平成28年度　修士論文

ソース-to-C型トランスレータ向けデバッガの実装手法

電気通信大学大学院 情報システム学研究科
情報システム基盤学専攻

1553011　柳 震

指導教員
小宮 常康 准教授
本多 弘樹 教授
新谷 隆彦 准教授

平成29年2月8日

目 次

図一覧

ソースコード目次

第1章

序論

1.1 研究背景

プログラムは，人々の言語と同じように，異なるプログラミング言語間で，通訳，変換できる．高級言語のプログラムは直接にマシンで実行できないので，コンパイラあるいはインタープリタなどの言語処理系を使って低級言語へ変換する必要がある．場合によっては，プログラミング言語の移植性を向上させるためや異なるプラットフォームで実行できるようにするため [1]，サーバ側の言語で書くプログラムをブラウザ側で実行できるようにするために [2]，ある高級言語を別の高級言語へ変換する場合もある．

ソース-to-ソース型トランスレータは高級言語（ソース言語）を別の高級言語（ターゲット言語）へ変換する言語処理系である．そのようなトランスレータの開発者はよく C 言語をターゲット言語としてソース-to-C 型トランスレータを作る．その理由が 3 つある [3]：

- ほとんどのプラットフォームで実行できる
- 汎用性が高い
- 別のプログラミング言語をエミュレーションしやすい

特に新しく設計されるプログラミング言語に対して，C 言語へ変換するトランスレータを作れば，その言語は直接たくさんのプラットフォームで実行できて，言語用の VM などを用意する必要がない [4].

プログラムのバグと欠陥を発見して，修正するために，プログラムをデバッグすることが必要である．よく使われるプログラミング言語では，その言語用のデ

バッガが存在する．そのため，Java 言語を C 言語へ変換するトランスレータを自分で作ったとしても Java 言語のデバッガは自分で作ることなく，既存のものを利用すれば良い．

しかし，ソース言語が新しく設計された言語の場合や既存の言語を拡張した言語の場合，その言語用のデバッガは存在しておらず，トランスレータの開発者はトランスレータのソース言語用デバッガを用意することが望まれる．そのソース言語で書かれたプログラムをターゲット言語である C 言語のデバッガでデバッグすることもできるが，デバッガの利用者にとって，望ましいことではない．そのために，ソース言語用のデバッガを開発して，ソース言語レベルのデバッグを実現できることが望ましいが，トランスレータの開発者にとって，新しい言語用のデバッガを開発するのは大変である．

1.2　研究目的

本研究では，C 言語のデバッガを用いてソース-to-C 型トランスレータのソース言語レベルのデバッガを実現する手法を提案する．その提案手法では，ソース-to-C 型トランスレータの改造と C 言語のデバッガの拡張を通じて，ユーザが入力するソース言語レベルのデバッグコマンドと引数を，対応する C 言語レベルのデバッグコマンドと引数へ変換する．そして，C 言語のデバッガで解釈して，実行する．

本研究では C 言語のデバッガとして LLDB を用いた．本論文では Scheme 言語 [5] をソース言語として，Scheme 言語を C 言語へ変換する Hobbit コンパイラ [7] で生成されるコードを例として，研究手法を説明する．

本研究の実装では Scheme 言語と Hobbit コンパイラだけを対象としたが，本手法は Hobbit コンパイラ以外のソース-to-C 型トランスレータにも適用できる．

1.3　論文の構成

本論文は，以下のように構成される．

- 第 2 章 関連技術
- 第 3 章 基本アイデア
- 第 4 章 実装と実行例

- 第 5 章 結論

第 2 章では関連技術として，既存のソース-to-C 型トランスレータがどのように C コードを生成しているのかを紹介する．また Scala 言語と Clojure 言語のデバッガを実現する技術を紹介する．第 3 章では C 言語用デバッガによってトランスレータのソース言語レベルデバッガを実現する手法を説明する．第 4 章では具体的な実装について述べる．第 5 章では本研究の成果をまとめ，結論を述べる．そして，今後の展望について議論する．

第2章

関連技術

2.1 関数閉包を備える言語とC言語の違い

近年のプログラミング言語には，関数閉包 (クロージャ，closure) をサポートするものが多い．関数閉包をサポートすると，関数は第一級オブジェクト (first-class object)*となり，また関数によるネストしたスコープが実現できるようになる [6]．すると，こうした言語の変数や関数は意味レベルでも実装レベルでも C 言語のものとは大きく異なるものとなる．以下では，関数閉包をサポートする言語として Scheme 言語を用いて，関数閉包と，関数をサポートする言語における変数のスコープと寿命について説明する．

2.1.1 関数閉包

関数閉包はプログラミング言語における関数オブジェクトの一種であり，あるスコープの中で関数がに定義される時に生成される [8]．

`(let ((y 4)) ((lambda (x) (* x y)) 3))` という let 式を評価する時には，システムは lambda 式 `(lambda (x) (* x y))` が局所変数 y のスコープの中にあって，その局所変数 y の値が 4 だということを知っている．それで，その情報を使って，`(* x y)` の値 (12) を求めることができ，let 式の値 (12) も求めることができる．

このように lambda 式を与えると，システムは lambda 式が単なるリストではなく，関数を定義するものとして解釈する．そして，この関数を呼び出す時に，あ

*オブジェクトとはプログラミング言語のデータのことであり，第一級オブジェクトとは，オブジェクトを関数に渡すことや返り値として返すこと，他のオブジェクトへ格納することなど，言語の基本操作が許されるオブジェクトのことである．

らゆる必要な情報を lambda 式に付加した関数閉包という関数データを生成して，それを lambda 式の値とする [9].

関数閉包の用途はたくさんある．

- 関数閉包は自分のスコープ以外に宣言される変数を使える．それで，プログラムの大域変数の宣言を減らすことができる．
- 遅延評価を実現できる（実際の計算を値が必要になるまで行わない）ので、制御構造の定義に用いることができる．
- 前回に呼び出されて実行された時の計算結果を内部で保存して，次回に呼び出される時に，前回の結果に対して，さらに同じ計算を行うことができる．

C 言語は大域スコープだけに関数を宣言することが許されるので，ネストしたスコープに宣言される関数閉包をサポートしない．それに対して，Scheme 言語を含む全ての関数型言語と Ruby スクリプト言語，Java 言語，JavaScript 言語などをはじめとする多くのは関数閉包をサポートする．

2.1.2　変数のスコープと寿命

プログラムのオブジェクトに対して，そのオブジェクトを参照することが可能な，プログラムテキストの範囲がある．その範囲はオブジェクトの有効範囲あるいはスコープである [9].

C 言語に対して，大域変数のスコープはプログラムの全体である．大域変数の寿命はプログラムの実行が始まる時から終わる時までである．関数に宣言される局所変数のスコープは，局所変数を宣言する所からその関数の終わりまでの範囲である．局所変数の寿命は宣言される時から関数の実行が終わる時までである．

図 2–1 に示す Scheme プログラムに対して，C 言語と比べて，Scheme などの関数閉包を備える言語は局所変数のスコープと寿命は無限である．関数閉包の中の `n` は関数 `make-accumulate` のパラメータを参照する．つまり，関数閉包の外の変数を参照している．

```
(define (make-accumulate n)
  (lambda()
    (let ((m 1))
      (set! n (+ n m)))
    n))
```

図 2–1: 関数閉包を使う累積計算

その Scheme プログラムの実行は図 2–2 のように示す．毎回に関数閉包を呼び出す時，n は前回の結果（値）を保持する．

```
> (define a (make-accumulate 9))
> (a)
10
> (a)
11
> (a)
12
>
```

図 2–2: make-accumulate の実行

外側の関数 (`make-accmulate`) の実行が終わった後，その返り値である関数閉包がどこかに保存されていれば，それを呼び出すことができる．その際，n の値は関数閉包を生成した当時の値となる．このような振る舞いを実現するために関数閉包の外のスコープの変数の値は関数閉包の中に置く (関数閉包の中に閉じ込める)．

そこで，Scheme などの言語を C 言語へ変換するには，ソース言語の本来のスコープ構造の意味を保存する必要がある．そのために，ソース言語の関数閉包の機能を実現するために，関数閉包の局所変数を C 言語の変数ではなく，違う形式の C 言語コードに変換される可能性がある．

2.2　ソース-to-C 型トランスレータの例：Hobbit

Hobbit コンパイラは Scheme コードを Scheme インタープリタである SCM 上で実行可能なコードに変換するソース-to-C 型トランスレータである [7].

2.2.1　関数閉包に対する処理

ソースコード 2.1 に示す Scheme プログラムでは，関数 b が参照する変数 x は関数 a のパラメータである．

ソースコード 2.1: 関数閉包を使う Scheme プログラム

```
1 (define (a x)
2   (define (b y) x)
3   (b x)
4   (b 2))
```

Hobbit コンパイラはソースコード 2.1 のようなプログラムに対して，まず lambda-lifting という手法で処理する [11].

lambda-lifting はネストしたスコープに定義される関数を大域スコープに移動する手法である．Hobbit コンパイラ以外に，Scheme 言語を C 言語へ変換する CHICKEN コンパイラ [12] などのトランスレータもその方法でネストしたスコープを削除する．ソースコード 2.1 のプログラムを lambda-lifting で処理した結果を，ソースコード 2.2 に示す．

ソースコード 2.2: lambda-lifting された Scheme プログラム

```
1 (define (a x)
2     (b x x)
3     (b x 2))
4 (define (b x y) x)
```

Hobbit コンパイラはまず関数 a のパラメータ x に対応する同名のパラメータを，関数 b に追加する．それで，ソースコード 2.2 のプログラムに変数 x は関数 a のパラメータを参照しなくなり，関数 b のパラメータ x を参照する．そして，関数 b を大域スコープに移動する．最後に C 言語へ変換する．変換された C 言語のプログラムはソースコード 2.3 に示す．

ソースコード 2.3: Hobbit による生成コード

```
1 SCM a(x)
```

```
2  SCM x;
3  {
4    a_fn1(x,x);
5    return a_fn1(x,MAKINUM(2));
6  }
7
8  SCM a_fn1(x,y)
9  SCM x,y;
10 {
11   return x;
12 }
13 ...
```

ソースコード 2.3 の C プログラムによって，Hobbit コンパイラは本来の関数閉包 b を a_fn1 という名前に変換する（8 行目）．本来のプログラムの意味を変えることなく，関数閉包を大域スコープに移動する．

Scheme 言語の関数は C 言語と違い，第一級オブジェクトである．Scheme 言語の関数は他の関数のパラメータあるいは返り値になれる．ソースコード 2.4 に示す Scheme プログラムでは，関数閉包 b は関数 a の返り値としてリターンされる．

ソースコード 2.4: 関数閉包に閉じ込められる変数

```
1 (define (a x)
2   (define (b y) x)
3   (b x)
4   b)
```

ソースコード 2.4 の Scheme プログラムの場合，関数 b が参照する変数 x は外側の関数 a の変数なので，関数閉包に閉じ込められる．ソースコード 2.1 の局所変数 x も関数閉包に閉じ込められているのであるが，そのコード生成においては最適化によって実際には閉じ込めることはしなかった．

ソースコード 2.4 の Scheme プログラムを Hobbit コンパイラで変換した C プログラムをソースコード 2.5 に示す．

ソースコード 2.5: Hobbit による生成コード (関数閉包に閉じ込められる変数)

```
1 SCM a(x)
2 SCM x;
3 {
4   SCM b,newclosure;
5
6   newclosure=makcclo(a_cl1_clproc0,2);
```

```
7    VECTOR_SET(newclosure,MAKINUM(1),x);
8    b=newclosure;
9    apply(b,x,listofnull);
10   return b;
11 }
12
13 SCM a_cl1(closurearg_0)
14 SCM closurearg_0;
15 {
16   SCM closurearg_car_0,x,y;
17
18   closurearg_car_0=CAR(closurearg_0);
19   x=VECTOR_REF(closurearg_car_0,MAKINUM(1));
20   closurearg_0=CDR(closurearg_0);
21   y=CAR(closurearg_0);
22   return x;
23 }
24 ...
```

局所変数 x が関数閉包に閉じ込められるので，Hobbit コンパイラはそれらの変数の値をベクタで保存して，newclosure という関数閉包オブジェクトで保持する (6〜7 行目)．その関数閉包オブジェクトをパラメータとして大域スコープに移動された関数閉包 a_cl1 に送る (14 行目)．局所変数 x はパラメータになった関数閉包オブジェクトのデータを参照する (19 行目)．関数 a は関数 apply で関数閉包の呼び出しを実現する (9 行目)．

そこで，外側の関数の局所変数あるいはパラメータを参照する関数閉包の局所変数に対して，Hobbit コンパイラは lambda-lifting 手法で処理して，変数ではない C コードに変換しない．

2.2.2　set!操作 (代入) に対する処理

Scheme 言語は変数の値を変更する時，(set!　<<変数名>> <<式>>) という形の式を使う．“ 変数名 ” なる名前の変数の値は “ 式 ” の値に変換される．図 2--3 のように示して，x の値を 1 で初期化する．式 (set!　x 2) は，変数 x の値を 2 に変える．set!は副作用 (side-effect) を生じる．

```
> (define x 1)
> (+ x 1)
2
> x
1
> (set! x 2)
> (+ x 1)
3
> x
2
```

図 2–3: set!操作

Hobbit コンパイラは Scheme 言語の局所変数が，set!操作される場合，局所変数を C 言語の変数より複雑なデータ構造や制御構造を含む C 言語コードに変換される．

具体例を示す．ソースコード 2.6 の Scheme プログラムでは，局所変数 x は関数閉包に閉じ込められて，set!操作される．これを Hobbit コンパイラで C 言語へ変換すると，ソースコード 2.7 のようになる．

ソースコード 2.6: set!操作を含む Scheme プログラム

```
1 (define (a x)
2   (lambda (y)
3     (set! x (+ x y))
4     x))
```

局所変数が set!操作され得る場合，変数の値をヒープに保存する．6〜7 行目のように，Hobbit コンパイラは set!操作され得る変数 x の値をベクタで保存して，clargsv_1 という変数で保持する．

局所変数 x は関数閉包に閉じ込められるので，Hobbit コンパイラは関数閉包オブジェクト内に変数の値を保存する．9 行目のように，先ほどの変数 x を保存するベクタは，同じくベクタで作られる関数閉包オブジェクト (newclosure) に格納される [13]．2.2.1 節の例と異なり，変数の値を関数閉包のベクタに直接保存しないのは複数の関数閉包オブジェクト間で同一のコンテナを共有しなければならないからある (set!の結果を他の関数閉包にも反映させる)．

ソースコード 2.7: Hobbit による生成コード (set!操作)

```
1 SCM a(x)
```

```
2  SCM x;
3  {
4    SCM clargsv_1,newclosure;
5
6    clargsv_1=make_vector(MAKINUM(1),EOL);
7    VECTOR_SET(clargsv_1,MAKINUM(0),x);
8    newclosure=makcclo(a_cl1_clproc0,2);
9    VECTOR_SET(newclosure,MAKINUM(1),clargsv_1);
10   return newclosure;
11 }
12
13
14 SCM a_cl1(closurearg_0)
15 SCM closurearg_0;
16 {
17   SCM closurearg_car_0,clargsv_1,y;
18
19   closurearg_car_0=CAR(closurearg_0);
20   clargsv_1=VECTOR_REF(closurearg_car_0,MAKINUM(1));
21   closurearg_0=CDR(closurearg_0);
22   y=CAR(closurearg_0);
23   VECTOR_REF(clargsv_1,MAKINUM(0))=MAKINUM(INUM(VECTOR_REF(
24                            clargsv_1,MAKINUM(0)))+INUM(y));
25   return VECTOR_REF(clargsv_1,MAKINUM(0));
26 }
27 ...
```

第 3 章で詳しく述べるが本研究では，例えば変数 x に対するデバッグコマンドの適用は C 言語のコード VECTOR_REF（clargsv_1,MAKINUM(0)) に対して行うようにする．

2.3　ソース-to-C 型トランスレータの例：Bartlett の Scheme → C

Bartlett の Scheme → C は Scheme 言語の移植性を向上するための Scheme 言語を C 言語へ変換するトランスレータである [3]．

2.3.1　関数閉包に対する処理

ネストしたスコープに対する処理は Hobbit コンパイラと違い，ディスプレイというものを採用する．

関数閉包をサポートする言語は，関数閉包が外側の関数のスコープを参照できるようにするために，内側の関数のフレームに外側の関数のフレームを指すリンクを保存する．そのリンク静的リンクと言われ，静的スコープの実現に使用される．ディスプレイはこの静的リンクをたどるコストを減らす手法である．

ディスプレイは関数閉包が参照する可能性のある全ての関数フレームへ指すポインタを，関数閉包の関数フレーム内の連続する空間で格納するものである．静的リンクと比べて，ディスプレイを利用したら，関数閉包の局所変数の参照が速くなる [10].

Bartlett の Scheme → C トランスレータは外側の関数を参照する関数閉包の局所変数に対して，ディプレイで保存する．ディプレイは大域変数として実現する．ソースコード 2.1 の Scheme プログラムに対して，Bartlett の Scheme → C トランスレータで変換される C プログラムをソースコード 2.8 に示す．

ソースコード 2.8: Bartlett の Scheme → C による生成コード (関数閉包に閉じ込められる変数)

```
1  TSCP  gg_b2004( y2006 )
2          TSCP  y2006;
3  {
4          PUSHSTACKTRACE( "B [inside A]" );
5          POPSTACKTRACE( DISPLAY( 0 ) );
6  }
7
8  TSCP  gg_a( x2002 )
9          TSCP  x2002;
10 {
11         TSCP  SD0 = DISPLAY( 0 );
12         TSCP  SDVAL;
13
14         PUSHSTACKTRACE( t2009 );
15         DISPLAY( 0 ) = x2002;
16         gg_b2004( DISPLAY( 0 ) );
17         SDVAL = gg_b2004( _TSCP( 8 ) );
18         DISPLAY( 0 ) = SD0;
19         POPSTACKTRACE( SDVAL );
20 }
21 ...
```

ソースコード 2.8 のプログラムによって，関数 a (gg_a) の実行中に，まずディスプレイの状態を退避する (11 行目)．そして，局所変数 x の値が display に置かれて

(15 行目)，そのディスプレイをパラメータとして，関数 b (gg_b2004) を呼び出す (16 行目)．実行の最後に，ディプレイを元の状態に戻す (18 行目)．

ソースコード 2.8 では，関数 a の局所変数 x は x2002 という C 言語の変数に変換されているが，x はスコープをまたいで関数 b から参照されるので C 言語のスコープでは不十分である．そのため，ディプレイを用いた管理が行われている．

そこで，Bartlett の Scheme → C トランスレータは関数閉包の外と内の同じ変数を違う形式の C コードに変換する．

2.3.2　set!操作に対する処理

Bartlett の Scheme → C トランスレータは set!操作される変数に対して，ヒープ上のペアデータを一つ用意しそうこへ値を格納する．その理由は Hobbit コンパイラの時と同じである．ソースコード 2.6 の Scheme プログラムを Bartlett の Scheme → C トランスレータで変換した C プログラムをソースコード 2.9 に示す．

ソースコード 2.9: Bartlett の Scheme → C による生成コード (set!操作)

```
1  TSCP  l_l2003( y2004, c2013 )
2          TSCP  y2004, c2013;
3  {
4          TSCP  X4, X3, X2, X1;
5
6          PUSHSTACKTRACE( "l_l2003 [inside A]" );
7          X1 = DISPLAY( 0 );
8          DISPLAY( 0 ) = CLOSURE_VAR( c2013, 0 );
9          X4 = PAIR_CAR( DISPLAY( 0 ) );
10         if  ( BITAND( BITOR( _S2CINT( X4 ),
11                              _S2CINT( y2004 ) ),
12                       3 ) )  goto L2016;
13         X3 = _TSCP( IPLUS( _S2CINT( X4 ), _S2CINT( y2004 ) ) );
14         goto L2017;
15 L2016:
16         X3 = scrt2__2b_2dtwo( X4, y2004 );
17 L2017:
18         SETGEN( PAIR_CAR( DISPLAY( 0 ) ), X3 );
19         X2 = PAIR_CAR( DISPLAY( 0 ) );
20         DISPLAY( 0 ) = X1;
21         POPSTACKTRACE( X2 );
22 }
23
24 TSCP  l_a( x2002 )
```

```
25          TSCP  x2002;
26  {
27          TSCP  SD0 = DISPLAY( 0 );
28          TSCP  SDVAL;
29
30          PUSHSTACKTRACE( t2011 );
31          DISPLAY( 0 ) = x2002;
32          DISPLAY( 0 ) = CONS( DISPLAY( 0 ), EMPTYLIST );
33          SDVAL = MAKEPROCEDURE( 1,
34                                 0,
35                                 l_12003,
36                                 MAKECLOSURE( EMPTYLIST,
37                                              1, DISPLAY( 0 ) ) );
38          DISPLAY( 0 ) = SD0;
39          POPSTACKTRACE( SDVAL );
40  }
41  ...
```

局所変数 x は関数閉包に閉じ込められたので，x は display に置かれるが，関数閉包 (l_12003) に x は set!操作されるので，x の値はペアに格納されて，そのペアへのポインタがディスプレイに格納される（31〜32 行目）．そして，x を set!で変更する際はそのペア（の CAR 部）に格納される値を変更する（18 行目）

そこで，関数閉包を備えるソース言語を C 言語へ変換する時，ソース言語は C 言語にない概念を使う場合，ソース言語のオブジェクトはよく違う C 言語の形式に変換される．そして，ソース言語のスコープのルールは C 言語と違う場合，参照場所によって，同じ変数でも，違う C 言語コードに変換される場合がある．

2.4　関連研究

本研究は Scala 言語のデバッガと Clojure 言語のデバッガの実現方法を関連技術として調査した．

2.4.1　Scala 言語のデバッガ

Scala 言語はオブジェクト指向言語と関数型言語の特性を統合したプログラミング言語であり，バイトコードにコンパイルされて，JVM 上で実行できる [14]．

Scala 言語に対するデバッグ方法が幾つがある．Java 言語の開発環境（例: intellij）で Scala 言語をデバッグできる．

intellij は Java 言語など多言語対応の統合開発環境である．intellij[15] にウェジット sbt(simple-build-tool)[16] をインストールすることを通じて，Java の環境で Scala 言語に対するデバッグを実現できる．

Eclipse のビルドパスの設定を通じて，Scala ライブラリを追加して，Eclipse で Scala プログラムに対するデバッグを実現できる．

そして，intellij と eclipse は Java 言語の開発環境であるので，Scala プログラムをデバッグできるが，デバッガの出力メッセージは Scala 言語レベルではなく，Java 言語レベルである．

intellij 上のでで Scala プログラムに対するデバッガを図 2–4 に示す．

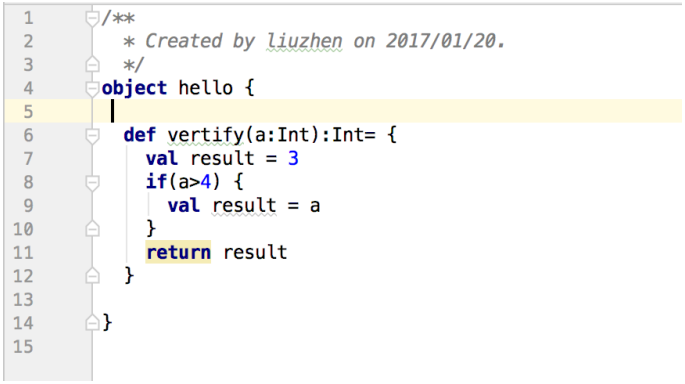

図 2–4: Scala プログラム

その Scala プログラムに main 関数がない．デバッガの出力メッセージを図 2–5 に示す．エラーメッセージによって，その Scala プログラムに main 関数を定義しない．しかし，“ public static void main(String[] args) ”というメッセージは Java 言語レベルである．

図 2–5: Scala プログラムのエラーメッセージ

2.4.2　Clojure 言語のデバッガ

Clojure 言語は Scheme 言語と同じく，Lisp 系言語の方言の一つである [17]．Clojure 言語のプログラムはバイトコードにコンパイラされて，JVM 上で実行できる．Clojure 言語に対するデバッグ方法が幾つがある．

clj-debugger は一つの Clojure 言語用のデバッガである [18]．このデバッガは Clojure 言語をデバッグできるが，デバッガの出力メッセージは Clojure 言語レベルではなく，Java 言語レベルである．例えば，図 2–6 に示す Clojure 言語プログラムをデバッグして，出力メッセージを図 2–7 に示す．変数 z についての情報は Java 言語レベルある．

```
user=> (dotimes [n 5] (debugger.core-test/foo n))

Break from: /Users/razum2um/Code/debugger/src/debugger/core_test.clj:12 (type "(help)" for help)

   17:         e (fn [] nil)
   18:         x "world"
   19:         y '(8 9)
   20:         z (Object.)
=> 21:         ret (break (inc 42))]
   22:     (println "Exit foo with" ret)))
```

図 2–6: breakpoint の設置

```
debugger.core-test/foo:21=> (l)
{x "world",
 a [1 2],
 y (8 9),
 args nil,
 e #<core_test$foo$e__4735 debugger.core_test$foo$e__4735@6735cbba>,
 z #<Object java.lang.Object@2094643d>,
 h {:k "v"},
 b #{4 3},
 d nil}
nil
debugger.core-test/foo:21=> z
#<Object java.lang.Object@3dc76ae9>
```

図 2–7: デバッグメッセージ

Clojure と Java は同じ特性があるので，Clojure が Java Debug Interface を利用して, デバッグすることが可能である．CDT(Clojure Debugging Toolkit) は Java Debug Interface を使う Clojure のコマンドラインデバッガである．CDT を利用して，Clojure プログラムのデバッグを図 2–8 に示す.

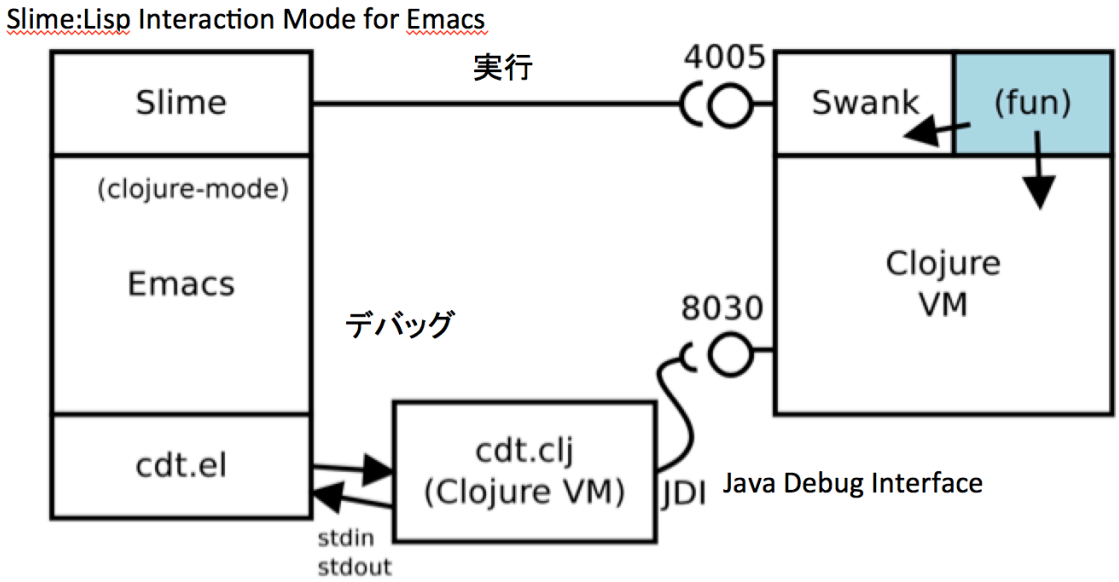

図 2–8: Clojure Debugging Toolkit の概観

図 2–8 は CDT の概観を表す．Swank は CDT のバックエンドである．Swank を通じて，Clojure プログラムに対するステップ実行，式の評価，breakpoint の設置などのデバッグ操作を実現できる．しかし，CDT は Java Debug Interface を使い，デバッグ時のメッセージは Clojure 言語レベルではなく，Java 言語レベルである [19].

Cider-debug を利用して，Clojure 言語の視覚的，対話的なデバッグを実現できる [20]．例えば，Emacs で書く Clojure 言語のプログラムに対して，監視したい関数の中あるいは直後で M-x cider-debug-defun-at-point を実行して，その関数に対する監視が始まる．Cider-debug は Emacs で実行できて，Emacs の命令 (C-u，C-M-x) を利用して，プログラムの式と関数をステップ実行できる．しかし，Cider-debug はデバッガは Emacs を基づくデバッガであり，LLDB などのデバッガと比べて強くない.

Scala 言語と Clojure 言語は Java 環境で実行とデバッグが行える．しかし，その

二つ言語のデバッグ時のメッセージは Java 言語レベルである．

第3章

基本アイデア

デバッガに対するユーザからの入力は，コマンドと引数からなる．例えば，C 言語の変数 x に対するプリント操作では，C 言語デバッガに“ print x ”と入力する．print はデバッガのコマンドであり，x はデバッガの引数である．

C 言語のデバッガを用いてトランスレータのソース言語レベルデバッガを実現するために，ソース言語レベルのデバッガコマンドと引数が入力されると，それを C 言語レベルのデバッガコマンドと引数へ自動的に変換するようにする．そのために，以下の手法が必要となる：

- ソース言語のコード片と C 言語のコード片の対応関係を表す手法
- ソース言語レベルのデバッガコマンドと引数を受け取り，上記の対応関係を用いて，C 言語レベルのデバッガコマンドと引数に変換する手法

本研究では，ソース言語のコード片として関数名と変数名のみを扱う．この他にコードの行番号の対応関係などが考えられるが本研究では扱わない．

3.1 対応関係

トランスレータのソース言語とターゲット言語である C 言語の間で識別子を表す規則に違いがある場合，識別子の変換が行われる．さらにソース言語や処理系の実装によっては，ソース言語の局所変数をターゲット言語の局所変数に変換するのではなく，実行時環境をアクセスする C 言語コードへ変換することもある．

ソース言語が Scheme 言語の場合，大域変数は C 言語の大域変数として実現され，識別子の規則の違いだけを考えれば十分であることが多い，Hobbit コンパイラの

場合はこれに当てはまる．例えば，ソースコード 2.1 の Scheme プログラムの先頭に“(define double 100)”という大域変数の宣言の場合．“double”は C 言語のキーワードであるので，Hobbit コンパイラによって変数名は double_nonkeyword という大域変数に変換する．

一方，Scheme 言語の局所変数については，関数閉包に閉じ込められる場合，set! 操作 (代入) される場合などで，C 言語への変換方法は異なり，C 言語の変数ではなく，より複雑なデータ構造や制御構造を含む C 言語コードに変換される．

以下の内容でその対応関係を表す手法についての考え方を紹介する．

ソースコード 2.1 に“(define double 100)”を追加したプログラムに現れる変数と Hobbit で変換したソースコード 2.3 の C プログラム間の対応関係を一つの写像で表示すると，図 3–1 のように示す。

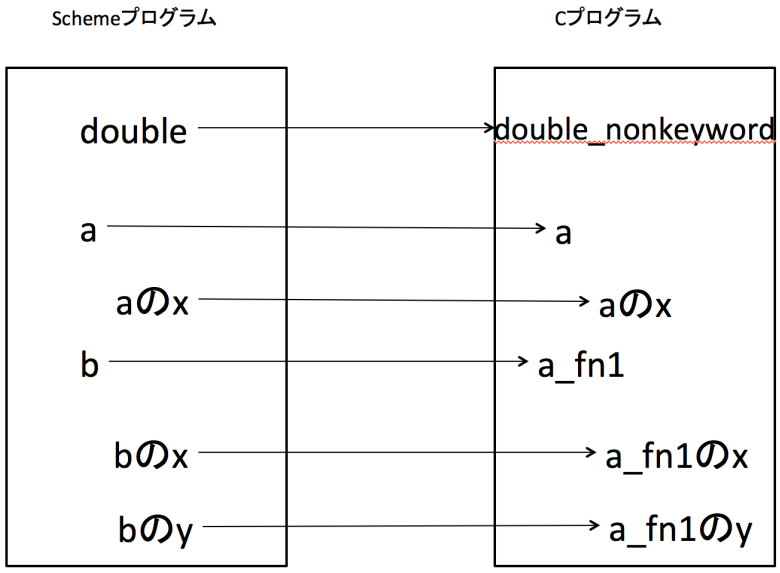

図 3–1: ソースコード 2.1 とソースコード 2.3 の対応関係

本研究に対して，生成される C プログラムに Scheme プログラムと C プログラムの対応関係を表す必要がある。生成される C プログラムに一ペアのリストを埋め込み，一つは Scheme プログラムの関数名あるいは変数名を保持して，もう一つは対応する C プログラムの関数名あるいは変数の実装コードを保持すると，Scheme プログラムと C プログラムの対応関係を表すことができると考える。

一ペアのリストを作って，全ての関数と変数間の対応関係を表すより一つのスコープによって，一ペアのリストを作ることはより明確的に対応関係を表すこと

ができると考える。それで，対応関係を表すリストを埋め込んだ C プログラムをソースコード 3.1 のように示す。

ソースコード 3.1: 対応関係を表すリストを埋め込んだCプログラム

```
1  char* listCfun[]={"a","a_fn1","double_nonkeyword"};
2  char* listSchemefun[]={"a","b","double"};
3  SCM a(x)
4  SCM x;
5  {
6    static char* listScheme[]={"x"};
7    static char* listC[]={"x"};
8
9    a_fn1(x,x);
10   return a_fn1(x,MAKINUM(2));
11 }
12
13 SCM a_fn1(x,y)
14 SCM x,y;
15 {
16   static char* listScheme[]={"x","y"};
17   static char* listC[]={"x","y"};
18
19   return x;
20 }
21 ...
```

デバッガに対して，あるスコープをデバッグする時，このスコープに埋め込まれた二つのリストをアクセスしては良い。

第 4 章にそのような対応関係を表すリストを作る方法を紹介する．

3.2　デバッガコマンドと引数の変換

入力されるソース言語レベルのデバッガコマンドと引数を C 言語のデバッガで実行できるために，C 言語レベルのデバッガコマンドと引数に変換することが必要である．デバッガコマンドと引数の変換用の変換器を開発するより，既存の C 言語のデバッガを拡張しては良いと考える．図 3–2 に示す構造のように，ソース言語レベルのデバッガコマンドと引数を C 言語のデバッガに入力して，まず C 言語のデバッガに埋め込まれたスクリプトで C 言語レベルのデバッガコマンドと引数に変換する．そして，C 言語のデバッガで解釈して，実行する．

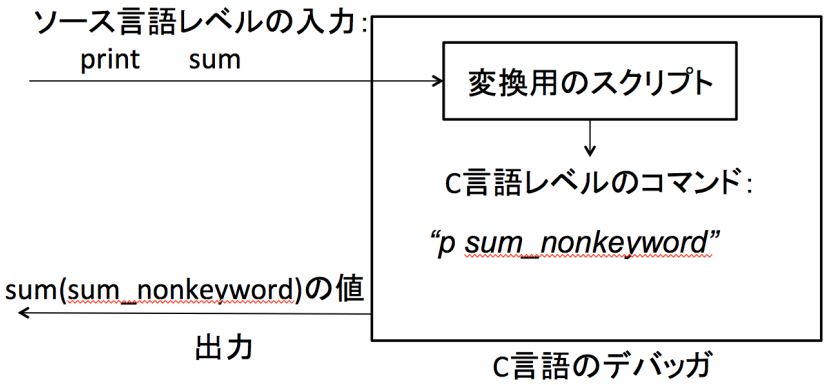

図 3–2: デバッガコマンドと引数の変換

第 4 章に LLDB に埋め込まれた Python によるスクリプトによって，ソース言語レベルのデバッガコマンドと引数を C 言語レベルのデバッガコマンドと引数に変換する実装を紹介する．

3.3　出力の変換

本研究に対する解決すべき問題はどうやって C 言語レベルに変換したデバッガコマンドと引数を通じて，C 言語のデバッガからソース言語レベルのデバッグのメッセージを出力できるかである．

例えば，プリントデバッガコマンドはプリント，引数は変数の値を表示する場合，Scheme プログラムのその変数に対応する値を Scheme のデータの表し方でプリントしてほしい．Scheme 処理系はこのような時に write という Scheme の組み込み関数を使ってプリントするので，単にそれを実装する C 言語の関数を呼び出せば良い．関数呼び出しの履歴を表示するバックトレースコマンドは今回は実装していないが同じように出力時の変換を行えば良い．

第4章

実装と実行例

4.1 実装

本手法の実装はHobbitコンパイラとLLDBに対して行った．

実装は二つの部分がある．一つはソース-to-C型トランスレータであるHobbitコンパイラに対する改造である．その改造の目的は，生成するコードにScheme言語の関数名と変数名とそれに対応するC言語のコード片の対応関係情報を含ませることである．

もう一つはLLDBに埋め込まれたPython上のスクリプトによってLLDBを拡張することである．その拡張の目的は，拡張されるLLDBはソースレベルの入力を受け取り，上記の対応関係情報を用いて，C言語レベルに変換することである．以下で二つの部分を具体的に説明する．

4.1.1 トランスレータの改造

本研究はトランスレータに対する改造を通じて，生成されるC言語プログラムに二種類のリストを埋め込むようにした．

C言語の関数内で宣言されるこれらのリストは，一方のリストでソース言語の関数の局所変数を文字列で保持する．他のリストは対応するC言語による変数の実装コードを文字列で保持する．

大域スコープで宣言される二つのリストは，一つはソース言語の関数名と大域変数名を文字列で保持する．もう一つは対応するC言語による関数名と大域変数名を文字列で保持する．

`write-c-wholefun`はSchemeプログラムの関数をC言語の関数に変換する関数

である．その関数のパラメータ def は Hobbit コンパイラで前処理 (lamda-lifting などの操作) した Scheme プログラムである．本研究で改造した関数 write-c-wholefun をソースコード 4.1 に示す．19 行目と 20 行目は今回埋め込んだコードである．関数 read_List のコードを 23〜27 行目に示す．

ソースコード 4.1: 改造された write-c-wholefun

```
1  (define (write-c-wholefun def port)
2    (let* ((fun (caddr def))
3               (top-let (caddr fun)))
4      (set! *c-port* port)
5      (set! *current-fun-name* (cadr def))
6             ...
7    (if (not (null? (cadr fun)))
8          (begin
9            (let ((scm-args (filter (lambda (x) (symbol? x))
10                     (cadr fun)))
11           ...
12             (if (not (null? scm-args))
13                 (begin
14                   (display-c *scm-type*)
15                   (display-c #\space)
16                   (display-c-lst scm-args #f #f)
17                   (display-c #\;)
18 ;;;;;;;;;;;;;;;;;;;;;;;;埋め込むコード;;;;;;;;;;;;;;;;;;;;;;;;
19   (read_List scm-args);パラメータ
20   (read_List (map car (cadr top-let)));局所変数
21 ;;;;;;;;;;;;;;;;;;;;;;;;埋め込むコード;;;;;;;;;;;;;;;;;;;;;;;;
22             ...
23 (define (read_List list)
24         (if (and (not (null? list))(pair? list))
25                 (embed (car list) (car list)))
26         (if (not (null? list))
27                 (read_List (cdr list))))
28  ...
```

関数 write-c-wholefun は def から Scheme プログラムの関数の名前，関数のパラメータと局所変数などの情報を取り出す．*current-fun-name*に関数名あるいは大域変数名を保存して，scm-args に関数のパラメータを保存して，top-let に関数の局所変数の宣言を保存する．

19 行目の関数 read_List は本研究で定義した関数であり，関数 write-c-wholefun で処理される関数のパラメータと局所変数を受け取り，リストで保存する．

関数 read_List で生成するリストは Scheme 言語の局所変数を記録するリストである．Scheme 言語の関数に，関数閉包に閉じ込められる変数と set!操作される変数がない場合，Hobbit コンパイラの識別子に対する再命名規則によって，関数 read_List で生成するリストの情報で C 言語による局所変数を記録するリストを作る．ソースコード 4.2 は関数 read_List で生成するリストに対する処理を示す．

ソースコード 4.2: C 言語の局所変数の実装コードを記録するリストを作る関数

```
1  (define (find list); Scheme listでC listを作る
2    (if (not (null? list))
3        (find_a list)))
4  (define (find_a list);
5    (if (check_keyword (cdar list)); Cのキーワードを探す
6        (change_keyword list)); キーワードを処理
7    (if (check_number (cdar list));数字を探す
8        (change_number list));数字を処理
9    (if (check_number (cdar list));不正文字を探す
10       (change_number list)); 不正文字を処理
11          ...
```

関数 find は関数 read_List で生成するリストから C のキーワードなどの内容を探して，処理する．そして，Scheme 言語の局所変数に対応する C 言語の局所変数の実装コードを記録するリストを作る．

Hobbit コンパイラは大域変数と関数に対して，変数と関数の名前の変換を行うだけである．関数 find を利用して，Scheme プログラムの大域変数名と関数名を記録するリストによって，対応する C プログラムの大域変数名と関数名を記録するリストを作ることができる．“(define double 100)”を埋め込むソースコード 2.1 の Scheme プログラムを本研究で改造された Hobbit コンパイラで変換された C プログラムをソースコード 4.3 に示す．1〜2 行目，8〜9 行目と 18〜19 行目が改造されたトランスレータによって埋め込まれた対応関係情報である．

ソースコード 4.3: 改造された Hobbit コンパイラによる生成コード

```
1  char* listCfun[]={"a","a_fn1","double_nonkeyword"};
2  char* listSchemefun[]={"a","a-cl1","double"};
3
4  SCM a(x)
5  SCM x;
6  {
7
8    static char* listScheme[]={"x"};
```

```
9    static char* listC[]={"x"};
10
11   a_fn1(x,x);
12   return a_fn1(x,MAKINUM(2));
13 }
14
15 SCM a_fn1(x,y)
16 SCM x,y;
17 {
18   static char* listScheme[]={"x","y"};
19   static char* listC[]={"x","y"};
20
21   return x;
22 }
23 ...
24  GLOBAL(double_nonkeyword)=MAKINUM(100);
25 ...
```

大域スコープのリストは関数と大域変数の対応関係を表す．listSchemefun は Scheme 言語の関数名を記録する．ソースコード 2.1 の関数閉包は Hobbit コンパイラによって，a_fn1 という名前がつけられた．本研究では関数 a の中の一番目の関数閉包を意味する a-cl1 という名前をソース言語レベルの名前として記録する．それで，Scheme 言語レベルでは名前のついていない関数閉包に対してもアクセスできる．

関数閉包と set!操作の変数に対する処理

ソースコード 4.4 に示すコードは Hobbit コンパイラの関数閉包を処理する関数 try-closure-making-def である．関数閉包に閉じ込められる変数を lvars に置く (4 行目)．関数閉包に閉じ込められる変数が set!操作される場合，closurevars に置く (7 行目)．本研究は lvars と closurevars の情報を取り出して (12〜13 行目行目)，Scheme プログラムの局所変数を記録するリストに追加する．

ソースコード 4.4: 改造された try-closure-making-def

```
1 (define (try-closure-making-def def)
2     ...
3    (let* ((lterm (caddr def))
4           (lvars (args->list (cadr lterm)))
5           (letvars (collect-local-vars (cddr lterm)))
6           (vars (union lvars letvars))
```

```
7             (closurevars (closure-building-vars (cddr lterm) vars))
8             (vectname (make-closure-vector-name)))
9         (set! *current-fun-name* (cadr def))
10         ...
11 ;;;;;;;;;;;;;;;;;;;;;;;;埋め込むコード;;;;;;;;;;;;;;;;;;;;;;;;
12    (set! listClosure (cons lvars listClosure))
13    (set! listSetFree (cons closurevars listSetFree))
14 ;;;;;;;;;;;;;;;;;;;;;;;;埋め込むコード;;;;;;;;;;;;;;;;;;;;;;;;
15       ...
```

ソースコード 2.6 の Scheme プログラムとそれに対応する C プログラム間に，図 4–1 に示す写像関係がある．

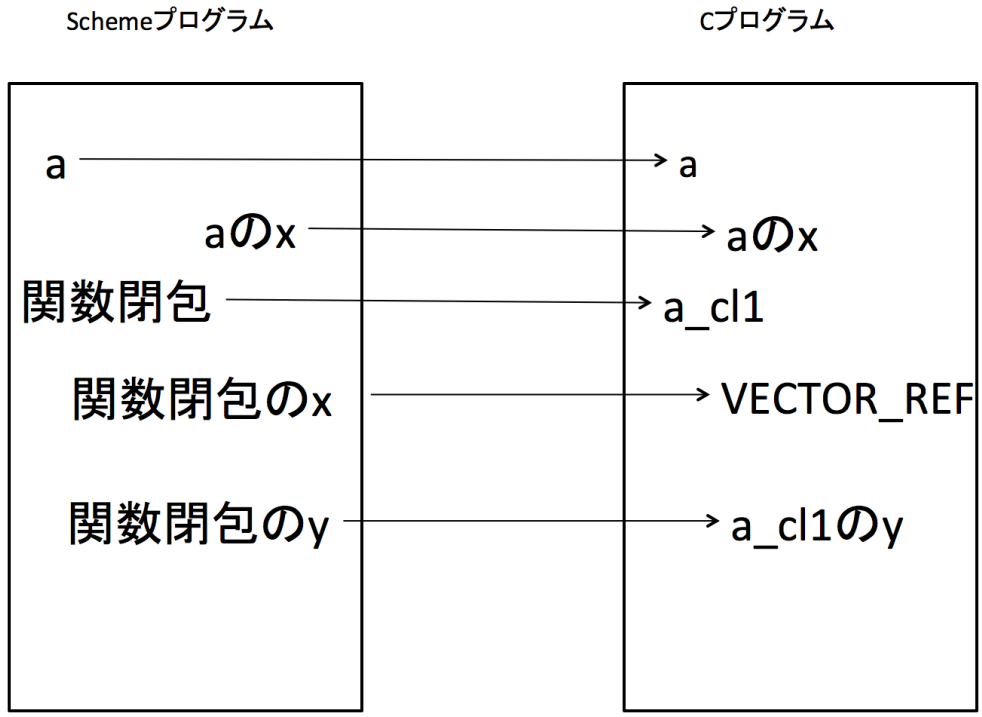

図 4–1: ソースコード 2.6 の変数間と関数間の写像

関数閉包の局所変数 x は VECTOR_REF(clargsv_1,MAKINUM(0)) という C 言語コードに変換された．ソースコード 2.6 の Scheme プログラムを変換した結果をソースコード 4.5 に示す．本研究はその x に対する“ -0 ”で listC に記録する．“ -0 ”の“ - ”は clargsv_1 に保存されることを表し，“ 0 ”はそのインデックス 0 の位置に保存されることを表す．

ソースコード 4.5: 改造された Hobbit コンパイラによる生成コード (set!操作)

```
1  char* listCfun[]={"a","a_cl1"};
2  char* listSchemefun[]={"a","a-cl1"};
3
4  SCM a(x)
5  SCM x;
6  {
7
8    static char* listScheme[]={"x"};
9    static char* listC[]={"x"};
10
11   SCM clargsv_1,newclosure;
12
13   clargsv_1=make_vector(MAKINUM(1),EOL);
14   VECTOR_SET(clargsv_1,MAKINUM(0),x);
15   newclosure=makcclo(a_cl1_clproc0,2);
16   VECTOR_SET(newclosure,MAKINUM(1),clargsv_1);
17   return newclosure;
18 }
19
20
21 SCM a_cl1(closurearg_0)
22 SCM closurearg_0;
23 {
24   static char* listScheme[]={"x","y"};
25   static char* listC[]={"-0","y"};
26
27   SCM closurearg_car_0,clargsv_1,y;
28
29   closurearg_car_0=CAR(closurearg_0);
30   clargsv_1=VECTOR_REF(closurearg_car_0,MAKINUM(1));
31   closurearg_0=CDR(closurearg_0);
32   y=CAR(closurearg_0);
33   VECTOR_REF(clargsv_1,MAKINUM(0))=MAKINUM(INUM(
34         VECTOR_REF(clargsv_1,MAKINUM(0)))+INUM(y));
35   return VECTOR_REF(clargsv_1,MAKINUM(0));
36 }
37 ...
```

本研究はトランスレータの改造を通じて，生成される C 言語にソース言語と C 言語の対応関係を表すリストを埋め込む.

次の 4.1.2 節では，埋め込んだリストを通じて，ユーザがデバッガに対して入力するソース言語レベルの引数を C 言語レベルに変換する方法と C 言語レベルの引

数を通じて，ソース言語レベルのデバッガコマンドの実行結果を出力する方法を説明する．

4.1.2　LLDB の拡張

LLDB は C 言語をデバッグできて，Python 上のスクリプトで拡張できるプラグイン機構を持つデバッガである [21]．

本研究では LLDB に埋め込まれた Python 上のスクリプトを通じて，二つのリストにアクセスして，ソース言語レベルの引数に対応する C 言語レベルの引数を得る．そして，デバッガコマンドと引数を Python に用意された LLDB の API を用いて LLDB で実行する [22]．

本研究は二つのソース言語用のデバッガコマンドを作った．一つは変数の値あるいは関数の情報をプリントするデバッガコマンド myprint である．もう一つは関数名を通じて breakpoint を設置するデバッガコマンド mybreak である．以下でその二つコマンドを説明する．

mybreak

mybreak の機能を実現する Python スクリプト mybreak.py は，まず入力されるデバッガコマンドの引数を受け取る．そして，C プログラムの大域スコープに埋め込まれたリスト `listSchemefun` と `listCfun` を読み込む．リストの情報を利用して，入力された引数を C 言語レベルに変換する．mybreak の機能を実現する関数をソースコード 4.9 に示す．

ソースコード 4.6: mybreak.py

```
1  def mybreak(debugger, command, result, internal_dict):
2    names=str(command)
3    listSchemefun=[]
4    for i in lldb.frame.variables:
5        if (cmp(i.name,"listSchemefun")==0):
6            for j in i:
7                x = str(j).find('"')
8                y = str(j).find('"',x+1)
9                listSchemefun.append(str(j)[x+1:y])
10   listCfun=[]
11   for i in lldb.frame.variables:
12       if (cmp(i.name,"listCfun")==0):
```

```
13              for j in i:
14                  x = str(j).find('"')
15                  y = str(j).find('"',x+1)
16                  listCfun.append(str(j)[x+1:y])
17      m=0
18      while(len(listSchemefun)!=0):
19          m=m+1
20          if m >= len(listSchemefun):
21              break
22      if m >= 0 and m < len(listSchemefun):
23         debugger.HandleCommand("b "+listCfun[m])
24      ...
```

関数 mybreak は listSchemefun と listCfun に保存する情報を通じて，ソース言語の関数名を対応する C 言語の関数名に変換する（19〜22 行目）．そして，その関数の所に breakpoint を設置する（23 行目）．

mybreak を利用して, ソースコード 2.6 の Scheme プログラムに対するデバッグのステップを図 4–2 に示す:

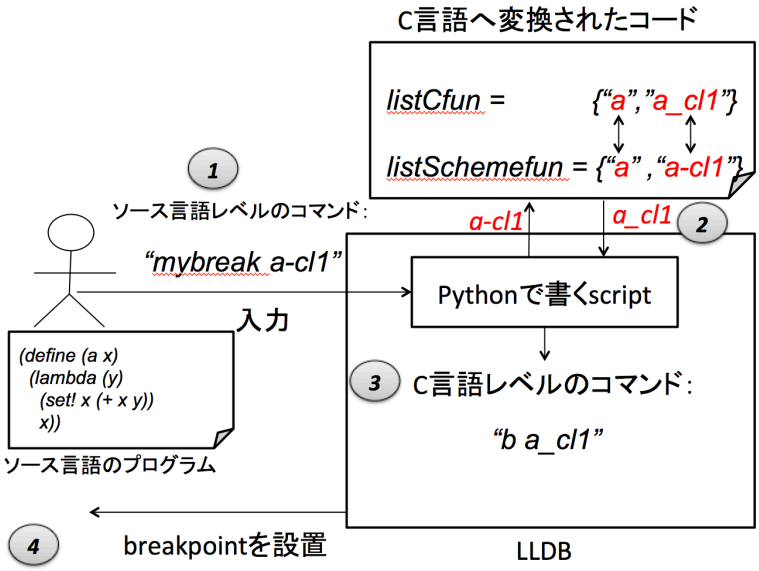

図 4–2: デバッガコマンド mybreak の引数の変換の流れ

1. ユーザが入力するソース言語レベルのコマンド mybreak と引数 a-cl1 を受け取る．

2. Python 上のスクリプトは mybreak を C 言語レベルのデバッグコマンド b(breakpoint)

に変換する．listCfun から a-cl1 と対応する引数 a_cl1 を取り出す．

3. 変換された C 言語レベルのデバッグコマンドと引数を LLDB で実行する．

4. 関数 a_cl の所に breakpoint を設置する．

mybreak は大域スコープの二つのリストにアクセスする. しかし,C 言語では, プログラムが実行しないと, 大域スコープに宣言されるリストは空リストである. それで, 本研究で作ったデバッグコマンド mybreak を初めて使用する場合, 先に図 4–3 に示す操作が必要である.

```
(lldb) b main
Breakpoint 1: where = fft`main + 38 at scmmain.c:88, address = 0x0000000100002aa6
(lldb) r
Process 9696 launched: '/Users/liuzhen/scm/fft' (x86_64)
Process 9696 stopped
* thread #1: tid = 0x12df1e, 0x0000000100002aa6 fft`main(argc=1, argv=0x00007fff5fbff978)
+ 38 at scmmain.c:88, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x0000000100002aa6 fft`main(argc=1, argv=0x00007fff5fbff978) + 38 at scmmain
.c:88
   85        int argc;
   86        const char **argv;
   87   {
-> 88     char *script_arg = 0;         /* location of SCSH style script file or 0. */
   89     char *implpath = 0, **nargv;
   90     int nargc, iverbose = 0, buf0stdin;
   91     SCM retval;
(lldb) breakpoint delete 1
1 breakpoints deleted; 0 breakpoint locations disabled.
```

図 4–3: mybreak を使用するための操作

まずプログラムに一つの breakpoint(例:b main) を設置する. プログラムが一度実行して, 止まって, 関数 mybreak は listSchemefun と listCfun に保存する情報を読み込むことができる. そして, 先ほどの設置された breakpoint を削除する. 以 上の操作が終わる後,mybreak を使用することができる.

関数名によって breakpoint を設置すると，breakpoint が関数の先頭に設置される．しかし，ソースコード 2.4 のように局所変数は関数閉包に閉じ込められる場合，ソースコード 2.5 の 16 行目から 21 行目の C プログラムのコードが Scheme プログラムのスコープ構造を実現するために，Hobbit コンパイラによって埋め込められる．関数名 a_cl1 によって，breakpoint を設置すると，プログラムは 16 行目に止まって，局所変数 x と y の初期化をまだしないので，変数の値をプリントできない．

そこで，本研究は各 Scheme 関数の本体の先頭にダミーの関数の呼び出しを埋め込み， そのダミー関数に対して breakpoint を設置するようにする （ダミー関数の呼び出し時点では初期化が済んでいる）．ここでのダミー関数とは本体が空の関数であり，各 Scheme 関数に用意する．ダミー関数 a_break を用意したソースコード 2.4 をソースコード 4.7 に示す．

ソースコード 4.7: ダミー関数を埋め込む Scheme プログラム

```
1   ...
2  (define (a_break) '());関数 a用のダミー関数の定義
3  (define (b_break) '());関数 b用のダミー関数の定義
4  (define (a x)
5    (define (b y)
6      (b_break);ダミー関数の呼び出し(トランスレータで挿入可能)
7      x)
8    (a_break)
9    (b x)
10   b)
```

関数閉包 b に breakpoint を設置する場合，関数 b の代わりに，関数 b_break に breakpoint を設置する (“mybreak b”を“mybreak b_break”と自動変換することも可能)．

この方法で関数 b に breakpoint を設置する様子を図 4–4 に示す．

```
(lldb) b b_break
Breakpoint 1: where = j0`b_break + 9 at f.c:17, address = 0x0000000100035d29
(lldb) r
Process 5430 launched: '/Users/liuzhen/scm/j0' (x86_64)
SCM version 5f2, Copyright (C) 1990-2006 Free Software Foundation.
SCM comes with ABSOLUTELY NO WARRANTY; for details type `(terms)'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `(terms)' for details.
;loading /Users/liuzhen/slib/require
;done loading /Users/liuzhen/slib/require.scm
;loading /Users/liuzhen/scm/Transcen
;done loading /Users/liuzhen/scm/Transcen.scm
> (a 3)
Process 5430 stopped
* thread #1: tid = 0xe2086, 0x0000000100035d29 j0`b_break + 9 at f.c:17, queue = 'com.appl
e.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x0000000100035d29 j0`b_break + 9 at f.c:17
   14     static char* listScheme[]={};
   15     static char* listC[]={};
   16
-> 17     return EOL;
   18   }
   19
   20
(lldb) n
Process 5430 stopped
* thread #1: tid = 0xe2086, 0x0000000100035de3 j0`a_cl1(closurearg_0=4295925824) + 67 at f
.c:51, queue = 'com.apple.main-thread', stop reason = step over
    frame #0: 0x0000000100035de3 j0`a_cl1(closurearg_0=4295925824) + 67 at f.c:51
   48     closurearg_0=CDR(closurearg_0);
   49     y=CAR(closurearg_0);
   50     b_break();
-> 51     return x;
   52   }
   53
   54
```

図 4–4: 関数閉包に breakpoint を設置

関数 b_break の所に停止したら，関数 b_break をスキップするために，n コマンドによって，実行を 1 ステップ進める．すると，ソースコード 4.7 の 7 行目に相当する位置で止まる．

以上のダミー関数の定義と呼び出しの挿入は自動的に行うことも可能であるが，今回は手動で行った．

myprint

myprint の機能を実現する Python スクリプト myprint.py は，まず入力されるデバッガコマンドの引数を受け取る．そして，4.1.1 節で説明した C プログラムに埋め込まれた `listScheme` などのリストを読み込む．`listScheme` を読み込む Python コードをソースコード 4.8 に示す．

ソースコード 4.8: myprint.py:埋め込まれたリストを読み込む部分のコード

```
def myprint(debugger, command, exe_ctx,result, internal_dict):
    names=str(command)
    listSchemefun=[]
    for i in lldb.frame.variables:
        if (cmp(i.name,"listSchemefun")==0):
            for j in i:
                x = str(j).find('"')
                y = str(j).find('"',x+1)
                listSchemefun.append(str(j)[x+1:y])
                ...
```

入力されるデバッガコマンドの引数を関数 myprint のパラメータ command に保存する．myprint.py は C プログラムの `listScheme` から情報を取り出して，Python スクリプトに定義されるリスト `listScheme` に保存する．他の C プログラムに埋め込まれたリストもその方法で処理する．

myprint.py はそれらのリストを利用して，ソース言語レベルの引数を C 言語レベルの引数に変換する．その変換の実現をソースコード 4.9 に示す．

ソースコード 4.9: myprint.py:Scheme レベルを C レベルに変換

```
 ...
 m=0
 while(len(listScheme)!=0 and cmp(str(names),listScheme[m])!=0):
     m=m+1
     if m >= len(listScheme):
         break
 if m >= 0 and m < len(listScheme):
     temp=listC[m]
     if (cmp(temp[0],"-")==0):
         number=temp[1:]
         debugger.HandleCommand
               ("p scm_write(((SCM *)(((cell *)(((SCM *)(((cell *)
                 (closurearg_car_0))->cdr))[1]))->cdr))
                       ["+number+"],sys_protects[2])")
     else:
         debugger.HandleCommand("call scm_write
                                ("+listC[m]+",sys_protects[2])")
 ...
```

まず入力された引数によって，`listScheme` から変数名を取り出す．そして，`listC` に同じインデックスの内容は対応する C 言語レベルのデバッガ引数である．

本研究で作られる myprint は Hobbit コンパイラの `scm_write` 関数を利用して，C 言語のコードによって，対応する Scheme プログラムの変数の値をプリントする．Hobbit コンパイラでは Scheme プログラムの変数を C 言語のコード片に変換する場合，マクロを含んだコードが使われる．そのマクロ展開は手動で行う必要がある．9〜14 行目は LLDB に埋め込む Python スクリプトで使用するマクロ展開のコードである．15〜17 行目は変数が C 言語の変数に変換される場合，変数の値をプリントするためのコードである．

myprint を利用して, ソースコード 2.6 の Scheme プログラムに対するデバッグのステップを図 4–5 に示す：

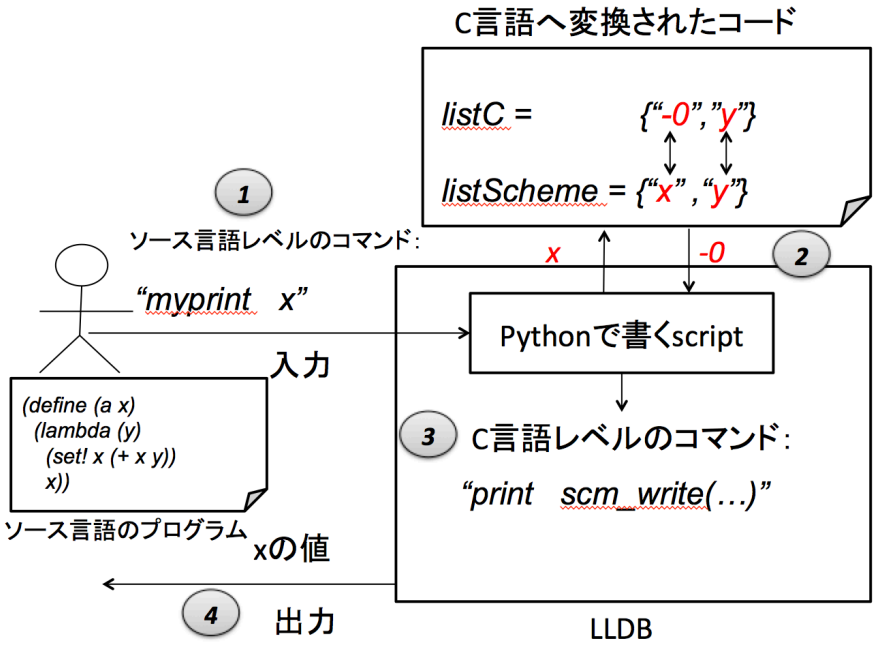

図 4–5: デバッグコマンド myprint の引数の変換の流れ

1. ユーザが入力するソース言語レベルのコマンド myprint と引数 `x` を受け取る．
2. Python 上のスクリプトは myprint を C 言語レベルのデバッガコマンド print に変換する．`listC` から `x` と対応する引数 `-0` を取り出して，`x` の値を `clargsv_1` のインデックス `0` に置かれていることを知って，マクロ展開する．そして，`scm_write` 関数を呼び出す．
3. “`print scm_write(...)`” を LLDB で実行する．

4. x の値をプリントされる．

本研究で作られる myprint は scm_write 関数を利用して，Scheme プログラムの変数をプリントする．Scheme プログラムの変数は C 言語のコードに変換される場合，myprint はマクロ展開を通じて，変数の値をプリントできる．

4.2　実行例

C 言語のデバッガコマンドでソースコード 2.6 のプログラムに対するデバッグを図 4–6 に示す．ソース言語のオブジェクトを C 言語に変換する後，名前が変わる場合，本来の C 言語のデバッガコマンドでデバッグできない．名前が変わらない場合，C 言語のデバッガコマンドでデバッグする結果はソース言語レベルではなく，C 言語レベルである．

```
(lldb) p x
error: use of undeclared identifier 'x'
error: 1 errors parsing expression
(lldb) p a
(SCM (*)(SCM)) $0 = 0x0000000100036200 (fi`a at sum.c:5)
(lldb) b a-cl1
Breakpoint 2: no locations (pending).
WARNING:  Unable to resolve breakpoint to any actual locations.
(lldb)
```

図 4–6: C 言語のデバッガコマンドでデバッグ

本研究で作られる mybreak コマンドと myprint コマンドを通じて，ソースコード 2.6 のプログラムに対する Scheme 言語レベルのデバッグを実現できる．

ソースコード 2.6 の Scheme プログラムの関数閉包に breakpoint を設置するために，ソースコード 4.10 のように示して，まず a_break 関数を呼び出す．

ソースコード 4.10: ダミー関数を埋め込む Scheme プログラム Scheme プログラム

```
1 (define (a_cl1_break) '())
2 (define (a_break) '())
3 (define (a x)
4   (a_break); a での停止用
5   (lambda(y)
6     (a_cl1_break); 関数閉包での停止用
```

```
7       (set! x (+ x y))
8       x))
9   ...
```

そして，図 4–7 のように示して，mybreak を利用して，関数閉包の所に breakpoint を設置できる．

```
(lldb) mybreak a_cl1_break
Breakpoint 2: where = jsum`a_cl1_break + 9 at sum.c:8, address = 0x0000000100036189
(lldb) r
There is a running process, kill it and restart?: [Y/n] y
Process 5594 exited with status = 9 (0x00000009)
Process 5598 launched: '/Users/liuzhen/scm/jsum' (x86_64)
SCM version 5f2, Copyright (C) 1990-2006 Free Software Foundation.
SCM comes with ABSOLUTELY NO WARRANTY; for details type `(terms)'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `(terms)' for details.
;loading /Users/liuzhen/slib/require
;done loading /Users/liuzhen/slib/require.scm
;loading /Users/liuzhen/scm/Transcen
;done loading /Users/liuzhen/scm/Transcen.scm
> (define i (a 3))
#<unspecified>
> (i 7)
Process 5598 stopped
* thread #1: tid = 0xe405d, 0x0000000100036189 jsum`a_cl1_break + 9 at sum.c:8, queue = 'c
om.apple.main-thread', stop reason = breakpoint 2.1
    frame #0: 0x0000000100036189 jsum`a_cl1_break + 9 at sum.c:8
   5       static char* listScheme[]={};
   6       static char* listC[]={};
   7
-> 8       return EOL;
   9    }
   10
   11
(lldb) n
Process 5598 stopped
* thread #1: tid = 0xe405d, 0x0000000100036253 jsum`a_cl1(closurearg_0=4295921792) + 67 at
 sum.c:52, queue = 'com.apple.main-thread', stop reason = step over
    frame #0: 0x0000000100036253 jsum`a_cl1(closurearg_0=4295921792) + 67 at sum.c:52
   49      closurearg_0=CDR(closurearg_0);
   50      y=CAR(closurearg_0);
   51      a_cl1_break();
-> 52      VECTOR_REF(clargsv_3,MAKINUM(0))=MAKINUM(INUM(VECTOR_REF(clargsv_3,MAKINUM(0)))+
INUM(y));
   53      return VECTOR_REF(clargsv_3,MAKINUM(0));
   54   }
```

図 4–7: mybreak の実行：breakpoint を設置

図 4–7 によって，(define i (a 3)) と (i 7) で関数を呼び出して, プログラムは“ return x ”の所に止まる．変数の初期化を完成したので，変数の値をプリントできる．

myprint を使って，ソースコード 2.6 の Scheme プログラムに対するデバッグを図 4–8 に示す．

```
> (define i (a 3))
#<unspecified>
> (i 7)
Process 5598 stopped
* thread #1: tid = 0xe405d, 0x0000000100036189 jsum`a_cl1_break + 9 at sum.c:8, queue = 'c
om.apple.main-thread', stop reason = breakpoint 2.1
    frame #0: 0x0000000100036189 jsum`a_cl1_break + 9 at sum.c:8
   5       static char* listScheme[]={};
   6       static char* listC[]={};
   7
-> 8       return EOL;
   9    }
   10
   11
(lldb) n
Process 5598 stopped
* thread #1: tid = 0xe405d, 0x0000000100036253 jsum`a_cl1(closurearg_0=4295921792) + 67 at
 sum.c:52, queue = 'com.apple.main-thread', stop reason = step over
    frame #0: 0x0000000100036253 jsum`a_cl1(closurearg_0=4295921792) + 67 at sum.c:52
   49      closurearg_0=CDR(closurearg_0);
   50      y=CAR(closurearg_0);
   51      a_cl1_break();
-> 52      VECTOR_REF(clargsv_3,MAKINUM(0))=MAKINUM(INUM(VECTOR_REF(clargsv_3,MAKINUM(0)))+
INUM(y));
   53      return VECTOR_REF(clargsv_3,MAKINUM(0));
   54   }
   55
(lldb) script
Python Interactive Interpreter. To exit, type 'quit()', 'exit()' or Ctrl-D.
>>> ^D
(lldb) command script import ~/myprint.py
The "myprint" python command has been installed and is ready for use.
(lldb) myprint x
3(SCM) $0 = 18804
(lldb) myprint y
7(SCM) $1 = 18804
```

図 4–8: myprint の実行：変数をプリント

図 4–8 によって，myprint は x の値 (3) と y (7) の値をプリントできる。

myprint コマンドの引数は関数名の場合，結果を図 4--9 に示す．C 言語デバッガ のプリントコマンド（図 4--6）と比べて，myprint コマンドは Scheme 言語レベルの 関数情報をプリントできる．

```
(lldb) command script import ~/myprint.py
The "myprint" python command has been installed and is ready for use.
(lldb) myprint a
#<procedure:a>
(lldb) myprint a-cl1
#<procedure:a-cl1>
(lldb)
```

図 4–9: myprint の実行：関数をプリント

そこで，本研究で作られる mybreak コマンドは Scheme プログラムによって，変

換される C プログラムに breakpoint を設置できる．myprint コマンドは Scheme プログラムの変数名によって，変数の値をプリントできる．Scheme プログラムの変数が C 言語のコードに変換されても，マクロ展開を通じて，変数の値をプリントできる．

第5章

結論

本論文では，C 言語デバッガを用いてソース-to-C 型トランスレータのソース言語レベルデバッガを実現する方法を提案した．

この提案手法では，まずソース-to-C 型トランスレータの改造を通じて，生成された C 言語にソース言語と C 言語の対応関係を表すリストを埋め込む．大域スコープに埋め込まれるリストはソース言語の関数と大域変数と C 言語の対応関係を表す．関数スコープに埋め込まれるリストはこの関数の局所変数と C 言語の対応関係を表す．

そして，LLDB に埋め込まれた Python 上のスクリプトでそれらのリストをアクセスして，入力されるソース言語レベルのデバッガコマンドと引数を C 言語レベルに変換する．最後に LLDB で解釈して，実行する．

5.1 評価

本研究では Hobbit コンパイラを改造して，LLDB を拡張した．本来の LLDB と比べて，本研究で拡張された LLDB はソース言語レベルのデバッガコマンドと引数を利用して，ソース言語レベルのプリントと breakpoint の設置を実現できた．

Scala 言語と Clojure 言語のデバッグと比べて，本研究で実現したデバッグはソース言語レベルのメッセージを出力できる．Clojure 言語の視覚的なデバッガ Cider-debug と比べて，本研究は LLDB を用いて，より多くの機能を実現する可能である．

5.2　今後の課題

本研究では，ソース言語のコード片として変数名と関数名のみを扱う．ソース言語と C 言語のコード片の行番号の対応関係を表すことができると，行番号を通じて breakpoint の設置などの機能を実現できる．

4.1.2 節の内容によって，mybreak コマンドを初めて使う場合，先にたくさんの操作が必要である．.lldbinit というファイルは LLDB の初期化を行うファイルである．そのファイルを利用して，プログラムを実行した前に，大域スコープに埋め込まれた二つのリストを読み込んで，mybreak を直接に使う可能である．

4.1.1 節の内容によって，Hobbit コンパイラで前処理した Scheme プログラムから情報を取り出した．それで，Scheme 言語レベルでは名前のついていない関数閉包をアクセスできた．関数閉包に閉じ込められて，set! 操作された変数を取り出す機能を実現するための hobbit.scm に埋め込まれたコード量が少ない．しかし，前処理された Scheme プログラムは一部分のオブジェクト (例 : 関数閉包) の名前が変換された．それで，Scheme 言語に名前のついている関数閉包を直接に対応できない．前処理した Scheme プログラムと前処理した前の Scheme プログラムを利用して，情報を取り出すと，今よりソース言語レベルのデバッグに接近すると考える．

本研究は Hobbit コンパイラとトランスレータを調査した．しかし，この二つトランスレータは Scheme 言語の末尾再帰の最適化を実現できない．それで，本研究の実装はトランスレータが末尾再帰の最適化を実現するための変換を対応できない可能がある．その問題を解決するために，CHICKEN コンパイラと sml2c コンパイラ [23] などの関数型言語の末尾再帰の最適化を実現できるトランスレータに対する調査が必要である．

5.3　まとめ

本論文は LLDB を用いてトランスレータ用ソース言語レベルデバッガの実装手法を紹介した．本研究はソース言語レベルのプリントコマンドと breakpoint の設置するコマンドだけを作ったが，埋め込まれたリストの情報を利用して，他のソース言語レベルのデバッガコマンドを作る可能である．

Scheme 言語と Hobbit コンパイラを対象としたが，Hobbit コンパイラ以外のソース-to-C 型トランスレータにも適用できるはずである．将来は共通 API を用意することで，様々なトランスレータへ少ない手間で対応することを考えている．

謝辞

本研究を遂行するにあたっては，指導教員の小宮常康先生には研究の方向性や, プログラムの実装方法, また関連論 文の選定に至るまで日頃からいろいろな方々にお世話になりました.

講座内進捗や講座内輪講などにおいて研究を進める上での貴重な意見や助言を頂いた．皆様への心から感謝 の気持ちと御礼を申し上げたく、謝辞にかえさせていただきます.

参考文献

[1] Todd A.Proebsting,Gregg Townsend,Patrick Bridges,john H.Hartman,Tim Newsham and Scott A.Watterson.Toba:Java For Application A Way Ahead of Time (WAT) Compiler.In Proceedings of the Third Conference on Object-Oriented Technologies and Systems (COOTS ’97).

[2] Arno Puder, Victor Woeltjen, and Alon Zakai. 2013. Cross-compiling Java to JavaScript via tool-chaining. In Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ ’13). ACM, New York, NY, USA, 25-34.

[3] Joel.F.Bartlett. Scheme → C a Portable Scheme-to-C Compiler. WRL Research Report，January 1989.

[4] Ankush Varma and Shuvra S. Bhattacharyya. 2004. Java-through-C Compilation: An Enabling Technology for Java in Embedded Systems. In Proceedings of the conference on Design, automation and test in Europe - Volume 3 (DATE ’04), Vol. 3. IEEE Computer Society, Washington, DC, USA, 30161-.

[5] N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson. 1998. revised5 report on the algorithmic language scheme. SIGPLAN Not. 33, 9 (September 1998), 26-76.

[6] Peter Sestoft. Programming Language Concepts.Undergraduate Topics in Computer Science.Springer-Verlag London 2012

[7] The SCM Implementation of Scheme，
http://people.csail.mit.edu/jaffer/SCM (2017 年 1 月 24 日参照).

[8] Harold Abelson, Gerald Jay ,Sussman and Julie Sussman.Structure and Interpretation of Computer Programs (SICP) . MIT Press.1996

[9] 湯浅太一.Scheme 入門. 岩波書店.1991.

[10] 湯浅太一，コンパイラ，昭晃堂，2001.

[11] Tanel Tammet. hobbit.doc,documentation for hobbit version 1. 1992.

[12] Henry G. Baker. 1995. CONS should not CONS its arguments, part II: Cheney on the M.T.A.. SIGPLAN Not. 30, 9 (September 1995), 17-20.

[13] The Hobbit compiler for SCM.
http://people.csail.mit.edu/jaffer/hobbit.pdf (2017 年 1 月 24 日参照).

[14] Lukas Rytz and Martin Odersky. 2010. Named and default arguments for polymorphic object-oriented languages: a discussion on the design implemented in the Scala language. In Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10). ACM, New York, NY, USA, 2090-2095.

[15] Dmitry Jemerov. 2008. Implementing refactorings in IntelliJ IDEA. In Proceedings of the 2nd Workshop on Refactoring Tools (WRT '08). ACM, New York, NY, USA, , Article 13 , 2 pages.

[16] http://www.scala-sbt.org/documentation.html (2017 年 1 月 24 日参照).

[17] Rich Hickey. 2008. The Clojure programming language. In Proceedings of the 2008 symposium on Dynamic languages (DLS '08). ACM, New York, NY, USA, , Article 1 , 1 pages.

[18] https://github.com/razum2um/clj-debugger (2017 年 1 月 24 日参照).

[19] http://georgejahad.com/clojure/cdt.html (2017 年 1 月 24 日参照).

[20] https://github.com/clojure-emacs/cider/blob/master/doc/debugging.md (2017 年 1 月 24 日参照).

[21] http://lldb.llvm.org/python-reference.html (2017 年 1 月 24 日参照).

[22] Bill Lubanovic. Introducing Python Modern Computing in Simple Packages. O ' Reilly Media. November 2014.

[23] David Tarditi, Peter Lee, and Anurag Acharya. 1992. No assembly required: compiling standard ML to C. ACM Lett. Program. Lang. Syst. 1, 2 (June 1992), 161-177.