日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

# ビットレベル表現をサポートする低水準言語における BDD を利用したプログラム検証に向けて

八杉 昌宏

本研究では，既存言語よりも汎用性の高い型付中間言語，すなわち広く様々な言語からの翻訳の共通のターゲットとしての利用に優れた低水準の型付中間言語の設計を，安全で高性能な言語処理系が作成可能な点を重視しつつ，進めている．本論文では，いわゆる fixnum などのためのビットレベル表現を柔軟かつ効率良くサポートしつつ，安全性や情報流の検査を実現するために，現在検討を行っている BDD を利用したプログラム検査の方針について提案する．

## 1 はじめに

PPL2007 で発表した低水準の型付中間言語 MIL の設計 [5] に関する発展途上の検討について報告する．PPL2007 の論文には様々な技術的問題があり，様々な点で大きな修正を加えている．しかし，本論文も主に例を用いたインフォーマルな議論である点，ご容赦いただきたい．

本論文ではいわゆる fixnum などのためのビットレベル表現を柔軟かつ効率良くサポートしつつ，安全性や情報流の検査を実現するために，現在検討を行っている BDD を利用したプログラム検査の方針について提案する．また型システムについては,「場合」を型レベルで扱い,「場合依存型」という考えに基づく方式を提案している．

fixnum のようなデータは JVML などを含む既存の型付き中間言語では通常，扱えない．本研究で開発しようとしている型付中間言語 MIL では，ビットフォーマットをプログラムで指定することで，fixnum のようなデータを比較的自由に設計できる．例えばスクリプト言語などの動的型付け言語の実装を MIL 上で行うことで，fixnum による（boxing を要する整数等と比較して）高速な実行を可能としつつ，ごみ集めの実装などは MIL 処理系に委ねることが可能となる．

## 2 型付中間言語 MIL

型付中間言語 MIL に関して次の関連言語を考える．

- MIL(U)：MIL の untyped 版．分かりやすさを生かして，主には MIL の説明やごみ集め（GC）の定義に利用する．各データは（構文上での区別に用いるような実行時の）データ型を持つと考える．データ型が合わないとき実行は get stuck する．get stuck しない部分は基本的には MIL から型消去で得る．ただし，今回，提案する「型レベルの “場合”」は untyped において対応する（データ型を持つ）データがないため,「値レベルの “場合”」に対応させ,「場合依存の命令（列）」はそのまま残す．
- MIL：MIL(U) に依存型などの型システムを加えたと考えてもよい．（値レベルの場合を型レベルに戻さなくてはならない問題を除き．）変数は型を持ち，変数の値となるデータはデータ型を持つ．MIL(U) と違い，型検査をパスするプログラムのみを許し，そのプログラムの実行は get stuck しない．今回の提案では，`case` というカインドを考え，このカインドを持つ型変数に新しい役割を持たせる．各場合で別の役を持つ「場合依存の命令（列）」を考えることができるが，後

---

Towards BDD-based Program Verification for Low-level Languages That Support Bit-level Representations.

Masahiro Yasugi, 京都大学大学院情報学研究科, Graduate School of Informatics, Kyoto University.

述の LL-MIL(U) にコンパイルされた時には同一の命令（列）とならなくてはならない．MIL(U) では各データは実行時のデータ型を持つと考えて GC を考えたが，実際には，次に述べる実行時のデータ型を持たない低レベルの言語において静的な情報（コンパイル時の型）から GC が可能である必要がある．また，コンパイル時の型に基づいてトレースされる「参照」が MIL(U) で考えるものと一致する必要がある．MIL のそのような部分をトレース可能な部分と呼ぶことにする．ただし，トレース可能性の定義などは今回の論文では扱わない．

- LL-MIL： MIL からのコンパイル先となる中間言語である．MIL で型検査済みなので get stuck しない．変数（や定数）は型を持つが，その値となるデータは，データ型を持つのではなく，ただのビット列とする．MIL のトレース可能な部分からコンパイルされる．コンパイル時の型に基づいて「参照」をトレースできる．ただし，具体的なトレース方法は今回の論文では扱わない．
- LL-MIL(U)： LL-MIL の untyped 版．通常，言語処理系の作成者が想定しているコンパイルしきった環境で動作するプログラムのための言語である．LL-MIL から型消去で得る．ただし，トレース可能でないといけないので，LL-MIL の型からオブジェクトのトレース方法情報や，コード上の GC 可能点におけるトレース方法情報を生成する．これらの情報は，オブジェクトやコードに付加する．ただし，具体的なトレース方法情報は今回の論文では扱わない．型消去においては場合のための型も消去されるが「場合依存の命令（列）」は同一の命令（列）にコンパイルされるため問題ない．これにより，意図しているのは参照または fixnum といったビット列のデータについて同一の命令（列）を実行できることであり，高水準関数型言語に見られるようなインジェクションタグを用いなくても，自前でタグ付きデータやタグ分離データを設計でき，総称的な扱いやデータに応じた分岐を行うことができる．

## 3 本研究で扱う依存型の例

C や Java で `int` として提供される型は，本研究おいて `(exists x1 i32 (i32s x1))` のような型となる．ここで `x1` はカインド `i32` を持つ型変数であり何か整数値を表す．カインドは「型の型」であり，`i32` は,「型レベルにおいて」変数が 32-bit 整数値をとることを示す．`(i32s x1)` という型は，その型を持つ式（実際には変数のみ）のとれる値が `x1` の値のみとなるシングルトン型であり，通常の型としてのカインドを持つ．例えば，型定数 5 を用いて `(i32s 5)` という型を考えると，この型を持つ変数の値は 5 と確定する．`exists` は存在型を構成し，`(exists x1 i32 (i32s x1))` は「何かしら整数値が存在し，その値のみをとるような型」である．本研究における依存型は [4] を参考にしつつ，[3] で行われているように，型変数とカインドによる型レベルの値の扱いを行っている．

値の範囲に制約を設けたいときは，`(exists x1 i32 (exist-c (> x1 0) (i32s x1)))` のような型とする．これは正の整数の型となる．次の例は，この型の変数に対して比較を行って条件分岐をする例である．

```
! (:t x (exists x1 i32
!          (exist-c (> x1 0) (i32s x1))))
(unpack x x2 i32)
! (:k x2 i32) (:t x (exist-c (> x2 0) (i32s x2)))
(unpack-c x)
! (:c (> x2 0)) (:t x (i32s x2))
(when (< x 0)
  ! (:c (< x2 0))
  ...
  (goto ···)
)
! (:c (not (< x2 0)))
```

ここで，行頭 `!` の部分は型検査時の型環境をコードに付加した部分である．型環境は基本的には前提のリストであり，前提 `(:t` $x$ $\tau$`)` は変数 $x$ が型 $\tau$ を持つこと，前提 `(:k` $\alpha$ $k$`)` は型変数 $\alpha$ がカインド $k$ を持つこと，前提 `(:c` $c$`)` は制約 $c$ の成立を示す．ここでは，`(unpack x x2 i32)` 命令により，`x` の存在型が unpack され，pack されていた型変数の部分を型変数 `x2` で受け取って，前提に移し，後続の命令列で利用される．`(unpack-c x)` 命令は pack されていた

制約を前提に移す．比較 (< x 0) において，x は値 x2 をとると確定しているため，比較後の分岐の結果，分岐先に応じて制約 (< x2 0) またはその否定 (not (< x2 0)) が前提に追加される．この例では，when の本体において (> x2 0) かつ (not (< x2 0)) となり，後述の BDD などを利用すれば，これは成立させられないと分かるため，when の本体は dead code であると分かる．

以下は逆にこのような正の整数を pack する例である．

```
! (:k x2 i32) (:c (> x2 0)) (:t x (i32s x2))
(pack-c x (> x2 0))
! (:k x2 i32) (:t x (exist-c (> x2 0) (i32s x2)))
(pack x x1 i32 x2 (exist-c (> x1 0) (i32s x1)))
! (:t x (exists x1 i32
!          (exist-c (> x1 0) (i32s x1))))
```

ここで，(pack-c x (> x2 0)) 命令は制約を pack し，x の型を更新する．(pack $x$ $\alpha$ $k$ $\tau_1$ $\tau_2$) 命令は変数 $x$ の型 $\tau_2[\tau_1/\alpha]$ を，型 (exists $\alpha$ $k$ $\tau_2$) に更新する．ここで，$\tau_2[\tau_1/\alpha]$ は，型 $\tau_2$ 中の，カインド $k$ を持つ型変数 $\alpha$ に，同じカインドを持つ型 $\tau_1$ を代入した型を表す．

MIL の操作的意味論では，(pack $x$ $\alpha$ $k$ $\tau_1$ $\tau_2$) 命令で，$x$ の値は，$\alpha$，$k$，$\tau_1$ $\tau_2$ を伴うようになる．これらの情報は以下の例のように unpack 時に取り出される．

```
! (:v x 5) (:t x (i32s 5))
(pack x x1 i32 5 (i32s x1))
! (:v x (pack 5 x1 i32 5 (i32s x1)))
! (:t x (exists x1 i32 (i32s x1)))
(unpack x x2 i32)
! (:v x 5)
! (subst x2 5)
! (:t x (i32s x2))
```

ここで，(subst x2 5) は，これ以降の型環境やプログラムにおける x2 の自由な出現に対し，5 を代入する．

## 4 場合依存型

プログラミング言語の中には，型 $\tau_1$ と型 $\tau_2$ に対して，disjoint ではない union 型 $\tau_1 \vee \tau_2$ を用いることができるものがある．disjoint ではない，つまりインジェクションタグは付いていないので，型 $\tau_1$ と型 $\tau_2$ で許されるデータ自体に両者を区別する情報がなければ，両者を区別することはできない．例えば，C 言語における union はこのような union である．

今回，場合を表す型変数 (type variable of kind case) と，それに従った場合依存型 (case-dependent type) を提案する．場合を表す型変数の（型レベルの）値が何であるのかを単純に存在型で隠すことで，以前からある union として働かせることができる．

MIL では，場合依存型を (case $\alpha$ ($sym_1$ $\tau_1$) ($sym_2$ $\tau_2$) $\cdots$ ($sym_n$ $\tau_n$)) の構文で用いる．$\alpha$ は場合を表す型変数であり，カインド case を持つ．$sym_i$ は場合を表すシンボルであり，それぞれの場合に $\tau_i$ が選択される．選択に当たっては，通常の「代入」を**拡張**する．すなわち，場合依存型の $\alpha$ に場合 $sym_i$ を代入したものは，(case $sym_i$ ($sym_1$ $\tau_1$) ($sym_2$ $\tau_2$) $\cdots$ ($sym_n$ $\tau_n$)) などではなく，$\tau_i$ とする．

pack，unpack などは前節で示した依存型のものがそのまま使える．以下が，pack したものを（説明のため直後に）unpack する例である．

```
! (:t x type1)
(pack x c1 case ctype1
   (case c1 (ctype1 type1) (ctype2 type2)))
! (:t x (exists c1 case
          (case c1 (ctype1 type1) (ctype2 type2))))
(unpack x c2 case)
! (:k c2 case)
! (:t x (case c2 (ctype1 type1) (ctype2 type2)))
```

やや繰り返しになるが，(pack $x$ $\alpha$ case $\tau_1$ $\tau_2$) 命令は変数 $x$ の型 $\tau_2[\tau_1/\alpha]$ を，型 (exists $\alpha$ case $\tau_2$) に更新する．$\tau_2[\tau_1/\alpha]$ は，型 $\tau_2$ 中の，カインド case を持つ型変数 $\alpha$ に，同じカインド case を持つ型 $\tau_1$ を代入した型を表すが，ここで上で述べた「拡張した代入」を用いている点に注意してほしい．また，コンパイル時の型検査時には，unpack しても元には戻らない．型変数 c2 の具体的な型レベルの値は不明なので，x の型は type1 にも type2 にも限定することはこのままではできない．2 節で述べたトレース可能性はこのような場合も x の値を（型から計画を立てた方法で）調べることで，どちらなのか分からなくてはならない（ただし，分からなくても同じようにトレースできるなら同じようにトレースすればよい）という条件である．

一方，操作的意味論は以下のようになる．

```
! (:v x v) (:t x type1)
(pack x c1 case ctype1
   (case c1 (ctype1 type1) (ctype2 type2)))
! (:v x (pack v c1 case ctype1
         (case c1 (ctype1 type1) (ctype2 type2))))
! (:t x (exists c1 case
         (case c1 (ctype1 type1) (ctype2 type2))))
(unpack x c2 case)
! (:v x v)
! (subst c2 ctype1)
! (:t x (case c2 (ctype1 type1) (ctype2 type2)))
```

ここで，(subst c2 ctype1) は，これ以降の型環境やプログラムにおける c2 の自由な出現に対し，ctype1 を（拡張）代入する．

図 1 には，正の整数 (exists x1 i32 (exist-c (> x1 0) (i32s x1))) と偶数 (exists x1 i32 (exist-c (== (mod x1 2) 0) (i32s x1))) の union 型を用いた例を示す．ここでは，場合依存型以外に，場合依存型環境と場合依存命令を提案し，示している．場合依存型環境は，(open-case x) によって，x の場合依存型を，型環境が場合に依存する形に変形することで得ている．それぞれの場合における型環境において x の型は，選択したときの型となっている．場合依存命令も場合に依存して命令を選択する形となっている．ただし，図 1 においては，すべての場合依存命令において各場合が偶然同じ命令となっている．場合依存型環境を用いることで，unpack 後に前提として移す先が確保されている．

3 節では dead code が分かることを示したが，図 1 では，dead case が分かる．when の本体において，型変数 c2 が c-pos の場合に型環境に (:c (> x2 0)) (:c (< x2 0)) が含まれ，後述の BDD などを用いればこれが成立しないことが分かる．よってこの場合を除去することができる．実際，(del-case c2 c-pos) 命令はこの場合を除去している．場合除去により場合が 1 通りに定まることがあるが，実際図 1 ではそのようになっている．そこでさらに，(uncase c2) 命令で場合依存の型環境を，通常の型環境に戻している．

## 5 fixnum の例

fixnum とは，Lisp 言語などにおいて，マシンワードに収まるような小さな整数のためのデータ型である．1 つのマシンワードが，参照を保持していることもあるし，fixnum のこともあるとすることで，小さな整数については boxing，unboxing のオーバヘッドなしに 1 つのマシンワードで表現できる．

ここでは簡単のために，次の 2 つの場合を考える．

- fixnum の場合

  (exists x1 i30
    (b32s (bm 32 (0 0i2) (2 x1))))

  という型を用いる．ここで，型変数 x1 のカインド i30 は型レベルにおける 30-bit 整数を表す．(b32s $\tau$) は $\tau$ の値のみのシングルトン型である．(bm 32 (0 0i2) (2 x1)) は，型レベルにおける 32 ビット表現であり，0〜1 ビット目の 2 ビットが 0i2 で，2〜31 ビットの 30 ビットが x1 である．0i2 は 2 ビットで値が 0 の型レベルにおける定数を表す．
- cons の場合

  (exists rc1 r29
    (b32s (bm 32 (0 7i3) (3 rc1))))

  という型を用いる．ここで型変数 rc1 のカインド r29 は型レベルにおける 29-bit 参照を表す．cons の場合は，下位 3 ビットがすべて 1，つまり 0〜2 ビット目の 3 ビットが型レベル定数 7i3 である．なお本当はヒープ（参照からオブジェクトへの関数）についての制約（参照先のオブジェクトは 2 つの sexp 型のフィールドを持つという意味の (om rc1 (fm 8 (0 sexp) (4 sexp))) のような制約）を伴う (exists rc1 r29 (exist-c (om rc1 (fm 8 (0 sexp) (4 sexp))) (b32s (bm 32 (0 7i3) (3 rc1))))) という型を用いる必要があるが，本論文ではヒープの扱いを省略する．

図 2 に示す例は，4 節と同様に場合依存型，場合依存型環境，場合依存命令を用いている．図 1 との差は，定義を後述する fixnump という関係判定命令を用いている点，それを用いて型適用 tapp をしている点，関係判定命令の結果として加わる制約が，(偶然)

```
! (:t x (exists c1 case
!         (case c1
!           (c-pos (exists x1 i32 (exist-c (> x1 0) (i32s x1))))
!           (c-even (exists x1 i32 (exist-c (== (mod x1 2) 0) (i32s x1)))))))
(unpack x c2 case)
! (:k c2 case)
! (:t x (case c2
!         (c-pos (exists x1 i32 (exist-c (> x1 0) (i32s x1))))
!         (c-even (exists x1 i32 (exist-c (== (mod x1 2) 0) (i32s x1))))))
(open-case x)
! (case c2
!   (c-pos (:t x (exists x1 i32 (exist-c (> x1 0) (i32s x1)))))
!   (c-even (:t x (exists x1 i32 (exist-c (== (mod x1 2) 0) (i32s x1))))))
(case c2 (c-pos (unpack x x2 i32)) (c-even (unpack x x2 i32)))
! (case c2
!   (c-pos (:k x2 i32) (:t x (exist-c (> x2 0) (i32s x2))))
!   (c-even (:k x2 i32) (:t x (exist-c (== (mod x2 2) 0) (i32s x2)))))
(case c2 (c-pos (unpack-c x)) (c-even (unpack-c x)))
! (case c2
!   (c-pos (:k x2 i32) (:c (> x2 0)) (:t x (i32s x2)))
!   (c-even (:k x2 i32) (:c (== (mod x2 2) 0)) (:t x (i32s x2))))
(when (case c2 (c-pos (< x 0)) (c-even (< x 0)))
  ! (case c2
  !   (c-pos (:k x2 i32) (:c (> x2 0)) (:c (< x2 0)) (:t x (i32s x2)))
  !   (c-even (:k x2 i32) (:c (== (mod x2 2) 0)) (:c (< x2 0)) (:t x (i32s x2))))
  (del-case c2 c-pos)
  ! (case c2
  !   (c-even (:k x2 i32) (:c (== (mod x2 2) 0)) (:c (< x2 0)) (:t x (i32s x2))))
  (uncase c2)
  ! (:k x2 i32) (:c (== (mod x2 2) 0)) (:c (< x2 0)) (:t x (i32s x2))
  ...
  (goto ...)
)
! (case c2
!   (c-pos (:k x2 i32) (:c (> x2 0)) (:c (not (< x2 0))) (:t x (i32s x2)))
!   (c-even (:k x2 i32) (:c (== (mod x2 2) 0)) (:c (not (< x2 0))) (:t x (i32s x2))))
```

**図 1　場合依存型，場合依存型環境，場合依存命令**

真を表す 1 ビット定数 `1i1` か偽を表す 1 ビット定数 `0i1` のどちらかとなっている点である．`when` の後というもうひとつの分岐先には否定がとられた制約が加えられる．ここでは，それぞれの分岐先で，直ちにどちらかの場合を除去でき，(場合が 2 通りだけだったため）最終的に場合依存のない型環境になっている．ここから先は，fixnum と cons の union ではなく，例えば，fixnum と分かる場合に限定したプログラム（四則演算を行うなど）を書いてよい．

関係判定命令 `fixnump` の定義方法に関する本研究での提案について述べる．定義には，低レベル定義と高レベルの定義の両方を用いる．まず低レベルの定義は以下のようになる．

```
(def-compound-ll-insn (fixnump x)
  ((:inp x 32) (:temp temp 32))
  (bit-and x 3 temp)
  (== temp 0))
```

これはより低レベルな 2 命令（短い命令列）からなる複合命令となっている．このプログラムの意味としては，32 ビット表現 `x` を入力とし，一時変数 `temp` に 32 ビット定数 3 とのビット毎の AND をとったものを格納する．そして，一時変数 `temp` と 32 ビット定数 0 が等しいかを判定する命令列となっている．この低レベルの定義は，2 節で述べた，ターゲット言語により近い低レベルの中間言語 LL-MIL(U) でのプロ

```
! (let bm1 (exists x1 i30 (b32s (bm 32 (0 0i2) (2 x1)))))
! (let bm2 (exists rc1 r29 (b32s (bm 32 (0 7i3) (3 rc1)))))
! (:t x (exists c1 case (case c1 (cfixnum bm1) (ccons bm2))))
(unpack x c2 case)
! (:k c2 case) (:t x (case c2 (cfixnum bm1) (ccons bm2)))
(open-case x)
! (case c2 (cfixnum (:t x bm1)) (ccons (:t x bm2)))
(case c2 (cfixnum (unpack x x2 i30)) (ccons (unpack x rc2 r29)))
! (case c2 (cfixnum (:k x2 i30) (:t x (b32s (bm 32 (0 0i2) (2 x2)))))
!          (ccons (:k rc2 r29) (:t x (b32s (bm 32 (0 7i3) (3 rc2))))))
(when (case c2 (cfixnum ((tapp fixnump cfixnum x2) x))
               (ccons ((tapp fixnump ccons rc2) x)))
  ! (case c2 (cfixnum (:c 1i1)) (ccons (:c 0i1)))
  (del-case c2 ccons)
  ! (case c2 (cfixnum (:k x2 i30) (:t x (b32s (bm 32 (0 0i2) (2 x2))))))
  (uncase c2)
  ! (:k x2 i30) (:t x (b32s (bm 32 (0 0i2) (2 x2))))
  ···
  (goto ···)
)
! (case c2 (cfixnum (:c (not 1i1))) (ccons (:c (not 0i1))))
(del-case c2 cfixnum)
! (case c2 (ccons (:k rc2 r29) (:t x (b32s (bm 32 (0 7i3) (3 rc2))))))
(uncase c2)
! (:k rc2 r29) (:t x (b32s (bm 32 (0 7i3) (3 rc2))))
···
```

**図 2　fixnum についての場合依存型，場合依存型環境，場合依存命令**

グラムに相当している．

次に関係判定命令 fixnump の高レベルの定義は以下のようになる．(より一般的な定義は後述する．)

```
(def-synth-insn (fixnump x)
  ((:k x1 i30) (:c 1i1))
  ()
  ((:inp x (bm 32 (0 0i2) (2 x1))) (:iff 1i1)))
```

この合成された命令 fixnump は，入力としてカインドが i30 の型を x1 として受け取り，真（1i1）であるという制約を満たす制約を受け取り，入力 x には (bm 32 (0 0i2) (2 x1)) というフォーマットの 32 ビットのデータを受け取ることを示している．また，出力は何もなし（括弧のみの部分）であり，関係判定結果は真になるということを (:iff 1i1) の部分が示す．実際には (:c 1i1) の部分は冗長なので以降では除去して

```
(def-synth-insn (fixnump x)
  ((:k x1 i30))
  ()
  ((:inp x (bm 32 (0 0i2) (2 x1))) (:iff 1i1)))
```

を考える．この合成命令 fixnump を利用するには，((tapp fixnump x2) x) のようにする．最初の x2 はカインドが i30 の型として x1 として受け取られ，(b32s (bm 32 (0 0i2) (2 x2))) という型 x を次に渡すことになる．判定結果は 1i1 であるため，(when ((tapp fixnump x2) x) ···) のように用いれば常に when の本体が実行されることが分かる．

簡単のため場合非依存の場合について述べたが，実際には，関係判定命令 fixnump の高レベルの定義は場合に依存して，以下のようになる．

```
(def-synth-insn (fixnump x)
 (case _
   (cfixnum
    ((:k x1 i30))
    ()
    ((:inp x (bm 32 (0 0i2) (2 x1)))
     (:iff 1i1)))
   (ccons
    ((:k rc1 r29))
    ()
    ((:inp x (bm 32 (0 7i3) (3 rc1)))
     (:iff 0i1)))))
```

ここで，場合依存定義 (case $\alpha$ ($sym_1$ $def\text{-}body_1$)

($sym_2$ *def-body*$_2$) $\cdots$ ($sym_n$ *def-body*$_n$)) の型変数 $\alpha$ の位置に “ _ ” を用いているのは型適用の際の型引数を受け取ることを意味している．受け取った型引数に応じて選択される各 *def-body*$_i$ に基づいて命令が実行されることになる．よって，`(when ((tapp fixnump cfixnum x2) x) ...)` のように用いれば，`cfixnum` の場合の定義が用いられ，常に `when` の本体が実行されることが分かる．高レベルの定義を用いれば，ビットフォーマットが独自設計されたようなデータに関しても，型変数間の関係や関係判定結果などが示されており，MIL 標準の命令と同様に，依存型による型検査が可能となっている．

依存型を用いた型システムでは，制約を満たせるかどうかといった判定が重要となるが，本研究ではこれに BDD を用いることを提案する．これについては，次の 6 節で簡単に述べる．また，本節では，合成命令について，低レベルの定義と高レベルの定義を与えることができたが，次の課題が残っている．

- 高レベルの定義が安全性や情報流の観点から妥当であるかの検査をしていない．これについては，7 節で述べる．
- 高レベルの定義と低レベル定義かマッチしているかの検査をしていない．これについては，8 節で述べる．

## 6 BDD

BDD（2 分決定グラフ）[1] は真偽値をとる変数をパラメータとする論理関数を効率よく表現したものである．2 分決定木に含まれる共有可能な部分，省略可能な部分を共有・省略することで無駄のない形をしている．変数順序を固定することで，BDD で表現された 2 つの論理関数の論理演算などが高速に実行可能である．

本研究では論理関数の NOT，AND，OR などや，変数の quantification（existential，universal，uniqueness がある）や，変数への代入を行う BDD ライブラリを Common Lisp で実装した．また，論理変数，論理演算，quantification からなる論理式（Quantified Boolean Formula; QBF と呼ぶ）の妥当性などを BDD を用いて検査できるようにした．これは単に BDD へと変換することで恒真性，充足可能性などが判定できる．さらには，さまざまなビット幅の整数変数や整数定数に関する整数制約，論理演算，quantification からなる論理式（Quantified Integer Formula; QIF と呼ぶ）についても，QBF に変換することで妥当性などを検査できるようにした．例えば，`(==+ 32 z x y)` は 32 ビット変数 `x`，`y`，`z` の間に，数学でいう $z = x + y$ に近い関係を計算機の加算命令などで通常行う 32 ビット演算に関して述べたものとなっている．BDD へと変換する際には，変数順序が重要であるが，`(==+ 32 z x y)` のような関係は，整数変数をビット変数配列と考えたときに，同じ桁のビットが隣り合うような順序にするとコンパクトな BDD となる．

前節までに述べた `(:c (> x2 0)) (:c (< x2 0))` のような制約のリストが充足できないことは，BDD を用いて簡単に判定できる．

## 7 安全性と情報流の検査

5 節で述べたような高レベルの命令定義は次の観点から安全性や情報流の検査をすべきと言える．

- 参照保存性： 参照はオブジェクト生成を行うような特別な命令以外では生成（ねつ造）されない．この安全性（ヒープにないぶらぶら参照）あるいは情報流の問題（あてずっぽの参照の利用の成功）の解決は，`(fixnump x)` の定義では「空」だった出力に関する型変数や制約の導入において，参照のカインドを持つ型変数の導入を禁止することで実現できる．
- オブジェクトへのアクセスと同値性判定に限定した参照値利用：これは参照の特定のビットに依存して結果が決まるような命令ではないということである．これは，制約のための論理式の構文では参照についての比較は同値比較のみとし，論理式において他の構文でも現れないようにし，制約の論理式を「カインド付け」することで実現できる．
- 出力の一意性：これは上記の「オブジェクトへのアクセスと同値性判定に限定した参照値利用」に対する補完としても必要である．出力において

導入される型変数について uniqueness quantification した論理式の恒真性で判定できる．なお，BDD における論理変数の uniqueness quantification は，その変数に真を代入したときとその変数に偽を代入したときとの排他的論理和で計算できる．(`:iff` $c$) が使われている場合は (`xor` $c$ (`not` $c$)) の恒真性の判定も必要なはずだが，これは if ではなく iff (if and only if) としたことにより既に成立している．(入力に関する) 出力の一意性により，高レベルでは無視しているが低レベルでは見えてしまうデータ（フォーマット指定に含まれないビットなど）に依存して出力が決まることがないこと，つまり，低レベルからではあるが意図しない不安定あるいはセキュリティリスクのある情報流が生じていないこと，が確認できる．

## 8 場合依存合成命令のビットレベル同一性による妥当性確認

5 節で述べた合成命令の高レベル定義と，低レベル定義とがマッチしているかの検査について述べる．ここで低レベル定義を用いているのは，高レベル定義は自由に設計できるわけではなく，最終的に LLMIL(U) のような低レベルの言語にコンパイルされたときに，場合に依存せず，(高レベル定義の意味は考えずにビット列としての操作を黙々と実行するための) 1 つの命令列となることを求めているからである．これにより，インジェクションタグなどに頼らずに自前で複数のフォーマットの可能性があるデータの違いを分析できるようになる．

マッチしているかの確認は次の 4 段階で行う．この確認は，高レベル定義が「場合依存定義」のときは，単にすべての場合について，それぞれの定義を確認すればよい．

(1) 高レベル定義の入力のビットフォーマットに対応する，低レベルのビット列表現が存在することの確認

(2) (1) に対応する低レベルのビット列表現に対する，命令列実行後のビット列表現が存在することの確認

(3) (2) の実行後の低レベルのビット列表現に対応する，高レベル定義の出力ビットフォーマットや関係判定が存在することの確認

(4) (1)〜(3) と低レベルを経由して定まる出力に関する条件の成立から，高レベル定義における出力に関する条件の成立を導けるかの確認

これらは高レベル定義のビットフォーマットと低レベルのビット列表現の間の対応関係を示す特別な述語を準備すれば，あとは QIF で表現可能である．そのような述語 `encode` を準備し，(`encode` ((`0 zt`) (`2 zd`)) `z`) のような論理式を書けることにした．ここで，`zt` や `zd` は高レベルの 2 ビット変数や 30 ビット変数であり，`z` は低レベルの 32 ビット変数である．その意味は単純に，対応するビットの真偽値が一致するというものである．この単純だが重要な述語の発見は本研究における主要な貢献の一つである．

最後に，`fixnump` の低レベル定義と，`cfixnum` の場合の高レベル定義に関する (1)，(3)，(4) について，その QIF にあたる論理式を図 3 に示す．ここでは，低レベルの関係判定結果を `cr` にとり，高レベルの関係判定結果を `c` として対応させている．これらの QIF は QBF へ変換したのち，BDD ライブラリを用いて恒真と確認できている．

## 9 まとめと今後の課題

いわゆる fixnum などのためのビットレベル表現を柔軟かつ効率良くサポートしつつ，安全性や情報流の検査を実現するために，現在検討を行っている BDD を利用したプログラム検査の方針について提案した．特に型システムとしては，型レベルで「場合」を導入し，場合依存型などを導入することを提案した．

今回の論文で示した範囲で，より微妙な問題についても扱える．例えば，複数の fixnum 候補のビット毎の OR をとってから，3 と AND をとることですべて fixnum と判定するといった命令も合成して確認できる．また，BDD を用いているので，数学的な自然数とは違って，定まったビット数で演算がなされるという点についても，そのまま対応できる．例えば，[2] で述べられているような整数オーバフローが原因となる確認ミスは生じない．

```
(1)
(forall ((xt i2)(xd i30))
  (implies
    (== xt 0)
    (exists ((x b32))
       (encode ((0 xt)(2 xd)) x))))

(3)
(forall ((xt i2)(xd i30)(x b32)
         (z b32)(cr b1))
  (implies
    (and
      (== xt 0)
      (encode ((0 xt)(2 xd)) x)
      (==bit-and z x 3)
      (iff cr (== z 0)))
    (exists ((c i1))
      (iff c cr))))

(4)
(forall ((xt i2)(xd i30)(x i32)
         (z b32)(cr b1)(c i1))
  (implies
    (and
      (== xt 0)
      (encode ((0 xt)(2 xd)) x)
      (==bit-and z x 3)
      (iff cr (== z 0))
      (iff c cr))
    (iff c 1)))
```

**図 3　高レベル定義と低レベル定義の対応をビットレベルで妥当性確認するための (1), (3), (4) に対応する QIF（すべて真となる）**

今回の論文で示した範囲は設計としては一応完結しているが，型付中間言語 MIL の設計のという研究自体においては，オブジェクトの初期化の問題，トレース可能性の問題，文字列や配列の一般的扱いなどの課題が残っている．また，MIL 処理系の作成も今後の課題である．

**謝辞** Common Lisp 上の BDD ライブラリの開発に携わった湯浅信吾氏，花岡俊行氏に感謝する．本研究の一部は，科学研究費挑戦的萌芽研究「安全で高速な共通計算基盤のための低水準の型付中間言語（20650004）」の補助を得て行った．

## 参 考 文 献

[1] Bryant, R. E.: Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers*, Vol. C-35, No. 8(1986), pp. 677–691.

[2] Chen, J., Hawblitzel, C., Perry, F., Emmi, M., Condit, J., Coetzee, D., and Pratikaki, P.: Type-Preserving Compilation for Large-Scale Optimizing Object-Oriented Compilers, *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, 2008, pp. 183–192.

[3] Hawblitzel, C., Huang, H., Wittie, L., and Chen, J.: A Garbage-Collecting Typed Assembly Language, *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI'07)*, 2007, pp. 41–52.

[4] Xi, H. and Harper, R.: A Dependently Typed Assembly Language, *Proceedings of the 6th International Conference on Functional Programming (ICFP'01)*, 2001, pp. 169–180.

[5] 八杉昌宏: 正確なごみ集めを前提とした低水準の型付中間言語の設計, 第 *9* 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 *(PPL2007)*, March 2007, pp. 111–122.