Vol. 46 No. SIG 6(PRO 25) 情報処理学会論文誌：プログラミング Apr. 2005

実行可能なコンパイラの形式化と検証

大熊浩示[†] 南出靖彦[†]

正しいコンパイラを得るためには，コンパイラの形式化が正しいだけでなく，その実装の正しさも検証しなければならない．我々は定理証明システム Isabelle/HOL を用いてコンパイラを形式化し，その正しさを検証した．さらに，Isabelle/HOL のコード生成機能を用い，Isabelle/HOL で記述されたコンパイラを実行可能なプログラムへ変換した．このコード生成機能は Isabelle/HOL の構文を Standard ML の構文に単純に変換するもので，生成されたプログラムの信頼性は高いといえる．コンパイラは構文解析を除くすべてのフェーズが Isabelle/HOL で記述されており，コード生成機能で変換されたプログラムに構文解析プログラムなどを加えることにより実行可能な検証されたコンパイラを得ることができる．コンパイラは Scheme の構文を持つ関数型プログラミング言語を入力とし，Java 仮想機械のアセンブリ言語を生成する．今回形式化したコンパイラはクロージャ変換やインライン展開など，これまで検証されたコンパイラでは扱われていない高度なプログラム変換を行っている．また，コンパイラの記述ではコンパイル時間も考慮し，効率の良いデータ構造を採用した．得られたコンパイラをいくつかの Scheme プログラムに対して適用した結果，コンパイル時間，生成されたプログラムの実行時間ともに既存のコンパイラに劣らない結果が得られた．

Specification and Verification of an Executable Compiler

Koji Okuma[†] and Yasuhiko Minamide[†]

To obtain a correct compiler, it is not enough to show its formalization is correct, but it is also required to show that the implementation of the compiler is correct. We formalized a compiler with the theorem prover Isabelle/HOL, and proved its correctness. Then, we translated the compiler definition from Isabelle/HOL into a Standrad ML program, using Isabelle/HOL's code generation facility. Since this code generation program only performs simple syntactic translation, the correctness of the obtained program can be highly trusted. The compiler we defined with Isabelle/HOL covers all phases of the compiler, except a front-end. We provided a hand-written parsing program, and obtained a verified, executable compiler. Our compiler compiles a functional language similar to Scheme into a Java virtual machine assembly language. It includes several non-trivial program transformations such as closure conversion and inline expansion. Taking compilation time into consideration, we chose efficient data structures in our specification. Benchmarks show that our compiler is comparable to existing compilers both in compilation time and running time.

1. はじめに

正しいコンパイラを得るためには，コンパイラの形式化が正しいだけでなく，その実装の正しさも検証しなければならない．従来の研究では，コンパイラの検証を行うだけにとどまることが多かったが，いくつかの研究では，検証されたコンパイラの定義をコンパイラプログラムに変換し実行できるコンパイラを構築していた．しかし，コンパイラプログラムへの変換が複雑であるため得られたプログラムの信頼性が低かったり，コンパイラの形式化が実行時の効率を考慮していないため現実的なコンパイラが構築できなかったりした．本研究では，コンパイラプログラムに直接対応するようにコンパイラの形式化を行い，その定義をコンパイラプログラムに変換し，検証された実行可能なコンパイラを構築した．また，コンパイラの形式化は変換されるコンパイラプログラムの実行時の効率も考慮に入れて行った．従来の研究での形式化に比べ，本研究の形式化はより実際のコンパイラのプログラムに近いものである．

本研究では定理証明システム Isabelle/HOL を用いてコンパイラを形式化し，その正しさを検証した．さらに，Isabelle/HOL のコード生成機能を利用して，

† 筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻
Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba

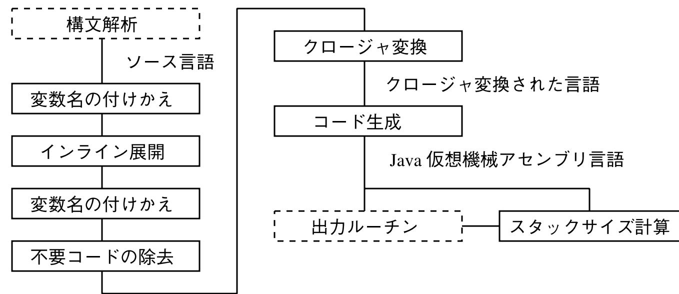

図 1 コンパイラの構成
Fig. 1 Organization of our compiler.

Isabelle/HOL で記述されたコンパイラを実行可能なコンパイラプログラムへ変換した．Isabelle/HOL のコード生成機能は，Isabelle/HOL で記述された定義を Standard ML プログラムに変換する機能である．コンパイラの形式化では，Isabelle/HOL の機能の中で Standard ML のプログラムに直接対応するもののみを使用した．生成されたプログラムに構文解析プログラムや出力プログラムを加えることにより，検証された実行可能なコンパイラを構築した．

我々が検証したコンパイラは，Scheme の構文を持つ関数型プログラミング言語を入力として受け取り，Java 仮想機械のアセンブリ言語を出力する．言語の主な機能として，静的スコープ，高階関数，リスト処理などがあげられる．基本的なデータ型として整数，真理値，記号などを扱うことができる．

本研究で形式化・検証したコンパイラの構成を**図 1** に示す．コンパイラはクロージャ変換とコード生成という 2 つのプログラム変換を行う．クロージャ変換では入れ子になった関数を閉じたものに変換して，すべての関数をトップレベルに引き上げる．コード生成ではクロージャ変換後の言語から Java 仮想機械のアセンブリ言語を生成する．また，このコンパイラはインライン展開と不要コードの除去という 2 つの最適化を行う．これらの最適化はソース言語上で行われる．変数名の付けかえのフェーズはその後のフェーズで正しくコンパイルできることを保証するために挿入されている．スタックサイズ計算のプログラムは，コンパイルされた仮想機械プログラムが使用するスタックの最大値などを求める．図に現れるフェーズの中で，実線でかこまれた部分はすべて Isabelle/HOL で形式化・検証を行った．

図中で点線でかこまれた部分は，コンパイラを実行可能にするために用意した補助的なプログラムである．構文解析プログラムには，マクロの展開や相互再帰関数をまとめるプログラムなども含まれる．出力プログラムは Isabelle/HOL で定義した Java 仮想機械のアセンブリ言語を Jasmin バイトコードアセンブラ[1) の形式に沿って出力する．そのほかに，言語のプリミティブを実現するために記述した Java ライブラリ，Scheme で記述したライブラリを与えた．

得られたコンパイラをいくつかの Scheme プログラムについて実行し，コンパイル時間，コンパイラが生成したプログラムの実行時間について既存の Scheme コンパイラと比較した．その結果，コンパイル時間，コンパイルされたプログラムの実行時間ともに既存のコンパイラに劣らない結果が得られた．

本研究は国際会議 APLAS '03 で報告した研究[2) を発展させたものである．前回の報告と比べ，より現実的なプログラムをコンパイルできるよう，手続き的な機能の追加や相互再帰関数のサポートなど，言語機能を拡張した．また，生成されるコンパイラプログラムの実行効率を考慮してより効率の良いデータ構造やアルゴリズムを採用した．

本研究は従来のコンパイラの検証の事例と比べ，1）高度なプログラム変換を検証している，2）最適化など複数のフェーズを扱っている，3）コンパイル時間を考慮して比較的複雑なデータ構造を用いて記述されている，などの点で大規模なものであるといえる．また，これまで定理証明システムでは検証されていなかったプログラム変換も扱っている．コンパイラの形式化・検証を行った証明スクリプトは `http://www.score.cs.tsukuba.ac.jp/~okuma/vc/` から入手できる．

本論文の構成は以下のようになっている．まず，2 章では定理証明システム Isabelle/HOL の機能と記法を紹介する．特に，コンパイラのプログラムを生成するために利用した Isabelle/HOL のコード生成機能について取り上げる．次に，3 章ではコンパイラが扱う言語の機能を紹介し，Isabelle/HOL での形式化について述べる．4 章ではコンパイラの主要な変換である

クロージャ変換について，その形式化と検証について説明する．5 章では，コンパイラが扱う最適化の中でインライン展開について説明する．6 章では，前章までで説明されなかったフェーズについて簡単に説明する．7 章では Isabelle/HOL で生成しなかったプログラムについて説明し，得られたコンパイラの実験・評価について述べる．最後に関連研究について述べ，まとめる．

2. Isabelle/HOL とコード生成機能

この章では定理証明システム Isabelle/HOL の基本的な機能を紹介し，本論文で用いる記法を説明する．その後，コンパイラの仕様から実行可能なプログラムを生成するときに利用する Isabelle/HOL のコード生成機能について詳しく述べる．

2.1 Isabelle/HOL

Isabelle は汎用の定理証明システムであり，一階述語論理，高階論理，公理的集合論などのさまざまな論理体系で証明を行うことができる．本研究で用いた Isabele/HOL は Isabelle で Church の高階論理の体系を実現したものである[3]．Isabelle/HOL は関数型プログラミング言語で用いられるデータ型や再帰的な関数などを用いて仕様を記述することができる．我々は Isabelle の新しい証明記述言語である Isar[4] で形式化・証明を行った．Isar はこれまでの記述言語に比べ，可読性の高い，構造化された証明を記述することができる．以後 Isabelle/HOL を単に HOL と呼ぶ．

HOL では標準的な論理記号である $\vee$，$\wedge$，$\rightarrow$，$\forall$，$\exists$ などを用いる．また，$\equiv$，$\Longrightarrow$ はそれぞれ，メタ論理である Isabelle の同値と含意を表す．論理式 $[\![P_1; \cdots ;P_n]\!] \Longrightarrow Q$ は論理式 $P_1 \Longrightarrow \cdots \Longrightarrow P_n \Longrightarrow Q$ の省略形である．型についての記法は，関数の矢印が $\Rightarrow$ で表されることを除いて，Standard ML のものと同様である．型 $[\tau_1, \cdots ,\tau_n] \Rightarrow \tau$ は型 $\tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau$ の省略形である．例として，リストの連結関数 @の型は下のような定義になる．

```
consts
  "@" :: "['a list, 'a list] ⇒ 'a list"
```

本研究では HOL の帰納的に定義される集合をさまざまな場面で利用している．帰納的に定義される集合は，キーワード **inductive** に続いて記述されている規則に関して閉じた最小の集合を定義する．また，定義した集合に関する導入規則や除去規則，帰納法の規則なども生成される．帰納的な集合はコンパイラの検証の過程で，言語の意味を定義するときなどで利用している．

2.2 Isabelle/HOL のコード生成機能

本研究では，HOL で定義したコンパイラから実行可能なプログラムを生成するときに HOL のコード生成機能[5] を利用した．HOL のコード生成機能は Isabelle/HOL で記述された仕様から Standard ML プログラムを生成する機能である．この機能はデータ型や原始帰納関数，帰納的に定義される集合など HOL の一部の機能を変換することができる．データ型や原始帰納関数は，対応する Standard ML の記法に単純に変換される．帰納的に定義される集合は，モード解析が行われ，論理プログラミング言語をシミュレートするプログラムに変換される．

形式化を行う際に，我々はコード生成が可能な HOL の機能の中で，Standard ML に直接対応する，データ型と再帰的な関数だけを利用することにした．利用する機能を制限することにより，1）単純な変換しか行われないので，生成されたプログラムがより信頼できる，2）生成されたプログラムとの連携がしやすくなる，などの利点が得られる．ここで注意してほしいことは，このように機能を制限して定義する部分はプログラムとして生成されるコンパイラの仕様に限られることである．言語の意味や証明で用いられる補助的な定義などはコード生成されないので，HOL のすべての機能を利用している．

次に，コード生成機能によってどのようなプログラムが得られるかを例を使って見ていきたい．下の HOL の仕様ではデータ型を用いて自然数を定義し，さらに原始帰納関数として自然数上の足し算を定義している．

```
datatype
  nat = 0 | Suc nat

primrec
  "add 0 y = y"
  "add (Suc x) y = Suc (add x y)"
```

この仕様をコード生成機能で Standard ML プログラムに変換すると，次のプログラムが得られる．

```
datatype Code_nat = nat_0
                  | nat_Suc of Code_nat


fun add nat_0 y = y
  | add (nat_Suc x) y = nat_Suc (add x y)
```

得られたプログラムは，型の名前が変更されていることを除けば，変換前の仕様とまったく変わらないことが分かる．このようにコード生成では単純な変換しか行われないので，得られたプログラムの信頼性も高いといえる．

しかし，上で示したような変換だけでは問題が起こる．HOL の仕様をコード生成機能で変換すると，HOL の型は Standard ML プログラムではユーザ定義の型に変換されてしまう．したがって，生成されたプログラムから Standard ML の基本型を利用することができない．このような問題に対処するため HOL では，HOL の仕様をどのように Standard ML に変換するかを指定できる．

```
types_code
  "nat"        ("int")

consts_code
  "0"          ("0")
  "Suc"        ("(_ + 1)")
```

ここでは，HOL での *nat* 型を Standard ML の `int` 型として生成し，構成子 *0* は Standard ML の整数 `0` として生成されるよう指定している．構成子 *Suc* に関する指定では，その右側に書かれている Standard ML のプログラムに変換するよう指定している．生成されるプログラムは，右側の式のアンダースコア部分を構成子の引数に対応するプログラムで置きかえたものとなる．この変換は単純な構文的な操作であり，文法的にも意味的にもチェックがいっさい行われないので指定するときに注意が必要である．このような指定は構文解析プログラムなどと連携を図るときに必要になる．また，コンパイラの形式化では自然数をさまざまな部分で用いているので，自然数などの型を変換しない場合，得られるプログラムの効率は明らかに悪くなる．

こうした変換を行う場合，関数についても変更が必要になる場合がある．上の例のように構成子を変換してしまうと，その型についてパターンマッチングを行っている関数が動作しなくなってしまうからである．たとえば，次の関数はリストを受け取り要素を *n* 個取り除いたリストを返す関数である．

```
primrec
  "drop n [] = []"
  "drop n (x#xs) = (case n of 0 ⇒ x#xs
                          | Suc(m) ⇒ drop m xs)"
```

ここで，`#`はリストのコンスを表す構成子である．この関数では，*n* についてのパターンマッチングを行っている．しかし，上記の変換を指定してコード生成を行うと次の Standard ML プログラムが得られる．

```
fun drop n [] = []
  | drop n (x::xs) =
     (case n of 0 => (x::xs)
              | (xa + 1) => drop xa xs)
```

このプログラムは，構成子 *Suc* が式で置きかえられ，`case` 式の中で式 `xa + 1` がパターンとして現れてしまっている．このような定義は正しい Standard ML プログラムではなく，実行することができない．HOL ではこのような問題にも対処するため，関数についても変換を指定することができる．まず，自然数についてのパターンマッチを使わないように変更した関数 *drop'* を定義する．

```
primrec
  "drop' n [] = []"
  "drop' n (x#xs) = (if n = 0 then
                       x#xs
                     else
                       drop' (n - 1) xs)"
```

そして，関数 *drop'* が関数 *drop* と同値であることを証明し，コード生成では *drop'* を使用するように指定することができる．

```
lemma [rule_format, code]:
  "∀n. drop n xs = drop' n xs"
proof(induct xs)
...
```

この指定を行うと，関数 *drop* の定義の代わりに *drop'* の定義が生成される．コンパイラプログラムの生成時にはこのような指定をいくつかの関数で行う必要があった．

3. コンパイラのソース言語

この章では，コンパイラが扱う言語を紹介し，HOL で構文や意味を定義する．今回形式化したコンパイラはソース言語として Scheme の構文を持つ関数型プログラミング言語を受け取る．言語の主な機能として，静的スコープ，高階関数，リスト処理などがあげられる．また，基本的なデータ型として，整数，真理値，記号を扱うことができる．データはヒープ上に置かれ，`set-car!`，`set-cdr!` など一部の手続き的な機能を持つ．我々のソース言語と Scheme の主な違いを以下にまとめる．

- `set!`関数がない．
- 入出力を扱っていない．
- 末尾再帰の除去を行っていない．

まず，HOL での抽象構文の形式化について説明する．定理証明システムで変数束縛を持つ式を形式化する場合，de Bruijn インデックスを用いた形式化がよく行われている．しかし，我々は de Bruijn インデックスを用いず，変数を名前で表現する手法を用いた．名前を用いた表現を採用したのは，高度なプログラム変換では de Bruijn インデックスを用いた形式化が難しい，名前を利用した表現の方が人間にとってより理

```
datatype
  'a exp = NumExp int
         | BoolExp bool
         | SymExp symbol
         | NullExp
         | VarExp 'a
         | PrimExp primop "'a exp list"
         | IfExp "'a exp" "'a exp" "'a exp"
         | LetExp 'a "'a exp" "'a exp"
         | FunExp "('a * 'a list * 'a exp) list" "'a exp"
         | AppExp "'a exp" "'a exp list"
```

図 2 ソース言語の抽象構文
Fig. 2 Abstract syntax of the source language.

解しやすいなどの理由からである．たとえば，本研究でも扱ったインライン展開などのプログラム変換を de Bruijn インデックスを用いた式の変換として形式化すると複雑になることが予想される．また，定理証明システムでは人間が対話的に証明を行うので，表現が人間にとって理解しやすいかどうかということも重要になってくる．

抽象構文はデータ型として**図 2** のように定義した．以後この言語をソースの言語と呼ぶことにする．ここで，型 *int*，*symbol* はそれぞれ，整数や記号を扱うため導入した新しい型である．ソース言語の式は多相型を持ち，変数部分がパラメータ化されている．これは，プログラム変換が正しく動作することを保証するため，コンパイラが変数名の付けかえを行うためである．コンパイラが最初に行う変数名の付けかえのフェーズでは，文字列で表されている変数を新しい自然数で置きかえる．変数名の付けかえではすべての変数名が付けかえられ，束縛変数がすべて異なることを保証する．

関数定義式 *FunExp* では相互再帰的な関数を定義することができるように関数定義のリストを受け取るようになっている．しかし，コンパイラの形式化の説明が煩雑になるので，今後は相互再帰的な関数はないものとして説明する．例として，次の Scheme プログラム

```
(define (f x y) (+ x y))
(f 1 2)
```

は HOL では次のデータ構造として表すことができる．

```
constdefs
  sample_prog :: "nat exp"
  "sample_prog ≡
     FunExp 0 [1, 2]
       (PrimExp PlusOp [VarExp 1, VarExp 2])
       (AppExp (VarExp 0) [NumExp 1, NumExp 0])"
```

この例は変数として自然数を使用した場合で，関数 f が自然数 *0*，変数 x が自然数 *1* のように対応している．

ソース言語の意味は自然意味論で与えた．ソース言語ではヒープを扱うので，評価関係は，変数と値の対応関係を表す環境，評価前のヒープ，式，評価後のヒープ，値の帰納的関係として定義する．値はデータ型，環境は変数と値の連想リストとして以下のように定義できる．

```
datatype
  value = NullVal
        | LocVal nat

types
  'a env = "('a * value) list"
```

値 *value* はナル値かヒープのロケーションで，ロケーションは自然数で表している．ヒープ上に置かれる値はデータ型として以下のように定義できる．

```
datatype
  'a heapval = HeapInt int
             | HeapBool bool
             | HeapSym symbol
             | HeapCons value value
             | HeapCls 'a "'a env"
                          "'a list" "'a exp"
```

値 *HeapCls f E xs e* は関数 *f* のクロージャを表している．クロージャは関数の名前 *f*，関数が定義された時点での環境 *E*，仮引数のリスト *xs*，関数本体 *e* からなる．ヒープはロケーションからヒープ上の値への部分関数として定義されている．

```
types
  'a heap = "nat ⇒ 'a heapval option"
```

次に，評価関係を帰納的に定義する．評価関係 *eval* は**図 3** のように定義できる．この定義では変数，let 式，関数定義式の場合の評価関係をそれぞれ定義している．意味の定義は自然意味論をもとに行っている．変数の場合は環境で束縛されている値が評価結果となる．let 式の場合は，まず式 *e1* を評価し，変数 *x* についての束縛を加えた環境のもとで *e2* を評価する．

```
consts
  eval  :: "('a env * 'a heap * 'a exp * 'a heap * value) set"
inductive eval ···
  "lookup E x = Some v ⟹ (E, H, VarExp x, H, v) ∈ eval"
  "⟦(E, H, e1, H', v1) ∈ eval;
    ((x, v1)#E, H', e2, H'', v) ∈ eval⟧ ⟹ (E, H, LetExp x e1 e2, H'', v) ∈ eval"
  "⟦l ∉ dom H;
    ((f, LocVal l)#E, H(l ↦ HeapCls f ((f, LocVal l)#E) xs e1), e, H', v) ∈ eval⟧ ⟹
                                                (E, H, FunExp f xs e1 e, H', v) ∈ eval"
```

図 3 ソース言語の評価関係
Fig. 3 Evaluation relation of the source language.

関数定義式の場合は，クロージャを新しいロケーション *l* に作り，関数 *f* の束縛を環境に加えている．そしてこれらの環境，ヒープのもとで式 *e* を評価する．ここで，クロージャに保存する環境も *f* で拡張していることに注意してほしい．ヒープを持たない言語で同様の定義を試みた場合，これから作るクロージャの環境で自分自身を束縛してしまうことになるので，このような定義はできない．このような定義は，相互再帰的な関数を扱うときにも拡張しやすい．実際の定義では相互再帰的な関数も許している．相互再帰的に定義されている関数では，それらの関数すべてを束縛した環境を作り，各関数のクロージャはその環境を持つように定義している．詳しい定義は証明スクリプトを見ていただきたい．

4. クロージャ変換の形式化と検証

この章ではコンパイラが行う主要な変換であるクロージャ変換について説明する．そして，クロージャ変換とクロージャ変換後の言語の形式化を示し，正しさの証明について述べる．

4.1 クロージャ変換

クロージャ変換は，関数本体に現れる自由変数を環境への明示的な参照へ変換する．この変換で，自由変数を持つ関数定義は環境を引数として受け取るように変換され，自由変数の参照は受け取った環境への参照に変換される．この変換を行うと，関数は閉じたものとなりトップレベルで定義できるようになる．変換後のプログラムは，得られたトップレベルの関数定義のもとで評価される．そしてクロージャ変換後は，関数のクロージャが関数定義への参照と環境の組で表される．例として以下のプログラムについて考える．

```
(define (f x y)
  (let ((g (lambda (z)
             (+ x y z))))
   g))
```

このプログラムで，関数 g は自由変数 x と y を含んでいる．クロージャ変換では，関数 g を環境を引数にとるように変更し，自由変数 x と y の参照を環境への参照に置きかえる．クロージャ変換の結果得られるプログラムは以下のようになる．

```
(define (g env z) (let ((x (car env))
                        (y (cdr env)))
                    (+ x y z)))

(define (f x y) (cons g (cons x y)))
```

このプログラムでは，環境をリストで表し，クロージャを関数名と環境の対で表している．

クロージャ変換を形式化するには，まず変換後の言語の構文と意味を定めなければならない．クロージャ変換が行われた後の言語では，関数定義はトップレベルで行われるため局所関数を定義する関数定義式がなくなる．また，クロージャを作る作業が明示的に行われるようになるため，クロージャを作るための式が追加される．クロージャ変換の後ではプログラムはトップレベルの関数定義と評価する式の組で構成されるようになる．クロージャ変換後の言語は次のように変更される．

```
types
  var   = nat
  decl  = "var list * var list * cexp"
  decls = "(var * decl) list"

datatype
  cexp = ···
       | MkCls var "var list"
```

クロージャ変換は変数名の付けかえを行った後に実行されるので，この言語では変数が自然数で表されている．型 *decls* は関数定義を表している．関数定義は自由変数のリスト，仮引数のリスト，関数本体からなる．ターゲット言語の式 *cexp* ではクロージャを作る式 *MkCls* が追加されている．クロージャ変換後はクロージャの表現も変更され，クロージャは関数名と環境の

```
inductive ceval ···
  "⟦l ∉ dom H; (E, xs, vs) ∈ values⟧ ⟹
                 (D, E, H, MkCls f xs, H(l↦HeapCls f vs), LocVal l) ∈ ceval"
```

図 4 クロージャ変換後の評価関係
Fig. 4 Evaluation relation of the closure converted language.

```
consts
  values :: "(nat env * nat list * value list) set"
inductive values
  intros
  "(E, [], []) ∈ values"
  "⟦lookup E x = Some v; (E, xs, vs) ∈ values⟧ ⟹ (E, x#xs, v#vs) ∈ values"
```

図 5 環境のもとでの自由変数の値
Fig. 5 Values of free variables under an environment.

組で表される．変換後のクロージャは下のように定義される．

```
datatype
  heapval = ···
        | HeapCls var "value list"
```

この定義では環境を値のリストとして表現している．

次にこの言語の評価関係を定義する．ソース言語の評価関係と異なる点は，関係が関数定義を受け取るよう変更されていることと式の評価関係が若干変更されていることである．クロージャ変換された後は，つねにトップレベルの関数定義のもとで評価が行われるので，評価関係 *ceval* は関数定義を含むようになっている．また変換後は関数定義式がなくなり，クロージャを作る式が追加されたので評価関係もそれに合わせて変更する．先程追加したクロージャを作るための式の評価は**図 4** のように定義される．クロージャを作る式 *MkCls f xs* は，この式を評価するときの環境 *E* での自由変数 *xs* の値 *vs* を求め，クロージャを作る．ここで，関係 *(E, xs, vs) ∈ values* は環境 *E* での変数のリスト *xs* の値 *vs* を求める関係であり，**図 5** のような定義になっている．

次に，クロージャ変換を形式化する．概念的には，クロージャ変換は 2 つの部分からなる．1 つ目は環境を関数の引数に加えて関数本体で環境を参照するように変更する部分で，2 つ目は関数をトップレベルに引き上げる部分である．従来のクロージャ変換の形式的な扱いでは 1 つ目の部分しか形式化されていないが[6),7)]，通常コンパイラではこの 2 つの部分を 1 つの変換として行う．我々の定義もこの 2 つのフェーズを同時に行うものになっている．クロージャ変換は次の型を持つ原始帰納関数として定義した．

```
consts
  clsconv :: "var exp ⇒ cexp * decls"
```

上で述べたように，クロージャ変換はソース言語の式を，トップレベルの関数定義とターゲット言語の式の組に変換する．変換の一部を**図 6** に示す☆．変換の中心となるのは関数定義式の変換で，関数本体の自由変数 *ys* を *rem_list (fv e1) xs* として求め，変換後は変数 *f* をクロージャ *MkCls f ys* に束縛するよう変更し，トップレベルの関数定義にこの関数の定義 *(f, ys, xs, e1')* を追加している．ここで，関数 *fv* は式の自由変数の集合を求める関数，*rem_list X xs* は集合 *X* からリスト *xs* の要素を取り除く関数である．

HOL には集合を表現する組み込みの型 *set* と集合の操作関数が存在し，集合に関する証明もかなり自動化されている．しかし，これらの関数はコード生成をすることができない定義になっており，コンパイラの定義の中では用いることができなかった．そこで我々は 2 色木（Red Black Tree）を形式化し，このデータ構造を使用して集合とその操作関数を定義した．2 色木を採用したのは，形式化がしやすく，効率も良いという理由からである．2 色木の定義は標準的なもので，必要な定理の証明を含め 1,000 行ほどであった．2 色木の形式化と証明では，コンパイラの検証に必要な定理の証明のみを行い，定義した操作関数が 2 色木が満たすべきインバリアントを保つことなどは示さなかった．2 色木を使用した集合は多相型を持ち型 *'a rbset* で表される．クロージャ変換で使用した自由変数を計算する関数などは以下の型を持つ．

```
consts
  ins :: "['a::linorder, 'a rbset] ⇒ 'a rbset"
  rem_list :: "['a::linorder rbset, 'a list]
                                       ⇒ 'a rbset"
  fv :: "'a :: linorder exp ⇒ 'a rbset"
```

2 色木では要素に順序が付いていなければならないので，型に全順序であるという制約 *linorder* が加わっ

☆ 実際の定義では構成子は HOL のモジュール名で修飾されているが，可読性を高めるために省略した．以後もこの省略を適宜行う．

```
primrec
  "clsconv (VarExp x)         = (VarExp x, [])"
  "clsconv (FunExp f xs e1 e2) = let (e1', ds1) = clsconv e1 in
                                let (e2', ds2) = clsconv e2 in
                                let ys = rem_list (fv e1) xs in
                                  (LetExp f (MkCls f ys) e2',
                                   (f, ys, xs, e1') # ds1 @ ds2)"
  "clsconv (AppExp f es)       = let (f',  ds1) = clsconv f in
                                let (es', ds2) = clsconv_list es in
                                  (AppExp f' es', ds1 @ ds2)"
  ...
```

図 6 クロージャ変換
Fig. 6 Closure conversion.

ている．

コンパイラの検証では集合に関する多くの補題を用いている．2 色木で定義した集合操作関数についてもそのような補題が成り立つことが必要となる．そこで，これらの関数が組み込みの集合操作関数と対応することを証明した．例として次の補題は木 *t* が 2 分探索木になっているならば，2 色木の挿入関数 *ins* が集合の挿入関数 *insert* と対応することを示している．ここで，*toSet* は 2 色木を集合に変換する関数である．

```
lemma assumes "isBST t" shows
  "toSet (ins x t) = insert x (toSet t)"
```

このような補題を各操作関数に対して証明することで，HOL の集合操作関数の補題を利用することができた．

2 色木は集合を表すだけでなく，辞書を実現するデータ構造としても利用している．辞書のデータ構造はコンパイラにたびたび現れ，多くのフェーズで利用している．型 *('a, 'b) dic* を型 *'a* をキーとして型 *'b* を値とする辞書を表すものとし，以下の型を持つ基本的な操作を定義した．

```
consts
  put :: "[('a::linorder, 'b) dic, 'a, 'b]
                              ⇒ ('a, 'b) dic"
  get :: "[('a::linorder, 'b) dic, 'a]
                              ⇒ 'b option"
```

コンパイラの形式化と検証を行うだけならば，組み込みの型を利用するだけでよいが，本研究では実行可能なコンパイラを生成するということと，得られたコンパイラの実行時間も考慮に入れ，これらのデータ構造を採用した．

4.2 クロージャ変換の検証

前節で定義した，クロージャ変換の正しさを HOL 上で証明した．プログラム変換の正しさの検証では，変換後のプログラムの評価結果が変換前のプログラムの評価結果と対応することを証明する．そこで，まずソース言語の値とターゲット言語の値の対応関係を定義する．この対応関係は変換の正しさを表すだけでなく，証明するときに帰納法が機能するようにうまく決めなければならない．

クロージャ変換後の言語では，関数の定義はトップレベルで行われているので，ソース言語のクロージャとの対応関係を定義するには関数定義の情報も必要となる．この対応関係を *D ⊢ v1 = v2* で表すことにする．整数などの単純な値については対応関係は自明である．クロージャの対応を考えた場合，必要となる条件として，

- 変換前のクロージャの関数に対応する定義が関数定義に存在する，
- 変換後のクロージャの環境に自由変数の値が収められている，

の 2 つが考えられる．具体的な定義は以下のようになる．

```
inductive heapval_eq
  "D ⊢ HeapInt n = HeapInt n"
...
  "⟦lookup D f =
      Some (rem_list (fv e) xs, xs, e');
   clsconv e = (e', D');
   subset D' D;
   (E, rem_list (fv e) xs, vs) ∈ values⟧
  ⟹ D ⊢ HeapCls f E xs e = HeapCls f vs"
```

クロージャの対応では，1 つ目の条件で関数定義 *D* に関数 *f* のクロージャに対応する定義が存在することを表している．2 つ目の条件は，式 *e'* が式 *e* をクロージャ変換したものであること表している．3 つ目の条件では，クロージャ変換の結果得られた関数定義 *D'* が現在の関数定義 *D* に含まれることを示している．変換後のクロージャに収められている環境には関数本体 *e* に現れる自由変数の環境 *E* での値が収められていなければならない．変換後は環境は値のリスト *vs* として表されており，最後の条件でこれを要求している．このヒープ上の値の対応関係をヒープに拡張し，*D ⊢ H ∼ H'* で表す．

```
primrec
  "ie D (FunExp f xs e1 e) =
     (let e2 = ie D e1 in
       (if f ∉ fv e1 ∧ proper f e then
          FunExp f xs e2 (ie (put D f (xs, e2)) e)
        else
          FunExp f xs e2 (ie D e)))"
  "ie D (AppExp f es) =
     (if isVar f then
       (case get D (varname f) of
          None ⇒ AppExp (ie D f) (ie_list D es)
        | Some ab ⇒ makelet (fst ab) (ie_list D es) (snd ab))
      else
        AppExp (ie D f) (ie_list D es))"
```

図 7 インライン展開の関数定義
Fig. 7 Inline expansion function.

最終的に得られた定理は次のものである．

```
theorem assumes
  "([], empty, e, H, v) ∈ eval"
  "fv e = {}" "nodup (fn e)"
shows
  "let (e',D) = clsconv e in
    (∃H'. (D, [], empty, e', H', v) ∈ ceval ∧
          D ⊢ H ∼ H')"
```

この定理は，変換前の式を空の環境，空のヒープで評価したときにヒープ H，値 v という結果が得られるならば，変換後の式を評価しても，ヒープ H に対応する H' と値 v という結果が得られることを示している．この定理を示すためには，より一般的な補題を証明しなければならなかった．証明はソース言語の評価関係の導出についての帰納法で行われ，1,200 行ほどの長さだった．この定理で，ソース言語についての条件 *nodup (fn e)* が必要となった．ここで，*nodup* はリストに同じ要素が存在しないことを表す述語で，*fn* は式に現れる関数名をリストとして集める関数である．この条件は，関数をトップレベルに引き上げるときに局所関数が大域関数と同じ関数名を持たないことを保証する．我々のコンパイラでは変数名の付けかえのフェーズでこの条件を満たすようにしている．

5. インライン展開

この章では，今回形式化したコンパイラが扱う最適化の中からインライン展開を取り上げる．インライン展開はコンパイラの最適化として一般的であるが，我々が知るかぎり定理証明システムでの検証はなされていない．

5.1 インライン展開の形式化

インライン展開は，関数呼び出しを呼ばれた関数の本体で置きかえる最適化である．関数型プログラミング言語では小さい関数をいくつも使うので，関数呼び出しのコストが関数本体の評価コストに比べて高い．そのため，インライン展開は関数型プログラミング言語のコンパイラでは一般的に行われている．例として，下のプログラムは

```
(define (f x y) (+ (* x x) y))

(f 1 (+ 2 3))
```

インライン展開を行うと次のプログラムに変換される．

```
(let ((x 1)
      (y (+ 2 3)))
  (+ (* x x) y))
```

関数呼び出し `(f 1 (+ 2 3))` が `f` の関数定義で置きかえられていることが分かる．変換後のプログラムでは引数の評価順序が変わらないよう `let` が挿入されている．インライン展開を行う際に，関数を展開しすぎると実行時プログラムが大きくなってしまうので，一定の条件をヒューリスティックスで与えその条件を満たす関数のみを展開するのが一般的である．

我々のコンパイラでは，インライン展開はソース言語上で行われる．インライン展開は次の型を持つ原始帰納関数として定義できる．

```
types
  'a fundef = "('a, 'a list * 'a exp) dic"
consts
  ie :: "['a fundef, 'a exp] ⇒ 'a exp"
  ie_list :: "['a fundef, 'a exp list]
                              ⇒ 'a exp list"
```

インライン展開を行う関数 *ie* は引数として，展開する関数の定義を保持するデータ構造 *'a fundef* と式を受け取り，関数を展開した式を返す．インライン展開の定義の主な部分を**図 7** に示す．図で示したのは関数定義式と関数適用の場合である．関数定義式では，

```
inductive inline inlines ···
  intros
  fun1 : "⟦D,(rev xs@f#X)⊢ e1 ⇝ e2;
          nodup xs; disjoint xs X; f ∉ set xs; f ∉ set X;
          D(f↦(xs,e2)),(f#X)⊢ e ⇝ e'⟧
                              ⟹ D,X ⊢ (FunExp f xs e1 e) ⇝ (FunExp f xs e2 e')"
  app1 : "⟦e = VarExp f; D(f) = Some (xs, e');
          nodup xs; disjoint xs X;
          D,X ⊢ es [⇝] es'⟧
                              ⟹ D,X ⊢ (AppExp e es) ⇝ (makelet xs es' e')"
```

図 8 インライン展開の関係としての定義
Fig. 8 Inline expansion relation.

まず関数本体 *e1* をインライン展開し，*e2* を得る．関数 *f* が再帰的でなく，ヒューリスティックス proper を満たすときは，関数 *f* の定義を，式 *e* をインライン展開するときの関数定義 *D* に加えるようになっている．関数適用の場合，式の関数部分 *f* が変数の場合は関数定義 *D* から *f* の定義を検索する．*f* が展開する関数であった場合は *D* に定義が収められており，*f* の仮引数と本体の組 ab が得られる．その場合，実引数のリストをインライン展開した後，`let` 式を作っている．ここで *makelet xs es e* は，式のリスト *es* を変数のリスト *xs* で順次束縛し，*e* を評価する式を作る補助関数である．

この定義では，展開した関数の定義もそのまま残すようになっている．展開した定義を取り除いてしまうと，プログラムの実行時の環境も変化してしまい証明が複雑になるからである．インライン展開では展開された関数の定義も残ってしまうが，不要コードの除去のフェーズでこれらの関数は取り除かれる．

5.2 インライン展開の検証

前節の定義に従ってインライン展開の正しさの検証を行うと，証明に関係しない条件などが現れて証明が複雑になってしまう．そこで，我々はインライン展開を演繹体系として定義し，この演繹体系をもとに正しさの証明を行った．この体系では，束縛変数の名前の衝突をふせぐ条件などが加わっており，上で定義したインライン展開を行う関数 *ie* よりも条件が多い．そのため，インライン展開を行う関数 *ie* で求めた結果がこの関係に含まれることを示すときには，変換する式に一定の条件を仮定した．

インライン展開が正しく行われるためには，自由変数の捕捉が起きないことなどを保証しなければならない．そのため，関数を展開する時点で束縛されている変数の情報が必要となる．インライン展開を表す関係は，関数定義，束縛されている変数のリスト，変換前の式，変換後の式の 4 項関係として定義した．この関係を *D,X* ⊢ *e* ⇝ *e'* で表す．定義の一部を**図 8** に示す．

定義 *fun1* は関数定義式の場合で，その関数を展開する候補に入れる場合を表している．変換では，まず関数定義 *(f, xs, e1)* の本体 *e1* を *e2* に変換する．このとき，束縛されている変数のリストに関数名 *f* と仮引数 *xs* を加えている．そして最終的に評価される式 *e* は，関数 *f* を追加した定義のもとで *e'* に変換されている．この定義にはインライン展開が正しく行われるために必要な条件も加えられている．たとえば，条件 *disjoint xs X* は引数 *xs* が現在束縛されている変数のリスト *X* と異なることを表している．ここで，*disjoint* は 2 つのリストの要素が互いに素であることを表す述語，*set* はリストを集合に変換する関数である．

定義 *app1* は関数を展開する場合を表しており，*ie* の定義と似た定義になっている．ここでも，定義 *fun1* と同じようにいくつか条件が加えられている．1 つは展開される関数の引数 *xs* が束縛されている変数のリスト *X* と異なるという条件である．この定義には，もう 1 つ条件 *nodup xs* が加わっている．我々のソース言語には，Scheme の `let` 式に対応する，複数の式を同じ環境で評価して束縛する式がない．したがって，式 *es* を 1 つずつ評価・束縛していく．その結果，式 *es* を順に評価するときに環境が変数のリスト *xs* で順に拡張されてしまう．この条件は式 *es* を評価する環境が変わらないことを保証するために必要となる．

インライン展開の正しさの証明はこの演繹体系に基づいて行った．次にクロージャ変換の場合と同様に，証明で利用する値の対応関係を定義する．値の対応関係はヒープのもとで定義され *H* ⊢ *v* ≈ *v'* で表す．

```
inductive valeq
  intros
  "⟦D,(rev xs@map fst E)⊢ e ⇝ e';
    nodup xs; disjoint xs (map fst E);
    E ⊆a E'; H,E⊨D⟧ ⟹
 H ⊢ (HeapCls f E xs e) ≈ (HeapCls f E' xs e')"
```

この定義はクロージャの対応を表している．まず，

```
inductive defenv
  intros
    "(∀f xs e'. D(f) = Some (xs,e') ⟶
      (∃l E1 D1 e. lookup E f = Some (LocVal l) ∧
                   H(l) = Some (HeapCls f E1 xs e) ∧
                   E1 ⊆a E ∧
                   H,E1⊨D1 ∧
                   D1,(rev xs@map fst E1)⊢ e ⇝ e'))
       ⟹ H,E⊨D"
```

図 9 ヒープ，環境，関数定義の対応関係
Fig. 9 Relation on a heap, an environment and a function definition.

式 *e'* が式 *e* をインライン展開したものであるという条件が必要となる．2 行目の条件は帰納法を利用するときに必要となる．条件 $E \subseteq_a E'$ は環境 *E'* が環境 *E* の拡張になっていることを意味する．インライン展開された式の評価を考えると，展開された式では `let` 式で引数を評価・束縛する．したがって，展開された式を評価するときは，展開される前の環境よりも多く変数を束縛することになり，このような条件が必要となる．条件 $H,E \models D$ は，関数定義 *D* とヒープ *H* に置かれたクロージャが対応するという関係を意味している．この関係は環境，ヒープ，関数定義の三項関係で**図 9** のような定義になっている．この関係は，関数定義 *D* に現れる *f* の定義と環境から得られる *f* のクロージャが対応することを示している．また条件 $E1 \subseteq_a E$ では，展開した式を評価するときの環境 *E* が，クロージャが作られたときの環境 *E1* の拡張になっていることを意味している．この条件は展開した式を評価するときの環境 *E* に式 *e* の自由変数が含まれることを保証するために必要となる．

この関係 $H,E \models D$ は次の理由から必要となる．証明において関数適用の場合を考えるとき，関数を展開しない場合は証明に必要な条件が値の関係 $H \vdash v \approx v'$ から導かれる．しかし，関数を展開した場合は対応するクロージャがないので，必要な条件を得ることができない．この関係は変換前のクロージャと関数定義から必要な条件を得るために用いられる．

最終的に得られた定理は次のものである．

```
theorem assumes
  "([], empty, e, H, v) ∈ eval"
  "empty,[]⊢e ⇝ e'"
shows
  "∃H'. ([], empty, e', H',v) ∈ eval ∧ H ∼ H'"
```

定理は変換前の式 *e* が，空の環境，空のヒープで評価され，ヒープ *H*，値 *v* が得られるならば，*e* をインライン展開した式 *e'* も対応するヒープ *H'* と値 *v* に評価されることを表している．この定理を証明するときも，クロージャ変換の正しさの検証と同じように，より一般的な形の補題が必要となった．証明はソース言語の評価の導出についての帰納法で行われ，1,800 行ほどの長さであった．

最後に，インライン展開を行う関数 *ie* で求めた結果が関係 $D,X \vdash e \rightsquigarrow e'$ に含まれることの証明を行った．この証明では，ソース言語の式が一定の条件を満たすときに，インライン展開の関係に含まれることを示した．最終的に次の定理が得られた．

```
theorem assumes "nodup (bvl e)" shows
  "get empty,[] ⊢ e ⇝ ie empty e"
```

ここで *empty* は空の辞書を表している．定理の中で *get empty* となっているのは関数 *ie* と演繹体系の型を合わせるためである．関数 *bvl* は式に現れる束縛変数をリストとして集める関数である．定理は，ソース言語の式に現れる束縛変数がすべて異なるならば，インライン展開を行う関数 *ie* の結果がインライン展開の関係に含まれることを示している．この条件も変数名の付けかえフェーズを利用することにより満たすことができる．

6. その他のプログラム変換

この章ではここまでで説明されなかったコンパイラのフェーズについて，その概要とポイントとなった部分について簡単に説明していく．

6.1 Java 仮想機械とコード生成の形式化

コード生成のフェーズでは，クロージャ変換された言語を Java 仮想機械のアセンブリ言語に変換する．我々が Java 仮想機械をターゲットに選んだ理由は 2 つあげることができる．1 つ目は，Java 仮想機械では，メモリ管理が行われる，局所変数の数に制限がないなど実際の計算機よりも抽象度が高いという理由があげられる．このことにより，コード生成を行う際にメモリ管理や変数の割付けなどを考慮に入れずにすむのでコンパイルがより単純になる．2 つ目は Java 仮想機械の仕様が明確であるということである．Java 仮想機械はその仕様が定められているのに加え，定理証明システムでの形式化がすでに行われている[8),9)] ため仕様に関して不明な点がほとんどない．

我々は，Java 仮想機械の形式化を独自に行った．既存の Java 仮想機械の形式化を利用した場合，コードを生成するのに必要な命令が不足していたり，コード生成の正しさの検証に不必要な部分が多く証明が複雑になったりするといった問題点があったからである．我々の形式化では Java 仮想機械の機能を制限し，既存の形式化よりも単純な定義をした．これによりコンパイルに関係しない条件が減り，証明を単純にすることができた．

我々が定義した仮想機械は基本的には Java 仮想機械のサブセットであり，コード生成に必要な機能のみを形式化したものである．主な制限を以下にあげる．

- 型情報を扱わない．
- クラスは 1 つしかメソッドを持たない．
- インタフェース，例外，スレッド，継承がない．

Java 仮想機械と異なる部分としては，分岐命令が後ろ向きにジャンプしないことなどがあげられる．

我々のコンパイラは，言語のプリミティブを実現するために，Java で記述されたライブラリを利用している．ライブラリの検証を行うには，ライブラリを Java 仮想機械のアセンブリ言語で記述し検証する方法とライブラリを Java で記述し検証する方法が考えられる．しかし，ライブラリを Java 仮想機械のアセンブリ言語で記述するのは，実際のコンパイラでは現実的ではない．また，ライブラリを Java で記述した場合は，どのようなコードが生成されるかはコンパイラに依存する．したがって，Java のコンパイラを形式化しなければライブラリの検証はできない．現実的なプログラミング言語は，プリミティブやライブラリに複雑な手続きもあり，その正しさの証明は単純ではない．そのため，コンパイラの検証とライブラリの検証は別の問題として考え，ライブラリの検証は行わなかった．

コード生成でこのライブラリを利用するために，仮想機械の命令セットにライブラリを呼び出す命令を加えた．ライブラリ呼び出し命令は，定義したライブラリがヒープなどに与える影響を記述することによって定義している．コード生成の正しさは，ライブラリがこの定義に沿っていることを前提に証明している．

コード生成では他の関数型プログラミング言語から Java 仮想機械へのコンパイラ[10),11)] と同様の処理を行っている．この変換ではソース言語の関数を Java のクラスに変換する．クラスは関数適用を行うメソッドと自由変数を格納するフィールドを持つ．ソース言語の関数定義はクラスに対応し，ソース言語のクロージャはそのクラスのインスタンスに対応する．紙面の都合上，ここでは変換の詳細については述べない．詳しくは実際のスクリプトか論文 2) を参照してほしい．

6.2 変数名の付けかえ

変数名の付けかえのフェーズでは，構文解析されたソースの式に現れる変数を新しいもので置きかえていく．コンパイラのさまざまなフェーズを形式化・検証した結果，我々の形式化では変換が正しく行われるためには変数に関する条件が必要になることが分かった．たとえば，インライン展開では束縛変数の名前がすべて異ならなければならない．このような条件を満たすために変数名の付けかえのフェーズを導入した．変数名の付けかえは次の型を持つ原始帰納関数として定義できる．

```
consts
  ren :: "[('a, nat) dic, nat, 'a exp]
                                  ⇒ nat * nat exp"
```

関数 *ren* は第 1 引数として付けかえの前後の変数の対応関係を保持するデータ構造を受け取る．第 2 引数はこれまで変換で使用された自然数の最大値をとる．変換ではこの自然数をインクリメントした値を使用していく．そして，第 3 引数として式を受け取り，変換で使用された自然数の最大値と式を得る．

検証では，変数名の付けかえで評価結果が変わらないことを示した．また，名前の付けかえによってコンパイラの他のフェーズで必要となる条件を満たすようになることを示した．例として，下の定理は式 *e* の変数名を付けかえたならば束縛変数がすべて異なることを示している．

```
theorem
  "nodup (bvl (snd (ren E n e)))"
```

コンパイラの処理の中で，変数名の付けかえはインライン展開の前後で行われる．インライン展開の後に変数を付け直すのは，インライン展開では関数定義がコピーされてしまう可能性があり，クロージャ変換で必要な，すべての関数名が異なるという条件を満たさなくなってしまうからである．

6.3 不要コードの除去

不要コードの除去のフェーズでは，式を評価するのに必要のない関数を除去する．このフェーズの形式化では，重複した計算を避けるように関数の定義を工夫した．

不要コードを除去する関数 *dce* は原始帰納関数として以下のように定義できる．

```
consts
  dce :: "'a exp ⇒ 'a exp"
primrec
...
  "dce (FunExp f xs e1 e) =
    (let e' = dce e in
      if f ∉ fv e then
        e'
      else
        (FunExp f xs (dce e1) e'))"
...
```

ここでは，関数定義式の場合を扱っている．関数 *f* が式 *e* の中で使用されていない場合は *f* の定義を除去している．この単純な定義では関数定義式が現れるたびに自由変数を計算してしまう．そこで，プログラムの自由変数を計算しておく関数 *fvl* を定義し，関数 *dce* ではその情報をもとに処理するように変更した．関数 *fvl* は，自由変数を求めるのと同時に関数定義式での自由変数を記録する．つまり，関数定義式 *FunExp f xs e1 e* が現れるたびに，自由変数 *fv e* を記録していく．関数 *fvl* は以下の型を持つ原始帰納関数として定義する．

```
consts
  fvl :: "'a exp ⇒
                ('a, 'a rbset) dic * 'a rbset"
```

この関数は式を引数として受け取り，関数定義ごとの自由変数を記憶したデータ構造と自由変数を返す．不要コードを除去する関数は，関数 *fvl* で求まったデータ構造を利用するよう変更する．

```
consts
  dce2 :: "[('a, 'a rbset) dic, 'a exp]
                                  ⇒ 'a exp"
```

この変更により，自由変数を重複して計算することがなくなる．そして，関数 *dce* と *dce2* が同値であることを証明した．

```
theorem assumes "nodup (fn e)"
  shows "dce e = dce2 (fst (fvl e)) e"
```

定理には条件 *nodup (fn e)* が加わっている．この条件は関数 *fvl* で求まったデータで，関数名と関数定義式を同一視しているため必要となる．異なる関数定義式に同じ関数名が使用されている場合には，これらの関数は同値にはならない．

6.4 スタックサイズの計算

スタックサイズの計算のフェーズは，仮想機械のアセンブリプログラムの各関数がどれだけスタックを使用するかを計算するフェーズである．求まったスタックサイズは仮想機械のアセンブリ言語に出力され，コードを実行するときにそのスタックサイズが確保される．このフェーズではスタックの使用量を計算する関数を形式化し，計算によって求まったスタックの使用量をプログラムが実際に超えないことを証明した．今回形式化した仮想機械では，後ろ向きのジャンプ命令がないので，プログラムにループが現れることがない．この制限により，スタックの使用量を計算するフェーズは単純な定義にすることができた．スタックサイズを計算する関数は下の型を持つ再帰関数として定義した．

```
consts
  cnt_code :: "(nat * instr list) ⇒ nat"
```

ここで，*instr* は Java 仮想機械のアセンブリ命令である．関数は引数として，現在のスタックの値とプログラムを受け取る．検証は，プログラムを実行したときにスタックの大きさがある数を超えないという関係を定義し，プログラムの実行時のスタックの大きさが関数 *cnt_code* で求まった値を超えないことを示した．しかし，上で定義した関数 *cnt_code* は条件分岐が起こるたびに両方の場合について計算する定義になっており，指数オーダの計算量がかかった．そこで，動的計画法を利用した関数を定義し関数 *cnt_code* との同値性を示した．

7. コンパイラの実行と評価

コンパイラの形式化とその正しさの証明は，約 2 万行の証明スクリプトからなる．**表 1** は主なモジュールのそれぞれの行数，証明した補題の数，証明にかかったステップ数を表している☆．これらの証明スクリプトから，HOL のコード生成機能を使用して，約 2,400 行の Standard ML プログラムを得ることができた．HOL が生成したコンパイラプログラムに構文解析プログラムや出力ルーチンといった入出力のプログラムを加え，実行可能なコンパイラを実現した．また，Java で記述されたプリミティブライブラリ，Scheme で記述されたライブラリも与えた．Standard ML，Java および Scheme で書かれたコードはそれぞれ約 1,200 行，150 行，220 行である．得られたコンパイラでいくつかの Scheme プログラムをコンパイル・実行し，コンパイル時間，コンパイラが生成したプログラムの実行時間について既存のコンパイラと比較した．

7.1 補助プログラム

我々はコンパイラを実現するために，構文解析プログラムや出力ルーチンなどのプログラムを実装した．構文解析部分では，構文解析のほかにマクロ展開を行

☆ 行数，補題の数，証明のステップ数はいずれも定義や証明のしかた，HOL のモジュールの分けかたなどにより大きく変わる．

表 1 証明スクリプトに関するデータ
Table 1 Data on our proof scripts.

HOL モジュール	行数	補題の数	ステップ数
ソース言語の仕様	578	12	39
クロージャ変換された言語の仕様	414	19	25
Java 仮想機械の仕様	1,136	64	115
クロージャ変換	1,306	41	254
コード生成	3,147	91	534
インライン展開	1,849	40	404
不要コードの除去	1,745	70	419
変数名の付けかえ	1,379	59	294

表 2 コンパイル時間（単位：秒）
Table 2 Compilation time (in seconds).

プログラム	行数	vc total	vc w/o jasmin	bigloo	kawa
symbdiff	379	0.76	0.19	0.11	0.67
boyer	551	0.84	0.23	0.08	0.67
sets	349	0.60	0.06	0.16	0.64
takr	520	1.00	0.13	0.17	0.67
art	3,008	4.70	3.83	0.95	0.79

うプログラムや相互再帰関数をまとめるプログラムなど，我々のコンパイラの抽象構文に合わせるためのプログラムを用意した．出力部分については，Java 仮想機械のバイトコードアセンブラである Jasmin[1)]の形式に合わせて出力を行った．今回形式化した仮想機械の命令は Java 仮想機械の命令と単純に対応がつく．ライブラリ呼び出し命令は，対応するメソッド呼び出しとして出力される．またスタックサイズを計算するプログラムもここで使用し，求めたスタックサイズの上限を Jasmin の疑似命令として出力した．

7.2 コンパイラの実行と評価

コンパイル時間，生成されたプログラムの実行時間について既存の Scheme から Java 仮想機械にコンパイルするコンパイラと比較した．コンパイル時間を比較するために用いたプログラムは比較的長いプログラムを選んだ：記号微分（symbdiff），トートロジーチェッカ（boyer），竹内の関数（takr），平衡 2 分木を用いた集合（sets）などである．実行時間を比較するときに用いたプログラムは実行時間が長いものを選んだ：フィボナッチ関数（fib），エラトステネスのふるい（sieve），8 クイーン（queens），トートロジーチェッカ，パズル（puzzle）などである．これらのプログラムは今回形式化したコンパイラでコンパイルできるよう，若干の変更を行った．例として，これらのプログラムでは `set!`関数の代わりに `set-car!`関数を使用するよう変更した．

比較対象としたコンパイラは Bigloo と Kawa である．Bigloo は C を用いた Scheme の実装ですぐれた最適化を行う．一方，Kawa は Java による実装で標準的な実装になっている．計測は Pentium 4（2.2 GHz），512 MB の計算機で行った．実験で使用したプログラムは，Poly ML 4.1.3，Blackdown JDK 1.4.1，Bigloo 2.6d，Kawa 1.7 である．

表 2 はコンパイル時間の計測結果である．本研究で構築したコンパイラは “vc”（verified compiler）で表記されている．“vc total” の列はバイトコードアセンブラ Jasmin の実行時間を含むコンパイル時間で，“vc w/o jasmin” の列は Jasmin の実行時間を除いたものを示している．“vc total”，“vc w/o jasmin” はともにインライン展開も行った場合のコンパイル時間である．我々のコンパイラは表中の上 4 つのプログラムについては，Jasmin アセンブラの実行時間を入れた場合は Kawa と，Jasmin の実行時間を除いた場合は Bigloo と同等の実行時間が得られた．プログラム “art” は実験のために書いたプログラムで多くの変数を含むプログラムである．このプログラムのコンパイル時間は既存の 2 つのコンパイラに比べ 3，4 倍長くかかっている．この結果は国際会議 APLAS での発表時に比べ改善されてはいるが，他のコンパイラに並ぶまではいたらなかった．前回の計測の時点では，コンパイラの形式化で集合などの表現にリストを使用しており，その実行時間は既存のコンパイラよりも 10 倍近く時間がかかっていた．また，インライン展開やスタックサイズの計算なども行っていなかった．今回の形式化では 2 色木などの効率の良いデータ構造の導入を行い，重複した計算を行わないアルゴリズムを

表 3 実行時間（単位：秒）
Table 3 Execution time (in seconds).

プログラム	vc	vc opt	bigloo	bigloo opt	kawa
fib	7.00	5.66	8.69	0.49	11.69
sieve	114.69	115.44	15.71	5.00	164.34
queens	0.45	0.77	0.58	0.22	3.58
boyer	2.17	1.54	0.37	0.21	9.83
puzzle	557.72	416.66	308.28	51.35	2,021.17

採用することによって．コンパイラの実行時間はかなり改善された．クロージャ変換を不要コードの除去のフェーズで行ったように自由変数をあらかじめ計算するように変更することでコンパイル時間をさらに短縮できるのではないかと考えている．

表 3 は生成したプログラムの実行時間である．表の “vc opt” はインライン展開を行った場合，“bigloo opt” はプログラムを Bigloo のベンチマークモードでコンパイルした場合の実行時間である．多くのプログラムでは，最適化を行っていない Bigloo や Kawa の実行時間と同じような結果が得られた．いくつかのプログラムでは我々のコンパイラでより良い実行結果が得られている．これは，Bigloo や Kawa は Scheme の言語仕様をすべて扱っているのに比べ，我々が形式化したコンパイラは Scheme のサブセットしか扱っておらず，コンパイルがより単純になっていることからだと考えている．しかしすべてのプログラムで，最適化を行った Bigloo のプログラムには大きく実行時間が離されている．

8. 関連研究

本研究と同じ手法を用いた研究が Berghofer らによって行われている[12),13)]．彼らは Isabelle/HOL 上で Java コンパイラを形式化・検証し，Isabelle/HOL のコード生成機能を利用して実行可能なコンパイラを構築している．この研究は，我々のものとほぼ同時期に行われたものであるが，形式化の方針や手法が異なる．彼らの研究では，検証を念頭に置いて形式化が行われており，どのようなコンパイラプログラムが生成されるかについては考慮していない．コンパイラの仕様は単純なデータ構造を採用しており，我々が使用しなかった帰納的集合もコンパイラの形式化で用いられている．帰納的集合のコード生成で得られたプログラムは HOL の定義と意味のギャップがあるだけでなく効率的に実行されない．また，論文では得られたコンパイラの実験や評価については述べられていない．

コンパイラの検証に関する研究は数多く行われているが，その中で比較的有名なのが，Oliva らによる VLISP コンパイラである[14)]．VLISP コンパイラはシステムプログラミングのために設計された Scheme の方言をコンパイルする．検証では定理証明システムは利用されておらず，手による証明を行っている．コンパイラはこの形式化に基づいて実装されているが，実装についての検証はなされていない．

定理証明システムを用いたコンパイラの検証の事例には，Boyer-Moore の定理証明システム[15)] を用いた研究がある[16),17)]．Boyer-Moore の定理証明システムは一階論理の定理証明システムで，記述言語は Lisp をもとにしている．したがって，形式化されたコンパイラは Lisp プログラムとして実行することができる．しかし，形式化・検証において一階の論理でしか記述できないというデメリットがある．

Stepney による研究では，仕様記述言語 Z を用いてコンパイラの形式化を行い，その正しさの検証を手による証明で行った[18)]．そして，この仕様を Prolog プログラムに変換して実行可能なコンパイラを構築している．Stringer-Calvert はこの仕様を定理証明システム PVS[19)] に変換し，検証を行った[20)]．この 2 つの研究でも検証のしやすさが優先されており，得られるコンパイラプログラムについては考慮されていない．

検証された仕様からコンパイラのプログラムを生成するという手法は，Curzon による研究でもとられている[21)]．この研究では，Vista と呼ばれる高機能なアセンブリ言語のコンパイラの検証を Gordon による定理証明システム HOL[22)] で行っている．そして，検証されたコンパイラの仕様を Standard ML プログラムに変換し，実行可能なコンパイラを構築している．しかし，形式化ではコンパイラの検証に主眼を置いており，得られたコンパイラの実験や評価はなされていない．

9. まとめ

我々は，定理証明システム Isabelle/HOL を用いて Scheme の構文を持つ関数型プログラミング言語から Java 仮想機械へのコンパイラを形式化・検証した．コンパイラの形式化・検証ではクロージャ変換やインラ

イン展開など，これまで定理証明システムでは検証されていなかった高度なプログラム変換を扱った．さらに，Isabelle/HOL のコード生成機能を利用し，いくつかのプログラムを補うことにより，実行可能なコンパイラを構築した．コンパイラの形式化では，生成されるコンパイラの効率も考慮にいれ，効率の良いデータ構造やアルゴリズムを採用した．得られたコンパイラは数百行程度の Scheme プログラムをコンパイル・実行することができ，そのコンパイル時間，生成されたプログラムの実行時間ともに既存のコンパイラに劣らないものだった．

今回，我々は仮想機械へのコンパイラを形式化・検証したが，本研究の手法はネイティヴコードへのコンパイルにも適用できる．ネイティヴコードへのコンパイルではデータフロー解析やグラフの彩色によるレジスタ割付けなどに取り組む必要がある．これらの手法は一部，すでに HOL による形式化が行われている[9),23)]．これらの形式化を参考にして実行可能なプログラムとしての形式化を行うことも可能であると考える．

参 考 文 献

1) Meyer, J.: Jasmin Home Page. http://mrl.nyu.edu/~meyer/jasmin/
2) Okuma, K. and Minamide, Y.: Executing Verified Compiler Specification, *Proc. 1st Asian Symposium on Programming Languages and Systems* (*APLAS*), Lecture Notes in Computer Science, Vol.2895, pp.178–194, Springer-Verlag (2003).
3) Nipkow, T., Paulson, L.C. and Wenzel, M.: *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Lecture Notes in Computer Science, Vol.2283, Springer-Verlag (2002).
4) Wenzel, M.: Isar — A Generic Interpretative Approach to Readable Formal Proof Documents, *Proc. International Conference on Theorem Proving in Higher Order Logics*, pp.167–184, Springer-Verlag (1999).
5) Berghofer, S. and Nipkow, T.: Executing Higher Order Logic, *Proc. International Workshop on Types for Proofs and Programs*, Lecture Notes in Computer Science, Vol.2277, pp.24–40, Springer-Verlag (2002).
6) Hannan, J.: A Type System for Closure Conversion, *Proc. Workshop on Types for Program Analysis*, pp.48–62 (1995).
7) Minamide, Y., Morrisett, J.G. and Harper, R.: Typed Closure Conversion, *Proc. Symposium on Principles of Programming Languages*, pp.271–283 (1996).
8) Klein, G. and Nipkow, T.: Verified Lightweight Bytecode Verification, *Concurrency and Computation: Practice and Experience*, Vol.13, No.13, pp.1133–1151 (2001).
9) Klein, G. and Nipkow, T.: Verified Bytecode Verifiers, *Theoretical Computer Science*, Vol.298, No.3, pp.583–626 (2003).
10) Benton, N., Kennedy, A. and Russell, G.: Compiling Standard ML to Java Bytecodes, *Proc. ACM SIGPLAN International Conference on Functional Programming* (*ICFP '98*), pp.129–140 (1998).
11) Bothner, P.: Kawa — Compiling Dynamic Languages to the Java VM, *Proc. USENIX 1998 Technical Conference, FREENIX Track*, New Orleans, LA, USENIX Association (1998).
12) Berghofer, S. and Strecker, M.: Extracting a Formally Verified, Fully Executable Compiler from a Proof Assistant, *Proc. 2nd International Workshop on Compiler Optimization Meets Compiler Verification* (*COCV*), Electronic Notes in Theoretical Computer Science, Vol.82, pp.33–50, Elsevier (2003).
13) Strecker, M.: Formal Verification of a Java Compiler in Isabelle, *Proc. Conference on Automated Deduction* (*CADE*), Lecture Notes in Computer Science, Vol.2392, pp.63–77, Springer-Verlag (2002).
14) Oliva, D.P., Ramsdell, J.D. and Wand, M.: The VLISP Verified PreScheme Compiler, *Lisp and Symbolic Computation*, Vol.8, No.1/2, pp.111–182 (1995).
15) Boyer, R.S. and Moore, J.S.: *A Computational Logic Handbook*, Academic Press (1988).
16) Young, W.D.: A Verified Code Generator for a Subset of Gypsy, Technical Report 33, Computational Logic Inc. (1988).
17) Moore, J.S.: A Mechanically Verified Language Implementation, Technical Report 30, Computational Logic Inc. (1988).
18) Stepney, S.: *High Integrity Compilation: A Case Study*, Prentice-Hall (1993).
19) Rushby, J., et al.: The PVS Specification and Verification System. http://pvs.csl.sri.com/
20) Stringer-Calvert, D.W.: Mechanical Verification of Compiler Correctness, Ph.D. Thesis, Department of Computer Science, University of York (1998).
21) Curzon, P.: A Verified Vista Implementation Final Report, Technical Report 311, University of Cambridge Computer Laboratory (1993).
22) Gordon, M.J.C. and Melham, T.F.: *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University

Press, Cambridge (1993).
23) Bauer, G. and Nipkow, T.: The 5 Colour Theorem in Isabelle/Isar, *Theorem Proving in Higher Order Logics*, Carreño, V., Muñoz, C. and Tahar, S. (Eds.), Lecture Notes in Computer Science, Vol.2410, pp.67–82, Springer-Verlag (2002).

(平成 16 年 9 月 30 日受付)
(平成 16 年 12 月 27 日採録)

大熊　浩示

1999 年筑波大学第三学群情報学類卒業．2001 年同大学大学院工学研究科にて修士（工学）取得．現在，同大学院システム情報工学研究科コンピュータサイエンス専攻在学中．定理証明システムによるソフトウェア検証の研究に従事．

南出　靖彦

1993 年京都大学大学院理学研究科数理解析専攻修士課程修了．同年同大学数理解析研究所助手．1999 年筑波大学電子・情報工学系講師．現在，筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻講師．博士（理学）．プログラミング言語およびソフトウェア検証に興味を持つ．