情報処理学会論文誌　プログラミング　Vol. 2　No. 3　48–56 (July 2009)

Haskellプログラムの開発を支援するGHCiデバッガフロントエンド

根岸純一[†1]　岩崎英哉[†2]

遅延評価型関数型言語 Haskell で書かれたプログラムのデバッグは，処理系 GHC に組み込まれている GHCi デバッガを用いて，実行時の変数の情報を確認する等して行うことが一般的である．GHCi デバッガは，対話的な環境におけるコマンド入力により，ブレイクポイントの設定，ステップ実行，状況に応じた変数の評価や追跡機能等の機能を提供する．しかし，GHCi デバッガは，グラフィカルなユーザインタフェースによる，ステップ実行の可視化が必要であることが，開発者らにより指摘されている．そこで，本論文では GHCi デバッガに対して，デバッグ実行時の表示を追加し，操作を簡易化するための機構を，ウェブブラウザを利用したフロントエンドとして実装する．本フロントエンドは，GUI を用いてソースコードと評価順序を確認しつつ履歴を簡単に戻りながら遅延オブジェクトの内容を評価できるデバッグ支援環境を提供する．実装には，Python および jQuery ライブラリを用いた．本フロントエンドは，GHCi デバッガの出力を JavaScript を用いた動的 HTML として整形する機構とウェブブラウザの操作を GHCi デバッガのコマンドに対応付ける HTTP サーバの 2 つで構成されている．GHC とウェブブラウザが使用できる環境であれば，環境を選ばず簡単に導入が行えることが特長となっている．

GHCi Debugger Front-end for Haskell Programming

Junichi Negishi[†1] and Hideya Iwasaki[†2]

To debug a program written in Haskell, it is usual to use the GHCi debugger in the distribution set of the GHC (Glasgow Haskell Compiler) interpreter. The GHCi debugger provides various interactive abilities such as setting breakpoints, single-step execution, inspecting variables, and execution in trace mode. However, it is pointed out by the developers of the GHCi debugger that visualization of step executions in a graphical user interface is necessary. Thus, we designed and implemented a front-end of the GHCi debugger with a graphical user interface that runs on an usual Web browser to help the debugging operations easier. This front-end offers a debug-supporting environment that enables users to easily confirm the evaluation order and to force delayed objects. The front-end is implemented by Python and jQuery library. The front-end is composed of two parts; one converts an output of GHCi into a dynamic HTML with JavaScript, and the other associates the operations of the user on a Web browser with the commands of the GHCi debugger. It can be used in any programming environment in which GHCi and a Web browser are installed.

1. はじめに

Haskell 等の最近の純粋な関数型言語は型推論，遅延評価といった多くの先進的機能を備えており，注目を集めている．しかし，遅延評価を行う言語のデバッグは，評価のタイミングが実行時まで予測できないため，C 言語等の先行評価を行う言語と同様のデバッグ手法を用いることができない．これは，値の確認のために式を評価しようとすると，本来必要ではない評価をしてしまい，評価順序の変更や余計な遅延オブジェクトの評価を招き，エラーにより実行が停止したり，無限リストを延々に評価したりする等して，実行結果を変化させる可能性があるからである．そのため，遅延評価を行う言語で書かれたプログラムのデバッグの際には，余計な遅延オブジェクトを評価せずに，デバッグの対象となる関数から必要な情報を取得する必要がある．

現在，Haskell プログラムのデバッグに有効な手段は，Haskell の標準的な処理系である GHC（Glasgow Haskell Compiler）付属の GHCi デバッガ[1),⋆1]を使用することである．処理系組み込みの機能で，使用に際して制約が少なく，ブレイクポイント設定の機能等により，既存の遅延評価型言語のデバッガではできなかった，ステップ実行によるデバッグが可能になった．これは C 言語における gdb と同様に，対話的操作で実行を進める．

しかし，Haskell プログラムには，遅延評価に起因する以下のような特有の実行形態がある．

- 連続して実行される処理がプログラムにおいて隣接しているとは限らない．
- 変数の値が遅延されている．

前者は，C 言語等のように基本的にはプログラムに記述された文の順に実行されるとい

†1 電気通信大学大学院電気通信学研究科
Graduate School of Electro-Communications, The University of Electro-Communications
†2 電気通信大学情報工学科
Department of Computer Science, The University of Electro-Communications
⋆1 GHCi デバッガが GHC に付帯するようになったのはバージョン 6.8.1 からである．

う保証がないため，GHCi デバッガを使っても直感的なステップ実行ができないことを意味する．また，後者は，GHCi によるブレイクポイントの停止時に，値が未評価のため内容が分からないことを意味する．値の確認のために評価を強制すると，後続の実行に影響を与え，望みのブレイクポイントを通過してしまい，最初から再実行をする必要が生じることもある．これらの問題点に対応するには，実行過程の視覚化により履歴の情報を分かりやすくし，操作の簡易化でステップ実行を容易にすることが有効であることが，文献 1) で指摘されている．これらの機能により，デバッグが容易になると期待できる．

本研究では，遅延評価型の関数型プログラミング言語として Haskell，処理系は GHC を対象として，GHCi デバッガを用いたデバッグを支援する環境を開発する．目的は GUI を用いた視覚的なデバッグ環境により，直感的な操作を可能にして，デバッグやトレースを簡易化することである．具体的には，クリック等の操作でコマンド入力を代替することで使用の敷居を下げる．そして，本機構自体は小さなものとすることで，導入を簡易化することを目指す．

本論文の構成は次のとおりである．まず，2 章で Haskell を対象としたデバッグの関連研究について説明する．3 章では，GHCi デバッガについて説明し，続いて 4 章で GHCi デバッガの簡易使用のための方針を述べ，5 章では本機構の実装の詳細について述べる．6 章では実装した機構について評価を行う．最後に，7 章で本論文をまとめる．

2. 関 連 研 究

Haskell のデバッグはコンパイル時の静的な型推論や対話環境上のユニットテスト，`trace`[⋆1]関数による出力の追加等が標準的だが，他にも以下のような機構がある．

Hood [2] は余計な遅延オブジェクトを評価しない出力関数 `observe` を提供する．`trace` と異なる点は，無理にデバッグ対象を文字列化する必要がないことである．制限として，適用可能なデータ構造が Hood で `observe` 用の出力が定義されているものに限定される[⋆2]．`observe` の場合は，デバッグ対象の式が評価済みかどうかを気にせず適用させることができる．

Hat [3] はコンパイルの際にソースコードに専用の出力関数を追加するとともに，ライブラリをすべて Hat が用意した出力関数が追加されたものを使うように変更する．コンパイルして生成された実行ファイルには，実行時の情報を出力する機能が加えられている．プログラムを実行すると，通常と同様の処理が終わったあとで，実行時の情報が保存されているファイルが生成される．このファイルをもとに，実行のトレースやコールグラフの生成といった，多彩な処理を行うことができる．

Buddha [4] は，アルゴリズミックデバッグを行う．Hat と同様に，コンパイルの際に出力関数を挿入する手法を用いているが，生成されるのはアルゴリズミックデバッグ用の実行ファイルである．通常の処理と同時に実行時の情報を木の形で保持し，その後，関数の引数と返り値が表示されるので，ユーザが正誤を入力として与える操作を繰り返していくと間違いを含む関数名を指摘する．

GHood [5] は，Hood に視覚的な出力の機能を追加したもので，デバッグ対象の式が評価されていく過程を木の形で図示化し，アニメーション表示をする．図の表示には Java を用いている．

Haskell Program Coverage [6] は，GHC に組み込みの機構で，プログラムの実行時にプログラムの分岐処理に関して真偽値を記録しておき，通過した回数や真偽値の出現の割合を出力する．また，真偽値の出現パターンに応じてソースコードに色付けをした HTML を出力させることができる．

Visual Haskell [7] は，Microsoft の Visual Studio で Haskell のプログラムを開発するための統合開発環境である．ソースコードの入力支援等の機能を提供しているが，デバッグに関しての特別な機能はない．

これらと比べると GHCi デバッガは，以下の点で優れている．

- ブレイクポイントが設定可能である．
- ソースコードに変更を加えずに使用可能である．
- 処理系組み込みのため適用可能なデータ構造の制約が少ない．

また，Hood のようにソースコードに変更を加えるものや，Hat のように適用ライブラリに制限のあるものに比べ，簡単に使用することができる．しかし，目的の値を確認するために，大量のコマンド入力が必要かもしれない点や，表示領域に対して，出力が大量になり，履歴の巻き戻りの際に出力が流れてしまうような不便な点があるので，本研究ではこれらの問題点を解決する．

3. GHCi デバッガ

GHCi デバッガは GHC に組み込まれており，対話環境で解釈実行されているモジュー

⋆1 使用のためには Debug.Trace モジュールをインポートする必要がある．
⋆2 Haskell の処理系 Hugs では，より多くのデータ構造に適用可能な HugsHood が提供されている．

```
1: qsort [] = []
2: qsort (x:xs) = qsort left ++ [x] ++ qsort right
3:   where (left,right) = (filter (<=x) xs, filter (>x) xs)
```

図 1　qsort.hs のソースコード
Fig. 1　Source code of qsort.hs.

ル[*1]に対してデバッグを行う．ブレイクポイントの設定，ステップ実行，実行の追跡といった機能を提供する．まず，関数や行にブレイクポイントをセットし，デバッグ対象となる関数を含む式を実行する．すると，ブレイクポイントで処理が停止するので，その後はステップ実行で処理を進めながら，状況に応じて，評価が遅延されている変数を強制的に評価し，内容を確認する．さらに，実行の追跡機能により，各ステップ実行における停止時のソースコードの位置，束縛変数の情報が履歴[*2]として保存されるので，以前のブレイクポイントに戻って，変数の内容を確認することもできる．特徴としては，処理系組み込みのため言語の拡張やライブラリに制限がなく使用できることや，ソースコードに変更を加えずに使用可能なこと，作成途中のプログラムでも，GHCi で実行可能な状態になりさえすれば，適用可能であることがあげられる．

ブレイクポイントの設定，ステップ実行，実行の追跡の機能のひととおりを**図 1** のクイックソートのプログラムに対して使用した実行例を**図 2** に示す．ただし，一部の出力を省略し，変数の情報を 1 行にまとめている．

1 行目で qsort.hs という，クイックソートを行う関数 qsort が定義されたファイルを読み込み，2 行目の:break コマンドで関数 qsort にブレイクポイントを設定している．続いて，4 行目の:trace コマンドで各ステップの履歴をとるように指定しつつ，qsort に適当な引数を与えて実行する．7 行目で qsort のブレイクポイントに到達したため，一時停止しているが，この時点では関数の内部まで評価が進んでいないため，:step コマンドで評価を進める．8 行目でソースコード中の 2 行目の 15 文字目から 46 文字目の範囲で評価が停止しており，9 行目に，部分式の結果_result と，束縛している変数名一覧が表示されている．左辺が変数名と型，右辺が値だが，現時点では値が未評価のため，右辺が_と表示されていて詳細は分からない．10 行目では，停止した位置周辺のソースコードを表示する:list コマンドを使用している．13 行目の^で示されている部分が停止した場所である．15 行目の:step コマンドで評価を進めたのち，前のステップに処理を戻したいと思った場合，18 行目のように:history コマンドで戻るステップの情報を確認し，22 行目のように:back コマンドでステップの履歴を戻ることができる．25 行目では，left という変数の値を:force コマンドにより評価して返り値を確認している．

ほかにも，行や例外に対してブレイクポイントを設定することもできる．この場合も，上

```
 1:  Prelude> :load qsort.hs
 2:  *Main> :break qsort
 3:  Breakpoint 0 activated at qsort.hs:(1,0)-(3,55)
 4:  *Main> :trace qsort [8, 4, 0, 3, 1, 23, 11, 18]
 5:  Stopped at qsort.hs:(1,0)-(3,55)
 6:  _result :: [a] = _
 7:  [qsort.hs:(1,0)-(3,55)] *Main> :step
 8:  Stopped at qsort.hs:2:15-46
 9:  _result::[a]=_  left::[a]=_  right::[a]=_  x::a = _
10:  [qsort.hs:2:15-46] *Main> :list
11:  1  qsort [] = []
12:  2  qsort (x:xs) = qsort left ++ [x] ++ qsort right
13:                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
14:  3    where (left,right) = (filter (<=x) xs, filter (>x) xs)
15:  [qsort.hs:2:15-46] *Main> :step
16:  Stopped at qsort.hs:2:15-24
17:  _result :: [a] = _  left :: [a] = _
18:  [qsort.hs:2:15-24] *Main> :history
19:  -1  : qsort (qsort.hs:2:15-46)
20:  -2  : qsort (qsort.hs:(1,0)-(3,55))
21:  <end of history>
22:  [qsort.hs:2:15-24] *Main> :back
23:  Logged breakpoint at qsort.hs:2:15-46
24:  _result::[a]    left::[a]    right::[a]    x::a
25:  [-1: qsort.hs:2:15-46] *Main> :force left
26:  left = [4,0,3,1]
```

図 2　GHCi デバッガの使用例
Fig. 2　Example of use of the GHCi Debugger

ほかにも，行や例外に対してブレイクポイントを設定することもできる．この場合も，上

⋆1 コンパイル済みコードに対してはデバッグできない．
⋆2 現在の GHCi は最後の 50 ステップを履歴に記憶する．

で述べたように GHCi デバッガを操作することで，処理過程を確認することができる．この場合，自動的に関数の中までステップ実行を行い，目的の部分式が評価されるときに停止をする，という違いがある．

4. GHCi デバッガ簡易使用のための方針

GHCi デバッガを用いれば，対話的にブレイクポイントを用いて，GHCi の評価の進行順で処理をたどることができる．これにより，Hood，Hat 等の既存の Haskell を対象としたデバッガと異なり，未完成のプログラムにソースコードの変更とコンパイルをすることなく，デバッグを行うことができる．しかし，GHCi デバッガを用いてデバッグを行おうとした場合，以下のような問題点がある．

- 繰り返しコマンドを入力する必要がある．
- 表示される情報が不親切である．

前者は，複数箇所にブレイクポイントを設定した場合や，ステップ実行により少しずつ評価を進める場合に起きる．評価を進めるには，処理に応じた分だけのコマンド入力が必要となるため，非常に手間がかかる．また，ブレイクポイントを抜けた場合は，履歴を戻ることができないため，最初からステップ実行をやり直す必要がある．

後者は，たとえば，`:history` コマンドで戻れる履歴を確認する際にソースコードが表示されず，どのような処理が行われていたか分かり難いことである．また，ターミナルで実行をした場合に，大量の出力を分析するのは大変である．

このような問題点を解決するためには，以下のような仕組みが必要である．

- 大量のコマンド入力の代替手段
- 実行履歴のソースコードを簡単に確認する機構

前者は，個々のコマンドや操作方法の習得の手間を軽減するとともに，デバッグを行うための敷居を低くし，簡単にデバッグを行えるようにする．後者は，ステップごとにソースコードを表示させつつ，色付け等の強調表示を用いて，注目すべき情報を分かりやすく表示し，デバッグの経過を簡単に確認できるようにする．

これらの機能を実現するため，GHCi デバッガを子プロセスとして起動し，入出力を操作するプログラムを作成するという方針をとる．

出力を視覚的に閲覧するためのビューアには，ウェブブラウザが適していると考えられる．大量の情報が出ても，スクロールで見る場所を容易に変更できるうえに，字の大きさの変更や文字列の検索等を標準的な機能として備えているので，自力で多機能な GUI を作る手間を削減できるうえに，可搬性も確保できる．表示系をエディタから分離することで，フォーカスの手間が増えるものの，開発環境を選ばず使用できるという利点もある．

GHCi デバッガをそのまま使うことにより，ソースコードを変更なしに使用でき，さらに，型推論，ブレイクポイント，ステップ実行といった機能を利用できるといった利点が生じる．処理系組み込みなのでデータ構造やライブラリに関しての制約が最も少なく，処理系変更の影響もほぼ受けない．

5. 実　　装

5.1 概　　要

前章で述べた方針を実現するために，**図 3** の構成でデバッグの支援環境を構築した．本機構は，実行情報を取得するための GHCi デバッガと，実行時の情報の表示を行うウェブブラウザと，両者を仲介する部分の 3 つから構成される．実装した仲介部分は，さらに GHCi デバッガラッパと HTTP サーバの 2 つに分けることができる．GHCi デバッガラッパは，GHCi の入出力を操作し，入力用にコマンドをまとめて行い，入出力を記録しておくとともに，HTML に変換することで，視覚的に内容を参照可能にする準備を行う．HTTP サーバは，ブラウザからの要求を解析して適切な GHCi デバッガラッパのコマンドを実行させ，デバッガの状態の操作や要求に応じた HTML を表示させる役割をする．

各仲介部分は Python の標準ライブラリのみで実装しており，追加のライブラリやパッケージのインストールをせずに利用できる．

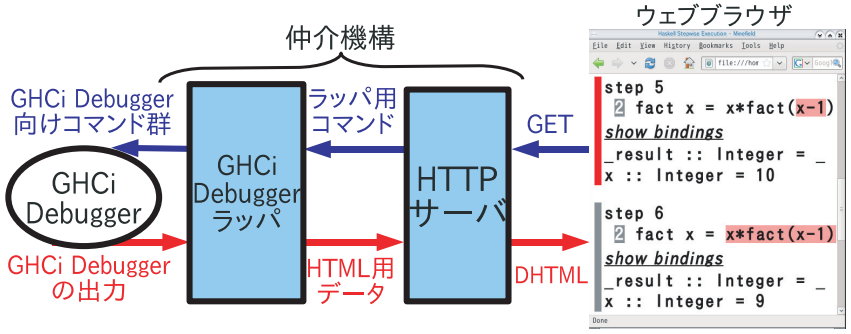

図 3　デバッグ支援環境の構成
Fig. 3　Overall structure of debugging support environment.

5.2　GHCi デバッガラッパ

GHCi デバッガラッパは HTTP サーバからの要求に対して適切に GHCi デバッガを操作し，出力を記録する．

具体的には以下のような機能を備えている．

- GHCi デバッガの起動と操作
- 入出力の記憶
- ステップ実行をまとめて行う機能
- 各ステップ実行の出力から HTML への変換

これらの機構によって，GHCi デバッガでよく行う操作や大量のコマンドが必要となる操作を簡単に実行可能にする．また，表示の視覚化のために適宜保持していた出力を解析して HTML 化を行う．

5.2.1　GHCi デバッガの起動と操作

GHCi デバッガラッパは GHCi を子プロセスとして起動して入出力を操作する．最も基本的なメソッドとして `talk` というメソッドを定義しており，引数に通常 GHCi に与える文字列を受け取り，その出力を返すことで，通常の対話環境の操作を外部プログラムから簡単に行えるようにする．

GHCi デバッガラッパが扱う主な GHCi ないし GHCi デバッガのコマンドを以下にあげる．

- `:browse`（関数名の表示）
- `:break`（ブレイクポイントの指定）
- `:step`，`:continue`（評価を進める）
- `:print`，`:force`（遅延オブジェクトの評価）
- `:trace`（トレース機能を使う）
- `:back`，`:forward`（履歴の移動）

たとえば，ブレイクポイントの適用可能な関数名の取得には，`talk(":browse")` を実行すれば，返り値として関数名の一覧が得られる．同様に，ブレイクポイントを関数 `f` に対して設定する場合は，`talk(":break f")` とする．

このほかには，現在何ステップ実行したか，ブレイクポイントで停止をしているか，ステップごとの出力といった情報を保持して簡単に外部から状態を把握できるようにしている．

5.2.2　ステップ実行をまとめて行う機能

GHCi デバッガを用いてデバッグを行う基本的な操作は，以下のような順番となる．

(1)　`:break` でブレイクポイントを指定する．
(2)　デバッグ対象の式を実行する．
(3)　`:step` や `:continue` 等で評価を進める．
(4)　適宜 `:force` 等で変数の評価と `:back` 等で履歴を移動をする．
(5)　ブレイクポイントを抜けデバッグ対象の式の評価を終える．

注意すべき点は (2) で `:trace` コマンドを使ったときのみ追跡機能が使用可能になることと，(3) は複数回実行する必要があり，(4) の処理はブレイクポイントで停止したときしか行えないことである．特に，`:trace` コマンドを使用しなかった場合や (5) までたどりついた場合には追跡機能は使用できないので最初から再実行をする必要がある．

以上をふまえて，追跡機能を行うために式の最初の評価には `:trace` コマンドを使い，回数を指定してステップ実行をまとめて行う `exec_nstep` というメソッドを実装している．たとえば，関数 `f` を `:continue` コマンドを使って 10 回評価を進めた後に，続けて `:step` コマンドで 30 ステップ評価を進める場合は，`exec_nstep(n=10,step="continue",expr="f")` の後に続けて `exec_nstep(n=30,step="step")` を実行する．この処理で 10 回ないし 30 回分のコマンドの入力の手間を省ける．また，適宜 `exec_nstep` の処理の間に `:force` コマンドや `:back` コマンドを与え，そのコマンド相当の処理を行う．

5.2.3　各ステップ実行の出力からの HTML への変換

前述の `exec_nstep` メソッドは，ステップ実行ごとに `talk` メソッドの出力を記録用のリストに保持している．サーバ側から HTML として実行内容を問い合わせられたときに，保持内容を解析，HTML 化し記録用のリストを更新しつつサーバに HTML を返す．記録用のリストのデータを HTML 化するためのメソッドとして `exec_html`，記録用のリストの部分的な取得および不足分をステップ実行するメソッドとして `exec_range` を定義している．

5.3　HTTP サーバ

ブラウザからの要求に応じて，適切な処理をする GHCi デバッガラッパのメソッドを実行し，実行結果の出力となる HTML を送信するのが HTTP サーバの役割である．なお，このサーバはローカルサーバとして個人で使うことを念頭としているため，セキュリティを考慮しておらず，また，複数のユーザで同時に操作するような使用法も想定していない．

このサーバは，以下の機能を持っている．

(1)　GHCi デバッガの初期条件の設定をする HTML の生成と表示
(2)　実行情報を含む HTML の表示
(3)　GHCi デバッガラッパの状態の取得と設定

(1) の HTML は**図 4** のように表示される．まず，ファイル名を指定することで自動的に

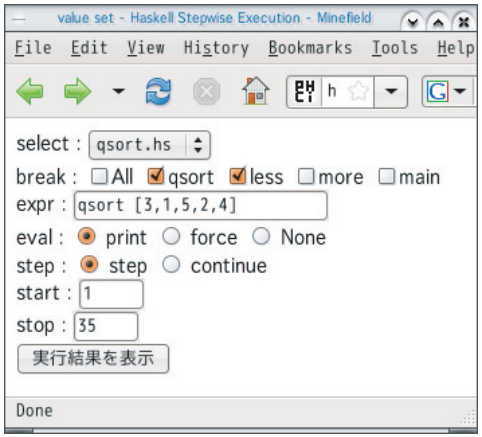

図 4　デバッグの条件設定
Fig. 4　Parameter setting.

関数名が抜き出され，関数名にチェックをするとブレイクポイントをセットする．そして実行する式を記述し，遅延オブジェクトに遭遇した際の評価方法とステップ実行コマンドの指定と最初に何ステップだけ実行するかを決めて実行する．

(2) の HTML は**図 5** のように表示される．ここでは部分的なソースコードとともに，処理系の注目範囲の色づけや，束縛されている変数名と値が表示される．ここで，遅延オブジェクトの変数名をクリックすると値を強制的に評価する．また，以前の実行のログをマウスカーソルでクリックすれば，履歴を巻き戻り，変数の評価を行える．この HTML は，前述の GHCi デバッガラッパのメソッドの `exec_range` を実行することで生成される．

(3) の処理は JavaScript の Ajax ライブラリ jQuery を用いて実装している．HTTP サーバからの GHCi デバッガラッパへの代表的な要求には，以下のようなものがある．

- ブレイクポイントを適用できる関数名の取得
- 指定した回数の評価
- `:print` や `:force` の遅延オブジェクトの評価
- `:back` や `:forward` の履歴の移動

たとえば，ブラウザ上でユーザが，遅延オブジェクトを含む情報をクリックすると JavaScript のイベントが実行され，以下のリクエストがサーバに送られる．

```
GET /thunk?variable=x&evaluate=force
```

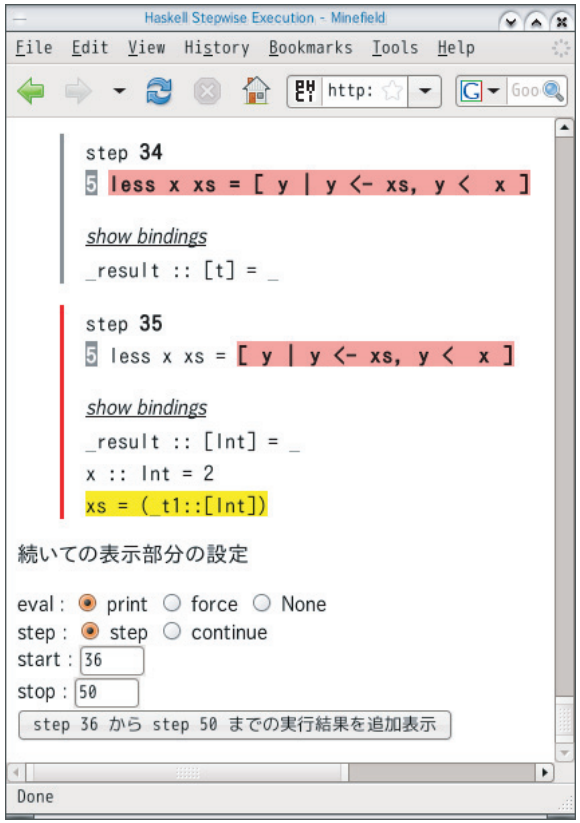

図 5　デバッグで値を表示した様子
Fig. 5　Printing values.

この場合は，遅延オブジェクトを評価して結果の文字列を受け取るという HTTP サーバのメソッドが実行される．このメソッドは GHCi デバッガラッパの `talk(":force x")` というメソッドを実行して，結果の文字列を返す．その後，サーバ側で整形をして，JavaScript でクリックされた部分を得られた文字列で動的に書き換える．このように，HTML の各要素にイベントを設定して，各イベントの要求に応じて適切に GHCi デバッガラッパのメソッドを実行する．

表 1　実行時間（秒）とオーバヘッド（%）
Table 1　Execution times (in seconds) and overhead (%).

	GHCi	50 ステップ		100 ステップ		200 ステップ		400 ステップ	
queen	4.34	4.76	(9.7)	4.99	(15.0)	5.35	(23.3)	6.02	(38.7)
prime	3.39	3.95	(16.5)	4.16	(22.7)	4.60	(35.7)	5.49	(61.9)
tak	8.83	9.30	(5.3)	9.57	(8.4)	10.32	(16.9)	11.71	(32.6)
rfib	10.30	10.86	(5.4)	11.05	(7.3)	11.67	(13.3)	12.71	(23.4)

6. 評　　価

6.1　オーバヘッド

本機構は GHCi デバッガを用いて実行時の情報を取得するが，コンパイル済みコードは GHCi デバッガで使用できず，GHCi 上でコードを解釈実行する場合は，コンパイル済みのコードを実行する場合と比較して 10 倍から 15 倍の時間がかかることと，`:trace` コマンドを用いた追跡機能を使用しているというオーバヘッドの要因がある．

これらの要因が実行時間に対して与える影響を確認するため，いくつかのプログラムに対して実行時間を計測した．各実行時間は処理系の起動からテスト用のプログラムを実行し処理系の終了を行うまでの時間である．各テスト用プログラムは，GHCi で解釈実行したもののみを計測の対象とする．計測した条件は以下のとおりである．

- GHCi はブレイクポイントを使用せずに関数を実行した場合
- 50，100，200，400 ステップは GHCi ラッパを用いてブレイクポイントを設定し，それぞれの回数だけ `:step` コマンドによるステップ実行をして残りは通常の実行をした場合

50 ステップというのは，現状の `:trace` コマンドの追跡機能で履歴を戻れる最大のステップ数である．GHCi ラッパを用いた処理はステップ実行の後で，全ステップ実行の情報を HTML 化して出力をするまでの時間を含んでいる．実行環境は CPU が Intel Core2 Duo 2.16 GHz，メモリ 2 GB，GHC 6.8.2，Python 2.5.2 の Linux 上で動作をさせた．

テスト用プログラムは，Haskell 用のベンチマーク nofib[8] の一部を使用した．queen は 10 クイーンの解の数を数える．prime は 1500 番目の素数を出力する．tak は tak 関数に引数 24，16，8 を与える．rfib は 30 番目のフィボナッチ数を求める．実行時間とオーバヘッドの割合を**表 1** に示す．

結果から，GHCi デバッガの機能を使って情報を取得し，HTML 化の処理を追加した場合，400 ステップの実行で 3 秒未満，30%から 60%程度の実行時間の増加があり，ステップ数と実行のオーバヘッドは比例している．ステップ実行を行うことに対する代償としては，大きな問題ではないと考える．

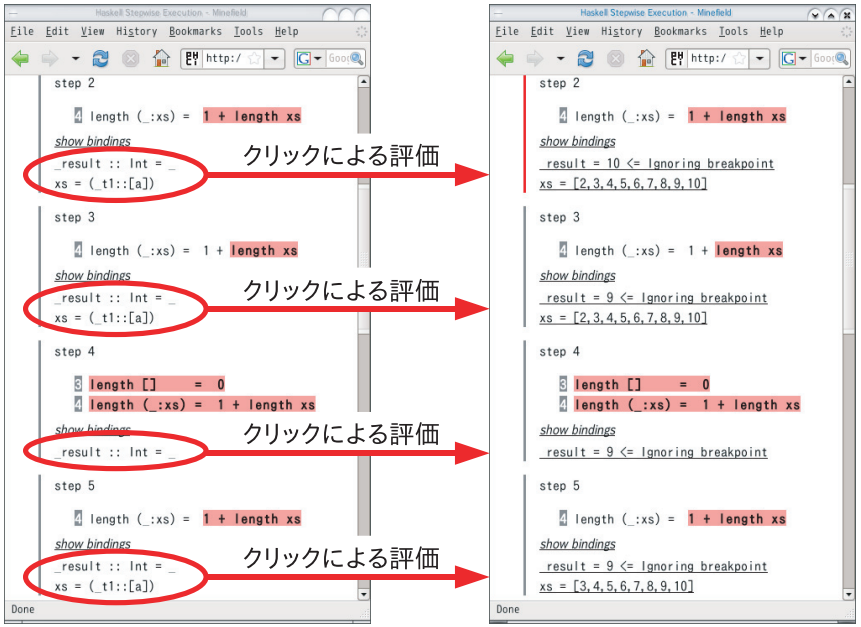

図 6　length 関数を実行した様子
Fig. 6　Executing the function `length`.

また，ウェブブラウザの HTML の表示とサーバ間の通信に関しても，通常の GHCi での実行に比べて時間がかかる要因となる．サーバに関してはローカルサーバとして外部非公開の環境で動作させることを前提としており，外部との通信によるネットワークの遅延等は発生しないと考える．このような理由により，本機構のオーバヘッドは主に GHCi ラッパによって発生しているため，ブラウザとサーバ間のオーバヘッドはあまり重要な要素ではないと考える．なお，GHCi のステップ実行の繰返しによる実行時間とオーバヘッドについては，文献 1) の 4.5 節に記述されている．

6.2　動　作　例

6.2.1　リストの長さ

単純な使用例として，リストの長さを返すプログラムに本フロントエンドを適用すると**図 6** のようになる．ここでは，冒頭の処理を数ステップ実行したのちに，遅延されている

```
1: main = print (sort [3,1,2])
2: sort [] = []
3: sort (x:xs) = insert x (sort xs)
4: insert x [] = [x]
5: insert x (y:ys) | x <= y = x:ys          {- x:y:ys が正しい -}
6:                 | x > y  = y:insert x ys
```

図 7　誤りを含む挿入ソートのソースコード
Fig. 7　An incorrect source code of insertion sort.

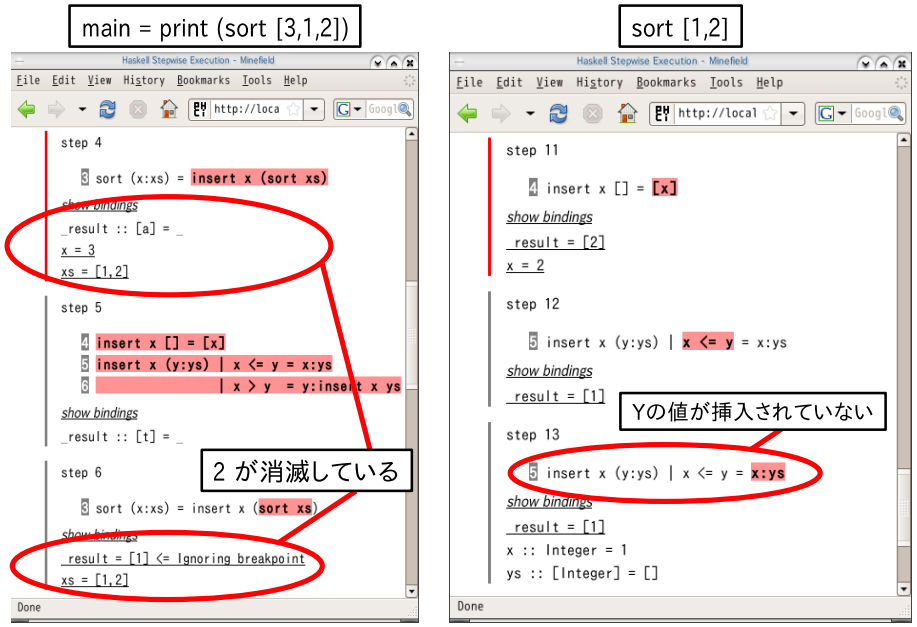

図 8　挿入ソートの実行とバグを含む関数を絞り込んだ様子
Fig. 8　Executing insertion sort and narrowing down the bug.

変数をクリックして内容を確認している．このように値の内容を調べることで，再帰的に呼ばれる関数にどのような値がわたっているかを簡単に確認できる．この場合には，ステップを進めるための :step，履歴の移動と変数を確認するための :back，:force といったコマンド入力の手間を削減できている．

6.2.2　挿入ソート

実行結果から簡単なバグを類推する例として，**図 7** の記述ミスを含む挿入ソートを実行したものが，**図 8** である．左図が定義してある main 関数を実行した結果，右図が記述ミスの絞り込みのために sort [1,2] を実行した結果である．まず左図の結果を見ると，ステップ 4 で insert 3 (sort [1,2]) の結果から 2 が消えている．同様に，ステップ 6 で sort [1,2] の結果から 2 が消えている．この段階で sort [1,2] の処理が疑わしいと分かるので，ブラウザのページを戻り，実行する式を sort [1,2] に変更して得られた結果が右図である．出力を眺めていくと，ステップ 13 で insert 1 (y:[]) | 1<=y = (1:[]) が [1] を返して y の値を消している．この時点で，5 行目の値の挿入の記述にミスがあると気付くことができる．この例は，ソースコード中の疑わしい場所が分からず，全体の処理過程を眺めている．このように，複数回のデバッグを実行する際には，その分だけコマンド入力の手間が省けており，操作の簡単化の恩恵が得られている．また，実行する式を変更した後に，ステップ実行を再びまとめて行うことも簡単にできた．

7. おわりに

本論文では Haskell のデバッグを支援する GHCi フロントエンドについて述べた．プログラムの実行の情報は GHCi デバッガを用いて取得し，表示にはウェブブラウザを使用することで，既存の各機構の恩恵を得るとともに，主要な実装は，GHCi デバッガとウェブブラウザのデータの仲介を行う部分のみで済んでいる．

また，本機構により GUI を用いて単純な操作に GHCi デバッガのコマンド入力の操作を割り当て，繰返し入力の操作を省略したことで，詳しい使い方が分からなくても GHCi デバッガの多くの機能を簡単に使用できる．

提案システムの適用範囲と限界は，GHCi の適用範囲と限界，および jQuery を実行できるブラウザが必要であることである．前者は，提案手法は GHCi を用いているため，GHCi の弱点をそのまま引き継ぐということを意味している．たとえば，現状の GHC（バージョン 6.10.1）では並行プログラムのデバッグができないことは，文献 1) でも指摘されている．これは，本論文で改善しようとした視覚化の問題と並ぶ，GHCi デバッガにおけるもう 1 つの重大な問題である．

今後の課題としては，まず，式の入力フォームの複数行対応や行番号と例外へのブレイクポイントの設定の対応といった，GHCi デバッガ関連の処理の追加があげられる．ほかに

は，無限ループからの脱却用のタイムアウト値の設定の追加，他のプログラムとの協調のためのウェブ API の用意，Haskell 開発支援ウェブサービスとの連携を可能にすること等が考えられる．

参 考 文 献

1) Marlow, S., Iborra, J., Pope, B. and Gill, A.: A Lightweight Interactive Debugger for Haskell, *Proc. 2007 ACM SIGPLAN Workshop on Haskell* (*Haskell 2007*), pp.13–24 (2007).
2) Gill, A.: Debugging Haskell by Observing Intermediate Data Structures, *Proc. 2000 ACM SIGPLAN Workshop on Haskell* (*Haskell 2000*), Technical Report of the University of Nottingham (2000).
3) Wallace, M., Chitil, O., Brehm, T. and Runciman, C.: Multiple-View Tracing for Haskell: A New Hat, *Proc. 2001 ACM SIGPLAN Workshop on Haskell* (*Haskell 2001*), Technical Report UU-CS-2001-23, Institute of Information and Computing Sciences, Utrecht University, pp.151–170 (2001).
4) Pope, B. and Naish, L.: Practical Aspects of Declarative Debugging in Haskell 98, *Proc. 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming* (*PPDP 2003*), pp.230–240 (2003).
5) Reinke, C.: GHood: Graphical Visualisation and Animation of Haskell Object Observations, *Proc. 2001 ACM SIGPLAN Workshop on Haskell* (*Haskell 2001*), Technical Report UU-CS-2001-23, Institute of Information and Computing Sciences, Utrecht University, pp.121–149 (2001).
6) Gill, A. and Runciman, C.: Haskell Program Coverage, *Proc. 2007 ACM SIGPLAN Workshop on Haskell* (*Haskell 2007*), pp.1–12 (2007).
7) Angelov, K. and Marlow, S.: Visual Haskell: A Full-Featured Haskell Development Environment, *Proc. 2005 ACM SIGPLAN Workshop on Haskell* (*Haskell 2005*), pp.5–16 (2005).
8) Partain, W.: The nofib Benchmark Suite of Haskell Programs, *Proc. 1992 Glasgow Workshop on Functional Programming*, pp.195–202 (1992).

(平成 20 年 12 月 18 日受付)
(平成 21 年 3 月 11 日採録)

根岸　純一（学生会員）

1984 年生．2007 年 3 月電気通信大学電気通信学部情報工学科卒業．2009 年 3 月電気通信大学大学院電気通信学研究科情報工学専攻博士前期課程修了．プログラミング言語とその処理系，特にスクリプト言語と関数型言語に興味を持つ．

岩崎　英哉（正会員）

1960 年生．1983 年東京大学工学部計数工学科卒業．1988 年東京大学大学院工学系研究科情報工学専攻博士課程修了．同年東京大学計数工学科助手．1993 年東京大学教育用計算機センター助教授．その後，東京農工大学工学部電子情報工学科助教授，東京大学大学院工学系研究科情報工学専攻助教授，電気通信大学情報工学科助教授を経て，2004 年より電気通信大学情報工学科教授．工学博士．記号処理言語，関数型言語，システムソフトウェア等の研究に従事．日本ソフトウェア科学会，ACM 各会員．