Assignment 5

Joshua Dunfield
due: Monday, 19 March 2018

Name(s): Estimated time spent (per person):

Submit your modified a5.rkt by email (joshuad@cs.queensu.ca) as a5-yourname.rkt.
Note: Questions marked with a + are bonus questions. You can receive full marks without doing them.

Types

\[ S, T := \text{unit} \quad \text{unit type} \]
\[ \quad | \text{int} \quad \text{type of integers} \]
\[ \quad | \text{bool} \quad \text{type of booleans} \]
\[ \quad | S \rightarrow T \quad \text{type of functions on } S \text{ that produce } T \]
\[ \quad | S \times T \quad \text{type of pairs of one } S \text{ and one } T \]

Typing contexts

\[ \Gamma := \emptyset \quad \text{empty context} \]
\[ \quad | \Gamma, x : S \quad x \text{ has type } S \]

[\[ \Gamma \vdash e : T \]] Under assumptions \( \Gamma \), expression \( e \) has type \( T \)

\[
\frac{\text{type-assum}}{(x : S) \in \Gamma}{\Gamma \vdash x : S} \\
\frac{\Gamma \vdash x : S}{\Gamma \vdash \text{unitIntro}}{\Gamma \vdash \text{unit}} \\
\frac{\Gamma \vdash e_1 : \text{int}}{\Gamma \vdash (+ e_1 e_2) : \text{int}}{\text{type-add}} \\
\frac{\Gamma \vdash e_1 : \text{int}}{\Gamma \vdash (- e_1 e_2) : \text{int}}{\text{type-sub}} \\
\frac{\Gamma \vdash e_1 : \text{int}}{\Gamma \vdash (\text{Abs } e_1) : \text{int}}{\text{type-abs}} \\
\frac{\Gamma \vdash e_1 : \text{int}}{\Gamma \vdash (\times e_1 e_2) : \text{bool}}{\text{type-equals}} \\
\frac{\Gamma \vdash e_1 : \text{int}}{\Gamma \vdash (< e_1 e_2) : \text{bool}}{\text{type-lt}} \\
\frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash \text{Ite } e e_{\text{then}} e_{\text{else}} : T}{\text{type-ite}} \\
\frac{\Gamma \vdash e_1 : S_1 \quad \Gamma \vdash e_2 : S_2}{\Gamma \vdash (\text{Pair } e_1 e_2) : (S_1 \times S_2)}{\times \text{Intro}} \\
\frac{\Gamma \vdash e : (S_1 \times S_2)}{\Gamma \vdash (\text{Proj}_1 e) : S_1}{\times \text{Elim1}} \\
\frac{\Gamma \vdash e : (S_1 \times S_2)}{\Gamma \vdash (\text{Proj}_2 e) : S_2}{\times \text{Elim2}} \\
\frac{\Gamma \vdash e : (S_1 \times S_2) \quad \Gamma, x_1 : S_1, x_2 : S_2 \vdash e_{12} : T}{\Gamma \vdash (\text{Split } e_0 (x_1 x_2 \Rightarrow e_{12})) : T}{\text{type-split}} \\
\frac{\Gamma \vdash f : S \vdash e : S}{\Gamma \vdash (\text{Rec } f e : S)}{\text{type-rec}}
\]

Figure 1 Typing with functions, integers, booleans, sums (unions), and pairs (structs)
Implementing Typing in Racket

1 Implementing Typing in Racket

Question 1. The file a5.rkt includes a Racket procedure typeof, intended to take a context \( \Gamma \) and expression \( e \) and either

- returns a type \( S \) such that \( \Gamma \vdash e : S \) according to the rules in Figure 1, or
- returns \#f (Racket’s “false”), if there is no such type.

However, the typeof procedure is incomplete: the branches for Ite, Split and Rec always return \#f instead of implementing the respective rules (type-ite, type-split, type-rec).

Try to understand the existing code in relation to the corresponding rules. The hardest part of implementing type-ite will probably be checking that the type returned for \( e_{\text{then}} \) matches the type returned for \( e_{\text{else}} \), so looking at the code for \( \rightarrow \text{Elim} \) may be helpful, because \( \rightarrow \text{Elim} \) has to check that the type returned for \( e_2 \) matches the domain \( S \) of the type returned for \( e_1 \).

Also, the rule for Rec is fairly similar to the rule for Lam—both rules have one premise, and both rules add to the typing context in that premise.

2 Emptiness (bonus)

With sadness, I had to add the argument type \( S \) to the expression form Lam. Otherwise, the code wouldn’t know what type \( S \) to give for \( x \) in the context in the \( \rightarrow \text{Intro} \) premise \( \Gamma, x : S \vdash e : T \).

Similarly, I had to include the type \( S \) in the recursion expression Rec.

By accident, this allows something rather strange. On a3 we saw that the formula \( \neg A \) can be simulated by the formula \( (A \supset False) \). We can translate this formula by replacing \( A \) with \( S \) and \( \supset \) with \( \rightarrow \), but how can we translate False? We can’t use the expression False because it is an expression, not a type.

To have a type that corresponds to the formula False, we need a new type, for which we will use the symbol \( \bot \). There will be no values of type \( \bot \), but there are expressions of this type.

Question 2(a)+. Design a single elimination rule for \( \bot \) (based on FalseElim from a3).

\[
e =
\]

Question 2(b)+. Find an expression \( e \) such that \( \emptyset \vdash e : \bot \). Do not add an introduction rule for \( \bot \) ! Your expression should have type \( \bot \), in the empty context, using only the rules from Figure 1 (and possibly your \( \bot \)Elim, but that is unlikely to help since you are trying to find a “sneaky” way to introduce \( \bot \), not to eliminate it).

Question 2(c)+. Write the derivation showing that \( \emptyset \vdash e : \bot \).
§2 Emptiness (bonus)

3 Progress

For this question, we need the small-step semantics.

Expressions  \( e \ ::= \ (\) \hspace{1cm} \text{Contexts} \hspace{1cm} C \ ::= \ [\] \\
\| (Rec \ x \ S \ e) \hspace{1cm} | (Abs \ e) \hspace{1cm} | (\ + \ e \ e) \hspace{1cm} | (\ - \ e \ e) \hspace{1cm} | (\ = \ e \ e) \hspace{1cm} | (\ < \ e \ e) \\
\| n \hspace{1cm} | (\ True \ | \ False \ | (Ite \ e \ e \ e) \hspace{1cm} | (\ Proj_1 \ e) \hspace{1cm} | (\ Proj_2 \ e) \hspace{1cm} | (\ Split \ e \ (x \ x \ =\Rightarrow \ e)) \\
Values \hspace{1cm} v \ ::= \ () \hspace{1cm} | n \hspace{1cm} | (\ True \ | \ False \ | x \hspace{1cm} | (\ Lam \ x \ S \ e) \hspace{1cm} | (\ Pair \ v \ v) \\

\[ e \mapsto_R e' \] Expression \( e \) reduces to \( e' \)

\[ (+ n_1 n_2) \mapsto_R (n_1 + n_2) \] \hspace{1cm} \text{red-add} \hspace{1cm} \[ (- n_1 n_2) \mapsto_R (n_1 - n_2) \] \hspace{1cm} \text{red-sub} \hspace{1cm} \[ (Abs \ n) \mapsto_R n \] \hspace{1cm} \text{red-abs} \\
\[ (= n_1 n_2) \mapsto_R (n_1 = n_2) \] \hspace{1cm} \text{red-equals} \hspace{1cm} \[ (< n_1 n_2) \mapsto_R (n_1 < n_2) \] \hspace{1cm} \text{red-less-than} \\
\[ (Ite \ True \ e_{\text{then}} \ e_{\text{else}}) \mapsto_R e_{\text{then}} \] \hspace{1cm} \text{red-ite-then} \hspace{1cm} \[ (Ite \ False \ e_{\text{then}} \ e_{\text{else}}) \mapsto_R e_{\text{then}} \] \hspace{1cm} \text{red-ite-then} \\
\[ (Call \ (Lam \ x \ S \ e) \ v) \mapsto_R [v/x]e \] \hspace{1cm} \text{red-beta} \\
\[ (Proj_1 (Pair \ v_1 \ v_2)) \mapsto_R v_1 \] \hspace{1cm} \text{red-proj1} \hspace{1cm} \[ (Proj_2 (Pair \ v_1 \ v_2)) \mapsto_R v_2 \] \hspace{1cm} \text{red-proj2} \\
\[ (Split \ (Pair \ v_1 \ v_2) \ (x_1 \ x_2 \Rightarrow \ e_{12})) \mapsto_R [v_1/x_1][v_2/x_2]e_{12} \] \hspace{1cm} \text{red-split} \\
\[ (Rec \ f \ S \ e) \mapsto_R [(Rec \ f \ S \ e) / f]e \] \hspace{1cm} \text{red-rec} \\

\[ e \mapsto e' \] expression \( e \) takes one step to \( e' \)

\[ C[e] \mapsto_R C[e'] \] \hspace{1cm} \text{step-context}
§3 Progress

Question 3(a). Progress says that

If $\emptyset \vdash e : S$ then either (1) $e$ is a value, or (2) there exists $e'$ such that $e \mapsto e'$.

For most languages, including ours, it is impossible to prove progress without first proving a lemma known as canonical forms or value inversion.

The first name, canonical forms, comes from the idea that the values of a given type—as opposed to expressions that are not values—are the original or canonical forms of that type. For example, while $(+ 11)$ and $(- 53)$ and $(- (\text{Abs} - 3) 1)$ are all expressions of type int—and, in a sense, represent the same integer 2 since they all eventually step to 2—we would not consider these expressions as defining the set of integers. But we can say that the values of type int—which are the integer constants $n$—define the integers.

The second name, value inversion, comes from the fact that the lemma uses inversion on a given derivation—but not the inversion we have often used, where we reason either from (a) knowing that we have an expression $e$ of a particular form, say $(\text{Call } e_1 e_2)$, or (b) knowing that the conclusion of a derivation is by some particular rule, say $\rightarrow$Elim. Instead, the inversion is based on the combination of two facts:

- We know that the expression is a value.
- We know something about the expression’s type.

Here is the complete value inversion, or canonical forms, lemma for our current language. There is one part for each production in the grammar of types.

**Lemma 1** (Value Inversion).

1. If $\emptyset \vdash v : \text{unit}$ then $v = ()$.
2. If $\emptyset \vdash v : \text{bool}$ then either $v = \text{True}$ or $v = \text{False}$.
3. If $\emptyset \vdash v : \text{int}$ then there exists $n$ such that $v = n$.
4. If $\emptyset \vdash v : (S_1 \times S_2)$ then there exist $v_1$ and $v_2$ such that $v = (\text{Pair } v_1 v_2)$ and $\emptyset \vdash v_1 : S_1$ and $\emptyset \vdash v_2 : S_2$.
5. If $\emptyset \vdash v : (S_1 \rightarrow S_2)$ then there exist $x$ and $e$ such that $v = (\text{Lam } x S_1 e)$ and $x : S_1 \vdash e : S_2$.

**Proof.** [Unusually, this proof does not need induction.]

Part 1: The only rule whose conclusion can be $\emptyset \vdash () : \text{unit}$ is unitIntro. By inversion on unitIntro, we have $v = ()$. [In full detail for the impossible cases:

- In type-assum, we have $x$ (which is a value) but the context is $\emptyset$, so the premise is $(x : \text{unit}) \in \emptyset$ which is impossible.
- In $\rightarrow$Intro, the expression being typed is a value, but the type is a $\rightarrow$ which does not match the given unit, so this case is impossible.
- In $\rightarrow$Elim, the expression being typed has the form $(\text{Call } e_1 e_2)$, which is not a value.
- In type-true, type-false and intIntro, the expression being typed is a value but the type does not match.
§3  Progress

- In type-add, type-sub, type-abs, type-equals, type-lt, type-ite, ×Elim1, ×Elim2, type-split and type-rec, the expression being typed is not a value.

- In ×Intro, the expression being typed could be a value (if the two subexpressions \(e_1\) and \(e_2\) are values, then \((\text{Pair } e_1 \ e_2)\) is a value), but the type does not match.

End of the detail for the impossible cases.]

Part 2: [proof omitted]

Part 3: [proof omitted]

Part 4: Question 3(a). Fill in the four listed cases.

Consider cases of the rule concluding \(\emptyset \vdash v : (S_1 \times S_2)\). [Either explain why the case is impossible, even if you are only repeating what I gave for Part 1, or show the goal for Part 4 as stated by the lemma.]

- type-assum:

- \(\rightarrow\text{Intro}:

- \(\times\text{Elim}:

- \(\times\text{Intro}:

- The remaining cases are impossible for reasons similar to those given in Part 1.

Part 5: Question 3(b). Complete the missing case.

- \(\rightarrow\text{Intro}:

- The remaining cases are impossible for reasons similar to those given in Part 1.
§3 Progress

**Question 3(c)+.** For the type \( \perp \) (see Question 2), value inversion is a little unusual. State and prove the appropriate lemma.