SML#のためのコードレベルデバッグ環境の構築に向けて

大野 一樹　上野 雄大　大堀 淳

関数型言語である SML#は，C や Java などの手続き型言語に比べて，高い信頼性と安全性を持つという点で優れており，安全で信頼のおけるソフトウェアの生産に大きく貢献すると期待されている．しかし SML#には十分なデバッグ環境が存在しない．現状では，SML# で書かれたプログラムを逐次実行し，動作を確認するというようなことができない．そこで，SML#の実用化にとって重要なデバッグ環境の構築を目指す．本研究ではその第一歩として，ステップ実行とブレークポイントの設定を目指して研究を行った．実現のために，FFI を拡張し，既存のデバッガである GDB を SML#から用いる方針を用いた．本研究の最終目標は，手続き型言語同様にコードレベルのデバッグ環境を導入することとする．

1　はじめに

関数型言語におけるデバッグ環境は，C などの主流な言語に比べて十分であるとは言えない．C や C++においては，GDB [1] などの対話型デバッガを用いることで，ブレークポイントの設定やステップ実行が，機械語で書かれ CPU が直接理解できるネイティブコードに対して行うことができる．一方，関数型言語においてはネイティブコードに対して対話型のデバッグを行う環境は存在せず，限定的な機能を提供するに留まっている．一例として，関数型言語におけるデバッグ環境としては OCaml [2] におけるバイトコードデバッガ ocamldebug が挙げられる．しかし，仮想マシンで実行するための中間表現であるバイトコードは，機械語で書かれ，CPU が直接理解できるネイティブコードとの環境の違いにより，実際に動作するプログラムと挙動が違う可能性がある．そのため，ネイティブコードを直接デバッグする環境を構築することは，プログラムの挙動を実際の環境に準じた形式で確認することができるようになり，有用であると言える．そこで，本研究では，関数型言語 SML# [3] で書かれたプログラムのネイティブコードを逐次実行し，動作を確認するためのデバッグ環境の構築を目指す．

ネイティブコードのデバッグのための対話型デバッガを新規に構築することは，機械語レベルの処理を理解する必要があり非常に難易度が高いと予想される．そこで我々は，SML#コンパイラが，C との直接連携機能を持ち，かつバックエンド以降のコンパイルのシナリオが C と同じであることに着目した．すなわち，フロントエンドでデバッグのために必要な情報を C と同様に出力することができれば，バックエンドでは C と同じ形式の実行ファイルが生成されるため，GDB などの C と同じデバッガを用いることができると考えられる．そこで，我々はネイティブコードをデバッグする環境を SML#に導入するために，C で用いられているデバッガである GDB に対応したデバッグ情報をコンパイラフロントエンドで出力することを目指す．ここではその第一歩として，プログラムの動作を途中で止めるブレークポイントの設定と，プログラムを 1 ステップごとに実行するステップ実行の導入を目標とする．この目標を達成するために以下のような C のシナリオを模倣することが，C と互換

Towards Construction of Source Level Debugging Environment for SML#

Kazuki Ono, 東北大学情報科学研究科, Graduate School of Information Sciences, Tohoku University..

Katsuhiro Ueno, Atsushi Ohori, 東北大学電気通信研究所, Research Institute of Electrical Communication, Tohoku University.

性のある SML#では可能なはずである．

C において，デバッガを用いてデバッグを行うためには，プログラムをコンパイルする際にソースコードについての追加情報を生成する必要がある．ここで生成される追加情報はソースコードにおける行番号や列番号などの，コードレベルのデバッグにおいて必要な情報であり，以下デバッグ情報と呼ぶ．デバッグ情報をコンパイラフロントエンドで生成し，デバッガが受け取ることができる適切なデータ構造に変換することで，デバッガにデバッグ情報を渡すことができる．デバッガは，受け取ったデバッグ情報を解釈し，ソースコードとマシンコードの対応関係をもとに，ユーザーにデバッグ環境を提供する．

よって，同様のシナリオを SML#で模倣することで，SML#にデバッグ環境を導入することを目指す．本論文では，以上の方針をもとに，ブレークポイントの設定とステップ実行の実現のためにフロントエンドで出力するべき適切なデバッグ情報の形式と意味についての概要，実装方針を述べる．本稿の構成は以下のとおりである．2 節で,LLVM フレームワークによって提供されている中間表現である LLVM IR におけるデバッグ情報の形式について簡単に述べる．3 節でデバッグ情報を SML#コンパイラで生成し，バックエンドまで伝えるための実装方針について述べる．最後に 4 節でまとめと今後の課題について述べる．

2 LLVM IR におけるデバッグ情報の形式

本節では，SML#コンパイラにおいて用いられている中間表現 LLVM IR におけるデバッグ情報の形式について述べる．LLVM IR は，LLVM フレームワークにおいて提供されており [4]，LLVM IR において，プログラムは命令列にコンパイルされる．ここで，LLVM IR においてはデバッグ情報を構築するための API が用意されており，構築したデバッグ情報を命令列に付与することでソースコードと命令の対応関係を表現することも，API を用いることで可能になる．

また，LLVM IR ではデバッグ情報は DWARF3.0 [5] の形式で表現される．DWARF3.0 ではデバッグ情報はもととなるノードから，適切なノードや値を参照する木構造を用いて表現される．実際に，図 1 に，デバッグ情報を生成する際のもととなる CompileUnit ノードの例を示す．

```
!DICompileUnit(language: SML#, file: !4,
isOptimized: false, runtimeVersion: 0,
emissionKind: 1, subprograms: !5)
```

図 1　compile_unit の例

図 1 のようなデバッグ情報を，LLVM で用意されている API を用いて構築し，命令列に付与するための方針を次節で述べる．

3 SML#コンパイラにおける実装方針

本節では，2 節で述べたデバッグ情報を SML#コンパイラで生成し，バックエンドまで伝えるための実装方針について述べる．SML#コンパイラにおいて，ブレークポイントの設定とステップ実行を行うために，LLVM IR 生成フェーズにおいて以下のような拡張を行う．

1 デバッグ情報を生成するために必要な情報（行番号情報）を入手する
2 入手した情報をもとに，デバッグ情報（DIE）を生成する
3 生成したデバッグ情報を命令列に付与する

SML#コンパイラにおいて，命令に対応したソースコードの行番号情報は，LLVM IR 生成フェーズより手前のフェーズで生成され，対話モードでのエラーメッセージの生成などで用いられている．そのため，第 1 の拡張を行うために，手前のフェーズから行番号情報を関数の返り値として渡すことで，LLVM IR 生成フェーズにおいて行番号情報を活用することを可能にする．

入手した行番号情報をもとに，第 2，第 3 の拡張を行う．デバッグ情報の生成と命令列への付与は，LLVM フレームワークによって提供されている C/C++ API を用いる．C との直接連携機能を持つ SML#では，C で書かれた関数をインポートし，使用することが可能である．この API を用いることで，行番号情報をもとに適切なデバッグ情報を生成し，命令にデ

バッグ情報を付与することが可能である．以下に，CompileUnit ノードを生成するための API（createCompileUnit 関数）をインポートし，使用した例を図に示す．ここで，CreateCompileUnit 関数の第一引数である LLVMDiBuilderRef は，LLVM フレームワークで提供されているデバッグ情報生成のためのクラスへの参照を表す．

```
type LLVMMetadataRef = unit ptr

val sml_CreateCompileUnit =
 _import "CreateCompileUnit"
 : (LLVMDIBuilderRef, string, string, string)
       -> LLVMMetadataRef
```

図 2　API のインポート

図 2 の関数を用いることで，コンパイルユニットを表すノードと，ファイル名とディレクトリを表すノードを生成することができる．図 3 に使用例と生成されるデバッグ情報を示す．

```
（インポートした API に情報を渡す）
val compileunit =
 sml_CreateCompileUnit (dibuilder,
 "file.sml", "path/to/file", "SML#")

（出力されるデバッグ情報）
!0=!DICompileUnit(language: DW_LANG_C, file:!1,
   producer: "SML#", isOptimized: false,
   runtimeVersion: 0, emissionKind: 1)
!1=!DIFile(filename: "file.sml",
   directory: "path/to/file")
```

図 3　API の使用例と生成されるデバッグ情報

これらの API を活用し，適切なデバッグ情報を生成することで，SML#コンパイラのデバッグ環境を構築する方針とする．

4　まとめ

本稿では，SML#で書かれたプログラムに対して，GDB デバッガを用いてブレークポイントを設定し，ステップ実行を行うための調査と，デバッグ情報を生成するための実装方針について述べた．本稿で挙げた方針に基づき，SML#コンパイラの中間表現生成フェーズにおいて適切に実装すれば，目標を達成できると期待できる．現在，実際のコンパイラを用いてフェーズの拡張を行っている．

今後の課題は，関数型言語における適切なステップ単位の考察である．現状，C の手法を元にしているため，ソースコードにおける行をステップ単位として実装しているが，関数型言語においては，より適切なステップ単位があるのではないかと考えられる．適切なステップ単位を考察し，そのステップ単位に適したデバッグ情報を生成するための改良をコンパイラに対して行うことで，より高機能なデバッグ環境を実現することが方針として考えられる．

謝辞 本研究の一部は JSPS 科研費 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 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ある．

参 考 文 献

[1] GDB: The GNU Project Debugger, https://www.gnu.org/software/gdb/
[2] The Caml Language, https://caml.inria.fr/.
[3] SML#プロジェクト, http://www.pllab.riec.tohoku.ac.jp/smlsharp/ja/.
[4] Source Level Debugging with LLVM, http://releases.llvm.org/3.7.0/docs/SourceLevelDebugging.html
[5] DWARF 3.0 Standard, http://dwarfstd.org/Dwarf3Std.php