4K-7 KL1のデバッグサポート機能

平野喜芳[1], 中越靖行[1], 宮崎芳枝[1], 西崎慎一郎[1], 木村康則[2]
1: (株) 富士通ソーシアルサイエンスラボラトリ
2: (財) 新世代コンピュータ技術開発機構

はじめに

KL1はFGHCをもとにICOTで開発された並列論理型言語である。このKL1のプログラミングスタイルの代表的なものに「プロセスとストリーム通信」がある。本論文ではこのスタイルを念頭においてKL1のデバッグサポート機能について考察するとともに、我々が開発したKL1の逐次処理系 - PDSS[1] 上で提供するデバッグサポート機能についても報告する。

なお、本研究は第5世代コンピュータプロジェクトの一環として行われたものである。

1 プロセスとストリーム通信

まず、「プロセス」という言葉について定義しなければならない。KL1における「プロセス」とは、再帰呼び出しにより実現されるある程度長いライフタイムを持つ計算過程のことである。以降、本論文では「プロセス」をこの意味で用いる。

これらのプロセスの間では、共有される変数を用いて通信が行われる。これは、通常、変数にリスト構造を具体化することによって実現されている。具体化されるリスト構造のCARにメッセージを書き、CDRには次の通信のための新しい変数を用意する。これを順次繰り返すことによりプロセス間の一連のメッセージ通信が実現される。このような共有変数とリスト構造を用いた通信を「ストリーム通信」と呼ぶ。

「プロセスとストリーム通信」のスタイルでは、複数のプロセスが御互いに通信を行いながら、処理を進めていく。各プロセスは自分の状態を持っており、処理の進行に従って、状態を変化させていく。これは、オブジェクト指向風のスタイルと見すこともでき、プログラムのモジュラリティを高め、大規模なプログラム開発に有効なプログラミングスタイルと言うことができる。

このストリーム変数やプロセスの状態は再帰呼び出しの各サイクルに渡って引数により保持される。従って、このプログラミングスタイルでは引数の数が多くなる傾向がある。

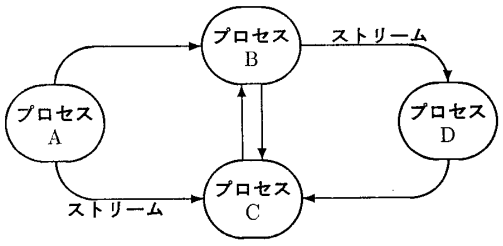

図1: プロセスとストリーム通信

2 KL1プログラムで発生し易いバグ

ここではKL1プログラムで発生し易いバグとそれによって発生し得る状態について考えてみる。

2.1 述語名等の間違い

KL1コンパイラでは述語名やモジュール名は、ソースプログラムに書いてあるものは全て正しいとしてコンパイルする。その述語やモジュールが存在するか、しないかは調べられない。そのため、タイプミス等で間違った名前を書いても、その通りにコンパイルされエラーにならない。これらの間違いは実行時に初めてエラーになる。

2.2 変数名の間違い

KL1では変数を使うために特別な宣言は必要ない。使いたい所にただ書くだけでよい。そのため、変数名を間違っても、本当に間違えたのか、意識的にそう書いたのか区別できない。

タイプミスの場合には、本来同じ変数になるべきものが別々の変数になってしまうので、変数の一方が具体化されても別の側は具体化されないままになる。その変数の具体化を待っているゴールがあれば、そのゴールはデッドロックすることになる。

同じ変数名を違う用途に使ってしまった場合には、1つの変数を別々の値に具体化しようとするので、一方の具体化が失敗することになる。この具体化の失敗は、変数名を間違えたところとは別のところで発生する場合が多いので、原因を見つけるのが難しくなる。

2.3 引数の順番の間違い

この場合、ストリームの接続先が違ってしまったり、入出力が合わなくなる。前者では、通信相手のプロセスが期待しないメッセージ受け取ってしまうことになり、そのプロセスが失敗することが多い。後者では、入力どうしだとデッドロックになり、出力どうしだと具体化が失敗する。

2.4 ',' と '.' の間違い

クローズのなかの述語の区切りの ',' (カンマ) を '.' (ピリオド) に間違う。この場合、クローズが2つに分かれてしまうが、KL1ではどちらもコンパイル可能で、エラーにならないことが多い。

プロセスは再帰呼び出しで実現されており、その再帰の記述はクローズの最後に書かれるのが普通である。クローズが2つに分かれてしまうと、再帰の部分がなくなり、プロセスが終了してしまう。

2.5 実行順序に依存したプログラム

KL1は並列言語であり、実行可能なゴールに関しては、それらをどのような順番で実行しても構わないことになっている。実際の実行順序はそのプログラムを実行している処理系のスケジューラに依存しており、実行環境によっても変わる可能性がある。そのため、特定の処理系 / 環境では正しく動いたものが、他の処理系 / 環境で動かそうとしたときに、全く動かなくなる場合がある。

このバグは再現しようとしても実行環境が変わってしまっている場合があり、デバッグは難しい。

3 デバッグサポート機能

PDSSでは、以上のようなバグを効率良く見つける為に、次のようなデバッグサポート機能を用意した。

3.1 トレーサ

トレーサは最も一般的なデバッグツールである。

Debugging support tools for KL1.
Kiyoshi HIRANO[1], Yasuyuki NAKAGOSHI[1], Yoshie MIYAZAKI[1], Shin'ichirou NISHIZAKI[1], Yasunori KIMURA[2]
1: Fujitsu Social Science Laboratory Ltd. 2: Institute for New Generation Computer Technology (ICOT)

トレーサでは通常トレースするべき範囲を何らかの方法で絞り込む必要がある。 Prologのトレーサには 特定の述語にトレースポイントを設定する機能がある。 KL1 の場合には、並列言語である為、このような述語オリエンテッドなトレースだけだと不十分である。トレースされる順序が、ユーザの期待通りになるとは限らないので、同じ名前の述語が並んで出てトレースされたとしても、それらの間にどのような関係があるか分からない。再帰的に呼ばれたのかもしれないし、全く別のところから呼ばれたのかもしれない。

「プロセスとストリーム通信」のスタイルで考えた場合、次の2つの事象をトレースできる機能が必要である。

3.1.1 プロセスオリエンテッドなトレース

プロセスは再帰呼び出しにより実現されているので、あるゴールから呼び出された子供ゴールだけをトレースする機能により実現できる。子供として呼び出すものにはサブルーチンもあるので、あるゴールの子供で、且つ、指定された述語名のゴールだけをトレースする機能があるとさらに便利である。

3.1.2 変数のモニタリング

変数が具体化されるタイミングで、その値をモニタリングする機能。変数がストリームの場合はそこを順次流れるメッセージがモニタリングできる。これにより、ストリームの接続がおかしい場合などは、予想外のデータが具体化されるのが分かる。また、途中のストリームが切れている場合には期待されるメッセージが来ないことでその状態が分かる。

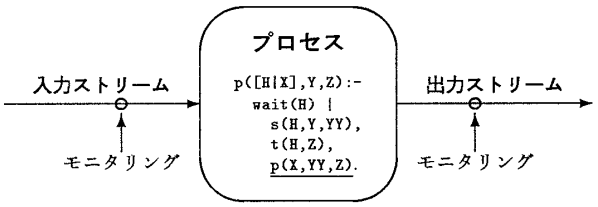

図 2: プロセスのトレース

3.2 デッドロック検出機能

KL1 におけるデッドロックは次のように定義出来る。あるゴールが変数に値が具体化されるのを待っているにもかかわらず、その変数が永久に具体化されない場合、このゴールはデッドロックしたと言う。これは変数名を間違えた等の理由で、書き込み側と読み出し側で同一の変数を参照すべきところが、別々の変数になっていた場合に起こる。

このようなデッドロックのために以下の2通りの検出方法を用意している。

3.2.1 一括GC時の検出

一括GCにおいてゴミとして棄てられてしまう変数は永久に具体化されない変数と言うことができる。もし、これらの変数の具体化を待っているゴールが存在した場合には、それらはデッドロックしたと言える。このようなゴールを検出するのが一括GC時のデッドロック検出[2]である。

しかし、この方式では一旦通常の処理を止めなければならないし、マルチプロセッサの場合には一括GC時を行うためのコストが非常に高いために、気軽に一括GCを行えないという欠点がある。

また、デッドロックの原因となる処理を行ってから、デッドロックが検出される(一括GCが起動される)までにある程度の時間がかかり、他の多くのゴールが実行される。 1つのゴールがデッドロックすると、そのゴールを待っていたゴールも芋蔓式にデッドロックしてしまうため、検出された時には非常に多くのゴールが同時に報告されることになる。このため、原因を調べるのが困難となる場合が多い。

この場合の対策として、芋蔓式にデッドロックしたゴールの依存関係を解析して、どのゴールが元となったかを調べるツールが必要となる。

3.2.2 実行時の検出

もう一つの方式は、実時間GCのように、通常の処理を行うのと同時にデッドロックも検出しようとするものである。これは、未定義変数への参照数、正確には未定義変数へ書き込む可能性のある参照数を管理することで実現できる。書き込む可能性のある参照数とは、参照数から、その変数の具体化を待っているゴールからの参照数を除いたもので、これが0になるとその変数は永久に具体化されない。この方式だと通常の処理を止める必要もないし、早い時期にデッドロックの発生を検出することができる。

3.3 プログラムアナライザ

プログラムアナライザとは、静的にプログラムを解析して初歩的なバグを見つける為のツールである。

現在のKL1コンパイラはエラーのチェックをほとんど行わない。また、コンパイラで多くのチェックをするようにすると、コンパイラの負担が増え、遅くなる。正しいことが分かっているプログラムをコンパイルする時もチェックが働くのは無駄である。そのため、チェックの部分だけを抜き出して、プログラムアナライザの形で使えるようになっているのが便利である。

3.3.1 変数チェッカ

これは変数の出現回数を調べるもので、1つのクローズ内に1回しか現れない変数は綴りミスの可能性が多いことを利用している。新しいプログラムはまずこれでチェックすることが望ましい。

3.3.2 クロスリファレンス

これは、述語の定義/参照の関係を調べるもので、未定義の述語を呼び出しているところや、どこからも呼ばれない述語定義を検出することができる。モジュール間呼び出しの場合もあるので、複数のモジュールに渡って定義/参照の関係を調べる必要がある。システムで提供されるモジュールを利用している部分もその述語が本当に提供されているかを調べる必要がある。

4 KL1 のシンタックスの改善

これは、バグが発生してからの対応でなく、積極的に、バグが入りづらいシンタックスを提供しようとするものである。

KL1 のシンタックスはかなり自由な記述が許される反面、少しぐらいの間違いは、間違いとして認識されず、そのままコンパイルされてしまう。また、「プロセスとストリーム通信」のスタイルではどうしても引数の数が多くなり、タイプミス等が起こり易くなる。

そこで、マクロ機能等を提供することにより、プログラマの負担を減らし、不注意によるバグを減らそうとするのがこの考えである。これは、デバッグサポート機能とは言えないかも知れないが、 KL1 を使い易い言語とする為には、是非とも必要な機能である。

現在、マクロとして次のような Implicit Argument の機能が提供されている。

```
:- implicit input:stream, output:stream, tty:stream.
io([write(X)   |L]) --> true | &output <<= [write(X)], io(L).
io([ttywrite(X)|L]) --> true | &tty    <<= [write(X)], io(L).
io([read(X)    |L]) --> true | &input  <<= [read(X)], io(L).
io([ttyread(X) |L]) --> true | &tty    <<= [read(X)], io(L).
io([]) --> true | true.
```

リスト 1: マクロを利用した記述の例

謝辞

日頃御指導頂いている ICOT 第4研究室の方々、デバッグツールの作成に関し数々の貴重な助言を頂いた宮崎敏彦氏(沖電気)に感謝します。

参考文献

[1] PDSS — 言語仕様と使用手引 — ,ICOT TM-437,1988

[2] 平野, 宮崎他: 汎用計算機上の KL1 処理系におけるメモリ管理とデッドロック検出, 情報処理学会第 36 回全国大会 7H-5,1988