プログラミング－言語・基礎・実践－ 18－6
（1994. 7. 21）

疎結合並列計算機上の遅延評価型関数型言語処理系の性能評価

田中哲朗, 山本貞英, 武市正人
{tanaka,yamamoto,takeichi}@ipl.t.u-tokyo.ac.jp
東京大学工学部

概要

関数型言語は言語中に逐次実行を陽に含まないので並列実行に向いていると考えられる. 関数型言語を疎結合並列計算機上で実行する場合, 共有メモリを前提にした逐次実行モデルをそのまま適用することはできない. 筆者らは, 関数型言語の逐次実行のモデルである STG(Spineless Tagless G-machine) を元に関数型言語の並列実行のモデルを提案し, 関数型言語 Gofer の処理系を疎結合並列計算機 AP1000 の上に実現した. 今回はその処理系の高速化のための手法を検討する.

Evaluation of lazy functional language implementation on loosely coupled multiprocessor

Tetsurou Tanaka, Tomohide Yamamoto, Masato Takeichi
{tanaka,yamamoto,takeichi}@ipl.t.u-tokyo.ac.jp
Faculty of Engineering, University of Tokyo,
Hongo 7-3-1, Bunkyo, Tokyo, 113 JAPAN Abstract

It has been suggested that functional programs are suitable for programming parallel computers owing to their inherent parallelism. We propose a parallel evaluation model of functional programs based on the STG(Spineless Tagless G-machine) model proposed for sequential evaluation, and describe our parallel implementation of a functional language Gofer on the AP1000 parallel computer.

1 はじめに

関数プログラムは評価の際に副作用を伴わないため，評価結果は評価の順序によらない．並列化の際に多くの言語では並列実行するプロセス間の実行順序を決めるための明示的な同期や通信が必要となるが，関数プログラムはそのままの形で並列に実行することができる．

先行評価型の関数型言語と遅延評価型の関数型言語とでは，並列化の事情が若干異なっている．前者では関数は引数に関して必須 (strict) なので，下手に並列実行しても並列化によるオーバヘッドがかかるだけだが，後者では並列実行させようとする引数が本当に必要かどうかは実行時まで分からない場合があり，並列実行の結果として無駄な実行が生ずることがある．この点で，先行評価の方が並列度を上げるのに向いているが，遅延評価の持つ要求駆動実行のメカニズムを並列の枝刈りなどに用いるアイデアもあり，優劣は単純にはいえない．

遅延評価型関数型言語の並列処理系は GAML[1], GRIP[2], $<\nu - G>$[3] などいくつか作られている．ただ，これらの多くは単一プロセッサ上の処理系の延長として実現できる密結合並列計算機上の処理系であって，疎結合並列計算機上の処理系を実現した例は少ない．

本研究では，他言語の疎結合計算機での実現技術を，既存の逐次関数プログラムの実行モデルである STG(Spineless Tagless G-machine) に適用し，遅延評価型関数型言語 Gofer の効率的な処理系を AP1000 上に実現した．

今回は，その処理系に静的解析等の最適化をほどこして，より高速な実行をさせることを試みた．実際に並列実行させるところまではいかなかったが，予備実験としておこなった逐次実行において，各種の最適化の効果が確かめられた．

2 遅延評価型関数型言語の実行

遅延評価型関数型言語を効率的に実行するために，template instantiation[5] , G-machine[6], TIM[7] など様々な実行モデルが提案されてきた．

我々が並列化のベースとして用いた Gofer[8] は Haskell 風の言語であり，処理系は G-machine に従ったバイトコードを生成する中間コードインタプリタと，バイトコードとほとんど1対1に C 言語に変換する C 言語ファイルを生成するコンパイラからなっている．

並列化するにあたってはこの処理系が出すコードを並列実行させることも考えたが，より効率的な実行のために G-machine を改良して作られた実行モデル STG(Spineless Tagless G-machine) にもとづいたコード生成をおこなった．

STG[4] は C 言語のような逐次言語に変換して効率的に実行できるように設計されている．以下では簡単にその特徴を述べ，並列化にあたっての特色を述べる．

2.1 クロージャの構造

遅延評価を実現するために，

```
g1 x y = h (f x y)
g2 f x y = h z z
  where z = f x y
```

に現れる`f x y`のような式は，値が必要になった時に評価できる構造を持つ必要がある．また，同じ式を一度しか評価しないように，この構造は評価後は評価済みの値に *update*(上書き) する必要がある．

遅延評価型関数型言語はでは式は弱頭部正規形 (Weak Head Normal Form) になるまで評価される．式を更に評価する必要があるかどうかを決めるため，一般の処理系では WHNF になっている場合は値，それ以外を thunk と呼んでタグなど区別することが多いが，STG では両方を区別せずにクロージャと呼ばれる図 1のように先頭ワードが info テーブルを指すという単一の構造で表す．

以下では，WHNF に対応するクロージャを評価済みのクロージャ，thunk に対応するクロージャを未評価のクロージャと呼ぶ．なお，図 1の info テーブル中の点線の部分は本研究で拡張した部分で元々の STG にはない．info テーブルに新たなエントリを加えて新たな機能を容易に実現できる点も STG の優れている点である．

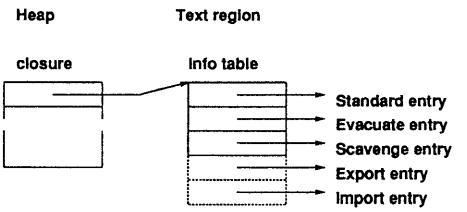

図 1: クロージャの構造

あるクロージャを評価する時には，評価済み，未評価にかかわらず大域変数Node にそのクロージャのアドレスを代入してから，info テーブルの 0 番目の標準エントリへジャンプする (この一連の操作を ENTER と呼ぶ)．評価済みのクロージャの標準エントリには，クロージャの中に保存されている評価結果を大域変数やスタックにコピーして即座に返すコードが，未評価のクロージャの標準

エントリには実際に評価を進めるためのコードが入っている.

STG は, G-machine のようにカリー化された関数適用を対応する二分木で表現せずに, flat な構造で表している (Spineless) ので, 関数定義を容易に得ることができる. また値と thunk を区別しないので, 特定の型の value を表すためのタグを必要としない (Tagless).

2.2 スタック

STG には A, B の 2 本のスタックがあり, A スタックは引数や実行結果の受渡しに, B スタックはコントロールと update のために用いる.

関数型言語では他言語におけるループを末尾再帰や末尾呼び出しの形で表現するが, これを他の C の関数を呼び出してその結果の値を return する形に変換すると再帰が深くなりスタックを大量に消費してしまう.STG では C の関数の末尾で他の C の関数に JUMP する形に変換することで, スタックの消費を少なくおさえている.

JUMP を C 言語で実現する方法はいくつかあって, これによって処理系の効率が変わってくる. 一般的に用いられるのは次に実行すべき関数のアドレス return 文で tiny interpreter と呼ばれる方法である.

`if (f x) then 1 else 2`の`f x`のような末尾呼び出し以外の関数呼び出しも`f x`の評価後に実行すべきコードのアドレスをあらかじめ用意したスタックに積んでから JUMP すれば, C 言語のスタックを用いるよりもスタック使用量が少なくてすむ. STG ではこの考えを更に進めて, ある型に属するコンストラクタの種類に応じた継続のアドレスの配列を B スタックに積む以下のような方法を用いている.

`if e1 then e2 else e3` において e1 は Bool 型だが, Bool 型は`False` と`True` の 2 種類のコンストラクタを持つので, B スタックにそれに対応する e3 と e2 への継続のアドレスの配列 (これはコンパイル時に作る) のポインタを積んでから e1 を評価する. e1 を評価すると, 最後に`False` か`True` のコードに入るが, これらの info テーブルの標準エントリには B スタックから配列のポインタをポップし, それぞれ第 0 要素, 第 1 要素に JUMP するコードが書かれているので, e1 の結果に応じた e2, e3 への分岐が実現される.

この「ベクターリターン」と呼ばれる技法により, if 文や case 文が効率的に実現できる. True や False は他に値を返す必要がないが, Int 型 Float 型はグローバル変数 RetInt, RetFloat に入れて値を返す. それ以外の値の返却は A スタックを通じておこなう. たとえば Cons の場合は head と tail の内容を A スタックに値を積んで値を返す.

B スタックは update にも用いられる. クロージャは, 複数のクロージャから参照されることがあるが, 計算は一度しか実行しない. 複数参照の可能性がある時は, そのクロージャに入った時に B スタックに update frame を push して自分自身を登録してから評価する. 評価した結果が WHNF になった時, B スタックの情報を元に対応するクロージャを update する. update もベクターリターンを利用して効率的に実現されている.

3 並列実行のための拡張

この節では, 遅延評価型関数型言語への並列性の導入と, それを STG にどのような拡張をほどこすことによって単一プロセッサ上に実現できるかを述べる.

遅延評価型関数型言語に並列性を導入する構文はいろいろ提案されているが, ここでは次のような組み込み関数spec による並列性の導入を扱う.

`spec f x = f x` (表示的意味)
x を評価するプロセスを `fork` し, 自身は `(f x)` を評価 (操作的意味)

x の評価が終わって x を値に update する前に, (f x) を評価するプロセスが x を必要として x を評価しにいくと, 同じものを 2 回評価することになって無駄な計算が生ずる. したがって, 未評価, 評価済みの他に評価中という状態も加える必要がある. 評価しにいったクロージャの状態が評価中の時は実行を中断して (サスペンド), 先に評価を始めたプロセスが update するのを待つ.

評価中という状態を表すために, suspend_self という info テーブルを用意し. 未評価のクロージャを評価して update frame を作る時には, 同時にクロージャの先頭ワードを suspend_self に書き換えるようにする. suspend_self のクロージャに ENTER すると, suspend_self の標準エントリのコードを実行するが, そこにはプロセスをサスペンドさせるコードが入っている.

suspend_self クロージャはそのクロージャを評価しにいってサスペンドしているプロセスのリストのポインタを含んでいて, クロージャの値が決まってそのクロージャを update する際に, そのリスト中に含まれるプロセスをアクティベイトする.

サスペンドしたプロセスのコンテクストの保存法も, 効率的な処理系を作る上では問題になる. 実行中のコンテクストはスタックに対応するが, スタックは大部分が使われないので最大コンテクスト数のスタックを用意するのはメモリの無駄であ

る．そこで，実行中でないコンテクストのスタックはクロージャの形でヒープ中に持つことにする．

プロセスの実体も図2のような形のクロージャでヒープ上に持つ．このようなクロージャをゴールと呼ぶ．アクティブなプロセスに対応するゴールは図2のように大域変数goalqueueから伸びるゴールキューにつなげられる．サスペンド中のプロセスに対応するゴールは，サスペンドする原因になったクロージャから伸びるサスペンドキューに同様の構造でつなげられる．

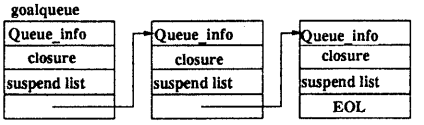

図 2: ゴールの構造

4 疎結合並列計算機上の実行

疎結合並列計算機上の処理系は単一プロセッサ上の疑似並列処理系に，プロセスの輸出入と，クロージャの移動機能を付け加えることによって実現される．その際の諸問題と解決法について以下に述べる．

4.1 リモートポインタの処理

遅延評価型関数型言語の実現では他のプロセッサ上のクロージャを指すポインタが必要となる．リモートポインタを直接プロセッサ番号とオブジェクトのヒープ上のアドレスで表現すると，そのプロセッサ上でローカルGCを行ないオブジェクトの場所を移動することができなくなる．

そのため，あるオブジェクトへのポインタを公開する時はヒープ上とは別のGCで動かない領域にテーブルを作り，そのオブジェクトを指すようにし，他のプロセッサにはそのテーブルのアドレスを渡す方法が他言語の実現において用いられている．このテーブルを輸出テーブルと言う．

一方，値を評価する必要が生じて，輸出テーブルの要素に対して値の要求を出した時に，値を受け取って代入する場所が必要となるが，この場所もローカルGCで動かない必要があるので，ヒープ外にテーブルを作る．この場所は値が必要になってから確保しても良いが，GCの都合でポインタの輸入時には必ず確保することにする．このテーブルを輸入テーブルと言う．

輸出テーブルの要素に値の要求のメッセージ(read_var)が届いた時に，輸出テーブルの指すオブジェクトが評価済みの場合は，輸入テーブルにそのオブジェクトのコピーを書き込むメッセージ(write_import)を送る．未評価の場合は，そのオブジェクトを評価するゴールを生成し，後でwrite_importを送るべき輸入テーブルのアドレスをリストにつなげて記録しておく．評価中の時は，輸入テーブルのアドレスをリストにつなぐだけで，ゴールを生成する必要がない．

輸出テーブルは次のような要素を含んでいる．

- 公開するオブジェクトのアドレス
- 輸入テーブルのアドレスのリストへのポインタ
- 重みつき参照カウント
- 次の使用中の輸出テーブルへのポインタ
- GC用マーク

STGとの整合性を考え，輸出入テーブルの要素もヒープ上のデータ同様，先頭にinfoテーブルへのポインタを持つクロージャの形にしている．そこで，以後は輸出入テーブルの要素一つを指すために輸出クロージャ，輸入クロージャという用語を用いる．輸出テーブルをヒープから指すポインタは，サスペンドリストからのものしかないが，輸入クロージャは普通のクロージャから指されることがある．

普通の未評価のクロージャ同様，輸入クロージャを評価しにいくことがあるが，その時の動作は輸入クロージャの状態によって異なる．たとえば，輸入クロージャは生成直後はReadVarというinfoが書いてあるが，その標準エントリには対応する輸出クロージャに対する参照要求を出してサスペンドし，infoをWaitVarに書き換えるコードが入っている．WaitVar状態の輸入クロージャを評価する場合はすでに参照要求が出ているので，単純にサスペンドする．

WaitVar状態にある輸入クロージャに対してwrite_importメッセージが届くと，そのメッセージ中に含まれる評価済みのクロージャをヒープ上に移動し，輸入クロージャのinfoをGroundImportに書き換えそのクロージャを指すようにし，サスペンドしていたプロセスをアクティベイトする．なお，GroundImport状態の輸入クロージャは間接ノード(proxy)と同じ働きをする．

4.2 クロージャの移動

プロセッサ間のクロージャの移動は次のような場合に必要になる．

1. プロセスの輸出入に伴う移動
   specによってあるクロージャを評価するプロセスを他のプロセッサ上に作る場合は，そのクロージャ自体も移動する必要がある．通信回数を減らすため，投機的にそのクロージャ

から指されるクロージャもある程度まで移動する.

2. リモート評価に伴う移動
   あるクロージャを評価しようとした時に, そのクロージャの実体が他のプロセッサ上にある場合は, 輸入クロージャを評価することになる. その輸入クロージャがReadVar状態にある時は, その実体に対応する輸出テーブルへ参照メッセージ (read_var) を送る. 輸出テーブル側で未評価の場合は評価用のゴールを生成し, 評価の終了時に評価済みのクロージャのコピーを返す. そのクロージャから指されるクロージャもある程度, 投機的に移動する.
3. プロセスの返り値
   spec f xによって, xを評価するプロセスが生成されるが, f xの評価に本当にxが必要かどうかはわからないので, xの値の移動は通常のリモート評価の機構を用いて必要が生じておこなってもよい. ただ, 本研究の処理系ではサスペンドの回数を減らすために, f xを実行したプロセッサに積極的に書き戻すようにしている. その際のクロージャ移動は2と同様である.

AP1000の各セルに同じプログラムテキストが配布されるので, テキスト領域中にあるinfoテーブルのアドレスはどのプロセッサでも等しい. この性質を利用して, クロージャの移動はinfoテーブルを含むメッセージを生成し, infoテーブル中に拡張されたExportエントリ, Importエントリという2つのエントリを用いて実現している.

クロージャを移動する際には, そのクロージャのアドレスとExportエントリが再帰的に何回呼ばれたを示すlevelを引数としてクロージャのinfoテーブルのExportエントリを呼び出す. Exportエントリの中には, そのクロージャを再配置可能な形に直して, 用意された送信用のバッファにコピーしていく関数へのポインタが入っている.

クロージャの投機的な移動を行なうために, そのクロージャの中から指されているクロージャのExportエントリを再帰的に呼ぶ場合があるが, この時は, 第2引数としてlevel+1を渡す. levelが適当な深さを超えたら, 投機的な移動は行なわずに, 輸出テーブルを確保してそのアドレスをinfoテーブル付きで送信用バッファにコピーする.

図3は, [100,200,300]を深さ2までは投機的にコピーする戦略のもとで, 送信用バッファにコピーした様子を図示している. コピーの際に間接ノード (indirect) などは除かれるが, これはindirectのinfoテーブルのExportエントリに自分をコピーする代わりに, ポインタの示すクロージャのExportエントリを呼び出すコードを書くことによって実現される. この中身を通信によって受け取ったプロセッサは, これを受信バッファからヒープ上に展開する必要がある. 受信バッファからクロージャごとにヒープ上へ移動すると, 途中でGCが起きることを考慮する必要があり, オーバヘッドが生ずるので, 受信バッファの大きさの分だけヒープを確保して, 受信バッファの内容をヒープ上にコピーしてから, ヒープ上のオブジェクトを正しいアドレスを指すように直すことにする.

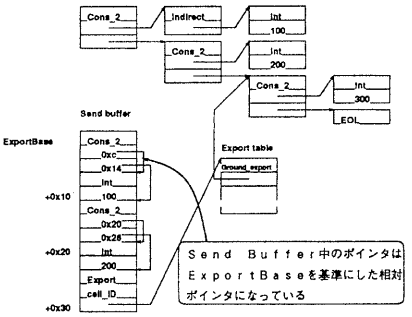

図 3: クロージャの輸出

ヒープ上のオブジェクトを正しいアドレスに直すために, コピーされた領域の先頭のワードからさされるinfoテーブルのImportエントリを領域のアドレスを引数として, 呼び出す. 図3のメッセージを移動した場合の様子は図4のようになっている.

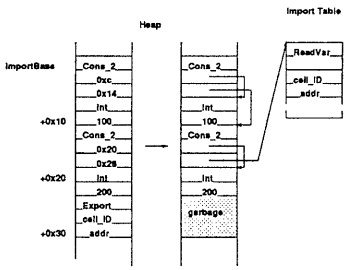

図 4: クロージャの輸入

5 評価

遅延評価型関数型言語 Goferの処理系に手を加えて, 内部で用いる中間形式からSTGにしたがって, C言語の目的コードを出すよう改造した. 本来Goferは対話的な実行が可能だが, Haskell同様にDialogue型の関数mainの評価によって実行を開始する非対話的な実行のみを扱う.

コンパイルして得られたCプログラムと, STGのランタイムライブラリ, AP1000用の入出力ラ

イブラリをリンクしてAP1000のセルプログラムが得られる．セルプログラムとセル上での入出力を肩代りするホストプログラムとによって実行する．

評価の対象としたのはベンチマークとしてよく用いられるn queensとfibonacciプログラムである(表1)．

表1: 逐次実行との比較(単位秒)

program	sequential	parallel	speedup
fib 30	36.4	0.74	49
queens 10	29.0	5.3	5.5

(AP1000 64 PE)

fibは，粒度を調節してfib 10以下は逐次実行をする．粒度を調節せずに最後まで実行すると却って時間がかかってしまう(111秒)が，適当に粒度の調節すると台数に見合う分の高速化が実現されることがわかる．

6 逐次実行の高速化

前節までで述べた並列処理系は，多くの密結合型並列計算機上の並列処理系の論文中での値と比べて勝っているが，これは主にハードウェア単体の能力によるものが多い．最近の高速な逐次処理系と比較すると同一の単一プロセッサ上での実行速度は大きく劣る．そこで，本節では逐次用の最適化とその効果について述べる．

6.1 1つのC関数にコンパイル

C言語上でJUMPを効率を落さずに実現する方法の1つとしてgoto文を使う方法がある．複数の関数を1つのC関数に入れるとそれらの関数の間の相互呼び出しはgoto文で実現できる．klic[9]では1つのモジュールに含まれる述語を1つのC関数にコンパイルしている．

Goferにはモジュールがないので，良い切り分け方を見つけるのが難しい．そこで，プログラム中で必要となるすべての関数の定義を1つのCの関数の中に入れる方法を取る．この方法では，Cの関数が巨大になりコンパイルのための時間とメモリを大量に消費する可能性があるので，main関数からの強連結成分を取り，不要なユーザ定義関数や組み込み関数は含めないなどのコード量を抑える努力が必要となる．また，型が違っても

```
type List a = Nil | Cons a (List a)
type Bool = False | True
```

の定義におけるNilとFalseの様にarityとnumberが等しいコンストラクタはコードを共有する．

すべての定義を1つの関数に入れることにより，レジスタに割り当て可能な変数の数が増え，値の受渡しにスタックではなくレジスタを積極的に使えるようになる．また，条件判断と呼び出しを一度におこなえるなどメリットが大きい．

STGの実行には間接ジャンプが必要になるが，GNU Cコンパイラでは，ラベルのアドレスを取る機能が用意されていて，ポインタを使ったgotoが実現できる．この機能を使うと移植性が制限されるが，この機能を使わないと効率が著しく低減するので使用する．

6.2 direct enter

STGでは，infoテーブルを標準エントリとは独立に持っている．これにより，同一の標準エントリを共有する別のinfoテーブルを作ることが可能になるが，クロージャ呼び出し際に2回メモリ参照が必要となり速度の低下につながる．

[4]の中では，標準エントリから負のオフセットで標準エントリ以外のエントリを参照するという形でinfoテーブルを共有することを提案しているが，標準的なC言語の枠組ではそのようなコードを生成することはできなかった．

本処理系では，標準エントリを1関数中のラベルとしているので，asm文を使ってテキスト領域にテーブルを作ることが可能になった．具体的には次のようなマクロを使ってinfoテーブルを定義する．

```
#define DefineEntry(info,entry,evac,scav,imp,exp)\
 entry:\
  asm (".word %0;.word %1;.word %2;.word %3" : :
    "i"(exp),"i"(imp),"i"(scav),"i"(evac));
```

複数のasm文で実現するとコンパイラが命令の入れ換えをおこなってしまうことがあるので，注意が必要となる．

6.3 組み込み関数のインライン展開

```
fib (n+2) = fib (n+1) + fib n
```

をそのまま，STGに変換するとfib (n+1)とfib nのクロージャを作ってから+のコードにジャンプする．+が2引数に関して必須であることを利用していないので，不要なクロージャを作ってしまっている．

そこで，このような組み込み関数はクロージャを作らずに直接fibを呼び出すような形にインライン展開する．これによって，作るクロージャの数が減るだけではなくupdateの回数も減る．組み込み関数のインライン展開は必須性解析と組み合わせると特に効力を発揮する．必須な変数が整数や浮動小数の時はあらかじめ評価しておけば，組み込み関数は即座に適用することが可能になる．

6.4 update機構の変更

STG は self update モデルを用いている．これは update する必要があるかどうかは呼び出し側が知っているという判断に基づくものである．実際，

```
case (f x) of ...
(f x) + ....
```

などの(f x) のように文脈から単一参照性が保証される場合は update の必要がない．両方の場合に対応するように f の定義を 2 種類用意するのは非効率的であるし，フラグ等をチェックして動作を変えるようにすると update しない場合の実行速度が落ちてしまうので，この判断は有効である．

STG では，ベクターリターンの機構に update 用のエントリをもうけるという方法で，呼び出し側が update をおこなう．しかし，この方法には，評価済クロージャの複製を作ることがあることと，評価側が型を知る必要があるという問題点がある．

並列評価では評価するクロージャの型が分からない場合があるので後者が問題になる．これまでは評価結果が残る可能性のあるスタックとレジスタをすべて含んだ型に依存しないクロージャを作ってこの問題を解決したが，ローカルレジスタを使う場合はこの方法では保存すべき情報が大きくなり問題である．

そこで，WHNF を返す際にレジスタにアップデート用の継続 (continuation) アドレスを入れてジャンプするようにする．アップデートフレームがはさまっている時は，その継続アドレスに飛ぶ．これにより通常のリターンは，即値をレジスタにロードする 2 命令分 (sethi, or) 遅くなるが，全体としてはほとんど影響がない．

6.5 境界検査の省略

関数型言語のように動的なメモリ管理をおこなう言語においては，領域の境界検出のオーバーヘッドは無視できない．Glasgow Haskell では，チェックの回数を減らすために関数のエントリの最初で一度だけ検査している．

このようなチェックのコストを省略するために仮想記憶のハードウェアを利用して，アクセスをするとトラップが生ずるようなページを境界部分に置く方法が古くから知られている．

SunOS では mmap を使ってこのようなページを容易に作ることができる．このページをアクセスすると Segmentation Violation の signal が発生するので，signal ハンドラで捕まえる．スタックオーバーフローの場合は，エラーメッセージを表示して終了するだけなのでよいが，ヒープのオーバーフローの場合は注意が必要となる．

ヒープがオーバーフローした時は GC を起こさなくてはいけないが，GC の際にローカルレジスタの一部もマークする必要がある．そのため，保存されたコンテクスト中のプログラムカウンタを変更して，元の関数中にある GC 用ラベルに変更する．真のリターンアドレスは大域変数に取っておいて，GC の最後でそこへ goto で戻る．

ページ切り離しは AP1000 でもハードウェア的には実現できるはずなので，OS に手を加えることも含めて調べてみることにする．

6.6 必須性解析による整数の先行評価

次のような簡単な必須性解析 (Strict Analysis) をおこなう．if 以外のコンビネータは略したが if に準ずる．|,& は集合における union と intersection をあらわす．

```
strict :: Exp -> [Var]
strict v -> [v]
strict (if e1 then e2 else e3) ->
  strict e1
strict (global_function e1 e2 .. en) ->
  (isStrict arg1 global_function) & (strict e1)
 |(isStrict arg2 global_function) & (strict e2)
  ..
 |(isStrict argn global_function) & (strict en)
strict (Construct e1 e2 .. en) -> []
strict _ -> []
```

以上の解析を最左最外順にほとこすと，停止するプログラムの解析では同じ関数に対して再帰的に適用することがない．再帰が生じた場合は，その関数が停止しない関数だということが検出できる．以上の解析は，1 つの式に関しては一度しかおこなわないので，プログラムの大きさに対して線形時間で終了する．

この解析は if に関して手を抜いたために，次のように解析能力が弱くなっている．

```
tak x y z =
  if(x>=y) then
    tak (tak (x-1) y z)(tak (y-1) z x)(tak (z-1) x y)
  else
    z
;    strict (if e1 then e2 else e3)
; -> strict (x>=y)
; -> [x,y] -- 実際には z に関しても strict
```

必須解析の結果として得られた必須な変数が Int の場合は先行評価をおこなう．fib の場合は，

```
fib 0 = 1
fib 1 = 1
fib (n+2) = fib (n+1) + fib n
->
fib =(strict_int (offset_1)
       (case (evaled_int offset_1)
```

表 2: 逐次実行の性能(SS/10, 単位秒)

	fib	nrev	queen	備考
	(30)	(3000)	(10)	備考
Utilisp	5.9	2.4	1.3	
klic	5.6	2.7	1.4	
hbc	4.2	9.4	1.0	
glhc	6.8	14.3	2.0	
gofstg	2.9	9.7	2.8	
gofstg	2.9	9.4	1.6	必須性解析
gofstg	2.5	8.1	1.5	境界検出省略
gofstg	12.3	19.4	13.2	旧版
gofc	32.1	117.0	21.7	

```
(0.1)(1.1)
((+2).
 (strict_int (offset_2)
  (inline_prim (+)
    (fib ((+) (evaled_int offset_2) 1))
    (fib (evaled_int offset_2)))))))
```

のようになり, (n+1) のクロージャを作る際に即座に計算をしてしまえるという利点がある.

6.7 最適化評価

今回の予稿には新処理系用の組み込み関数等のライブラリの書き直しが間に合わなかったので, AP1000 上での並列実行はできなかった. また, 必須性解析も実装が間に合わなかったので, ここに書いたアルゴリズムに従って手で解析し, その結果を機械的に適用した. 試したプログラムは, 整数演算を主におこなう fib と, リスト演算を主におこなう nrev と, その両方を扱う n-queen の 3 種のプログラムである.

図 2中で参考としてあげているのは, 先行評価を行なう言語として Lisp(UtiLisp) と KL/1(klic), 同じ関数型言語としては gofer の本来のコンパイラ gofc と STG モデルの本家の Glasgow Haskell(glhc), Chalmers の Haskell B(hbc) である. Haskell B 以外は C 言語のファイルを出力する形式を取っている.

遅延評価型関数型言語の処理系としては Chalmers の Lazy ML が名高いが, これとコード生成部をかなり共有している Haskell B コンパイラが, 遅延評価型関数型言語処理系の中では飛び抜けているが, 我々の処理系も C 言語を中間コードとしているにもかかわらず, かなり迫っている. また, lazy でない言語と比べると純粋にリストを扱う nrev で 3 倍以上の時間がかかっているが, これは構造をもったデータに対する必須性解析をおこなっていないためだと考えられる.

構造を持ったデータを扱うととたんに抽象領域が大きくなってしまって, 解析が困難になることが知られているが, 実用的な範囲である程度解析できるようにしていきたい.

7 おわりに

並列処理系の性能を向上させるにはまずは逐次部分の最適化が大切だとよく言われることだが, 遅延評価型の関数型言語の場合にもよく当てはまることを確認した.

今後は, 逐次実行で現在, 最速の遅延関数型処理系といわれる Haskell B を越えることを目標とすると共に, 静的解析を用いた並列実行の効率向上を更に検討していきたい.

参考文献

[1] Luc Maranget: GAML: a Parallel Implementation of Lazy ML. Proc FPCA'91, LNCS Vol. 523, Springer Verlag, pp. 102-103(1991).

[2] Peyton Jones, S., Clack, C. and Salkild, J.: High Performance Parallel Graph Reduction. Proc PARLE '89, LNCS Vol.365, Springer Verlag, pp. 193-206(1989).

[3] Augustsson, L. and Johnsson, T.: Parallel Graph Reduction with the $< \nu - G >$ machine. Proc FPCA'89, LNCS Vol. , Springer Verlag, pp.202-213(1989).

[4] Peyton Jones S.L., and Salkild J.: The Spineless Tagless G-machine. Proc FPCA'89, LNCS Vol. , Springer Verlag, pp.184-201(1989).

[5] SL Peyton Jones: The Implementation of Functional Programming Languages. Prentice-Hall International Series in Computer Science(1987).

[6] L Augustsson: A Compiler for Lazy ML, Proceedings of LFP 1984, pp.218-227(1984).

[7] J Fairbairn and S Wray: TIM – a simple lazy abstract machine to execute supercombinators, functional Programming Languages and Computer Architecture, LNCS 274, Springer Verlag(1987).

[8] Jones, M. P.: An Introduction to Gofer, University of Oxford(1991).

[9] 関田大吾, 近山隆: Portable な KL1 処理系の構想, 情報処理学会第 8 回プログラミング - 言語・基礎・実践 - 研究会 SWoPP'92, pp. 123-130(1992).

[10] 田中哲朗: 疎結合並列計算機用の関数型言語処理系, ソフトウェア科学会第 10 回大会論文集, pp. 329-332(1993).