情報処理学会論文誌　プログラミング　Vol.11　No.3　1–13 (Sep. 2018)

関数型言語SML#のための ネイティブコードレベルデバッグ環境の実現方式

大野 一樹[1,a)]　上野 雄大[2,b)]　大堀 淳[2,c)]

受付日 2018年1月29日, 採録日 2018年5月12日

概要：本論文では，ネイティブコードにコンパイルされたSML#プログラムをデバッグする環境を提案する．SML#は，C言語との直接連携や並列に動くマルチスレッドのサポートを持つ関数型言語である．これらSML#特有の機能を駆使して書かれた関数型のプログラムの動作は，関数型言語としての記号的なプログラムの評価規則だけでなく，C言語などと同様に，オペレーティングシステムの振舞いや，ネイティブコードレベルでのレジスタやメモリの扱いにも依存しうる．このようなプログラムのデバッグを行うためには，単にSML#のソースコードレベルで実行を追跡するだけでなく，SML#コンパイラが生成したマシンコードの振舞いを直接追跡できることが望ましい．以上のようなデバッグ環境を関数型言語においても実現することに向けた第一歩として，著者らはGDB（the GNU Project Debugger）を用いてSML#プログラムをデバッグ可能にするための方式の検討およびコンパイラの改良を行った．本論文では，その詳細を報告する．

キーワード：関数型言語，SML#，ネイティブコード，デバッグ

Implementation Method of Native Code Level Debugging Environment for SML#

Kazuki Ono[1,a)]　Katsuhiro Ueno[2,b)]　Atsushi Ohori[2,c)]

Received: January 29, 2018, Accepted: May 12, 2018

Abstract: This paper proposes a debugging environment for SML# programs compiled into native machine code. SML# is a functional programming language equipped with the seamless interoperability with the C language and native multithread support. By using these features, the programmer can enjoy both the declarative programming with the full-scaled functional programming language and system programming exploiting the operating system functionalities including multithreads. To debug an SML# program with such features, it is helpful to check the behavior of its compiled machine code with respect to the usage of the low-level memory, registers, and operating system features, as well as to trace the control flow of the source code. Towards realizing a software development environment that assists such the machine code level debugging for functional languages, we extend the SML# compiler to make GDB (the GNU Project Debbuger) available for SML# programs as well as C. This paper reports the details of the development.

Keywords: functional language, SML#, native code, debug

[1] 東北大学情報科学研究科
Graduate School of Information Sciences, Tohoku University, Sendai, Miyagi 980–8579, Japan
[2] 東北大学電気通信研究所
Research Institute of Electrical Communication, Tohoku University, Sendai, Miyagi 980–8577, Japan
[a)] k-ono@riec.tohoku.ac.jp
[b)] katsu@riec.tohoku.ac.jp
[c)] ohori@riec.tohoku.ac.jp

1. はじめに

近年のソフトウェア開発では，ソフトウェアは高い実行効率と機能性の両方が求められる．その需要に答えるために，ソフトウェアは，様々なライブラリや，マルチコアCPUなどの最新のハードウェアを駆使する，巨大かつ複

雑なコードから構成されている．そのコードには，関数型やオブジェクト指向の考え方に基づく高水準な計算の記述だけでなく，メモリアクセスやマルチスレッドの細やかな制御を含む，低レベルな記述も多く含まれる．もちろん，よく設計されたプログラムでは，それらはモジュール単位にまとめられるなど，プログラムの構造を簡潔にする工夫が用いられる．しかし，プログラムにバグが生じた場合には，それらモジュールの切り分けを超え，ソースコードレベルから機械語コード（ネイティブコード）レベルに至る横断的な挙動の観察がしばしば必要とされる．

産業界で用いられている主流のプログラミング言語には，プログラムの挙動を様々な視点から観察できるデバッガが提供されている．たとえば，GDB（the GNU Project Debugger）[3] は，C や C++などを含むいくつかの手続き型言語をサポートし，かつ多くのオペレーティングシステムで利用可能なデバッガの 1 つである．GDB は，コンパイラが生成する実行形式ファイルを読み込み，C プログラムを対話的に実行する．プログラマは，プログラムの任意の行でその実行を一時中断することができ，その時点でのローカル変数やメモリの内容をダンプすることができる．この機能を用いることで，ソースコード上の実行トレースに加え，ネイティブコードがメモリを操作する様子など，プログラムの低レベルな挙動を細やかに観察することができる．以下，本論文では，低レベルなプログラムの振舞いに基づきデバッグすることをネイティブコードレベルデバッグと呼ぶ．

本論文の一般的な目的は，上述のように産業的主流の言語で実現されているネイティブコードレベルデバッグ環境を，関数型言語においても利用可能とすることである．関数型言語が産業的利用可能なほどの実用性を得るためには，その高水準な記述体系や型システムによる安全性だけでなく，高度に洗練されたライブラリや，データベースエンジン，オペレーティングシステムなどと連携し，さらにマルチコア CPU やキャッシュメモリの振舞いを制御し，それらの機能を駆使した産業的価値のあるプログラムが生み出せることが必須である．このようなプログラミングが関数型言語でも可能になるならば，そのデバッグには，上述した GDB のように，ソースコードからネイティブコードに至る様々な視点からプログラムの振舞いを観察できるデバッガが必要不可欠であると考えられる．

いくつかの関数型言語では，その言語特有のソースコードデバッガが提供されているものの，ネイティブコードレベルデバッグを支援する環境は十分とはいえない．関数型言語のソースコードデバッグ環境のほとんどはバイトコードインタプリタの上に実現されており，ネイティブコードにコンパイルされたプログラムを対象としない．一方，OCaml [5] や Glasgow Haskell Compiler（GHC）[4] は GDB が解釈するメタ情報の出力にも対応しており，GDB を用いたネイティブコードレベルデバッグも可能ではある．しかし，これらのコンパイラが GDB に提供する情報は，ステップ実行のための行番号情報などに限定されており，GDB が C 言語などに提供するような，ソースコードからネイティブコードまで横断するデバッグ環境には至っていない．

関数型言語でのネイティブコードレベルデバッグを可能にするためには，主に 2 つの技術的な課題が存在する．1 つは，多相関数や自動的メモリ管理などの機能を達成するために，ネイティブコードレベルで独自のデータ構造を管理していることである．そのため，ネイティブコードレベルでのプログラムの追跡やメモリダンプの解読には，各言語のコンパイラに関する専門知識が必須となる．もう 1 つは，関数型言語の意味論と現行の計算機アーキテクチャをつなぎ高速な実行を実現するために，関数型言語のコンパイラは，最適化を除いても，しばしば大規模なコードの変形を行うことである．ソースコードとネイティブコードの対応付けは，この変形を考慮して行う必要がある．

これらの課題のうち，最初の課題は，SML#コンパイラ [15] によって解決がなされている．SML#は，すべてのデータ構造のメモリ上の表現に，C 言語と同様の，計算機アーキテクチャに応じて自然に定められるデータ表現を採用している．これは本来，C 言語ライブラリとのシームレスな連携のために開発された機能である．ネイティブコードレベルデバッグの観点からも，SML#コンパイラに関する専門知識なしに SML#プログラムのメモリダンプを人間が読むことができるという点において，この機能は有用である．

本論文では，主に 2 つ目の課題を取り扱う．GDB が解釈可能な，ソースコードとネイティブコードを対応付けるメタ情報（デバッグ情報）を生成するためには，SML#コンパイラがコンパイルの途中で破棄する，実行コードの生成に不要な情報の一部を，ネイティブコード生成フェーズまで保存する必要がある．また，単に保存するだけでなく，SML#の動的意味を実現するためにコンパイラが追加したコードに対しても，何らかのデバッグ情報を作り出すことが必要である．著者らは，SML#コンパイラ内部でのソースコード位置情報などの取扱いを見直し，デバッグに有用なメタ情報の多くがネイティブコードとともにオブジェクトファイルに出力されるように，SML#コンパイラの改良を行った．この改良によって，GDB を用いた SML#プログラムのステップ実行，およびステップ実行中におけるローカル変数の値の表示が可能となった．本論文では，これらの改良の詳細を報告する．

本論文は，SML#のデバッグ環境の開発に関する最初の報告である．本論文の成果に基づき，著者らは SML#のネイティブコードレベルおよびソースレベルのデバッグ環境の完成を目指す予定である．

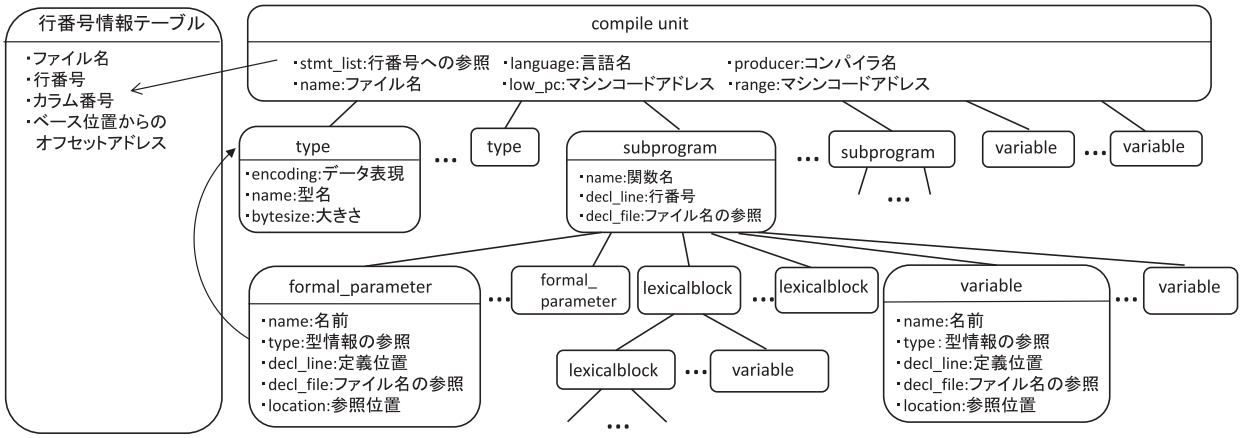


図 1　DWARF の木構造
Fig. 1　The tree structure of DWARF.

本論文の構成は以下のとおりである．2 章では，GDB などが採用するデバッグ情報の共通規格である DWARF とその生成方法を概観し，SML#コンパイラにおけるデバッグ情報生成の方針を与える．3 章では，DWARF データの生成に必要なメタ情報を得るために，SML#コンパイラ内部でのメタ情報の取り扱いを整理する．4 章では，SML#コンパイラの変更点について報告する．5 章では，さらなる実用的なネイティブコードレベルデバッグ環境に向けての展望を述べる．6 章および 7 章では，関連研究との比較および結論を述べる．

2.　デバッグ情報の構造とその生成

GDB などのデバッガは，実行形式ファイルに埋め込まれた，ソースコードとネイティブコードの対応を表すメタ情報（デバッグ情報）を解釈する．このデバッグ情報はコンパイラによってネイティブコードとともに生成されオブジェクトファイルに埋め込まれる．デバッグ情報は，ネイティブコードやメモリの番地，スタックフレームのオフセットなどの低レベルな要素に，ソースコードでの名前や行番号などの構造を対応付ける．この対応関係に基づき，ソースコードレベルからネイティブコードレベルに至る様々な段階におけるプログラムの挙動の観察を，1 つのデバッグ環境の中で一度に行うことができる．このデバッグ情報は，たとえば DWARF [1] など，言語から独立して規格化されており，コンパイラとデバッガの両方がその規格に対応しているならば，言語を問わず，同様のデバッグ環境が利用可能である．

本論文では，SML#から GDB を利用することを想定し，GDB が解釈する DWARF 情報の生成を試みる．本節では，SML#コンパイラにおける DWARF 情報の生成の観点から，DWARF の構造およびその生成方法について概観する．

2.1　DWARF の概要

DWARF は，デバッグ情報をオブジェクトファイルに格納する際に用いられるデータフォーマットの 1 つである．DWARF のデバッグ情報は，ソースプログラムの構造を抽象する木構造と，ソースコード上の位置とメモリ上の配置を対応付ける表（行番号テーブル）の 2 つからなる．その構造の概要を**図 1** に示す．

木構造の各ノードは，ソースプログラムに由来する様々な情報を表す．DWARF では，これらの木構造に含まれる各ノードを，Debug Information Entry（DIE）と呼ぶ．以下の DIE は，ソースコードのブロック構造を表す．

- コンパイル単位（compile unit）：木構造の根となる．属性情報として，その子ノードには，ソースコードでトップレベルに定義された型，変数，関数（サブプログラム）が並ぶ．
- サブプログラム：モジュール，サブルーチン，関数など，ネイティブコードを含む単位を表す．属性情報として，関数名，定義される位置情報，ファイル名への参照を持つ．その子ノードには，そのサブプログラムでローカルに定義された変数やそのスコープなどが並ぶ．
- レキシカルブロック：サブプログラムに含まれる変数の静的スコープを表す．レキシカルブロックは任意の深さにネストできる．その子ノードには，サブプログラムと同様に変数やレキシカルブロックが並ぶ．

これらのブロックには，その属性として，ソースコードにおける名前と出現位置などのソースレベルの情報と，ネイティブコードのアドレス範囲など機械語レベルの情報をともに持つ．さらに，以下の DIE が，ブロック構造に属する要素を表す．

- 型情報：`int` などの基本型やユーザ定義の型を表す．ポインタ型や定数型，構造体型などは，型情報ノードをネストして表現される．属性情報として，ソース

コードでの型名や，その型の値のメモリ上での表現などを持つ．

- 変数情報：ローカル変数，グローバル変数，関数の引数などを表す．属性情報として，ソースコードでの名前，ファイル名への参照，定義位置，および参照位置，スタックフレーム上でのオフセットなどを持つ．

これらのノードは，それがソースコード中で宣言された位置に合わせて，上述のブロック構造を表すブロックのいずれかの子ノードとして現れる．また，これらのノードは，親子関係にない他のノードへの参照を持つことがある．たとえば，変数情報は型情報への参照を持つ．サブプログラムノードも，その関数の型を表す型情報ノードへの参照を持つ．

DWARF のデバッグ情報は，この木構造に加えて行番号情報テーブルを持つ．行番号情報テーブルは，ソースファイル名，行番号，およびカラム番号と，ネイティブコードレベルでのオフセットアドレスの対応からなる．行番号情報テーブルの各エントリは，各 DIE やオブジェクトファイルに含まれる各命令などと対応する．各 DIE は，その DIE に属する要素に対応する行番号情報テーブルの範囲を持つ．

これらの構造の各部に配置されたソースレベルとネイティブコードレベルの情報が，ソースコードとネイティブコードを行き来するための足がかりとして用いられる．たとえば，コンパイル単位が持つソースファイル名の情報は，ステップ実行中に現在実行中の位置付近のソースプログラムを表示するために使用される．また，ファイル名と行番号情報の組がネイティブコードのあるアドレスに対応付けられ，ブレークポイントを置くときの位置指定などに用いられる．各命令に対応付けられた行番号は，ステップ実行機能が 1 ステップとして実行するネイティブコードの範囲を特定するためにも用いられる．

2.2　LLVM における DWARF 情報の取り扱い

DWARF はオブジェクトファイルに含まれるバイナリデータについての巨大な仕様であり，その全貌を理解しバイナリデータを生成することは，アセンブラが生成する命令列のバイナリを直接生成することにほぼ等しい．アセンブラやコンパイラ基盤の多くは，ネイティブコードの生成に加え，DWARF のデバッグ情報を生成するためのメタな記法を提供する．SML#がネイティブコードを生成する基盤として用いている LLVM [9] にも，DWARF の木構造を書くための記法が用意されている．本節では，LLVM における DWARF データ構造の記述について概観する．

LLVM は，プラットフォーム独立のコード記述言語 LLVM IR で書かれたプログラムを入力として受け取り，様々な CPU 向けのネイティブコードを生成するコンパイラ基盤である．1 つの LLVM IR プログラムは，関数または変数の宣言の列からなる．各関数は命令の列を持つ．これらのネイティブコードやメモリ上のデータ構造に直接対応する実体に加え，LLVM では，これらの実体に対する注釈として，デバッグ情報を書くことができる．

```
!0 = distinct !DICompileUnit
         (language: DW_LANG_C99,
          file: !1,
          producer: "Ubuntu clang version ...",
          ...,
          subprograms: !3)
!1 = !DIFile(filename: "func.c",
             directory: "/home/debug/...")
!2 = !{}
!3 = !{!4}
!4 = !DISubprogram
         (name: "main", scope: !1, file: !1,
          line: 3, type: !5, ..., scopeLine: 3,
          function: i32 ()* @main, variables: !2)
...
!11 = !DILocalVariable
         (tag: DW_TAG_auto_variable,
          name: "x", scope: !4, file: !1, line: 5,
          type: !7)
!12 = !DIExpression()
!13 = !DILocation(line: 5, column: 9, scope: !4)
...
```

図 2　LLVM IR での DWARF デバッグ情報の表現
Fig. 2　DWARF debug information in LLVM IR.

LLVM IR のデバッグ情報はすべて，プログラムのトップレベルに関数や変数とともに列挙され，それぞれにユニークな ID が振られる．ID が振られた各要素はそれぞれ，DIE ノードか行番号テーブルエントリのどちらかを表す．DIE 間の親子関係やネイティブコードとデバッグ情報の対応は，デバッグ情報間で ID を参照したり，デバッグ情報から実体のシンボル名を参照したり，あるいは実体からデバッグ情報の ID を参照したりすることで表現され，必ずしも DWARF のデータ構造とは一致しない．LLVM IR で書かれた DIE 構造の例を図 **2** に示す．この図において，`!0` や`!1` はデバッグ情報の ID である．`!DICompileUnit` や `!DISubprogram` は，DIE におけるコンパイル単位ノードやサブプログラムノードに直接対応する．それらの名前に続くレコード構造には，DWARF においてそれらのノードが持つ属性や，DIE での親子関係などを表すフィールドが並ぶ．たとえば，`!DICompileUnit` と`!DISubprogram` の親子関係は，`subprograms` フィールドから中間構造`!3` を介して参照されることで表される．`!DISubprogram` とコードの実体の対応は，`!DISubprogram` の `function` フィールドから，そのコードの実体に付けられたシンボル名（例では `@main`）を参照することで表現される．逆に，`!DILocation` や`!DILocalVariable` は，その位置情報に対応する命令や，その変数をアロケートする命令から，以下のようにして参照される．

```
%x = alloca i32, align 4
call void @llvm.dbg.declare
  (metadata i32* %x,
   metadata !11, metadata !12),
  !dbg !13
store i32 1, i32* %x, align 4, !dbg !13
```

このように，DWARF の構造と LLVM IR での表現の対応は一致するとは限らない．しかし，DWARF 構造を構成するために必要なソースコードに関する情報の種類と量はおおよそ等しい．これは，最終的な DWARF 構造に含める必要のあるすべてのデータを，余すことなくコンパイラが生成しなければならないことを意味する．

2.3 SML#からのデバッグ情報の生成

DWARF におけるソースコードの構造モデルや，その LLVM IR における記法は，どの言語にも現れる共通の構造を表している．関数型言語のプログラム構造も例外ではなく，この DWARF のソースコードモデルに基づいて自然に分類することができる．したがって，コンパイラが構文解析時に得たソースコードの位置情報やブロックのネスト関係を適切に保存し，実行コードとともに LLVM IR にするならば，ネイティブコードレベルデバッグに必要な多くの情報が自然と生成されるはずである．

一方で，DWARF で表現される要素が関数型言語の概念と一致しないものもいくつか存在する．その代表的なものが型情報である．DWARF の型情報は C 言語の型構造に従って設計されており，レコード型やバリアント型，第 1 級の関数型，多相型など，豊富な型構造を持つ関数型言語の型をすべて表現することは難しい．これらの型情報は主に，GDB がステップ実行中に変数の値を表示したり，実行を一時中断した文脈でユーザが C 言語の式を実行するときなどに使用される．したがって，関数型言語から GDB を使うという観点からは，これらの型情報は，表層言語での型情報よりもむしろ，その評価モデルを実現するためにヒープ中に作られるデータ表現に対するプリンタコードに対応すると考えられる．

C 言語と共通の自然なデータ表現を採用する SML#では，すべての基本型やレコード型は，データ表現に互換性のある C の型が存在するため，それらの型を用いた DWARF 型情報を構築することができれば，それらの型の値の GDB でのプリントは自然と達成されると期待できる．一方で，バリアント型や多相型の値のプリントには，データ表現レベルの議論を超える本質的な課題が存在する．それらへの対応は今後の課題とし 5 章で議論する．

変数の値の表示のためには，その値を保持するスタックフレーム上の領域とそのアドレスを LLVM IR に明示しなければならない．しかし，SML#は，最適なコード生成のために，ユーザ定義の変数を含むすべてのローカル変数を LLVM IR のレジスタにコンパイルする．したがって，`!DILocalVariable` を付与する対象の `alloca` 命令は出力されない．これは，メタ情報だけでなく，デバッグのためだけの，プログラムの実行とは無関係なコードも挿入しなければならないことを意味する．この方式の詳細は，実装とともに，4.2 節で述べる．

3. SML#コンパイラ内でのメタ情報の伝播

前節で述べたように，SML#コンパイラがソースコードの構造に関する情報を LLVM IR 生成フェーズまで保存するならば，DWARF 情報のほとんどは自然に出力できるはずである．しかし，計算機アーキテクチャに近い実行モデルを持ちソースコードとネイティブコードの対応がとりやすい C などの言語とは異なり，SML#は高度な言語機能を実現するため，大きな配置換えをともなうコード変換や，ソースコードには現れない項の生成などを繰り返し行う．ネイティブコードレベルデバッグに有益な形でデバッグ情報を生成するには，これらの変換に対しても位置情報が保存されるように考慮した，位置情報の伝播方式が必要である．

SML#コンパイラは，本研究に取り組む以前から，すべての中間表現のすべての項に，その項が由来するソースコードの位置情報を含めていた．SML#コンパイラは，ソースコード中のある 1 点の位置を，ファイル名，行番号，カラム番号の 3 つ組として，以下のレコードで管理する．

```
type pos = {fileName : string,
            line : int, col : int}
```

中間言語の項は，その項がソースコード中に存在している範囲を，この `pos` の 2 つ組として，以下の型で持つ．

```
type loc = pos * pos
```

構文解析時に生成される抽象構文木のすべてのノードに対して，SML#コンパイラはこの `loc` 型の位置情報を付与する．たとえば，抽象構文木を表すデータ型 `Absyn` において，組式のデータ構成子は以下のように定義されている．

```
datatype exp
  = ...
  | EXPTUPLE of exp list * loc
  ...
```

これらの位置情報の主な利用先は，後続する各コンパイルフェーズ（型推論など）が出力するエラーメッセージである．そのため，エラーメッセージを出力しうるフェーズに至るまでの間は，位置情報はよく管理されている．しかし，エラーメッセージに関係のないフェーズでは，位置情報は受け継がれてはいたものの，その位置情報はほとんど利用されてこなかった．

構文解析の後，SML#コンパイラは，高階関数や多相関数などの ML 系言語に共通した機能に加え，ランク 1 多相性，レコード多相性，C との直接連携，SQL との統合，自

然なデータ表現などを実現するために，23 のコンパイルフェーズと 12 の中間表現を経て，LLVM の C API の呼び出しを経由して C のヒープ内に作られた LLVM IR を出力する．著者らは，ネイティブコードレベルデバッグを実現するにあたり，すべてのフェーズを見直し，位置情報の伝播の過程や情報の正確さを調査した．以下，本章では，SML#コンパイラを構成するこれらのコンパイルフェーズのうち位置情報の伝播に注意を要するものについて，コンパイルフェーズごとにそれぞれ章を分けて，注意点，技術的課題，および本論文での対応を述べる．5 章では，ここで述べるフェーズごとの課題に加え，より実用的なデバッグ環境に向けた全体の課題を総括する．

3.1　ファイル読み込みフェーズ

このフェーズは，抽象構文木に含まれる外部ファイルへの参照をたどり，コンパイル単位を構成するすべてのファイルを読み込む．このフェーズではトップレベルのソースファイル名が失われる．ファイル名自体は，読み込んだ構文木の各ノードに `loc` 型の位置情報として含まれるものの，どの項がトップレベルのものかの判断はできなくなる．

この対策として，中間表現を変換するコンパイルフェーズの連鎖とは別に，トップレベルファイル名を LLVM IR 生成フェーズに送るように，コンパイラトップレベルを改良した．

3.2　名前評価

このフェーズでは，ML のモジュール言語を評価し，コア言語の同等な構造に変換する．この過程で，ストラクチャへのシグネチャ適用やファンクタは，コア言語の型注釈や関数に変換される．たとえば，

```
structure A : sig val x : int end =
struct
  local fun x y = y + 1
  in val x = x 3 end
end
```

は，このフェーズによって `structure` や `local` が取り除かれ，またそれに合わせて変数の ID が変更され，

```
fun A.x{1} y{2} = + (y{2}, 1)
val A.x{3} = A.x{1} 3
val A.x{4} = A.x{3} : int
```

に変換される（{} 内はコンパイラが内部的に割り当てる ID を表す）．

このフェーズではデバッグに大きな影響を与える変形を 2 つ行う．1 つは，識別子の変更である．SML#コンパイラはソースプログラム中の名前とは別にすべての識別子に一意の内部 ID を割り振るため，ソースプログラム中の名前はコード生成に不要である．しかし，変数の名前は，このフェーズより後のフェーズが出力するエラーメッセージに使用されるため，このフェーズでは注意深く管理されていた．これらの名前はデバッグ情報にもそのまま利用できる．

もう 1 つは，ソースコードのモジュールブロック構造と変数のスコープ構造の除去である．これらの構造は，DWARF ではレキシカルブロックに対応し，変数の値の表示などのために，GDB が実行コンテキストにおける変数名を管理するのに用いられる．スコープ構造の除去は，ネスト構造を有しないネイティブコードへのコンパイル過程において必要不可欠な処理である．そのため，スコープ構造をオブジェクトコードとして伝播することは難しい．DWARF のスコープ構造を生成するためには，位置情報のようにオブジェクトコードに乗せて伝播するメタ情報ではなく，オブジェクトコードとは独立なメタ情報として，オブジェクトコードと分けて伝播する必要がある．オブジェクトコードと独立したメタ情報の設計と実装は，変数の値のプリントともに今後の課題である．

3.3　型主導コンパイルのための型推論

このフェーズは，型推論の結果として型抽象式 $\Lambda t.e$ および型適用式 $e\ \tau$ をコードに挿入する．これらの式は単に静的な型付けのために導入されるものではなく，多相レコードや自然なデータ表現の実現のため，後続する型主導コンパイルフェーズにより実行コードにコンパイルされる [12], [13]．これらの式はソースコード中に存在しないため，抽象構文木に由来する位置情報をそもそも持たない．したがって，これらの式の位置情報を，挿入位置に合わせて，コンパイラが生成する必要がある．SML#コンパイラにおける位置情報を生成する一般的な方針は，挿入される項で囲われる式の位置情報を，挿入した項にコピーすることである．たとえば，多相関数 $\mathtt{id} : \forall t.t \to t$ に対する適用式

```
id 123
```

は，型適用式が挿入され，

```
(id int) 123
```

となる．このとき，式 (id int) の位置情報として，識別子 `id` のものをコピーする．

他のフェーズにおいても，生成される項がある 1 つの式を囲んでいたり，あるいはある 1 つの項に由来して生成されたりする場合は，それらの項の位置情報を生成された項にコピーしている．この方針は，期待されるとおりのデバッグ情報が受け継がれることが多く，また，コードの翻訳過程においても最も入手しやすい位置情報であるため，これらのどちらの観点からも多くの場合において自然かつ最良の方針である．著者らは，位置情報が流れる過程をたどる中で，多くの箇所で位置情報がこの方針に従って次の

フェーズに受け継がれていることを確認した．

3.4　パターンマッチコンパイル

制御の流れが分岐する位置は，ブレークポイントを置くなど，デバッグを開始する起点としてよく用いられる．したがって，パターンマッチを行う case 式へのブレークポイントの設定は，case 式全体よりもむしろ，個々の分岐先が選ばれたことに対して行えたほうが，デバッグの観点からは有利である．この点を考慮すると，前述の自然な方針により case 式から生成されるすべての式に case 式の位置情報をコピーするよりも，分岐コードへの関わりに応じて位置情報を付けた方が好ましい場合がある．

たとえば，以下のプログラムを考える．

```
1: fun f x =
2:     case x of
3:       (1, 2) =>
4:       1
5:     | (x, 3) =>
6:       f (4, f (5, x))
```

この case 式において実行時にどちらに分岐するかデバッガで調べたいとしよう．その場合，プログラマは，x が (1, 2) にマッチしたとき，あるいは (x, 3) にマッチしたときに止まることを期待して，3 行目と 5 行目にブレークポイントを置きたくなるかもしれない．ところが，SML#コンパイラは，この式を以下のようにコンパイルする．

```
1: fun f x =
2:   let fun g x =
6:           f (4, f (5, x))
2:       val y = #1 x
2:   in case y of
2:        1 =>
2:        (case #2 x of
2:           2 =>
4:           1
2:         | 3 => g y
2:         | _ => raise Match)
2:      | _ =>
2:        (case #2 x of
2:           3 => g y
2:         | _ => raise Match)
2:   end
```

行番号は，生成されたコードにコピーされた行番号を示している．マッチコンパイルで case から生成された式には case の行番号がコピーされるため，コンパイル後のコードからは 3 行目および 5 行目は失われている．そのため，3 行目および 5 行目には，そもそもブレークポイントを置くことができない．

この問題は，マッチコンパイルの過程で，パターンの位置情報がすべて無視されていることに起因する．しかし，マッチコンパイルは分岐を構成するパターンの並び全体から最適な分岐コードの生成を行うコンパイルアルゴリズムであるため，生成される分岐コードと個々のパターンの対応は必ずしも明確でない．マッチコンパイルフェーズを大きく変更することなくパターンの行番号をデバッグ情報に残すためには，分岐の本体式が評価される直前に何もしない命令を挿入し，その命令の行番号をパターンのものにするなどの工夫が必要である．

また，マッチコンパイラは，上述の例にもあるように，パターン集合から決定木を構築するときに分岐先を複製する．その際，コードの複製を避けるため，関数 g を作り，各分岐の先には g を呼び出す関数適用式を置く．これらの g を呼ぶ式は，分岐先が決定した後に実行されるコードである．したがって，分岐先にブレークポイントを置く観点からは，これらの関数適用式には，case のある 2 行目ではなく，分岐先コードである 4 行目や 6 行目の行番号を与えた方が，よりプログラマの直感に近い効果が得られる．

本論文で報告する実装では，これらの課題への対応は行っていない．上述した対応策の実装と評価は今後の課題である．

3.5　A 正規化

A 正規化アルゴリズム [8], [14] は，ネストした式のすべてに変数を束縛し，call-by-value 意味論に従った評価順に式を並べる．この変形により式のネストが直列化され，プログラムの構造がよりネイティブコードに近くなる．値呼び関数型言語をネイティブコードにコンパイルするときの基礎となるアルゴリズムの 1 つである．

しかし一方で，この変形のために，ステップ実行が奇妙な振舞いをするように見えることがある．たとえば，以下のように複数の関数適用を組み合わせることは関数型言語ではよくあることである．

```
1: fun f 0 =
2:     veryverylongfunctionname
3:       (veryverylongfunctionname y,
4:        f 2)
5:   | f _ = raise Error
```

最初の分岐に入ったときに実行を停止させようとして，ブレークポイントを 2 行目に設定したとする．このとき，ブレークポイントは確かに 2 行目に置くことができるが，実際には 2 行目で止まることなく，先に 4 行目の f 2 が実行され，5 行目の分岐に制御が移り，Error 例外が発生する．これは，A 正規化により，2〜4 行目の式が以下のコードにコンパイルされるからである．

```
3: val $1 = veryverylongfunctionname y
4: val $2 = f 2
3: val $3 = alloc ($1, $2)
```

```
2: val $4 = veryverylongfunctionname $3
```

Standard ML の評価規則 [11] に定義どおりに従うならば，3 行目以降よりも先に 2 行目の veryverylongfunctionname が評価されるはずである．その一方で，A 正規化アルゴリズムは，この値式の評価を正規化し除去する．結果として，プログラムは，ソースプログラムでの書かれた順番ではなく，関数適用が評価される順番に並べ替えられる．

この，ステップ実行からは奇妙に見える振舞いは，値への評価をモデルとする ML プログラムの意味と，プログラマが想起する実行トレースをモデルとする ML プログラムの意味が異なり，A 正規化アルゴリズムは前者を保存しても後者を保存しないことを意味する．これは，C 言語などの手続き型言語では自然に対応付けられるネイティブコードレベルとソースレベルのプログラムの振舞いが，関数型言語においては，たとえ call-by-value 意味論を採用する ML においても乖離することを示唆している．この乖離に対する根本的な解決には，ステップ実行単位を明確にしたうえでトレース意味論を定義し call-by-value 意味論との違いを明らかにするなど，緻密な分析が必要と考えられる．

本論文で報告する実装においては，A 正規化後の振舞いこそが観測されるべきネイティブコードの振舞いであると考え，この問題に対する解決を何も行っていない．ステップ実行の単位に関する問題は，A 正規化だけでなく，インライン展開などのコード変形にも関わりがある問題であり，完全な解決にはコンパイラ全体での包括的な対応が必要である．この問題も含めたステップ実行に関する今後の課題は 5 章で述べる．

4. LLVM IR デバッグ情報の生成

各フェーズによって伝播されたメタ情報は，最後に LLVM IR 生成フェーズにおいて LLVM IR のデバッグ情報として出力される．本章では，SML#コンパイラの LLVM IR 生成フェーズの拡張について報告する．この拡張は，ソースコードの構造と各要素の位置を表すデバッグ情報の生成，およびローカル変数の値の表示を実現する追加コードの挿入からなる．以下，これらの拡張についてそれぞれ章を分けて報告する．

4.1 位置情報の生成

SML#コンパイラは SML#自体で記述されている．各中間表現は再帰的データ型で定義されており，各コンパイルフェーズはある中間表現を受け取り別の中間表現を返す再帰的な関数として実装されている．

LLVM IR 生成フェーズは，クロージャ変換と A 正規化が行われた後に行われる．そのため，LLVM IR 生成フェーズへの入力は，関数や式がネストしない，LLVM IR に近い構造を持つ．LLVM IR 生成フェーズは，入力表現を構成する関数および命令を LLVM IR の関数および命令列にそれぞれ変換する．この変換は，トップレベル定義や関数，命令など，入力表現の構文構造それぞれについて LLVM IR の対応する構造に変換する関数の再帰的な組合せで構成される．一方，2 章で述べたように，LLVM IR のデバッグ情報はトップレベルに置かれ，関数や命令と相互参照する構造を取る．この構造の違いから，SML#の宣言的・再帰的な記述でコードの実体とメタ情報の間の相互参照関係を書くことに無理が生じた．そこで，本開発では，コンパイルフェーズを構成する関数は実行コードを返すこととし，メタデータ構造は書き換え可能な値（ref 型の値）を破壊的に更新して蓄積することで，個々の命令とメタデータを関連付けながら生成する方針をとった．コンパイルフェーズの最後に，蓄積されたメタデータは LLVM IR のトップレベル定義として実行コードに挿入される．

LLVM IR 生成フェーズは，次の 2 つのステップからなる．

- コンパイラ記述言語である SML#の内部データ構造として，LLVM IR の構文木を構築する．
- その構文木を再帰的にたどりながら，LLVM C API を手続き的に呼び出し，LLVM IR を C のヒープに構築する．

最初のステップは次の 3 段階で行われる．

- トップレベル定義の生成
- 関数の生成
- 命令の生成

このいずれの段階においてもメタデータの生成を行う．トップレベルで生成するメタデータは，コンパイル単位，ファイル，およびサブプログラムの集合を表すノード !DICompileUnit，!DIFile，および!DISubprogram である．このうち，!DIFile ノードの生成に必要なファイル名は，3 章で述べたとおり，中間表現とは別にファイル読み込みフェーズから受け渡すことで得る．!DIFile ノードはすべての!DISubprogram ノードから参照されるため，!DIFile ノードの ID をトップレベル変数として保持し，フェーズ全体から参照することとした．

入力表現に含まれる関数それぞれについて，!DISubprogram ノードと型を表すノードがそれぞれ生成される．これらの生成に必要な関数名や行番号情報には入力言語に付与されているものをそのまま用いる．

生成される LLVM IR 命令の多くには，1 つの命令に対して!DILocation ノードが 1 つ生成される．!DILocation ノードは，その命令が属する!DISubprogram ノードへの参照を含む．そのため，命令を生成する関数を，その命令を含む!DISubprogram ノードの ID を引数に追加して拡張した．

SML#における内部表現を LLVM IR に翻訳するためには，LLVM が提供する C の API を用いる．SML#の C との直接連携機能により，SML#で書かれた SML#コンパイ

```
 1: fun id x =
 2:     x
 3: fun fib 1 =
 4:     id 1
 5:   | fib 2 =
 6:     id 1
 7:   | fib n =
 8:     fib (n - 1) + fib (n - 2)
 9: fun main () =
10:     let
11:       val n = 3
12:       val result = fib n
13:     in
14:       print (Int.toString result);
15:       0
16:     end
17: val _ = main ()
```

図 3　ステップ実行例に用いるプログラム
Fig. 3　Program used for the experiments on step execution.

ラから，この API をそのまま利用することができる．ただし，いくつかの API は，C の API が存在せず，C++のメソッドとしてのみ提供されている．C++のメソッドを C の関数としてインポートするために，C++のメソッドそれぞれについて，C++のオブジェクトを C のポインタにラップするスタブ関数を実装した．

以上の拡張を施した結果，デバッグ情報が埋め込まれたオブジェクトファイルを生成する SML#コンパイラを得た．生成されたオブジェクトファイルをリンクし GDB から実行することで，ブレークポイントの設定とステップ実行ができることを確認した．以下に，SML#プログラムを GDB でステップ実行する例を示す．ステップ実行の対象プログラムを**図 3** に示す．なお，本プログラムは，フィボナッチ数の計算の過程を 1 行ずつ追うために意図的に改行を多くしている．

このプログラムをコンパイルし，生成したファイルを GDB で実行すると，埋め込まれたデバッグ情報が読み込まれる．

```
GNU gdb (Ubuntu 7.10-1ubuntu2) 7.10
...
Reading symbols from ./a.out...done.
```

break コマンドによってブレークポイントを設定しプログラムの実行を止めることで，プログラムの実行を対話的に観察することができる．たとえば，以下のコマンドはブレークポイントを 12 行目に設定する．

```
(gdb) break fib_ml.sml:12
Breakpoint 1 at 0x4036a2: file fib_ml.sml, line 12
```

実行すると 12 行目で止まる．

```
(gdb) run
...
Breakpoint 1, _SML_top () at fib_ml.sml:12
12              val result = fib n
```

```
(gdb) bt
#0  _SMLF3fib_27 () at fib_ml.sml:3
#1  0x000000000040371c in _SML_top ()
    at fib_ml.sml:12
#2  0x0000000000480526 in sml_run_toplevels
    (topfuncs=0x6da448, topfuncs@entry=0x6da350)
    at src/runtime/control.c:570
#3  0x000000000048481d in sml_run ()
    at src/runtime/top.c:482
#4  0x000000000040318e in main (argc=<optimized out>,
    argv=<optimized out>)
    at src/runtime/main.c:13
```

図 4　実行途中でのバックトレースの表示例
Fig. 4　Example of backtrace dump.

```
(gdb) list
1       fun id x =
2           x
3       fun fib 1 =
4           id 1
5         | fib 2 =
6           id 1
7         | fib n =
8           fib (n - 1) + fib (n - 2)
9       fun main () =
10          let
```

図 5　実行中のソースコードの表示例
Fig. 5　Example of source code listing.

```
(gdb) disas
Dump of assembler code for function _SMLF3fib_27:
=> 0x0000000000403360 <+0>:     sub    $0x58,%rsp
   0x0000000000403364 <+4>:     lea    0x2d5685(%rip),%rax
   # 0x6d89f0 <sml_check_flag>
   0x000000000040336b <+11>:    movq   $0x0,0x50(%rsp)
   0x0000000000403374 <+20>:    cmpl   $0x0,(%rax)
   0x0000000000403377 <+23>:    mov    %edi,0x44(%rsp)
...
```

図 6　逆アセンブルでマシンコードをトレースする例
Fig. 6　Example of tracing disassembled machine code.

実行制御のコマンド，`step`，`next`，`finish`，`continue` などはすべて使用可能である．たとえば，12 行目で止まった状況で `step` コマンドを実行すると，`fib` 関数の呼び出しが実行された後，`fib` 関数本体の先頭で停止する．

```
(gdb) step
_SMLF3fib_27 () at fib_ml.sml:3
3       fun fib 1 =
```

GDB によって提供されている他の機能もそのまま利用できる．バックトレースの表示，ソースコードの表示，逆アセンブルの機能を用いた例をそれぞれ**図 4**，**図 5**，**図 6** に示す．

4.2　ローカル変数の表示

2.3 節で述べたように，SML#は C 言語と同様の自然なデータ表現を採用しているため，SML#におけるローカル変数の値のいくつかは，GDB のプリント機能やメモリダン

プ機能をそのまま用いるだけで確認できると期待できる．しかし，これらの表示機能は，表示する値がメモリに書き込まれていることを前提とする．2.3 節で述べたとおり，SML#コンパイラはローカル変数を LLVM IR のレジスタにコンパイルするため，ローカル変数の値はレジスタが溢れない限りメモリに書き込まれない．変数の値の表示を実現するためには，メタレベルのデバッグ情報の生成に加えて，表示する値を保持するメモリ領域を確保する `alloca` 命令と，その領域に値を書き込む `store` 命令を，コンパイラが生成したコードに追加で挿入する必要がある．

著者らは，これらの手続きを以下の 3 段階で行うように SML#コンパイラを拡張した．

(1) 入力中間表現に現れるすべての変数定義についてその変数がソースコードに由来するものであるかどうかを判定する．ソースコード由来の変数である場合，以下を行う．
(2) 関数の先頭にその変数の型の `alloca` 命令を挿入し，その `alloca` が確保したメモリに対する `store` 命令を変数定義の直後に挿入する．
(3) その変数の名前と型を持つ`!DILocalVariable` ノードを，(2) で挿入した `alloca` 命令に関連付ける，`llvm.dbg.declare` イントリンシックの呼び出しを挿入する．その呼び出しには，変数の定義位置を表すデバッグ情報を付与する．

このうち，手順 1 および 2 は LLVM IR 生成フェーズの前処理フェーズの 1 つとして実装した．手順 1 における判定は，変数の名前に基づいて行う．SML#コンパイラはソースコード中の変数の名前を保存する．多くのコンパイルフェーズは，内部的に生成した変数に対して，Standard ML の構文が許さない名前を与える．したがって，Standard ML の構文が許す名前を持つ変数のほとんどは，ソースコードに由来するはずである．

手順 3 は，変数定義項の変数名，型注釈，および位置情報を用いて行う．変数名と位置情報はそのまま LLVM IR の形式で出力する．型情報については，SML#の型に対応する適切な DWARF の型を選んで出力する．本実装では，SML#における `int` 型，`real` 型，`real32` 型，およびリストや配列などのヒープアロケートされる値を持つ型を，それぞれ C 言語の `int` 型，`double` 型，`float` 型，および `void*`型を表す DWARF 型情報に対応付ける．レコード型など上記以外の型についても，2.3 節で述べたように，互換なデータ表現を持つ C 言語の型を表す DWARF 型情報を生成することで，その値を表示することができる．この方式で対応可能な型に対する網羅的な実装は今後の課題である．

図 7 に示す SML#プログラムを用いて，変数の値を GDB から確認する例を以下に示す．はじめに，ブレークポイントを設定しステップ実行することで，`makelist` 関数の内部にたどり着く．

```
 1: fun sumlist x =
 2:     case x of
 3:       nil => 0
 4:       | a::t => a + sumlist t
 5: fun makelist x =
 6:     case x of
 7:       0 => []
 8:       | n => n :: makelist (n - 1)
 9: fun main () =
10:     let
11:       val x = makelist 2
12:       val y = sumlist x
13:       val z = Int.toString y
14:     in
15:       print z
16:     end
17:val _ = main ()
```

図 7　ローカル変数の表示に使用するプログラム例
Fig. 7　Program used for the experiment on displaying local variables.

```
(gdb) break list.sml:5
Breakpoint 1 at 0x4032f7: file list.sml, line 4.
(gdb) run
...
Breakpoint 1, _SMLF7sumlist_27 () at list.sml:4
4               a + sumlist t
```

この場所で，info local コマンドによってスコープ内のプリント可能な変数を調べると，以下のように表示される．

```
(gdb) info local
T_1115 = 0x7fffe705b020
a = 2
t = 0x7fffe705b010
```

図 7 のプログラムにおいては，このブレークポイントに初めて到達したとき，`a` = 2 かつ `t` = 1 :: `nil` であると期待される．この表示では，変数 `a` の値が 2 であることが確認できる．`t` についてはメモリにアロケートされたコンスセルに束縛されるため，そのコンスセルのアドレスが表示されている．そのコンスセルの内容はそのアドレスのメモリをダンプすることで確認できる．`T_1115` は，ソースコードに現れない，コンパイラが内部的に生成した変数である．より実用的なデバッグ環境の実現のためには，ユーザ変数であることを表すフラグをすべての変数項に持たせるなど，名前から推測することなく内部変数とユーザ変数を区別する明確な基準を導入することが必要である．

5. 実用的なデバッグ環境に向けての課題

本論文で示したデバッグ環境によって，関数型言語 SML#で書かれたプログラムに対してブレークポイントの設定とステップ実行が可能になった．しかし，現状のデバッグ環境には様々な未解決な課題が残っている．それら

の課題の一部には，3 章や 4 章で指摘したものも含まれる．3 章でも述べたとおり，実用的なデバッグ環境を完成させるためには，個々のコンパイルフェーズや機能ごとだけではなく，コンパイラ全体として，これら課題への対応が必要である．本章では，これまでに指摘した問題も含めて，コンパイラ全体で包括的に対応すべき課題を列挙し，今後の展望を述べる．

5.1　適切な位置情報を得られない問題

本論文のコンパイラの拡張においては，ソースコードにおいて関数適用が行われる箇所の位置情報をもとに，命令にメタデータを付与することで，ブレークポイントの設定とステップ実行を可能とした．しかし，適切な位置情報を入手できず，ブレークポイントを設定することができなかった箇所も存在する．実装をもとにいくつかの例を試してみたところ，3 章で分析した箇所以外にも，ブレークポイントを設定できず不便に感じた箇所がいくつかあった．その代表的な例は，let 式や val 宣言など，変数の束縛が行われる箇所である．これらの式にブレークポイントが置けない原因は，3 章で触れた A 正規化と同様に，これらの構造と直接対応するネイティブコードが生成されるとは限らないからである．この問題を本質的に解決する 1 つの方法は，ここで述べた let や val，3 章で述べたパターンなど，プログラマがブレークポイントを置きたくなるソースコード上の構造を洗い出し，ステップ実行の単位を定め，トレース意味論を定義し，各コンパイルアルゴリズムがトレースを保存するように洗練することである．

また，インライン展開される式についても，コンパイル過程で位置情報が不適切に失われたり複製されたりするため，ソースコードの位置情報とネイティブコードのアドレスとの対応がプログラマの直感とは異なる形で出力されることがあった．インライン展開に対応するためには，インライン展開が行われたことを表す DWARF ノードの生成が必要である．この生成は，後述するソースコードのブロック構造を保存することとも密接に関連する．

5.2　適切なステップ単位

本論文では，GDB デバッガを用いて SML#のプログラムをステップ実行することを目標としてコンパイラの改良を行った．しかし，手続き型言語である C を主な対象とする GDB のステップ実行の単位は，言語の性質が異なる関数型言語 SML#に対して，必ずしも適切であるとはいえない．GDB デバッガは，ソースコードにおける「行」をステップ実行を行う単位として用いる．しかし関数型言語においてプログラムを適切なステップに分割することを考えると，関数適用や変数の束縛などの別のステップ単位が考えられる．言語の性質によるステップのとらえ方の違いをどのように対処するか，ステップ単位を変更するならばどのようにコード生成に変更を加えればよいか考察することは，前述した位置情報の適切な付与の基盤となる重要な課題である．

5.3　多相型を持つ変数のプリント

2.3 節で述べたように，レコード型を含む多くの型の変数の値は，互換な表現を持つ C 言語の型情報を生成することで，GDB の標準機能の範疇で表示が可能である．一方，多相関数の内側では．実行時でなければ型が定まらない変数が存在しうる．たとえば，多相関数

```
1: fun id x =
2:     x
```

の 2 行目にブレークポイントを置き，ステップ実行したとき，変数 `x` の値の表示方法は，`id` 関数の型インスタンスに依存する．したがって，この表示のためには，`x` に関する正確な実行時型情報が必要である．著者らは，動的型に関する理論 [7] を応用し，値だけでなく，実行時型情報も値と対にしてメモリに書き込み，値と型の対をプリントする対象として管理することで，多相関数内でも値のプリントを可能にすることを検討している．

5.4　ソースコードのブロック構造の DWARF 表現

2.1 節で概観したとおり，DWARF は，サブプログラムに加え，レキシカルブロックなど，ソースコードのブロック構造を表すノードを持つ．これらのノードをネストさせることで，ソースコード自体に含まれたり，コンパイル過程で生成されたりする，プログラムの構造を表現することができる．たとえば，インライン展開された関数本体を `inlined_subroutine` ノードとして表現することができる．これらを用いることで，ソースコードに関するより正確なデバッグ情報を GDB デバッガに伝えることができ，より緻密なステップ実行や変数の表示が可能となる．

しかしながら，3 章で述べたとおり，ソースコードのブロック構造に関する情報はコンパイル過程で除去されるべき情報であり，位置情報のようにオブジェクトコードに乗せて伝播することは困難である．ブロック構造を保存するために考えうる 1 つの戦略は，3 章でも述べたとおり，中間表現とは独立にソースコードのブロック構造を保存し，オブジェクトコードに対するメタ情報として管理することである．その詳細な検討は今後の課題である．

6.　関連研究

主要な関数型言語の処理系にはそれぞれ独自のデバッグ環境が存在する．

OCaml [5] では，バイトコードを用いたソースレベルデバッグと，DWARF の生成による GDB を用いたネイティブコードレベルデバッグの両方が提供されている．バイトコードでのデバッグは，専用のデバッガ ocamldebug が高

度な実行トレース制御と対話的な値のプリント機能を提供する．一方で，DWARF の生成では行番号情報のみ出力し，GDB で対話実行ができるにとどまっている．OCaml ではさらに，より高機能なネイティブコードレベルデバッグ環境を提供することを目指したコンパイラ拡張 libmonda [10] の開発が進められている．ドキュメントが少ないため詳細は比較は難しいが，libmonda と本手法は少なくとも，ソースレベルとネイティブコードレベルを横断する高機能なデバッグ環境を ML プログラムに対して提供するという方向性は共有しているといえる．

Haskell においても，GHC はバイトコードによる高度なソースレベルデバッグと，GDB によるネイティブコードレベルデバッグの両方を提供する．GHC の対話環境 ghci では，部分式にブレークポイントを設定しプログラムの動作を止める機能，停止した位置における自由変数について型や値を調べる機能，変数に対して評価を強制する機能，実行の履歴を遡る機能など，高度な機能が提供されている．多相関数の中での変数のプリントについても，変数を評価し値を得ることで型情報を回復し表示することができる．ネイティブコードレベルデバッグ環境としては，GDB などの対話型デバッガからブレークポイントやバックトレースなどの機能を用いることができることがユーザーズマニュアル [4] に示されている．しかし，ユーザーズマニュアルにも書かれているとおり，ソースコードとネイティブコードの対応関係は十分に保持されていない．

Scala はいくつかの統合開発環境でサポートされており，それら統合開発環境のデバッガに加え，より使いやすくするためのプラグインが利用可能である．たとえば，Scala IDE 3.0 [6] では，Scala Debugger と呼ばれる Scala に特化したデバッグ環境が提供されている．Java VM で動作する Scala プログラムでは，Java と同様に，Java VM が持つ高度なリフレクション機能を活用した可能である．ただし，値のプリントについては Scala の内部表現が見えるなど，完全な情報が出力されないケースがある．Scala Debugger はさらに，クロージャ内部に入り込むステップ実行など，Scala 独自の言語機能に対応する．

F# [2] は，その言語処理系が統合開発環境 Visual Studio とともに配布されており，他の.NET 上で動作するプログラミング言語と同様のデバッグ環境が提供されている．ただし，変数の持つ値の表示や，デバッグ中のコードの編集はできず，VisualStudio が提供するすべての機能が使用できるわけではない．

総じて，関数型言語に置けるデバッグ環境はバイトコードインタプリタを用いたソースレベルデバッグ環境が主流であり，高度な機能が提供されている一方，ネイティブコードに対して同じレベルの環境を提供するまでには至っていない．本論文は，ネイティブコードレベルでのデバッグに主軸を置き，C 言語などの手続き型言語との同水準の，ソースレベルとネイティブコードレベルの振舞いを横断的に観察できるデバッガの開発を目指している．

7. まとめと今後の課題

本論文では，SML#プログラムをネイティブコードにコンパイルして生成したファイルを GDB から実行し，動作を確認するための開発の報告を行った．C と同様のデータ表現と直接連携機能を持つ SML#の特長を利用し，コンパイラフロントエンドの中間表現生成フェーズを改良し，適切なデータ構造生成を行うことで，GDB が受理可能なデバッグ情報の生成を行った．コンパイラの改良により，位置情報とスコープについてのメタデータを生成することで，ブレークポイントの設定とステップ実行の機能を用いることが可能になった．また，デバッグのためにメモリ確保命令を挿入し，変数名とスタックフレーム上のオフセットを用いてデバッグ情報を生成することで，基本型を持つローカル変数のプリントが可能になった．

本論文は，SML#における実用的なデバッグ環境を実現することに向けた最初の試みである．本論文で報告したコンパイラの開発により，既存の高性能デバッガを SML#で活用するための準備は整ったといえる．本論文で未解決としてあげた課題に取り組み，SML#のデバッグ環境の完成を目指す予定である．

謝辞　本研究の一部は，JSPS 科学研究費補助金課題番号 25280019「ML 系多相型言語 SML# の実用化技術に関する基礎研究」および JSPS 科学研究費補助金課題番号 15K15964「実用プログラミング言語のための系統的言語開発基盤の実現」の助成を受けて実施されたものです．

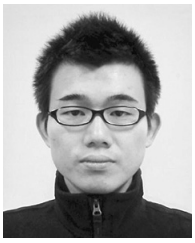
参考文献

[1] DWARF Debugging Information Format Version 3.
[2] F#, available from ⟨http://fsharp.org⟩.
[3] GDB: The GNU Project Debugger, available from ⟨https://www.gnu.org/software/gdb/⟩.
[4] Glasgow Haskell Compiler User's Guide, available from ⟨https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/index.html⟩.
[5] OCaml 4.00.0 Changes, available from ⟨http://caml.inria.fr/pub/distrib/ocaml-4.00/notes/Changes⟩.
[6] Scala IDE for Eclipse, available from ⟨http://scala-ide.org/⟩.
[7] Abadi, M., Cardelli, L., Pierce, B. and Plotkin, G.: Dynamic Typing in a Statically Typed Language, *ACM Trans. Program. Lang. Syst.*, Vol.13, No.2, pp.237–268, DOI: 10.1145/103135.103138 (1991).
[8] Flanagan, C., Sabry, A., Duba, B.F. and Felleisen, M.: The Essence of Compiling with Continuations, *Proc. ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, PLDI '93*, pp.237–247, DOI: 10.1145/155090.155113 (1993).
[9] Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proc. International Symposium on Code Generation and Optimization, CGO '04*, pp.75–86, DOI:

10.1109/CGO.2004.1281665 (2004).
[10] libmonda, available from ⟨https://github.com/mshinwell/libmonda⟩.
[11] Milner, R., Tofte, M. Harper, R. and MacQueen, D.: *The Definition of Standard ML*, The MIT Press, revised edition (1997).
[12] Nguyen, H.-D. and Ohori, A.: Compiling ML Polymorphism with Explicit Layout Bitmap, *Proc. 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, *PPDP '06*, pp.237–248, DOI: 10.1145/1140335.1140364 (2006).
[13] Ohori, A.: A polymorphic record calculus and its compilation, *ACM Trans. Program. Lang. Syst.*, Vol.17, No.6, pp.844–895 (1995).
[14] Ohori, A.: A Curry-Howard Isomorphism for Compilation and Program Execution, *Typed Lambda Calculi and Applications*, *TLCA1999*, Lecture Notes in Computer Science, Vol.1581, pp.280–294 (1999).
[15] SML# Project, available from ⟨http://www.pllab.riec.tohoku.ac.jp/smlsharp/⟩.

大野 一樹

1994 年生まれ．2016 年東北大学情報知能システム総合学科卒業．2018 年同大学大学院情報科学研究科システム情報科学専攻修了．採録時現在，日立建機株式会社に所属．ソフトウェアの開発に関する研究に興味を持つ．



上野 雄大　（正会員）

1981 年生まれ．2009 年東北大学情報科学研究科博士課程修了．博士（情報科学）．同年東北大学電気通信研究所助教．2016 年同研究所准教授，現在に至る．プログラミング言語に関する研究に従事．

大堀 淳　（正会員）

1957 年生まれ．1981 年東京大学文学部哲学科卒卒業．1989 年ペンシルバニア大学計算機科学科博士課程修了．Ph.D．1981 年沖電気入社．英国王立協会特別研究員（グラスゴー大学），沖電気関西総合研究所特別研究室長，京都大学数理解析研究所助教授，北陸先端科学技術大学院大学教授を経て，現在，東北大学電気通信研究所教授．プログラミング言語に興味を持つ．