プログラミング 17−2
(1995. 1. 23)

フロー解析に基づく意味的エラー検出方法の研究

宮坂 浩司* 渡辺 坦*

概要

構文規則に適合するソースプログラムに含まれている論理エラー等は、構文解析系で検出することが不可能である。これらを意味的エラーとして扱い、C/C++コンパイラにおいて静的に検出する処理系の実装を試みた。その全てを静的に検出することは不可能であるため、我々の経験則により検出対象を限定した。この処理系では、本来オブジェクト最適化に用いる情報であるデータフロー情報、コントロールフロー情報及び設定使用連鎖に着目し、エラー検出を行う。また、検出はTIL(Tree Intermediate Language)と呼ぶ特定の言語に依存しない中間語上で行っている。この処理系により、本来人間がデバッガ等を用いて実行時の情報を検証しながら検出していたバグや、テスト時の実行パス上にない潜在バグの一部が、コンパイル時に検出可能となった。

A system of detecting semantic errors by flow analysis

Koji Miyasaka*, Tan Watanabe*

Abstract

There are many program errors which a parser can not detect. They are syntactically correct,but semantically erroneous. In this paper we report a compiler system which partially detects such errors by using flow analysis techniques. From our experience,we limited a set of error patterns to be detected. We use data flow,control flow information and du-chain which a code optimizer usually uses. So far,programmers spent a lot of time in detecting such errors by using tools such as debugger at execution time. But if errors are not on executed paths,programmers can not detect them. Our system can detect some of them at compile time.

1 はじめに

プログラマはプログラムミング時に数多くのエラーに遭遇する。構文規則に沿っていない記述、すなわち構文誤りはかなりの割合を占めるが、通常のコンパイラでは検出可能であるため、比較的対処が容易である。その他には論理的誤りやポインタに関するエラーといったものがある。その性質は、構文規則に乗っ取って書かれたプログラムに含まれていることである。よって実行時に思いもよらぬ振舞から初めて気付くことが多く、またテスト時に実行されないパス上に潜在的エラーとして残っているものは、訂正に至る以前にエラーの特定が困難である。その検出のため、プログラマが多くの時間をかけソースコードを追跡したり、またはデバッガ等を用いて実行時の情報を検証するなど、高コストの作業を要する。さらにデバッガでも、実行されないパス上にあるエラーを検出することはできないという欠点がある。

本研究では、構文解析において検出できないエラーを意味的エラーとして扱い、それらをコンパイル時に検出する処理系の実装を目的としている。コンパイラによって静時に検出することにより、実行時の意図しない振舞を減少させ、またテスト時の実行パスにないエラーを事前に知ることが可能となる。

この処理系を実装することで、現在我々の研究室で開発中であるコンパイラシステム [1] の一つの特徴とする。

2 基盤となるコンパイラ

我々の研究室において研究中であるコンパイラは、以下の特徴を持つ。

*電気通信大学 情報工学科

- 言語と機種への依存性を隠したオブジェクト最適化向き中間語を持つ.
- 各モジュールをビルディングブロック方式で結合可能としている.
- 複数の対象機種に対して容易に実装可能である.

特定の言語、機種に依存しない中間語として、木構造中間語 TIL(Tree Intermediate Language) 及び、抽象レジスタマシン語 ParmCode(Parallel Abstract Register Machine Code) を設定しており、これらを各モジュール間共通のインターフェースとする。TIL は手続き型言語の要素的機能を抽象化した二分木の構造を持ち、ソースプログラムの論理構造を表現している。そのため、インライン展開等のソースプログラムに即した処理に適している。また、ParmCode は実行命令列を表現し、実際のマシンコードに近い構造を持っている。この二つの中間語とともに、記号情報を保持する記号表を持つ。コンパイルされるソースプログラムは、コンパイラ内部において中間語及び記号表によって表現される。

本研究ではソースプログラムに即した情報が必要であるため、その論理構造が隠蔽された ParmCode 上ではなく、TIL 上でエラー検出を行う。現時点で C 及び C++[3] の構文解析系を用い実験を行っているが、特定の言語に依存しない中間語上で処理を行っているので、エラー検出の処理が現行の他の手続き型言語に容易に対応できることが期待される。

3 検出対象

プログラミング時に発生するエラーの種類は膨大な数にのぼり、その全てを検出することは困難である。そこで本研究では筆者の経験上悩まされたものに限定し、検出の対象とした。以下の意味的エラーの一部を検出し適当な警告を出す処理系を実装した。

- 未設定変数の利用、特に未設定ポインタの利用.
- NULL ポインタのポイント先参照.
- 永久ループ、永久再帰の検出.

これらのエラーには、不適当な値やメモリにアクセスし異常終了する、プログラムの実行が終わらない、等の形で何度も遭遇する。また訂正を行うにしても、意図しない実行をするプログラムの外見上の動作はすぐに把握できるが、どの部分を訂正すれば良いのかわからない場合が多い。しかもこれらのエラーを含んだプログラムは、構文規則に沿って記述されているという性質を持つため、構文解析系での発見は不可能であり、その箇所を特定することもできない。

検出対象を上記に限定し実装を行ったが、その全てのケースを実行時の情報無しに検出することは不可能であるため、実際の検出対象はさらに範囲の狭いものとなっている。

4 フロー情報

対象となる意味的エラー検出には、フロー情報を用いる。フロー情報は本来、オブジェクト最適化に用いられる。ここでフロー情報に着目する理由は、検出の際プログラムの静的情報をできる限り把握するためである。中間語、記号表においてソースプログラムの実体は把握できるが、その制御構造、データの流れ等を知ることはできない。そのため、フロー解析 [2] によってソースプログラムのより詳しい情報を得る。

フロー解析によって収集される情報は、プログラムの制御フロー情報、基本ブロック単位のデータフロー情報 (In,Out 等)、各データの設定使用連鎖である。フロー解析は静的解析であるため、実行時の動的情報を知ることはできない。しかし、プログラムの全てのパスについて解析を行うため、テスト時に実行されないパスの情報も知ることができるという利点がある。

検出にはソースプログラムに即した情報が必要であるため、TIL 上でフロー解析を行い、サブツリーに対応させて設定使用地点、設定式、使用式を表す情報を集める。

5 意味的エラー検出の方法

TIL、記号表及びフロー情報を用いたエラー検出の方法を述べる。先に挙げた検出対象は、2つの性質に分類することができる。すなわち変数使用状況の分析および永久ループの検出である。同じ性質のエラーに関しては処理系を共通化することが可能である。また永久ループの検出では変数使用状況の分析を用い、その処理をほぼ包括した形になっている。

ここで変数の宣言、設定、使用という用語の使い

方を明らかにしておく。変数において、

- 宣言とは変数の領域が確保された状態.
- 設定とは変数に値が代入される状態 (define).
- 使用とは変数の値が評価される状態 (use).

をそれぞれ示す。文献等 [2] では設定を定義と呼ぶが、宣言と混同しやすいのでここでは設定として扱う。

表 1: 変数の宣言、設定 (define)、使用 (use) の例

int x,y	変数 x、変数 y の宣言
x = 1;	変数 x の設定
y = x;	変数 x の使用、変数 y の設定
func(x);	変数 x の使用
if (x == 10)	変数 x の使用

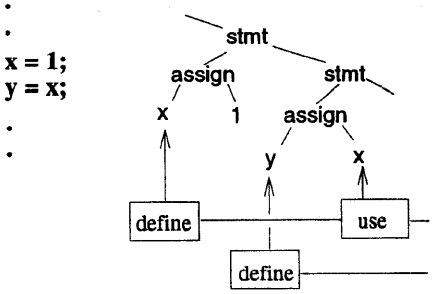

図 1: TIL によるソース表現及び使用定義連鎖

5.1 未設定変数使用、未設定ポインタ参照

未設定変数の使用とは、変数に値が設定される以前に変数が使用される場合を指す。特に一時的な auto 型の局所変数は、デフォルト値が設定されないため、その後の処理に悪影響を与える可能性がある。

そこで検出に際し、本稿で未設定変数を以下のように定義する。

- 未設定変数とは未設定を表す値が設定された変数である.
- 未設定の設定は、変数が宣言された副プログラムの先頭で一度だけ行われる (大域変数ならば main 内でデフォルト値を設定).

上記の未設定変数に対する定義は以下のような利点を持つ。

- 未設定という状態に実体を与えることにより、TIL で表現できる.
- フロー解析系の変更無しに、未設定変数の到達定義を調べることができる.

各副プログラムの中で宣言された全ての変数に、ダミーとして宣言しておいた未設定を表す変数を設定する。ここで TIL の仕様に従い、未設定変数の設定を表すサブツリーを TIL に挿入する。ただし仮引数は値が必ず設定されているものとする。フロー解析系は、未設定の設定式がデータとしてパス上をどのように流れるかを解析し、フロー情報及び TIL より未設定変数の使用を検出する。

- 変数 x の使用地点の到達定義に x の未設定式が含まれていない場合.
  - 変数 x は確実に設定されている.
- 変数 x の使用地点の到達定義に x の未設定式が含まれている場合.
  - 未設定式以外の設定式が含まれていなければ、変数 x は設定されていないことが確実である.
  - 未設定式以外の設定式が含まれていれば、変数 x は設定されていない可能性がある.

フロー解析はプログラムの全てのパス上を捜査するため、実際に実行されるパス以外の設定が到達定義に含まれている。従って、使用地点での到達定義に複数の設定式が含まれている場合、コンパイル時には可能性を示唆するに留まる。未設定ポインタに関しても同じアルゴリズムを適応させることができる。

この段階で、宣言のみで設定、使用の無い変数の検出も行う。

5.2 NULL ポインタ参照

NULL ポインタ参照の検出に関しても、未設定変数使用の検出とほぼ同じアプローチをとることができる。NULL の設定と未設定の大きな違いは、NULL 設定はソースプログラム上に実体として存在すること、未設定の設定は一つしか無いのに対し、NULL 設定は複数存在することにある。未設定検出は一つの変数について一つの未設定式に着目すれば充分であったが、NULL ポインタ参照については、全ての NULL 設定式に着目する必要がある。

- ポインタ p の使用地点の到達定義に NULL 設定式が含まれていない場合
  - ポインタ p は確実に設定されている.
- ポインタ p の使用地点の到達定義に NULL 設定式が含まれている場合
  - NULL 以外の設定式が含まれていなければ、ポインタ p は確実に NULL ポインタである (ただし NULL 設定式は複数存在する).
  - NULL 以外の設定式が含まれていれば、ポインタ p は NULL ポインタである可能性がある.

ただし、未設定ポインタの検出も行われるため、到達定義が未設定式を含んでいれば未設定ポインタである可能性がある。

この検出においても、実際は実行時に NULL ポインタを参照しない場合の検出を行ってしまうことになる。

変数、ポインタの未設定検出は永久ループの検出時に用いられる。

5.3 while 永久ループ

while 永久ループの検出には、ループの判定に着目する。while ループの判定が変数で行われている場合にはその変数を、判定が式で行われている場合には式に現われる (複数の) 変数を判定変数とする。ただし while(1) のように定数 1 が現われた場合はループ内の break 文による。(0 では、ループは行われない) この判定変数に関して while ループ内の到達定義、設定地点を調べる。ここで正常に終了する while ループと、判定変数の定義を以下のように定める。

- 判定変数はループ内で更新される。これを再設定と呼ぶ.
- while ループは判定式または break 文によって終了する.

このような定義に沿い、以下のようにして永久 while ループを検出する。

ループ内で判定変数の再設定が無いループは、確実に永久ループであることがわかる。さらに再設定された判定変数は、ループの判定ブロックに到達する必要がある。以上の事項から while 永久ループの検出が可能となる。

- ループ内で判定変数の再設定が無い場合、確実に永久ループである.
- ループ内で判定変数の再設定がある場合
  - ループ最終基本ブロックの最終地点に判定変数の再設定式のみが到達していれば、while は終了するとみなす.
  - ループ最終基本ブロックの最終地点に判定変数の再設定式以外が到達していれば、while は永久ループの可能性がある.(ただし正常に終了する while ループにも, この条件にあてはまるものが多い)
- 判定式が定数 1 の場合、ループ内に break 文があれば while は終了する可能性がある.

ループ最終ブロックの最終地点に着目するのは、ループ先頭の基本ブロックにおける到達定義には判定変数の初期設定式等が含まれているためである。ループ最終ブロックの最終地点に判定変数のループ内での再設定式しか無ければ、ループ内の全てのパスにおいて確実に判定変数の再設定が行われていると言うことができる。

5.4 永久再帰

永久再帰の検出も、while 永久ループの検出とほぼ同じアプローチをとる。while 永久ループ検出は判定変数、ループ最終ブロックに着目していたが、永久再帰検出でも、再帰呼出しの引数、再帰呼出しを含むブロックはほぼ同じ位置づけとなる。ただし、再帰呼出し地点はブロック中にもあり得るので、その地点での到達定義を解析する必要がある。(本システムでは while の判定節は常に基本ブロックの先頭である。) ここで、正常に終了する再帰呼出しを以下のように定める。

- 再帰呼出しの引数は再帰呼出しされる前に再帰関数内で確実に設定、再設定される.
- 再帰関数内の先頭ブロックから再帰呼出し地点を通らない return を含むブロックへのパスが存在する.

このように再帰呼出しを定めることにより、永久再帰を検出することができる。

- 再帰関数内の先頭ブロックから、再帰呼出し地点を通らないreturnを含むブロックへのパスが無ければ、確実に永久再帰である.
- returnへのパスが存在する場合、
  - 再帰呼出しの引数が再帰呼出し地点で確実に設定(再設定)されていなければ、永久再帰する可能性がある.
  - 再帰呼出しの引数が再帰呼出し地点で確実に設定(再設定)されていれば、永久再帰ではない.

上記の検出方法によると、たとえ引数が設定されていても、実行時の即値に変化が無い場合の永久再帰は検出不可能である。

6 エラー検出系による検出例

ここで実装した処理系を用いての、エラーを含んだソースプログラムの検出例を示す。

(1)未定義変数の使用

```
x = _UndefInt;   ----設定1
...
if ( a > b ) {
  x = func(a);   ----設定2
  ...
  }
...
y = x;           ----使用1
...
```

上記のプログラムにおいてif節以前に変数xの値が設定されていないとする。実行はif条件(a > b)によって分岐し、その結果xに値が設定されるパス、設定されないパスが生じる。テスト時に変数a,bの値によって常にxが設定されるパスを実行していれば、未設定変数使用は潜在バグとして残り得る。使用1においてx設定式の到達定義を見ることにより、設定1設定2ともに到達していることがわかり、実行時のパスによってはxの使用時にxが未設定である可能性を警告する。

Cの静的エラー検出系lint[1][4]では型チェックの他、未設定変数等の一部を検出するが、分岐を含んだこの例での検出は行わない。

[1]型チェック等を厳密に行うCの静的エラー検出系

```
MSG 0 Warning. 'filename' line . before :
variable 'x' is probably undefined-used.
```

(2)while永久ループ

```
while (sum < Required) {
  if ((a[i] > 0) && (a[i] < 10))
    sum = sum + (a[i] * 100 );
  else if ((a[i] > 10) && (a[i] < 100))
    sum = sum + (a[i] * 10 );
  else if ((a[i] > 100) && (a[i] < 1000))
    sum = sum + a[i];
  else if ((a[i] > 1000) && (a[i] < 10000))
    sum = sum + (a[i] / 10 );
  i++;
}
```

上記は、個々に異なる単位で設定されているa[i]を補正し、ある要求された値になるまで合計を求めるプログラムである。ループ内のif elseではa[i]の値によって変数sumの設定がされない可能性がある。つまりa[i]が0、10、100、1000及び10000以上の場合であり、この時sumはループ内で加算されず、永久ループに陥る。(ただしa[i]のインデックス領域逸脱により実行は停止し、a[i]の値がif-else条件をすり抜けない場合、ループは正常に終了する)この時ループの最終ブロックにおいてsumの到達定義より、永久ループを警告する。

```
MSG 0 Warning. 'filename' line . before :
while loop does not end probably in some paths.
```

(3)永久再帰

```
int p(int x,int n)
{
  int a;
  if (n == 0)
    a = 1;
  else
    n--;
    a = x * p(x , n);
  return(a);
}
```

上記は正の変数xのn乗を求めることを意図している。しかしelse節がブロックになっておらず、どのパスでも再帰呼出しがなされる。また、再帰呼出しの際の引数x,nともに更新されないパスが存在する。

(n の値が 0 の場合) よって、この再帰関数は完全に永久再帰であることがわかる。

```
MSG 0 Warning. 'filename' line . before :
recursive call can't return difinitly.
MSG 0 Warning. 'filename' line . before :
parameter'x' of recursive call isn't renewed
```

7 考察

本稿で述べた検出方法により、意味的エラーの一部を静的に検出できることが分った。現時点では複数ファイルにわたる制御構造や、複雑なデータ構造を持ったプログラムのエラー検出は行うことができないため、サンプルのプログラムに制限がある。しかし異常終了、永久実行するプログラム、実行パスに含まれないエラーをコンパイル時に検出可能とした。制御が複雑な場合は、実行テストを何度繰り返しても実行されないパスが生じてしまうことがあり、そこに意味的エラーが潜在的に残ってしまうことがあり得る。このようなエラーはデバグの対象とならず検出自体が困難であるため、静的検出の利点が発揮される。

本処理系を用いてのプログラム開発時間のサンプル抽出は行っておらず、コストの削減を数値的に示すことはできないが、有用性は確かであると思われる。

また本処理系と同じく静的エラー検出系である lint は、未設定変数の一部を検出するが、フロー情報を用いた検出を行っていないと思われ、分岐等を含んだプログラムでは警告を行えない。

しかし検出に際し、実際にはエラーではないコードを検出するケース、逆に実際にはエラーであるコードを検出しないケースがある。また確実にエラーであるか否か判定できない場合は、その可能性を示すに留まる。原因として、コンパイル時にはプログラムの動的情報、すなわち実行時のパスや設定された値を得ることができないことが挙げられる。これは静的検出の弱点であり、同じように静的検出を行う lint においても、実際の実行結果に反する警告がしばしば出される。これではかえって警告が混乱を招き得るので、常に検出を行うのでなく、オプションにより必要時に起動させたい。加えて実行時における情報を解析するエラー検出系の提案が望まれる。静的と動的両側面からエラーを検出することにより、より確実なエラー検出が可能となることが予測される。

また上記に関連して以下の問題点をもつ。

- 変数の別名を見ていない.

変数の影響波及分析を行っていないため、未設定である変数の値が他の変数に伝播された場合、変数の未設定等を検出できない。関数の引数に関しても呼出し時の引数が変数等の場合、値の伝播が行われている。しかし、現段階では引数には必ず値が設定されていることを前提としている。

- 外部リンケージ変数のデータフロー解析が行われていない.

外部リンケージ変数のような複数のファイル間にまたがるデータに関してフロー解析が行われていないため、未設定等を検出できない。

8 おわりに

本稿において構文誤りではないエラーを静的に検出する方法を述べ、処理系の実装を行った。検出対象であるエラーはその性質上、実行時に発見される可能性が高く、また実行頻度の低いパスに残り得るので、静的エラー検出の利点があるといえる。現時点で検出可能なエラーは一部に限定されているが、今後ターゲットを広げ問題点を解決していき、動的エラー検出系と組み合わせることで実用レベルに達することが期待される。

最適化コンパイラの目的は本来、ソースプログラムを効率の良いコードに変換することである。しかし本検出系をコンパイラに組み込むことによって、プログラミングコストを低下でき、ユーザーにやさしいコンパイラを提案できる

謝辞

本研究にあたり惜しみない協力をして下さった、電気通信大学情報工学科の鈴木貢助手に、厚く感謝致します。

参考文献

[1] 渡辺 坦 鈴木 貢: コンパイラ・エンジニアリング, 情報処理学会第 *38* 回プログラミングシンポジウム報告,*pp.135-142(1997.1)*.

[2] Alfred V Aho,Ravi Sethi,Jeffrey D Ullman: *Compilers-Principles Techniques andTools, Addison-Wesley Publishing Company (1986)*.

[3] Stanley B Lippman: *Inside The C++ Object Mmodel, Addison-Wesley Publishing Company (1996)*.

[4] 平林 雅英: *ANSI C/C++*辞典, 共立出版株式会社 *(1996)*.