6G−1

遅延ナローイング抽象機械のアーキテクチャ

中村敦司、　鈴木太朗、　中川康二、　井田哲雄
筑波大学

1　はじめに

我々は、遅延ナローイング(lazy narrowing)と呼ばれる計算モデルに基づいたプログラミング言語Evを設計し、その処理系を開発している[1]。この言語は、等式の反駁による証明をプログラムの実行の基本的な機構とするものであり、証明の過程の中で値の必要になった関数項を、遅延評価により簡約する。我々の処理系では、この言語で記述されたプログラムを、まず基本式(basic form)と呼ばれる形式に変換する。さらに、それを抽象機械の命令列に変換し、その命令列を実行する[2]。我々は、この抽象機械を遅延ナローイング抽象機械(Abstract Lazy Narrowing Machine: 以下、LNMと略す)と呼んでいる。

我々の言語は、証明を実行の機構とするという意味では論理型言語であり、項の書き換えにより簡約を行なうという意味では関数型言語である。LNMは、Prologの抽象機械であるWAM(Warren Abstract Machine)[3]に対して、関数型言語としての側面を実現する機能を付加したものと捉えることができる。本稿では、LNMの構成と命令セットを、WAMとの違いに重点をおいて説明する。

2　記憶領域の構成

LNMの記憶領域は、ヒープ、スタック、トレイルの3つの領域からなる。

ヒープ　ヒープは、リストやシンボルといったデータ構造を格納する領域である。この領域はWAMの大域スタックに対応するものであるが、関数項を表すデータ構造が格納できるように拡張されている。このデータ構造は他のいくつかの項の部分項として共有されることがあるため、それを簡約した値が繰り返し参照される可能性がある。参照されるたびに関数項を簡約するオーバーヘッドを避けるため、関数項が簡約されると、その値をこのデータ構造の中に覚えておき、二回目以降の参照では覚えている値を利用する機構を備えている。これを実現するため、関数項を表すデータ構造は、図1に示すように、その関数項の値を覚えておく場所と、引数をおく場所からなる。この値を覚えておく場所のことを、キャッシュフィールド(cache field)と呼んでいる。

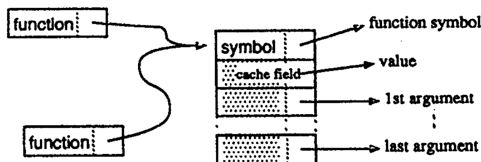

図1. 関数のデータ構造

スタック　スタックは、定義された関数を呼び出した時に、戻り番地や局所的な変数などを割り当てる領域である。この領域はWAMの局所スタックに対応するものである。WAMがレジスタを介して引数を渡すレジスタマシンであるのに対して、LNMはスタックを介して引数を渡すスタックマシンである。フレーム(frame)は、関数が呼び出される毎にスタックに割り当てられる。これは、図2に示すように、実引数を格納する領域、バックトラックの処理のためのチョイスポイント(choice-point)、戻り番地や局所的な変数を割り当てる環境(environment)の3つの部分からなる。このうち、チョイスポイントと環境は、必要に応じて割り当てられる。ある時点で簡約を行なっている関数のフレームの環境を環境ポインタEが指す。また、バックトラックが起きた場合に実行を再開する関数に対応したフレームのチョイスポイントをバックトラックポインタBが指す。

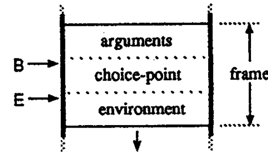

図2. スタックフレームの構造

トレイル　トレイルは、実行の途中で値の定まった変数の履歴を覚えておく領域であり、変数へのポインタが格納される。チョイスポイントの中にはトレイルへのポインタが格納されており、バックトラックが起きた時に、チョイスポイントが作られた時点よりも後に値の定まった変数を、未束縛の状態に戻す。LNMでは、変数だけではなく、関数項の値もバックトラックが起きた時点で未束縛に戻す必要がある。そのため、WAMとは異なり、トレイルには関数項のキャッシュフィールドへのポインタが格納される。

レジスタ　命令の中に明示的には現れないが重要なレジスタに、構造ポインタSとレジスタA、スタックポインタがある。Sは構造を持つデータを単一化する時に用いられる。Aは変数の値を辿った時(後述)に得られた値を、一時的に覚えておくためのレジスタであり、WAMにはないものである。LNMはスタックマシンとして設計されているため、WAMのA_nレジスタとX_nレジスタのようなレジスタファイルはない。代わりに、引数をスタックの先頭に積むために、常にスタックの先頭を覚えているスタックポインタも用意されている。

3　実行制御に関する命令

LNMの実行を制御する命令を図3に示す。このうち、右の欄に示す4つの命令はWAMにはない。

関数の実現　我々の言語では、$f(\vec{s}) = d$という形の等式は、記号fに関数として定義されている手続きを呼び出す(簡単のために、これを「関数を呼び出す」と表現する)ことにより証明される。callは、この手続き呼び出しを行なう命令である。

Architecture of Abstract Lazy Narrowing Machine
Atsushi Nakamura, Taro Suzuki, Koji Nakagawa and Tetsuo Ida
University of Tsukuba

call f/n	try_me_else ℓ	jmp_if_function x ℓ
execute f/n	retry_me_else ℓ	push_args
proceed	trust_me_else_fail	call_function
allocate n	jmp ℓ	pop n
deallocate	fail	

図 3. 実行制御のための命令

関数は、複数の条件付き等式(各々の等式を規則と呼ぶ)の集合として定義される。関数が呼び出されると、チョイスポイントを操作しながら、その一つ一つの規則を順番に試みる。try_me_else に続く 3 つの命令は、このチョイスポイントの操作を行なう。

規則は、左辺が関数項である等式(等式部と呼ぶ)が成り立つために証明されなければならない、いくつかの等式を与えるものである。規則の実行は、まず環境を確保し、証明されなければならない等式を左から順番に試みる。それらのすべてが証明されると、証明したい等式の右辺が等式部の右辺と等しいかどうかの証明を試みる。これも成功すると、最後に環境を捨てて証明を終了する。allocate と deallocate は環境を確保する命令と捨てる命令であり、proceed は証明の手続きを終了する命令である。execute は、最後の証明が関数を呼び出す場合に、末尾呼び出しの最適化を行なうものである。pop は、証明の手続きを終了した時に、必要に応じてフレームの中の実引数の領域を捨てる命令である。jmp は制御を移す命令であり、fail は明示的にバックトラックを起こす命令である。

動的な関数呼び出しの実現　証明する等式の左辺が変数であり、それが 2 回目以降に現れた変数であれば、その変数の値が関数項かどうかを動的に検査する。もしそれが関数項であればその関数を呼び出す。このように、変数に束縛されている関数を呼び出すことを、「動的な関数呼びだし」と呼ぶ。もし変数の値が関数項でなければ、後述する get 系の命令により証明を試みる。

jmp_if_function 命令は、変数の値を辿って関数項かそうでないかの検査を行なうものである。値を辿る操作は WAM のデレファレンス (dereference) に相当するが、LNM では、値の定まっている関数項も辿る点が異なる。この命令では、検査を行なうと同時に、辿った結果得られた値をレジスタ A に格納する。その値がまだ簡約されていない関数項であれば、まず、push_args 命令により、そのすべての引数をスタックの先頭に積む。次に、後述する bind 系の命令で証明する等式の右辺をスタックの先頭に積むと同時に、その値をキャッシュフィールドに設定する。そして、call_function 命令により関数を呼び出す。この一連の動作の間、レジスタ A は簡約されるべき関数項を保持し、この関数項から呼び出される関数や引数が動的に得られる。

4　項の操作に関する命令

項を渡す命令　等式の両辺のどちらかが初めて現れた変数の場合には、他方の辺の項をその変数の値とすることにより、その等式が成り立つような変数への代入を得ることができる。この処理は、図 4 の左の欄に示される put 系の命令により実現される。LNM の put 系の命令は、WAM の put 系の命令とほとんど同じであるが、WAM では引数レジスタに項を設定するのに対して、LNM ではフレームの中に割り当てられた変数に項を設定する点が異なる。

図 4 の中央の欄に示す push 系の命令と右の欄に示す bind 系の命令は、put 系の命令とほとんど同じ動作をする。push 系の命令は、関数呼び出しの際に実引数をスタックの先頭に積む。bind 系の命令は、動的な関数呼び出しの際に、等式の右辺の項をスタックの先頭に積むと同時に、それを関数項のキャッシュフィールドの中に設定する。

項を単一化する命令　単一化に用いられる命令を図 5 に示す。unify 系の命令は WAM と同じである。get 系の命令は、引数レジスタではなく、フレームの中に割り当てられた変数や引数に対して単一化を行なう点が WAM とは異なる。また、レジスタ A に格納されている項に対して単一化を行なう場合には、図中のパラメータ x を省略する。

5　おわりに

本稿では、遅延ナローイングに基づく言語 Ev を効率良く実現するための抽象機械 LNM のアーキテクチャを示した。このアーキテクチャは WAM に基づいたものであり、言語 Ev が Prolog と同様の技法により実現されることが示された。我々は、既にこの抽象機械のシミュレータを作成しており、言語仕様や細かい命令の動作の確認に用いている [4]。

参考文献

[1] 鈴木, 井田. 遅延ナローイング計算系に基づく言語 Ev とその処理系. 情報処理学会第 *44* 回全国大会予稿, *5F-7*, 1992.

[2] 鈴木, 中川, 井田. 遅延ナローイングに基づく言語 Ev の等式翻訳方法. 情報処理学会第 *44* 回全国大会予稿, *6G-3*, 1992.

[3] H. Aït-Kaci. *Warren's Abstract Machine.* MIT Press, 1991.

[4] 中川, 鈴木, 中村, 井田. 遅延ナローイング抽象機械のシミュレータ. 情報処理学会第 *44* 回全国大会予稿, *6G-2*, 1992.

put_variable	v	x	push_variable	v	bind_variable	v
put_value	v	x	push_value	v	bind_value	v
put_nil		x	push_nil		bind_nil	
put_constant	k	x	push_constant	k	bind_constant	k
put_list		x	push_list		bind_list	
put_structure	c/n	x	push_structure	c/n	bind_structure	c/n
put_function	f/n	x	push_function	f/n	bind_function	f/n

図 4. 項を渡す命令

get_variable	v		unify_variable	v
get_value	v		unify_value	v
get_nil		$[x]$	unify_nil	
get_constant	k	$[x]$	unify_constant	k
get_list		$[x]$		
get_structure	c/n	$[x]$	unify_void	n
get_function	f/n	x	unify_local_value	v

図 5. 項を単一化する命令