Assignment 5
COMP 302 Programming Languages and Paradigms

Brigitte Pientka
MCGILL UNIVERSITY: School of Computer Science

Due Date: 3 Dec 2015

Q1. Extending the language of our calculator [85 points]

We revisit here our simple calculator for arithmetic expressions from the first homework where we were able to to add, subtract, and multiply floating point numbers. Here we extend the language with booleans and boolean expressions such as conjunction (written as &&), disjunction (written as ||), and negation (written as not), and if-expressions. We also add an expression $\sum_{x=e_1}^{e_2} e$ which allows us to sum series. Finaly we include to_float that allows us to convert an integer to a float and trunc which truncates a float to an integer. In the grammar we use n to stand for integers while f stands for a float.

Binary operations op	::=	$+ \mid - \mid * \mid < \mid = \mid \&\& \mid \|\|$
Unary operations uop	::=	not | to_float | trunc
Expressions e	::=	$x \mid$ true $\mid$ false $\mid n \mid f \mid e_1$ op $e_2 \mid$ uop $e \mid \sum_{x=e_1}^{e_2} e \mid$ if e then e_1 else e_2
Values v	::=	$n \mid f \mid$ true $\mid$ false
Types T	::=	int | float | bool

Values in our language are booleans, integers and floats. Here are some examples:

$(\text{to\_float } 5) + \sum_{x=1}^{10} (\text{to\_float } x) + 2.0$	well-formed and well-typed
$\sum_{x=1}^{\text{trunc } (50.0/2.0)} (\text{to\_float } x) + 2.0$	well-formed and well-typed
$\sum_{x=1}^{\text{trunc } (50.0/2.0)} \sum_{y=0}^{x} (\text{to\_float } x) + (\text{to\_float } y)$	well-formed and well-typed
$\text{if } (\text{to\_float } 5)/2.0 = 0 \text{ then } 3.3 \text{ else } \sum_{x=1.0}^{10} (\text{to\_float } x) + 2.0$	well-formed and **not** well-typed
$\sum_{x=\text{true}}^{\text{false}} x \| \| \text{true}$	well-formed and **not** well-typed

Note that the last two expressions are syntactically allowed, but they are not well-typed. The range for a sum-expression must be given using integers and moreover we can only compare two expressions that have the same type.

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 `exp.ml`.

Please see the file `exp.ml` for the encoding of expressions in a data type `exp`. In the implementation, we model variables using strings.

10 points Extend on paper the definition for $\text{FV}(e)$ which computes the free variables of an expression to the new expressions (not , &&, ||, to_float , trunc and in particular define $\text{FV}(\sum_{x=e_1}^{e_2} e)$.

10 points Implement a function `fv` that takes as input an expression `e` and returns the names of the free variable in `e` (no duplicates).

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.

10 points Following your specification of $[e/x]e'$ implement a function `subst (e,x) e'` which replaces free occurrences of `x` in the expression `e'` with `e`.

Hint: We have already provided a function `genVar` which generates a new variable by taking as input a string and appending it to a unique number.

10 points Define on paper typing rules for each of the expressions. Note that not all expressions are closed. Consider for example the sum-expression $\sum_{x=0}^{n} (\text{to\_float } x) * (\text{to\_float } x)$. Here we can only type check the sub-expression $(\text{to\_float } x) * (\text{to\_float } x)$ knowing that x is an integer. We therefore 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 adopted below for your convenience.

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

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

$$\frac{\Gamma \vdash e_1 : \text{float} \quad \Gamma \vdash e_2 : \text{float}}{\Gamma \vdash e_1 + e_2 : \text{float}} \qquad \frac{\Gamma \vdash e_1 : \text{float} \quad \Gamma \vdash e_2 : \text{float}}{\Gamma \vdash e_1 * e_2 : \text{float}} \qquad \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 = e_2 : \text{bool}}$$

However the rules for some of the unary and binary operations as well as for $\text{to\_float } e$ and $\text{trunc } e$ are still missing. Most importantly, the typing rule for $\sum_{x=e_1}^{e_2} e$ must be defined. Your task is to add them on paper.

10 points Implement a type inference engine for our language following your specification of the typing rules by implementing the function `infer ctx e` which takes a context `ctx` and an expression `e` and returns the type of expression.

For simplicity, we are modelling our context using association lists pairing the variable name with its type.

15 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 $\sum_{x=e_1}^{e_2} e_3$.

10 points Implement an evaluator for our language following your specification of evaluation rules.

Q2. Type Inference [10 points]

As we have discussed in class, type inference often proceeds by introducing type variables for yet unknown types and ask whether we can find an instantiation for these type variables such that a given term becomes well-typed. For our arithmetic language above we were lucky that we always knew that the variable x in $\sum_{x=e_1}^{e_2} e$ stands for an integer. This may however not be the case, when our language supports general functions.

For the following examples, give an instantiation for the type variables, if there exists an instantiation such that the term is well-typed. The instantiation should be the most general one possible. If there is no such instantiation explain why.

- Does there exist an instantiation for the type variables α and β s.t.
  fun `x y -> (x y) + 2.0` has type $\alpha \Rightarrow \beta$ in OCaml?
- Does there exist an instantiation for the type variables α, β, γ, and δ s.t.
  fun `u w x -> u (w x) || (u 2.0)` has type $\alpha \Rightarrow \beta \Rightarrow \gamma \Rightarrow \delta$ in OCaml?

Q3. Course Evaluations! (5 points)

Done! Fantastic. Now, please take a moment to fill out course evaluations! How to submit the answer to this question and earn 5 points? – You tell us, if you have or intend to fill out your course evaluations by editing and submitting the file `course-eval.txt`.

Here a reminder why this is important:

- 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 doing the homeworks. 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!