを用いた論理型言語の解釈／実行方式を採用すると共に極力小型，シンプルに実現する方針でした。

ψの設計目標はVAXに代表される32ビットスーパーミニコンピュータ並みの性能で，Lispマシン等のビットマップディスプレイ機能を有するスーパーパーソナルコンピュータ並みの使い易さを兼ね備えたパーソナルマシンとなることであるが，性能については今後の開発ソフトウェアの大規模化を考慮してパーソナルマシンとしては若干高めに設定されている。

具体的な目標性能としては実行速度がDEC-10 PROLOGコンパイラ版(DEC2060)と同程度，メモリ空間は1桁以上大きい4M語を標準実装として最大16M語と設定した。この理由は既に述べたように実行速度の高速化よりも使用可能なメモリ空間の拡大の方が論理型言語プログラム環境を完成させる上で急務であるとの判断による。

図1にψのシステム構成を示す。ψは基本的にローカルエリアネットワークに結合されたワークステーションとして利用されることを前提にしている。従ってψ自身はあまり多くの入出力装置を備えてはいない。基本システム構成としては良好なマンマシンインターフェースを実現するためのビットマップディスプレイ装置，マウスと固定ヘッドディスク(Winchesterディスク)装置を備えている。但し，ψはスタンドアロンでの利用も可能であり必要に応じてプリンタ等を付加することも勿論可能である。

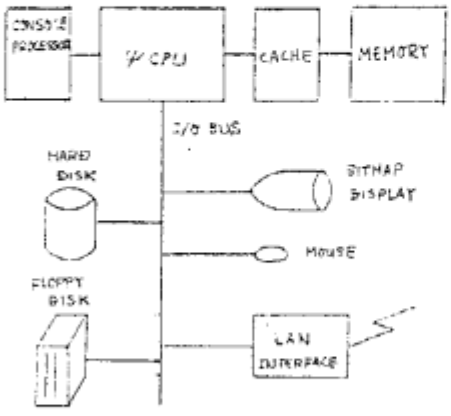


図1　ψシステム構成図

CPUはTTLにより実装することとし，充分な柔軟性を得るためにマイクロプログラム制御方式を採用し命令サイクルは200nsec以下を目標とした。アーキテクチャはPROLOG-likeな言語仕様を持つ機械語，核言語KL0の解釈実行に最適なように設計されており，最大の特徴は各語にタグを持たせ，その判定を高速に行なうハードウェア機構を持っていること(タグアーキテクチャ)，及びメモリアクセスの効率化を図っていることである。メモリアクセスはスタックを多用する論理型言語の解釈／実行方式では主要なオペレーションであり，その効率が直接マシンの実行性能に影響する。従ってψではキャッシュメモリの導入によりメモリアクセスの高速化を図った。仮想記憶については，現段階では論理型プログラムの動作特性がどのようなものかがわかっておらず，また仮想記憶システムへのガーベッジコレクションの導入の技術的困難さも考え合わせ，ψでは仮想記憶の採用は見合わせることとした。そのかわり実用規模のプログラムを動作させその挙動を測定するために充分と思われる主記憶を用意することにした。その大きさはDEC-10 PROLOGの使用経験等を踏まえて，標準4M語，最大実装16M語であれば適当であると判断した。またψではシステム立ち上げ，保守のためコンソールプロセッサとしてワンボードマイクロコンピュータを内蔵している。

入出力インターフェースについては極力標準化を図ることによってユーザーの要求に合わせた様々のデバイスを自由に付加できることを意図し，IEEE標準仕様であるIEEE-796バス(Multi-Bus)を採用した。

入出力制御はPSIのCPUが直接制御する方式とした。この理由はシステム構成の単純化を図ることにあるが論理型言語によるマルチウィンドウシステムなどの研究実験を行うために核言語KL0によって低レベルの入出力制御を行うことも考慮しているためである。

PSIは推論エンジンとしてのCPUのまわりに研究開発の努力を注いでおり、入出力バスは1本設けているが時に高速アクセスが要求されるデバイスとの接続や特殊デバイスの接続のために32ビットパラレルI/Oポートを設けている。

4. 設計上の選択点

核言語KL0はPSIの上での最も低いレベルの言語であり、従来のアセンブリ言語に相等する。コンパイラをはじめとしてPSIのオペレーティングシステムの核部分は核言語KL0で記述される予定である。

しかしながらKL0は述語論理に基づく高い言語仕様を持っているためPSIの機械語命令をどのように設定するかはマシンアーキテクチャを設計する上での重要な問題である。

以下に核言語KL0を高速に実行するために備えるべきマシンアーキテクチャについて、その設計上考慮したいくつかの選択点とPSIでの設計方針について述べる。

4.1. 核言語KL0の設定

PSIを実際に活用するためにはハードウェアと共にオペレーティングシステムの開発が必要である。オペレーティングシステムの記述には決定的処理と履歴依存処理の機能が必要であるが、これらの機能は本来述語論理の体系には異質のものである。従ってシステム記述用の目的のためには別の言語体系を導入するのが普通である。しかしこのように述語論理ベースの言語でシステム記述用の別言語の二つの部品を設けるとシステム全体の均質性が崩れ、ユーザーにとってシステムを統一的に理解することが困難となる。ことにアーキテクチャレベルでは、二種の異なる言語体系をサポートしなければならずそのハードウェア機能も複雑になることが予想される。

そこでPSIの機械語である核言語KL0はこれら本来異なる二種の体系を統合させ、簡素かつ統一的なハード、ソフトのインターフェースを作るよう設計された。即ち、PROLOGに代表される述語論理型言語の特徴の一つである宣言的にプログラムが書けるという良さと低レベルのシステム制御の記述のための組込述語を導入して機能拡張を図った。[2]

従って核言語KL0は2つの側面を持っている。一つはユーザー定義の述語であり、これはバックトラックを基本とするPROLOGに似た非決定的な制御の対象となるものである。もう一つは組込述語であり、これについてはなすべき処理は決定しており、バックトラックの対象とはならないものである。しかしいずれの場合もデータの受け渡しはユニフィケーションを基本とする。

4.2 核言語KL0の実行について

核言語KL0は実行時に決定される動的要素を多く含んでいるにもかかわらずコンパイル方式を採用したのは次の理由に依る。

1) 組込述語は決定的(deterministic)に実行される。従ってその内部表現はできるだけコンパクトにすれば、処理効率の向上とメモリ使用の効率の点から望ましいこと。

2) TRO(tail Recursion Optimization)[3]はコンパイラのレベルで対処しなければマシンレベルで動的に処理することが難かしいこと。

これらの理由により核言語KL0はコンパイルされ、マシン内部形式(オブジェクトコード)に変換される。

このオブジェクトコードを実行するマシンの制御方式としては、ハードで直接実行

を行なうハードワイヤド方式とマイクロコードにより解釈/実行を行なうマイクロプログラム方式が考えられる。どちらの方式を採用するかはオブジェクトコードのレベルと密接に関連している。ここではマイクロプログラムによりオブジェクトコードを解釈/実行する方式を取ったがそれは次の理由からである。

1) ユニフィケーション、クローズの実行順序制御等はメモリアクセスが中心であり、これらは分解され、低レベルのオブジェクトコードで処理されるよりも、一つのオブジェクトコードで処理される方が命令フェッチに要するメモリアクセスが少なくなる。そのようにハードウェアの単純化、簡素化のために命令フェッチユニットを持たないマシンでは、オブジェクトコードのレベルは高い方が望ましいこと。

2) 中のソフトウェア支援ツール、ハードウェア実験機としての性格を考えると、容易に機能の追加、修正のできるマイクロプログラム方式が望ましいこと。

3) 組込述語の処理は決定しており、この中にはハードウェアで直接実行した方が明らかに望ましいものもあるが、プログラムの実行順序制御、実行環境を変更する等の命令は、コンパクトなオブジェクト表現を取る限りマイクロプログラムによる実行が望ましいこと。

まとめると、中のオブジェクトコードの表現の高さのためと中のマシンとしての性格から、中の実現方式としてマイクロプログラム制御を採用した。

次に、マイクロインタプリタによる核言語KL0の実行方式について述べる。実行処理方式を逐次型に限って考えると、次の2つに絞ることが可能である。即ち、ヒープをベースにした実行メカニズムとスタックをベースにしたそれである。ヒープを基本とする処理方式では述語の呼び出しに伴い生成される変数領域は新しく作られるだけで解放つまり消滅はしない。一方スタックを基本とする処理では変数領域は、通常呼び出し元への復帰時に消滅する。

一般に通常の言語の処理方式としてスタックを基本とするものが多いのは、呼び出された手続きがバックトラックの対象とはならないので、実行終了時点の呼び出し元への復帰時にはその変数領域を解放してもよいからである。一方核言語KL0の仕様ではバックトラック制御が行なわれ、クローズによってはバックトラックの対象となり得るものが存在する。従ってその場合にはそのクローズの変数領域は解放できない。

さらにユニフィケーションでは変数と構造体がバインドすることがある。この場合構造体への参照ポインタが変数には格納されるが、その構造体を含む変数領域を解放してしまうと変数の参照先がなくなってしまい、いわゆるdangling referenceになってしまう。

このように核言語KL0の仕様では変数領域が割り当てられるスタックは単純には解放することができない。それにもかかわらず中でスタックをベースとする実行メカニズムを採用した理由を次に示す。

1) ヒープベースによる実行メカニズムではメモリ空間の消費が激しく、ガーベッジコレクションが実行される頻度が増すこと。

2) ヒープベースを採用することにすれば広いメモリ空間を揃えることが望ましく、従って仮想記憶方式が有効であるが中では採用を見合わせたこと。

3) DEC-10 PROLOGコンパイラ版は基本的にスタックをベースにした実行メカニズムを採用しているが高い処理効率が得られていること。

1.3 sharing / copy 方式について

ユニフィケーションによって変数に値が代入されるという操作は基本的にスタック上に生成された変数セル(ローカル)への値の代入として実現される。

しかし代入すべき値が構造体データの場合は、1語対複数語の対応関係を作りだす必要がある。最も単純な方法は代入すべき構造体データの複製を作ってそれへのポインタをセットするcopy方式であるが、複製を作らずに相手のデータ構造を共有することにより実行効率をあげることを意図したstructure sharing方式[4]もある。いずれの方式も長所、短所を持っているが今では後者のstructure sharing方式を採用したがその理由を以下に述べる。

図2は変数Xと構造体struct(A1, A2)がユニファイされたときのsharing方式と一般的なcopy方式を用いた場合に生成される環境を各々示したものである。

図2-(a)に示されるsharing方式では構造体の公な型(skelton)と実際の値とは分離して管理され、両者へのポインタを通にしたmoleculeによって構造体データが表現される。従ってsharing方式ではmoleculeの生成だけの手間で済む。一方図2-(b)に示されるcopy方式では、構造体のフレームが新しく生成され、A1, A2に対応する変数セルからそれぞれそのフレーム内変数セルへのポインタが格納される必要がある。即ちcopy方式では構造体の要素数に比例した複製するための手間がかかることになる。

この状態でさらに変数Xと構造体struct(a1, a2)がユニファイされる場合を考えてみる。どちらの方法によっても構造体の名前(functor)の一致を調べるのは同一であるが各要素へのアクセスの仕方に差が生じる。即ちstructの第1要素である変数A1とアトムa1のユニフィケーションが実行される場合のA1のセルアドレスの生成に要する手間が異なるのである。sharing方式では構造体のない型を先ずアクセスすることによりその第1要素A1の値領域であるA1変数セル位置を知り、その変数領域を示すベースに加算することで求めるセルアドレスA1が得られる。つまり1回のメモリアクセスと1回の加算が必要である。これに対してcopy方式ではXの持つ構造体フレームへのポインタにオフセットするだけでよく1回の加算で求めるセルアドレスが得られる。

このように構造体同志のユニフィケーションが行なわれる場合にはcopy方式のほうがオーバーヘッドは小さく、sharing方式では構造体要素へのアクセス毎に余計なメモリ参照が必要となる。そのようにアドレス情報が32ビットまで持てるマシンでは1ワード内にmoleculeを格納することは難しい。moleculeの表現に2語要する場合には、図2-(a)を例にとると変数セルXにはmoleculeへのポインタが格納されることになり、molecule読みだしのためにさらにメモリアクセスを要することになる。

ところが、今度は変数A1とアトムbが単にユニファイされる場合を考えてみる。sharing方式では変数セルA1へのアクセスは直接であるのに対し、copy方式では変数セルA1のポインタを参照するために同

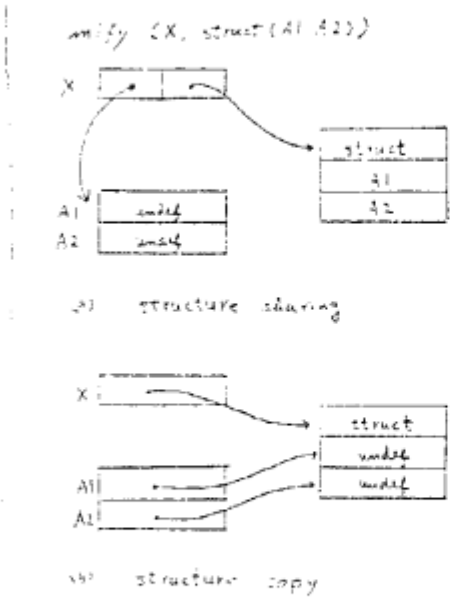


図2 変数Xと構造体struct(A1, A2)のユニフィケーション

待アクセスとなり sharing 方式に比べてオーバーヘッドは大きくなる。

以上のようにどちらの方式が優れているかはユーザーが構造体をどの様に扱うかという特性に強く依存しており一概には判定できない。Y ではこれまでの PROLOG 使用経験より copy 方式での複製の手間のほうが、sharing 方式での構造体要素へのアクセスに要する手間より大きいと考えて sharing 方式の採用に踏切ったが、この問題は引き続きプログラム特性の評価もふまえて研究する必要があろう。

4.4 ユニフィケーション高速化のサポート

被言語 KL0 の解釈/実行の中心はユニフィケーションであり、Y の実行効率を上げるためには、これを高速実行するためのハードウェアサポート機構が必須である。ユニフィケーションの処理の中心はメモリ参照とそのデータの属性判定である。従ってその高速化は結局

1) メモリ(特にスタック領域)アクセスの高速化.

2) データタイプの判定の高速化に帰結する。

スタック領域へのアクセスの高速化は単純な LIFO 形式のハードウェアスタックを用いるのが一般的である。しかしバックトラック処理を行なう被言語では単純に領域をポップアップ(即ち解放)できない。この場合のスタック動作を忠実にサポートするハードウェアスタックの機構はかなり複雑なものになることが予想され、その有効性等については充分な検討が必要であると思われる。

従って Y ではスタックアクセスの高速化のために1ワード分の変数フレーム(即ちスタックトップフレーム)を CPU 内のフレームバッファ(高速な専用レジスタ)に割り当てて、そこでユニフィケーションが実行されるようにした。これは TRO を実現するに当たっての引数レジスタ[3]としての使用を兼ねることもできかなりの高速化が期待できる。

一般のメモリアクセスの高速化にはキャッシュメモリを設けることにより対処することとした。キャッシュメモリの長所はスタックアクセスのみならず命令読み出しをも含めたすべてのメモリアクセスを均等に高速化できる点にある。キャッシュメモリの方式としてはライトスルー方式とライトバック方式が代表的であるが、若干性能がよいと言われているライトバック方式を採用した。

2)のデータタイプの判定の高速化のためにはすべてのデータにタグを付与するタグアーキテクチャを採用した。即ち、すべてのデータにそのデータタイプを表現するタグビットを付加することにもにタグビットを高速に判定するための専用ハードウェアを設けた。

4.4 オペレーティングシステムのサポート機能

Y は効率のよいプログラム作成環境をユーザーに提供することが求められておりこのための対話型オペレーティングシステムの開発が予定されている。このオペレーティングシステムはエンドユーザインタフェースシステム(コマンド・インタプリタ、ウィンドウシステム等)、プログラミングシステム(エディタ、デバッガ等)あるいはファイルシステム等の多種のソフトウェアシステムから成る。システムの均質性を重視していることからもオペレーティングシステムについても述語論理の体系で記述されるため、Y のサポートは必須である。

Y のオペレーティングシステムサポート機能については、相互述語を効率よく実行するという以外にハードウェアレベル、ファームウェアレベルでのサポートを設けている。即ち、記憶管理システムにおけるメモリ領域の割り当て、ガーベッジコレクション処理等は通常はマイクロプログラムで記述される。また、プロセス管理システムでの

プロセススイッチ機構もマイクロプログラムで記述される。ψではさらにプロセス情報を高速専用メモリ内に格納している。これは核言語KLθの解釈/実行のためには成される実行環境が通常の言語のものよりも大きく、プロセス環境を主記憶に格納する場合のメモリアクセスオーバーヘッドができるだけ小さくするためである。

このようなファームウェア、ハードウェアレベルのサポートの他にオペレーティングシステムの核部のためのサポートとしてガーベッジコレクションの対象とならない領域を設けた。この領域に格納されたプログラムはガーベッジコレクションの実行中でも処理が可能となっており、特にここに格納されたプロセスはSupra G.C. プロセスと呼ばれている。この特別な領域を設けた理由は、ガーベッジコレクションの実行途中においてもオペレーティングシステムの核部は実行できる形にしておいた方が対話型システム環境では望ましいためである。

5. ψのアーキテクチャ

ψ章で述べた様々の設計上の考慮に基づいてψのマシンアーキテクチャが決められている。

ψは基本的にはワードマシンであり、データ、機械語命令は一語を単位としてアクセスされる。またデータタイプ判定の高速化のためにすべてのデータにはタグが付与されておりタグマシンでもある。

以下にψのアーキテクチャ上の特徴のなかから、アドレス空間とデータおよび機械語命令の内部表現について述べる。

5.1 アドレス空間

ψは32ビットのアドレス空間を持っており、メモリアクセスは図3に示すように8ビットのエリア番号と14ビットのページ番号、10ビットのページ内オフセットという論理アドレス形式でアクセスが行なわれる。

ψの実際のメモリシステムは仮想記憶ではなく実記憶システムであるが、核言語KLθのプログラマからは論理的に16M語まで拡張できる256枚の"エリア"と呼ばれる論理アドレス空間により構成されているように見える。各エリアは全く独立した空間でありψのオペレーティングシステムにより管理される。このようなエリアの概念を導入した目的は次の理由に依る。

1) 核言語KLθの解釈/実行のために必要とされる各種スタック領域を論理的に別空間、即ち別々のエリアに割り当てることによって、同一空間に複数のスタックを生成する場合に生じるスタック領域同志の衝突の問題を論理レベルからなくしたかったこと。

2) ψは論理型プログラミング環境にプロセスを導入することを目標としておりプロセス毎に独立の空間を与えたいため。

これらが、"エリア"の概念を導入した理由である。

プログラムの実行環境として必要とされる各種スタックを別々のエリアに割り当てるとして1つのプロセスの実行には4枚のエリアが必要とされる。プログラム格納領域(ヒープ)は共有を許しており、これに4エリアを割り当てるとすると、256枚のエリアからは、最大63個までのマルチプロセスがサポートできる。

各エリアの空間は更に1K語の大きさの"ページ"を基本単位として管理されている。即ちエリアがスタックやヒープとして使用される場合にその領域の増減の基本単位は1K語である。

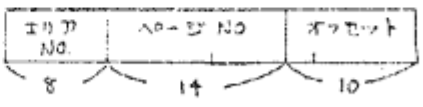


図3 論理アドレス

図4にアドレス変換のメカニズムを示す。論理アドレスから物理アドレスへの変換はテーブルを2回参照することで行なわれる。即ちまずエリア番号によりエリアテーブルがひかれこれによりページテーブルのベースが生成される。ページテーブルベースとページ番号を加算した結果でページテーブルをアクセスすると物理ページアドレスが得られる。これをページ内オフセットと結合することにより計24ビットの物理アドレスが生成される。

大型汎用機ではアドレス変換の高速化のためにTLB(Translation Lookaside Buffer)を持ち、求める論理ページエントリがその中にあれば主記憶中の変換テーブルを参照することなく高速に物理ページエントリを求める方式がよく用いられる。

Ψではこの方式は採用せずにエリアテーブル、ページテーブルを専用の高速メモリ内に持つこととし、この専用ハードウェアによりアドレス変換は常に1マシンサイクルで行なわれる。この方式を採用したのは以下の理由に依る。

1) ミスヒットが発生したときTLB方式では主記憶上の変換テーブルを参照せねばならず、それによるオーバーヘッドが存在すること。

2) Ψのキャッシュメモリは論理アドレスで管理されており、キャッシュミスヒット時には常に論理アドレスから物理アドレスへの変換が必要である。Ψではキャッシュミスヒット時に備えてキャッシュメモリへのアクセスと並行してアドレス変換を行うよう設計しており、ミスヒットが判明した時点で対応する物理アドレスが生成されている。このようなアドレス変換方式でTLB方式を採用すると更にTLBでのミスヒットも管理せざるを得ず、メモリシステムが複雑になる。

3) ガーベッジコレクション実行中では、すべてのメモリ空間を探索する必要がありその参照アドレスパターンにはあまり局所性のないことが予想される。従ってTLB方式ではあまりヒット率を高くすることができず、TLBミスヒットによるオーバーヘッドを伴うことが予想される。

4) エリアテーブルのエントリは256語であるが、ページテーブルのエントリはエリア当たり16kページを含むので論理的には256×16K≒4M語必要である。しかしΨでは仮想記憶方式を採用していないためシステム全体で使用できるページテーブルエントリの総和は実記憶のページ数以上大きくならない。即ちΨの最大実装主記憶の容量は16M語であるからページテーブルエントリの総和は16k個を越えないため変換テーブルをすべてアドレス変換ユニット内の高速メモリチップ上に格納することが可能である。

このような理由によりTLB方式は採用されなかったが、仮想記憶方式を用いる場合には有効な手段である。

エリアは論理的に各最大16M語まで拡張可能であるが、あるプログラムが使用するエリアがそれぞれ最大どれだけのページを必要とするかは事前に予測できない。更に1つのプロセスは少なくとも4つのエリアを使用するがプロセス毎にその使用状況は異なる。従って1枚のページマップメモリ上にすべてのエリアに対する使用中のページに関するページテーブルを格納する方式では、エリアのページ数の増加により他のエリアのページテーブルとの衝突が生じるこ

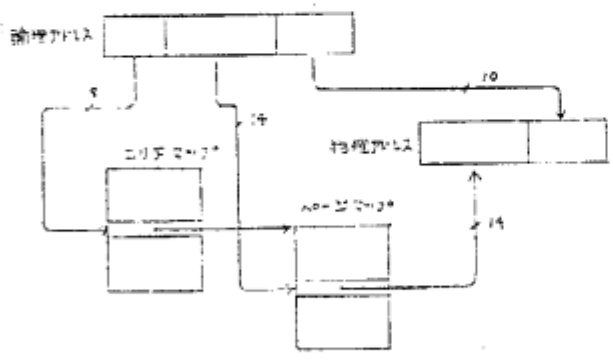


図4　アドレス変換メカニズム

ともあり得る。これをなるべく避けるためには実装ページ数の数倍程度のページ数が登録できるページマップメモリを用意してその中にあるエリアに対するページテーブルをなるべく分散させて配置するのが望ましい。Psiでは標準実装主記憶4M語に対しページマップメモリの大きさは16K語とし標準実ページ数の4倍を確保することとしている。もしもページマップメモリの中でページテーブル用意の衝突が生じた場合には割り出し(Trap)が生じページテーブルの再配置を行なう。このアルゴリズムも種々考えられるがこの方式がよいかは今後の課題である。

4.2 データ表現

Psiはパーソナルマシンであるが、今後の論理プログラムの大規模化にも充分耐えられるように、上述のアドレス空間の拡張とともにデータ表現能力も強化されている。

Psiの一語は図5に示すように32ビットのデータ部とaビットのタグ部から成る40ビットで構成される。タグ部の上位2ビットはガーベッジコレクション用のマークビットであり、次の6ビットはデータを識別するためのデータタグである。データ部は32ビット幅のデータもしくは論理アドレスを表現している。Psiの取り扱うデータのすべてにはタグが付与されており、主記憶上のすべてのデータは一意的に識別される。

ユーザーが取り扱えるデータタイプとしてPsiは次のものを用意している。

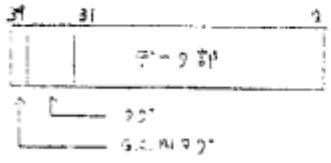


図5 ワードフォーマット

i) シンボル

シンボルは検索語KLのアトム情報を表現することを目的としているが、通常文字列表現との対応関係はない。この対応付けはオペレーティングシステムにより管理されている。

ii) 整数、実数

通常のマシンが扱う整数、実数と同一のものである。

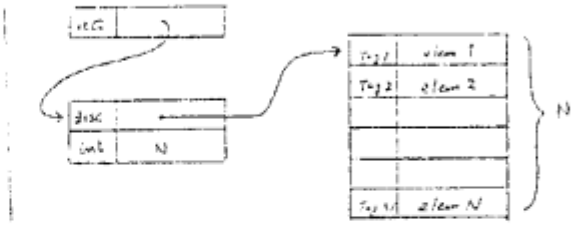
iii) ベクター

ベクターは決められた要素数からなる1次元配列であり、2進木リストをはじめとする各種構造体データを表現するのに用いられる。ベクターは標準形として図6に示すようにベクター記述子を介してアクセスされる。ベクター記述子のユワード目はベクターの要素数を示すものである。

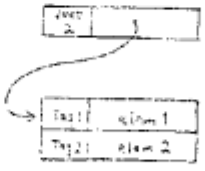
しかしこのような表現方法ではベクターへアクセスする際に必ずベクター記述子の読み出しが必要でありオーバーヘッドがある。例えば2進木リストのような要素数の少ない小規模のベクターは頻繁に利用されることが予想され、このようなベクターのアクセスを高速化する目的でPsiではデータタグ部に要素数情報を埋め込んだダイレクト型ベクターを設けた。

iv) ストリング

ビット列や文字列の表現のためにPsiではストリングが用意されている。ストリング



(a) 標準ベクター表現



(b) ダイレクト型ベクター表現

図6 ベクターの表現

の表現はベクターと同じように記述子を介してストリング本体へアクセスする方法を用いる。

ベクターとストリングの違いは次のとおりである。

(a) ベクターの中には変数を含むことが許されているがストリング中には含むことは許されない。

(b) ベクターの各要素は1語の任意データでよく別のベクターを要素として持つことも可能であるが、ストリングの各要素は1ビット、1バイト、2バイトのいずれかでなくてはならない。

次に、プログラムの内部形式中に現われるデータタイプとしては次のものがある。

1) ローカル変数

2) グローバル変数

これらは共に述語もしくは構造体のローカル変数、グローバル変数であることを示すとともに変数セル位置を指示するのに用いられる。

3) 述語呼び出し

クローズ中に表われるゴール呼び出しの述語名はコンパイラにより呼び出すべきクローズに対応する内部形式へのポインタに置き換えられ、タグ部にはコードを示す情報がつけられる。

4) 組込述語

Mはワードアクセスを基本としているためクローズの内部表現も1要素/語の形式で表現される。しかしながら組込述語の処理に於いては引数データの読み出し処理が通常のユニフィケーションに比べて簡単である。このことを利用して組込述語に対しては図7に示すように基本的には1語のコンパクトな内部表現形式を採用した。タグ部では組込述語であることが示され、データ部にはオペコードにつづくまでのコンパクト型の引数が格納される。

Mではさらにプログラムの実行時にマイクロインタプリタによりダイナミックに生成されるデータタイプも存在する。代表的なものをいくつか上げると、

1) レファレンス

変数同志がユニファイされたときにはじるポインタを示すものである。

2) モレキュール

変数と構造体がユニファイされたときに生じるものであり図8に示す表現がとられる。

3) コントロールブロック

クローズの実行に伴って発生するクローズ環境情報を保存するためのデータブロックであり、その中にはユニフィケーションを行なうのに必要な実行環境情報やバックトラック、呼び出し元クローズへのリターンを制御するための情報が格納される。

4.3 クローズ表現

核言語KLBにおけるクローズはPROLOGの仕様と同じであり、ヘッド部と多数のゴールの結合であるボディ部から構成される。1つのクローズはコンパイラにより図9に示すような語の並びである内部形式に変換される。クローズヘッドに対応

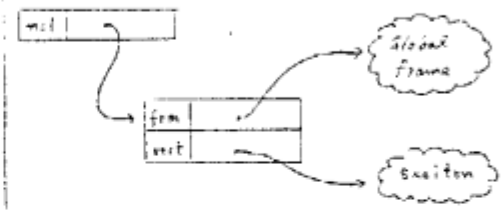


図8 モレキュールの表現

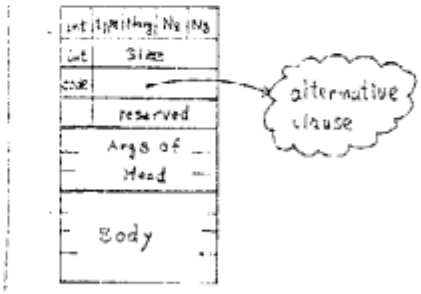


図9 クローズ内部表現

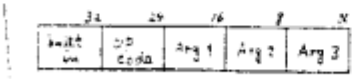


図7 組込述語のフォーマット

する部分はヘッダ部とヘッド引数から構成される。ヘッダ部はワードから成り、ヘッドの引数個数を示すNarg、クローズに含まれるローカル変数、グローバル変数の数を表わすNl, Ngとクローズのタイプ(ユニットクローズ等)を示すTypeフィールドから成る1語の他に、このクローズ内部形式のテーブルサイズを示す1語、他の選択肢へのポインタを格納した1語等から成る。クローズヘッドに含まれる引数は内部形式の引数部に展開される。

ゴールの結合のされ方には、次の3つのタイプを基本形式として設けている。

1) AND結合

ゴールとゴールが探索木ではand木として表現されるものであり、各ゴールは呼び出し述語へのポインタとその引数が展開された形で表現される。この結合の例を図10に示す。

2) OR結合

ゴールとゴールが探索木ではOR木として表現されるものである。実行順序はこの結合されたクローズの先頭以外はバックトラックの対象となるような制御を取る。

```
append ( [], X, X ).

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

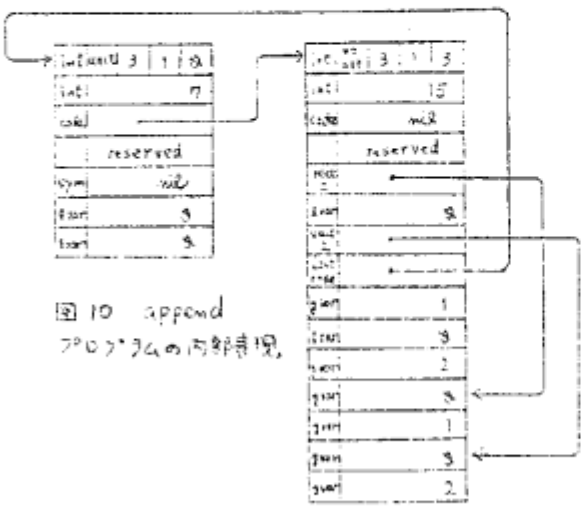


図10 append プログラムの内部構成

3) CASE結合

この形で結合されたゴール同志はIndexが付けられたものと見なすことができる。この形で結合されたゴールの一つが実行時に選択されるが、バックトラックの対象とはならない結合のされ方である。

5. おわりに

本論文ではICOTが中心になって開発を進めているパーソナル逐次型推論マシン、Pの開発目標、設計思想およびアーキテクチャの時敵としての機械語の内部形式、データ表現、メモリ空間について述べた。

現在アーキテクチャ設計は完了し、ハードウェア試作に向かって詳細設計を進めているところである。性能としてはDEC-10 PROLOGコンパイラ版(DEC-2060)相当の性能が実現できる見通しである。また機械語の内部形式についてはTRO等の最適化手法の導入によって、現在の語を基準にした形式ではなく組込述語と同様のコンパクトな表現形式も検討中である。今後、Pのマイクロプログラムレベルでの様々の最適化技法についても検討してゆく予定である。

最後に、本研究の機会を与えて下さった渕一博研究所長、村上国男オー研究室長、討討を通じて有益な助言を与えて下さった近山隆氏はじめICOTメンバー諸氏に深謝する。

参考文献

1. Warren, D.H.D: Implementing Prolog
- compiling predicate logic program
D.A.I Research Report no 39-40 (May 1977)

2. 他: 検証語その版の仕様
ロジックプログラミングコンファレンス(3月1983)

3. Warren, D.H.D: An improved Prolog
Implementation which optimises
Tail Recursion
Proc. of the Logic Programming
Workshop, Hungary (July, 1980)

4. Boyer, R.S and J.S. Moore:
The sharing of structure in theorem
proving programs
Machine Intelligence Vol. 1, 7
Edinburgh Up (1972)