ProveIt - An interactive *point-free* proof editor

José Pedro Correia

zepedro.correia@gmail.com

Techn. Report DI-PURe-05.08.02

August 2005

PURe

Program Understanding and Re-engineering: Calculi and Applications
(Project POSI/ICHS/44304/2002)

Departamento de Informática da Universidade do Minho
Campus de Gualtar — Braga — Portugal

DI-PURe-05.08.02
`ProveIt` - *An interactive* point-free *proof editor* by José Pedro Correia

Abstract

The growing use of the *point-free* notation as a way of representing functional programs has a purpose: to allow reasoning to be made about those programs.
There is nowadays a well-sustained *point-free* algebric theory that allows us to use and deduct certain laws over expressions that may represent computer programs. Those laws allow us, with assurance, to "transform" an expression into an equivalent one or to prove that same equivalence.
It is common tu use this mechanism to build proofs over *point-free* expressions in order to testify equivalences between functions or programs.
The goal of this task was to develop a tool that would allow the user to manage this type of proofs interactively.

1 Introduction

1.1 The *point-free* notation and its combinators

In *point-free* notation one represents functions without referring to variables. As so, in order to associate behaviours or results one needs special function combinators that sustain our notation. I will now resume the most common combinators' syntax (theoretical and in ASCII for the text boxes of the tool) and general behaviour (I'm assuming some basic knowledge of formal methods. For more detailed information on algebra of programming or the combinators please refer to [2] or [3]).

1.1.1 Composition

The composition of two functions $f : B \to C$ and $g : A \to B$ is a function $f \cdot g : A \to C$ defined as following:

$$(f \cdot g)\ x = f(g(x))$$

In `ASCII` we represent it simply as `f . g`. In `Haskell` exists predefined.

1.1.2 Split and product

These definitions use the definition of a *product* of two types A and B:

$$A \times B = \{(a, b) | a \in A, b \in B\}$$

wich is the `Haskell` predefined pair `(a,b)`. There are two projection functions associated with this constructor:

$$\pi_1(a, b) = a$$

wich is `Haskell` predefined `fst` and

$$\pi_2(a, b) = b$$

wich is `Haskell` predefined `snd`. So, the split of two functions $f : A \to B$ and $g : A \to C$ agregates the result of both functions, yielding a function $f \vartriangle g : A \to B \times C$ defined as following:

$$(f \vartriangle g)\ x = (f(x), g(x))$$

In `ASCII` we choose to represent it as `f /\ g`.

One can also define the product of two functions $f : A \to C$ and $g : B \to D$ as a function $f \times g : A \times B \to C \times D$:

$$f \times g = f \cdot \pi_1 \vartriangle g \cdot \pi_2$$

In `ASCII` is `f >< g`.

1.1.3 Either and sum

These definitions are built around the *sum* of two types A and B defined as:

$$A + B = (\{0\} \times A) \cup (\{1\} \times B)$$

This is `Haskell` predefined combinator `Either`. We associate with this combinator two injection functions:

$$i_1\ x = (0, x)$$

wich is the `Haskell` `Either` constructor `Left` (in `ASCII` `inl`) and

$$i_2\ x = (1, x)$$

wich is the `Haskell` `Either` constructor `Right` (in `ASCII` `inr`). The either between two functions $f : A \to C$ and $g : B \to C$ acts similar as a "choice" and is a function $f \nabla g : A + B \to C$ defined as following:

$$(f \nabla g)\ (i_1\ x) = f(x)$$

$$(f \nabla g)\ (i_2\ x) = g(x)$$

In `Haskell` exists as prefix `either`. In `ASCII` format it was chosen to represent it as infix `f \/ g`.

From this definition one can consider (as with the product) the sum of two functions $f : A \to C$ and $g : B \to D$ as a function $f + g : A + B \to C + D$:

$$f + g = i_1 \cdot f \nabla i_2 \cdot g$$

In `ASCII` is `f + g`.

1.2 The `wxHaskell` library

"`wxHaskell` is a portable and native GUI library for `Haskell`. The goal of the project is to provide an industrial strength GUI library for `Haskell`, but without the burden of developing (and maintaining) one ourselves.

`wxHaskell` is therefore built on top of `wxWidgets` - a comprehensive `C++` library that is portable across all major GUI platforms; including GTK, Windows, X11, and MacOS X. Furthermore, it is a mature library (in development since 1992) that supports a wide range of widgets with the native look-and-feel, and it has a very active community (ranked among the top 25 most active projects on sourceforge)." [4]

2 Initial considerations

The above *point-free* theoretical introduction is intended only as a motivation for the objectives of the task. If we take in consideration that `ProveIt` will manage *point-free* expressions only by applying certain calculation laws (wich associate two *point-free* expressions), we can see that the tool will only act in a syntax level, leaving the semantic interpretation for the user.
Let's now reflect on what's intended for the tool.

2.1 Behaviour of the tool

2.1.1 Proof style

In order to represent our *point-free* proof one can generally choose between two fashions:

$$\begin{array}{lll|ll}
 & & & & f_1 = f_3 \\
f_1 & & & \equiv & \{\text{by laws} \ldots\} \\
 & = & \{\text{by laws} \ldots\} & & f_1 = f_2 \\
f_2 & & & \equiv & \{\text{by laws} \ldots\} \\
 & = & \{\text{by laws} \ldots\} & & f_1 = f_1 \\
f_3 & & & \equiv & \\
QED & & & & True \\
 & & & & QED
\end{array}$$

and one should expect that the tool supports them both unambiguously.

2.1.2 Information representation

The information regarding the current proof should be concise and clear. The relevant information, when building a proof is, I believe, the current state of the expression one is manipulating and the steps already taken (laws applied so far).

2.1.3 User interaction

The fundamental action that this tool provides is the application of *point-free* laws. Therefore, the mechanism should be simple and intuitive to use. One can devise two ways of application of laws:

- selecting a specific part of the expression and the law to apply
- choosing a law and selecting from the possible points of application

Moreover, as the creation of a proof could be somewhat by attempt and error, it is fundamental that actions of undo and redo are allowed.

2.1.4 External actions

It is expected that actions like saving to a file and loading from a file are supported, in order to have persistancy in our proofs. Also it is useful to be able to export into a format like a text or a latex file. These are the simpler features the tool should support.

3 The tool - `ProveIt`

The tool has some `haddock` documentation that can be consulted in order to clear the behaviour of some functions. Nevertheless, this section intends to describe the internal structure of the tool as well a reference to data-structures used and algorithms.

3.1 Back-end

3.1.1 Data structures

The information represented to implement our proof environment consists of

- *Point-free* expression (represents a function, or part of it. Example: $f \triangle (g + h \nabla i)$). This structure is represented using a generic tree according to these definitions:

  ```
  data RTree a = Node a [RTree a]

  data PFComb =
            FuncVar String
          | Id
          | Fst
          | Snd
          | InL
          | InR
          | FuncRef String
          | MutVar Int String
          | Comp
          | Split
          | Either
          | Prod
          | Sum
    deriving Eq

  type PFExp = RTree PFComb
  ```

- *Point-free* equation (an equivalence between expressions. Can also be the value `True`)

These two datatypes are agregated into one that I describe as a *Point-free* term and is defined as:

```
data PFTerm =
          Exp PFExp
        | Eq  PFEq
```

- *Point-free* law (a *valid* equivalence between two terms)
- *Point-free* proof (consists of an initial expression and the sequence of steps (law applied and application point) that make the proof)

3.1.2 Law application algorithm

The algorithm of applying a law to an expression runs briefly in three steps:

matching - one tries to match the current expression to the first expression of the law, building a list of correspondences between variable names and sub-expressions

checking - because the list generated above might not yield a valid substitution, one must check if the repeated references to a variable correspond to the same sub-expression, and eliminate those repetitions

substituting - after having a valid substitution, one can apply it to the second expression of the law, obtaining the "conversion" of the given expression, according to the law

It is important to mention that the first two steps can fail, case where the law is not applicable to the expression.

3.2 Front-end

3.2.1 Visual representation and interaction

The visualization of the current term is done using a tree representation of the term and a textual representation in a textbox. One can select part of the term by clicking in a node of the tree (that represents a sub-expression) or by placing the cursor in a position above a combinator in the text and right-clicking, wich selects the respective sub-expression. A right-click pops up a menu with the laws from the repository applicable at that point.
There is also three buttons to allow operations of undo, redo and commit. This last one allows the user to commit the last changes to a log, shown in a textbox.

3.2.2 Menus

There are three menus, being:

File For operations of creating a new proof, opening a proof from a file (not working), saving to file, saving the log to a text file, finishing the proof and quitting.

Laws Menu containing the repository of laws. Futurely will support the adittion of laws.

Help For now, this menu only has minimal information about the tool. Futurely should allow the access to a user's guide among other possible features.

3.2.3 Data structures

For the manipulation of a *point-free* term it was necessary to maintain information, for a given sub-expression, about the position in the text, as well as the corresponding `wxHaskell` object that pointed to the tree node. For this purpose there is a data structure built "parallel" to the *point-free* term wich is named the *meta-term*. This datatype is built as an instance of a generic tree, and as the same shape as the expression, only with information about text ranges and `TreeItem`'s that corresponds to the same node in the expression.
All the front-end functions return the type `IOState x`. This is not a real state monad, as it should be. The datatype is defined as:

```
type IOState a = IOStateVars -> IO a
```

where `IOStateVars` is the tuple of variables that one needs to keep track along the program, like the main frame, the menus, the current expression, etc.

3.3 Modules

The modules organization is described in this section.

3.3.1 Back-end

Mpi This module is distributed with the course of "Métodos de Programação I" and has the basic *point-free* combinators, here used in some functions.

RTree A generic tree definition and som useful functions on it.

PFExp Datatype definition of a single *point-free* expression.

PFEq Datatype definition of a *point-free* equation.

PFTerm Datatype definition of a *point-free* term that agregates both *point-free* expression and *point-free* equation.

PFLaw Datatype definition of a *point-free* law and some functions to manipulate it.

PFProof Datatype definition of a *point-free* proof and some functions to manipulate it.

FunctionsExp Functions to manipulate a *point-free* expression, being the most important the law application algorithm for this datatype.

FunctionsEq Functions to manipulate a *point-free* equation, being the most important the law application algorithm for this datatype.

FunctionsTerm Functions to manipulate a *point-free* term, being the most important the law application algorithm for this datatype.

3.3.2 Front-end

MetaTerm Datatype definiton of a *meta-tem* and some functions.

IOState Dataype definiton and selector functions.

AuxMain This module contains all the auxiliary functions that manipulate the `wxHaskell` objects within the program.

Main This module is the Main module, and contains the main loop as also the functions that process actions over `wxHaskell` objects.

References

[1] *Tranformações pointwise - point-free*, José Proença, DI-PURe-05.02.01, February 2005.

[2] *Program Design by Calculation* (draft of textbook in preparation), J.N. Oliveira, Departamento de Informática, Universidade do Minho, 2005.

[3] *Algebra of Programming*, Richard Bird and Oege De Moor, Prentice Hall 1996.

[4] `wxHaskell` website: http://wxhaskell.sourceforge.net