インターコミュニケーション・プログラミング

「機能と構成」領域　　関口　龍郎

要　　旨

この研究ではインターネット上での人と人との双方向コミュニケーションを支援するソフトウェアを、安全で効率的に記述できるプログラミング言語システムの構築を目指した。

1．研究のねらい

アーキテクチャやOSに依存しないコード表現によってプログラムを配布しようとする発想は古くからある。OSをバイナリではなく高級言語のソースコードによって配布しはじめた1980年代のUNIXには間違えなくそのような思想が見られる。PascalのPコード、BASICの中間言語なども機種互換性を持つコード表現であった。"Write once, run anywhere" を歌うJavaでは設計の当初からアーキテクチャ非依存のコード表現によるプログラムの配布が考慮されていた。そしてその発想はマイクロソフト社の.NETにも受け継がれている。マイクロソフト社は.NETの仮想機械をIA32以外のアーキテクチャ上でも動作させ、将来的にはすべてのソフトウェアを.NETのコード表現により配布すると発表している。

またプログラミング言語の研究分野において、アーキテクチャ非依存のコード表現によるプログラムの配布を第一の目的としたものではないが、そのような目的にも流用できる研究として、多言語、多種のアーキテクチャに対応するコンパイラバックエンドの研究がある。そのような研究の中でもっとも有名なのがおそらくGNU Compiler CollectionのRTL（Register Transfer Language）であろう。他にも最適化のための共通のプラットフォームとして開発されたSUIF、普遍的なアセンブラであるC--、機種非依存のモーバイル言語であるOmnicode、高速な機械語生成を目的として設計されたvcode、Standard MLのバックエンドコンパイラであるMLRISC、JavaやSmalltalkなどのオブジェクト指向言語のための共通のバックエンドコンパイラであるVortexなどがある。

この研究はJava仮想機械や.NETなどのような言語非依存、アーキテクチャ非依存の普遍的なコード表現の開発とその実行環境の実装を目指している。

普遍的なコード表現の開発での最大の問題は実行効率と安全性の両立である。安全性とは

非常に広い意味を持つ概念であるが、ここで言う安全性とはあくまでもプログラミング言語が保証できる安全性を意味している。形式的にはプログラムがきちんと定義されている仕様通りに動作し、未定義な振る舞いを起こさないことである。したがってバッファオーバランを利用した攻撃を防ぐことは保証の対象となるが、通信プロトコルのバグを利用した「なりすまし」などを防ぐことは我々は考えていない。

実行効率と安全性のどちらか一方を実現するだけなら既存の様々な手法がある。例えば実行効率のみを重視するなら、カリカリに最適化したC言語のプログラムを利用すればよい。しかしもし悪意を持つ攻撃者がいればありとあらゆる危険な操作をプログラムに行わせることができるだろう。一方、安全性のみを重視するなら、Javaのようにプログラム検証を行う方法もあり、Proof-Carrying Codeのように証明を利用する方法もある。しかしそれには様々な制限が付きまとう。例えばJava仮想機械ではオブジェクト指向言語以外の言語で記述されたプログラムを効率よく実行することができない。特定の高級言語だけしか効率良く実行できないのでは普遍的な実行環境であるとは言えない。

.NETランタイム（CLR）では最初からC言語やC++言語を実行させることも想定していたため、実行効率と安全性の両立させる最もナイーブな、しかし現状ではベストかもしれない機構を持っている。それは効率を重視するモードと安全性を重視するモードの区別である。安全性を重視するモードではプログラムの実行直前にJavaのプログラム検証と本質的に等しい検査が行われ、プログラムが仕様通りに動作することが保証されるが、そのかわり一部の言語機能が制限されている。一方、効率を重視するモードではすべての言語機能を利用できるが、安全性の検査はプログラムに埋め込まれた署名によって確認される。つまりプログラムの安全性の問題はそのプログラムを作ったプログラマへの信頼に帰着させられている。

.NETランタイムにおいて安全性を重視するモードで禁止される言語機能の中で主要なものはポインタ演算である。C言語をコンパイルする時はポインタ演算を自由に使って良い。しかしJava言語と似た言語であるC#言語をコンパイルする場合では生成されたコードにはポインタ演算が含まれてはいけない。我々もこの「実行効率と安全性の分水嶺となるのがポインタ演算の有無である」という重要な認識を共有している。この認識が含意しているのはもし仮にポインタ演算を安全であると保証できるのならば、実行効率と安全性を両立できる、ということである。これがこの研究を始めるきっかけとなったアイデアであった。

2．研究方法と成果

2．1　アーキテクチャ非依存コード表現

本研究ではアーキテクチャに依存しない低水準コード表現を設計し、IA32とSPARCのコードを生成するJITコンパイラを実装した。我々は次のような目標を持ち、このコード表現を設計した。

1．多様なプログラミング言語のバックエンドコンパイラとして利用できること。

2．多様なアーキテクチャの機械語コードを生成できること。

3．完全な（accurate）GCをサポートできること。

この言語のプログラムの例としてN-Queenのプログラムを次のページに示す。このプログラム例から分かるように文法はcompound statementのないC言語に良く似ているが、semanticsはBCPLに非常に近いものである。変数には型があるが、この型は演算の種類を区別するためのものであり、いわゆる型安全性を保証するためのものではない。演算の種類とは、例えば加算が整数上の演算か浮動小数点数の演算かの違いやメモリへの読み書きのサイズの区別などを意味している。構造体、共用体に相当する型はない。ポインタ演算も自由に記述することができる。多様なプログラミング言語をサポートするためにこの言語には、現代的なプログラミング言語には必須の機能である例外機構を効率良く実装できる機能が提供されている。例外機能の仕様はC--にある類似の機能を深く参考にしている。完全なGCをサポートするために必要なのはスタックフレームとレジスタのどこにポインタが格納されているのかを示す情報（いわゆるスタックマップ）である。我々が実装したJITコンパイラはコンパイル過程の全段階

```
model32;
int8   nqueen_buf[20];
int32  nqueen ( int32 n ) [entry]
{  gc int32   p;
   int32      i, j, k, sum;
   if ( n < 12 ) goto body;
   return 1;
 body:
   sum = 0;
   p = nqueen_buf;
   i = 0;
 loop_i:
   if ( i >= 12 ) goto exit_i;
   j = 0;
 loop_j:
   if ( j >= n ) goto exit_j;
   k = (int8) p[j];
   if ( k == i ) goto next;
   if ( n + i == j + k ) goto next;
   if ( n - i == j - k ) goto next;
   j = j + 1;
   goto loop_j;
 exit_j:
   (int8) p[n] = i;
   k = nqueen (n + 1) ;
   sum = sum + k;
 next:
   i = i + 1;
   goto loop_i;
 exit_i:
   return sum;
}
```

N-Queenアセンブラプログラム

でポインタを保持するレジスタ、スタックスロットを覚えており、スタックマップを生成する機能がある。図中で整数型の変数宣言の前にgcというキーワードを追加している部分があるが、この宣言は変数がポインタを保持していることをコンパイラに通知している。

2.2 安全性の保証

安全なポインタ演算とは言っても、配列の添字解析が静的に決定不能であるようにもちろん一般の場合では安全性を保証できない。そこで我々はWalkerやXiによる一種のsoft typingのアイデアを採用し、静的検証器がポインタ演算を安全と証明できたものについては実行時にチェックを行わず、証明できなかったものは安全性のチェックを実行時に行うことにする。つまり我々はポインタ演算に関する制約を持ったアーキテクチャ非依存の低水準言語を設計し、その言語でのポインタ演算に関する静的解析器を作る。静的解析器によってあるポインタ演算が安全と証明されれば機械語コード生成時にチェックを挿入しない。そうでなければチェックを含んだコードを生成することになる。

我々の設計したポインタの制約システムは整数区間解析とポインタ解析を統合したもので、ポインタの動的な振る舞いを静的に規制することができる。この文書ではポインタに関する制約システムの最も重要な概念である抽象的な値と抽象的なメモリ領域について説明する。

2.2.1 値の抽象化

プログラムを静的に解析するためには、プログラム中で使われている値を見積もる必要がある。我々は安全性を重視しているので、プログラムがどんな実行パスを通ったとしても、値の取り得るすべての可能性をあらかじめ見積もっておかなければならない。なぜならもし想定外の値が生じた場合に安全性を保証できなくなるからである。

もちろん実行時に生じ得るすべての値を正確に知ることは一般にはできない。そのため一般に静的解析では値が取り得る可能性のある集合を近似して求める。例えばサイコロの出る目が分からなくても、その値は1から6までの整数であることは保証できる。サイコロの目が1から6までの整数であることが分かっても普通は役に立たないが、サイコロを振るプログラムによっては次に出る目が1から3であることが解析によって分かる場合がある。もしあるプログラムが、サイコロの目が6を出したときだけ危険な動作をするのであれば、我々は解析によって危険な状態が起こらないことを保証できたことになる。

解析によって有益な情報を引き出せるかどうかは、プログラムの使う値をうまく近似でき

る表現を見付けられるかどうかに依存している。我々の場合、ポインタの振る舞いをうまく近似する表現が必要なのである。

右図のようなC言語のプログラムを考えてみる。このプログラムは配列aを0で初期化しているだけのコードである。このプログラムのポインタpの取り得る値は、ポインタが32ビットで表されるアーキテクチャのマシンではa, a + 4, ..., a + 40の可能性がある。配列aの置かれるアドレスはプログラムのロード時に初めて決定され、静的には分からないので記号のままにしている。

```
int  a[10];
void  init  () {
   int  i;
   int  *p = a;
   for ( i = 0; i < 10; i++ )
     *p++ = 0;
}
```

プログラム1

ポインタpはfor文を抜けた後p+40の値を取るが、メモリへのアクセスを行うfor文の中に限定すればa, ..., a + 36のいずれかの値である。これを我々は{a} + 4 [0, 9]という記号で表すことにする。[0, 9]は0から9までの整数の集合を表している。4 [0, 9]はその要素をそれぞれ4倍にした集合{0, 4, ⋯, 36}である。

```
int  a[10], b[10];
void  init ( int *p ) {
   int  i;
   for ( i = 0; i < 10; i++ )
     *p++ = 0;
}
void  main  () {
   init (a) ;
   init (b) ;
}
```

プログラム2

この情報から変数pを使ったメモリのアクセスはベースアドレスに4の倍数を加えたオフセットを持つことが分かる。すなわちint型として正しいアドレスである。また変数pを使ったメモリのアクセスは配列aの範囲内であることも分かる。したがって我々は変数pを使ったメモリのアクセスは安全であると結論できる。

```
void  init  ( int *p, int len ) {
   int  i;
   for ( i = 0; i < len; i++ )
     *p++ = 0;
}
void  main  ( int argc, char *argv[] ) {
   int  len = atoi (argv[1]) ;
   int  *a = calloc  ( len, 4 ) ;
   init (a, len) ;
}
```

プログラム3

ベースアドレスを集合で表しているのは、ソースコード上の同じ変数が別の領域を指す場合があるからである。例えばプログラム2を考えてみる。このプログラムでは変数pは配列aと配列bの両方の領域を指す場合がある。我々の記号を使うと変数pが持つ抽象的な値は{a, b} + 4 [0, 9]である。

領域の範囲がプログラムだけからは決まらない場合がある。例えばプログラム3を考えてみる。この場合、初期化される領域の大きさはプログラ

ムの外部から与えられているためソースコードを解析しただけでは分からない。このような場合は上限が与えられないことを表す特殊な記号を使う。変数pが持つ抽象的な値は{a} + 4 [0, ∞]となる。この情報は、オフセットが上限は制限されておらず0より大きい4の倍数であることを意味している。また[-∞, ∞]は整数全体の集合Zと等しい。

解析アルゴリズムを考える際に重要なポイントとなるのは抽象的な値を利用してプログラムを仮想実行できるということである。言い替えるとC言語のすべての演算子に対して、抽象的な値の演算を定義できる。例えばプログラム中のp++はpのアドレスに4を加えるという演算であるが、これは抽象的な値に対しても（{a} + 4 [0, ∞]) + 4 = {a} + 4 [1, ∞]のように計算できる。

抽象的な値同士の演算が正しく定義できているかどうかは、抽象的な演算の結果が具体的な値の上での計算結果を必ず含んでいるかどうかによって判定される。つまりある具体的な演算・とそれに対応する抽象的な演算○があり、具体的な値同士の演算x・yに対応する抽象的な演算X○Yが正しいとは、任意のx∈Xと任意のy∈Yに対してその演算結果x・yが抽象的な演算の結果X○Yに含まれる、すなわち（x・y）∈（X○Y）となることである。これは抽象解釈での抽象化関数の作り方と全く同様の議論である。

2.2.2 メモリ領域の抽象化

値の抽象化と同様にメモリ領域についても抽象化しなければならない。なぜなら実行時に確保されるメモリ領域の量は一般に静的に読み切れないからである。そのためにはメモリの領域に対して何らかの抽象化を行い、実行時には別々の領域を静的には同一視することによって解析時に扱うメモリ領域を有限に押さえなければならない。

```
int  a[3] = { 11, 22, 33 };
```

プログラム4

メモリ領域とはオフセットから値への関数と考えることができる。例えばプログラム4で定義されたメモリ領域について考えてみる。このプログラムは整数の配列を初期化しているだけである。配列aは{0 → 11, 4 → 22, 8 → 33}という対応を持ったオフセットから値への関数であると考えることができる。

```
void  main ( int argc, char *argv[] ) {
   int  len = atoi (argv[1]) ;
   int  *a = calloc ( 3 + len, 4 ) ;
   a[0] = 11; a[1] = 22; a[2] = 33;
}
```

プログラム5

メモリ領域を単なる表ではなく、関数と解釈することには理由がある。例えばプログラム5のように実行時にcallocなどの関数によって動的に確保されるメモリ領域の大きさは静的には分からない。そこでこのような場合には何らかの方法によ

り安全に近似を行う。

近似の仕方には任意性がある。例えば $\{0 \to 11, 4 \to 22, 8 \to 33, 4\,[3, \infty] \to 0\}$ という関数でも良いし、また $\{4\,[0, \infty] \to 11\,[0, 3]\}$ という関数でも良い。ただし後者の方が近似としては荒いものになっている。

解析アルゴリズムは具体的な値とメモリ領域の代わりに抽象的な値とメモリ領域を使ってソースコードを仮想実行するといういわゆる抽象解釈である。実行時にしか分からない情報がある場合は近似を行うことによって解析を行う。近似によって情報の精度は失われてしまうが解析アルゴリズムは必ず停止しなければならないのでこれは仕方がないことである。できるだけ情報を失わずに済む近似法が良い近似法であると言える。

3．今後の展開

今後は設計した制約システムや静的解析器の有効性の検証を行いたいと考えている。そしてその過程でこれらのシステムについて改良できればよいと考えている。我々の設計したポインタ演算に関する制約システムが良いかどうかは、それができるだけ正確にポインタ演算の振る舞いを把握できているかどうかということである。具体的な基準としてはできるだけ多くの実行時チェックを外せるほど良いと言える。またポインタ解析の標準的な性能検証手法であるpoints-to setのサイズを使っても性能を計測することができる。Points-to setとは静的に求められた、プログラムの一つの変数が指す可能性のある値の集合のことであり、小さければ小さいほど良いとされている。測定に使うアプリケーションとしてはSPEC CPU 2000などのベンチマークやsendmailなどのインターネットのサーバを考えている。現在、大岩、末永、大山の協力のもとでC言語の現実的なアプリケーションから制約システムへの変換器を実装している。

本論文の執筆時点では残念ながら安全性を保証する機構がJITコンパイラに組み込まれていない。我々は早急に実装を進めており、完成次第我々の研究成果として一般に公開したいと考えている。

4．成果リスト

出版

1．Tatsurou Sekiguchi, Takahiro Sakamoto, and Akinori Yonezawa. Portable Implementation of Continuation Operators in Imperative Languages by Exception Handling. Advances in Exception Handling Techniques, Lecture Notes in Computer Science, 2022, ISBN: 3540419527, Springer Verlag, May 2001.

2. Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-Safe ANSI-C Compiler: An Approach to Making C Programs. Software Security -- Theories and Systems, pages 133 153, Lecture Notes in Computer Science, 2609, Springer Verlag, April 2003.

論文

1. 関口龍郎，大岩　寛，米澤明憲。オブジェクト指向言語によって記述された，携帯電話，PDAのアプリケーションプログラム圧縮方式。コンピュータソフトウェア。2002年1月。

2. 住井英二郎，関口龍郎，細谷春夫。PLI2002報告。コンピュータソフトウェア。Vol.20, No. 2, pages79-84, 2003年3月。

口頭発表

1. 関口龍郎，大岩　寛，米澤明憲。オブジェクト指向言語によって記述された，携帯電話，PDAのアプリケーションプログラム圧縮方式。第3回プログラミングおよびプログラミング言語ワークショップ。2001年3月。

2. Takeo Imai, Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. Dynamic Access Control of Mobile Objects by Switching Namespaces. OOPSLA Workshop of Patterns and Techniques for Designing Object-Oriented Mobile Wireless Systems. October 2001.

3. Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-Safe ANSI-C Compiler: An Approach to Making C Programs Secure - Progress Report. International Symposium on Software Security, November 2002.

4. 関口龍郎。Java のための二つのモバイル技術。オブジェクト指向2003シンポジウム。2003年8月。

コンテスト

1. Tatsurou Sekiguchi, Yutaka Oiwa, and Eijiro Sumii. Software: Rog-O-Matic Ⅲ, The 2002 International Conference of Functional Programming (ICFP' 02) Programming Contest Playoff, October 2002.