Assignment 5
COMP 302 Programming Languages and Paradigms

Francisco Ferreira and Brigitte Pientka
MCGILL UNIVERSITY: School of Computer Science

Due Date: 1 Dec 2017

Q1. All Things Grow with Love [50 points]

We revisit here our simple language for arithmetic expressions and booleans. Here we extend the language with pairs of expressions

Backus-Naur Form (BNF):

$$
\begin{array}{lll}
\text{Operations } \mathsf{op} & ::= & + \mid - \mid * \mid < \mid = \\
\text{Expressions } e & ::= & n \mid e_1 \ \mathsf{op} \ e_2 \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
 & & \mid x \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} \\
 & & \\
\text{Values } v & ::= & \mathsf{true} \mid \mathsf{false} \mid n \\
\text{Types } t & ::= & \mathsf{bool} \mid \mathsf{int}
\end{array}
$$

Values in our language are booleans, integers. This language is the language that we saw in class.

To grow our language we want to add the idea of pairs. We want to add an expression to build pairs (e_1, e_2) and an expression to pattern match on them:
$\texttt{let pair}\ (x, y) = e_1\ \texttt{in}\ e_2\ \texttt{end}$. Pairs of values are values, and their type is a called a product type of the types of each component.

We extend the grammar by adding the following expressions, values, and types:

$$
\begin{array}{lll}
\text{Expressions } e & ::= & \cdots \mid (e_1, e_2) \mid \texttt{let pair}\ (x, y) = e_1\ \texttt{in}\ e_2\ \texttt{end} \\
\text{Values } v & ::= & \cdots \mid (v_1, v_2) \\
\text{Types } t & ::= & t_1 \times t_2
\end{array}
$$

Some examples of these new operations are:

$(1, \text{true})$	:	is a value of type int $\times$ bool
$\texttt{let } z = (3,2) \texttt{ in let pair } (x,y) = z \texttt{ in } x + y \texttt{ end end}$	:	is well-typed with type int and it evaluates to 5.
$\texttt{let pair } (x,y) = (\text{false}, 7) \texttt{ in if } x \text{ then } 1 \text{ else } y \texttt{ end}$	:	is well-typed with type int and it evaluates to 7.
$\texttt{let pair } (x,y) = 14 \texttt{ in if } x \text{ then } 1 \text{ else } y \texttt{ end}$	:	is not well-typed because pattern matching lets require a pair to eliminate.

Note that the last expression is syntactically allowed, but it is not well-typed. Pattern matching let expressions expect an expression of product type.

Q1.1 (5 points) Extend on paper the definition for $FV(e)$ which computes the free variables of the new expressions.

Q1.2 (5 points) In the template file `hw5.ml` using the extended expression data-type, extend the definition of free variables from Q1.1

Q1.3 (5 points) Extend on paper the definition of substitution $[e/x]e'$ where we replace any free occurrence of the variable x in the expression e' with the expression e.

Q1.4 (7 points) Extend the definition of `subst` function as you did on paper in Q1.3

Q1.5 (5 points) Extend on paper the definition of well typed expressions to contemplate pattern matching lets.

Q1.6 (9 points) Extend the definition of function `infer` so it is able to infer the types of pairs and their pattern matching using.

We will use the typing judgment $\Gamma \vdash e : T$ which states that expression e has type T in the context Γ where Γ is defined as follows:

$$\text{Typing Context } \Gamma \quad ::= \quad \cdot \mid \Gamma, x : \text{int}$$

Recall that Γ records the type of variables. The rules for some of the expressions we have already seen in class; they are produced and slightly adapted below for your convenience. We define the rule for pairs, the pattern matching lets are up-to you.

$$\frac{}{\Gamma \vdash n : \text{int}} \qquad \frac{}{\Gamma \vdash \text{true} : \text{bool}} \qquad \frac{}{\Gamma \vdash \text{false} : \text{bool}}$$

$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : T}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \qquad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 * e_2 : \text{int}}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}$$

Q1.7 (5 points) Finally, we want to define the operational semantics for our language. Build on the big-step evaluation rules we have defined in the notes for part of our language and show the rules which must be added to cover the new constructs. In particular, think carefully how to define the operational semantics for evaluating `let pair` $(x, y) = e_1$ `in` e_2 `end`.

Q1.8 (9 points) Extend the definition of the `eval` function to support pairs and their pattern matching.

Q2. TIMTOWTDI – There's more than one way to do it [20 points]

While pattern matching let expressions are very convenient, there is another option to take apart pairs, namely projections. Projections allow one to get the first or the second component of a pair.

We add the following expressions to our language:

$$\text{Expressions } e ::= \cdots \mid \texttt{fst}\ e \mid \texttt{snd}\ e$$

These two new operations are an alternative way of writing programs without using pattern matching lets. Some examples of these new operations are:

`fst` $(1, \mathsf{true})$	:	is a value of type int
`let` $z = (3, 2)$ `in fst` $z +$ `snd` z `end`	:	is well-typed with type int and it evaluates to 5.
`let` $z = (\mathsf{false}, 7)$ `in if` `(fst` z`)` `then` 1 `else snd` z `end`	:	is well-typed with type int and it evaluates to 7.
`let` $z = 14$ `in if` x `then` 1 `else snd` y `end`	:	is not well-typed because projections require a pair to eliminate.

Q2.1 (10 points) Extend on paper the definition for $\mathrm{FV}(e)$ which computes the free variables of the new expressions and in the template file `hw5.ml` using the extended expression data-type, extend the definition of free variables.

Q2.2 (10 points) Extend on paper the definition of substitution $[e/x]e'$ where we replace any free occurrence of the variable x in the expression e' with the expression e and extend the definition of `subst` function.

Q3. Spring Cleaning During Fall [30 points]

Realistic implementations of languages often try to optimize the execution time of the expressions as written by the user. These are transformations from code to code (in our case values of type `Exp.exp` to `Exp.exp`). We can define the type of modules that perform this optimizations as follows:

```
module type Optimization =
  sig
    val optimize : E.exp -> E.exp
  end
```

In this question we will explore a common compiler phase called: dead code elimination. That is, we will eliminate expressions that are assigned to a variable but the variable is never used.

To illustrate the transformation, let's consider some examples:

`let` $x = 7$ `in` 9 `end` $\Rightarrow$ 9

`let` $x = 7$ `in` x `end` $\Rightarrow$ `let` $x = 7$ `in` x `end`

`let pair` $(x, y) = (3, \text{false})$ `in if` y `then` $x + 1$ `else` 7 `end` $\Rightarrow$
`let pair` $(x, y) = (3, \text{false})$ `in if` y `then` $x + 1$ `else` 7 `end`

`let pair` $(x, y) = (3, \text{false})$ `in` $x + 1$ `end` $\Rightarrow$
`let pair` $(x, y) = (3, \text{false})$ `in` $x + 1$ `end`

`let pair` $(x, y) = (3, \text{false})$ `in if false then` 1 `else` 2 `end` $\Rightarrow$ `if false then` 1 `else` 2

`let` $z = 3$ `in let` $x = 7$ `in let` $y = x$ `in` z `end end end` $\Rightarrow$ `let` $z = 3$ `in` z `end`

Q2.1 (15 points) Implement a module called `DeadCode` of type `Optimization` that implements this transformation. Notice that to correctly implement the last case you may need to perform more than one pass (depending on your implementation).

Another interesting transformation it is to eliminate all calls to pattern matching lets into regular lets and projections. The idea is to transform pattern matching lets into the regular lets and then take the pairs apart using projections. Let's consider some examples:

$(1, \text{true})$ $\Rightarrow$ $(1, \text{true})$

`let pair` $(x, y) = (3, 2)$ `in` $x + y$ `end` $\Rightarrow$ `let` $z = (3, 2)$ `in fst` $z + $ `snd` z `end`

`let pair` $(x, y) = (\text{false}, 7)$ `in if` x `then` 1 `else` y `end` $\Rightarrow$
`let` $z = (\text{false}, 7)$ `in if` (`fst` z) `then` 1 `else snd` z `end`

Q2.2 (15 points) Implement a module called `RemoveLetMatch` of type `Optimization` that implements this transformation.

The goal of this homework is to better understand how to define simple languages, and extend them. This will allow us to use important concepts such as variable binding, free variables, substitution, operational semantics, and typing. This homework is both solved

on paper and in OCaml. Your on paper solution should be handed in as a pdf-file. Your code should use the template provided in `hw5.ml`.

Please see the file `hw5.ml` for the encoding of expressions.In the implementation, we model variables using strings.

Q6. Course Evaluations! (Honour Points)

Done! Fantastic. Now, please take a moment to fill out course evaluations! Here a reminder why this is important for everyone:

- Course evaluation feedback improve courses.
- You and other students can view course evaluation results in Minerva - but only if more than 30% fill out the course evaluations!
- The more people fill out the course evaluations, the more representative the results.
- It is a way to give back to other students, since the results are shared with them, if more than 30% of you fill them out.
- Your professor enjoys reading about your experiences in the course, in class and in completing the homework. Even more, if you actually had fun!
- It is a way to give back and say thanks, if you enjoyed the course.
- It is a way to voice your concerns anonymously and provide suggestions.
- Your responses matter to professor's careers such as tenure and promotion and merit committees.

The conclusion is:

Fill Out Your Course Evaluations!