A Programming Tutor for Haskell
Exercises and Projects

Johan Jeuring, Alex Gerdes, and Bastiaan Heeren

CEFP, June 2011

In this document we provide both exercises and some projects. The exercises serve to introduce you to several aspects of our programming tutor. The projects are more challenging. Actually, some projects deal with some of our ongoing research, and we welcome contributions from participants. Substantial contributions on these projects are probably interesting enough for a scientific paper, and we would welcome co-authors/contributors.

Exercises

1. Familiarise yourself with the programming tutor for Haskell by playing with it. Please visit `http://ideas.cs.uu.nl/ProgTutor/`.
2. Install the programming tutor on your machine. The sources for the programming tutor can be downloaded from `http://ideas.cs.uu.nl/trac/wiki/Download`. Follow the instructions in the `README` file.
3. Take your favourite beginner's exercise, and add it to the programming tutor. Look at the file `src/FPTutor.hs` to find out how you add a new exercise to the programming tutor. You might be interested in how the strategies are inferred from model solutions: have a look at the function *compileStrategy* in the module *Domain.FP.StrategyInference*.
4. Add a strategy for the prelude function *scanr* to the strategy prelude file *Domain.FP.PreludeS*. Include some of the properties for *scanr*.
5. The goal of this exercise is to develop a simple strategy for rewriting a λ-term into SKI-combinators (see `Feedback/trunk/src/Domain/Lambda.hs`). For example, $\lambda x \rightarrow \lambda y \rightarrow x\ y$ can be transformed into $S\ (S\ (K\ S)\ (S\ (K\ K)\ I))\ (K\ I)$, where

   $$\begin{aligned} S &= \lambda x\ y\ z \rightarrow x\ z\ (y\ z) \\ K &= \lambda x\ y \rightarrow x \\ I &= \lambda x \rightarrow x \end{aligned}$$

   The file `Lambda.hs` contains a definition of a datatype for the λ-calculus, which can represent variables, application and abstraction. This datatype is an instance of the **class** *Uniplate*, so that you can use combinators like *somewhere*, *bottomup*, etc. to traverse over λ-terms. We can produce an SKI-term from a λ-term, by means of the following rules:
   - $\lambda x \rightarrow y \Rightarrow I$, if $x == y$,
   - $\lambda x \rightarrow a \Rightarrow K\ a$, if x does not appear free in a
   - $\lambda x \rightarrow a\ b \Rightarrow S\ (\lambda x \rightarrow a)\ (\lambda x \rightarrow b)$

   The document contains a strategy for calculating an SKI-term form a λ-term, but the definition of the rules is missing. Add these definitions.
6. Suppose we want to support a student with transforming her program to a point-free program. A point-free program is a program without variables, using only function composition, application (denoted by :@: here), and other constructs. For example,

   $\lambda x \rightarrow \lambda y \rightarrow x$:@: y
   $\Rightarrow$ { Write infix operation as prefix function }
   $\lambda x \rightarrow \lambda y \rightarrow (:@:)$:@: x :@: y
   $\Rightarrow$ { Remove an abstraction: $\lambda y \rightarrow f$:@: $y \Rightarrow (f$:@:$)$ }
   $\lambda x \rightarrow ((:@:)$:@: x :@:$)$
   $\Rightarrow$ { Write sectioned operation (last occurrence of :@:) as a prefix function }

$\lambda x \rightarrow (:@:)\ :@:\ ((:@:)\ :@:\ x)$

$\Rightarrow$ { Introduce composition }

$\lambda x \rightarrow ((:@:) \circ ((:@:)\ :@:))\ :@:\ x$

$\Rightarrow$ { Remove an abstraction }

$(((:@:) \circ ((:@:)\ :@:))\ :@:)$

Define rewrite rules and a strategy for transforming a λ-term (extended with infix operations, and possible other constructors for representing special λ-terms) to a point-free program. Termination will prove to be hard, if at all possible. You can reuse the file `Lambda.hs` for this exercise.

7. (a) The lecture notes contain definitions of the functions *empty* and *firsts*. Another grammar analysis function that is useful in top-down parsing is the function *follow*, which returns the symbols that can follow upon a non-terminal in a derivation using a context-free grammar. In our situation a non-terminal is a strategy, and a non-terminal that appears in the right-hand side of a production is a recursive non-terminal. Such a strategy is introduced by means of *Rec i* for some natural number *i*, and referred to by means of *Var i*. We want the function *follow* to determine the rules that can be applied after a strategy starting with a *Rec* has been used recognized. Define function *follow*.
   (b) Use the functions *empty*, *firsts*, and *follow* to determine whether or not a strategy is LL(1). A strategy is LL(1) if given the next input symbol, we know which path to take in a strategy, and no backtracking is required. If a grammar is not LL(1), left-factoring might help to make it LL(1).

Projects

1. At the moment our tutor does not support datatypes. Which refinement rules do we need to add to also support the development of datatypes in our tutor?
2. Sometimes we want to enforce a student to use a particular construct in a program, for example, we might ask: define *reverse* using the function *foldl*. Think about how to provide support for either enforcing or disallowing particular solutions in a strategy. Can we solve this by annotating model solutions using pragma's, for example?
3. If we accept a solution of a student, we know it is provably equal to a model solution, and hence correct. We cannot prove a solution to be wrong. Think about how to add support for proving a student to be wrong to our framework. Proving a student to be wrong can be done by other showing that a student solution doesn't pass some unit tests we specify for a solution, or by formulating a contract, and showing that a student solution doesn't fulfill the contract. To perform tests or verify contracts on incomplete solutions, we have to adapt existing unit testers or contract checkers.