Assignment 4

Joshua Dunfield

due: Friday, 15 March 2018

Name(s):          Estimated time spent:

Note: Questions marked with a + are bonus questions. You can receive full marks without doing them.
§1 Typing

1 Typing

Types

\[
S, T ::= \text{unit} \quad \text{unit type}
\]
\[
| \text{int} \quad \text{type of integers}
\]
\[
| \text{bool} \quad \text{type of booleans}
\]
\[
| S \rightarrow T \quad \text{type of functions on } S \text{ that produce } T
\]
\[
| S \times T \quad \text{type of pairs of one } S \text{ and one } T
\]
\[
| S + T \quad \text{disjoint union or sum type: contains either an } S \text{ or a } T
\]

Typing contexts

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

Now consider the typing rules in Figure 1.

\[\Gamma \vdash e : T\]

Under assumptions \(\Gamma\), expression \(e\) has type \(T\)

\[
\frac{(x : S) \in \Gamma}{\Gamma \vdash x : S} \quad \text{type-assum}
\]
\[
\frac{\Gamma, x : S \vdash e : T}{\Gamma \vdash (\text{Lam } x \ e) : (S \rightarrow T)} \quad \text{Intro}
\]
\[
\frac{\Gamma \vdash e_1 : (S \rightarrow T) \quad \Gamma \vdash e_2 : S}{\Gamma \vdash (\text{Call } e_1 \ e_2) : T} \quad \text{Elim}
\]
\[
\frac{\Gamma \vdash () : \text{unit}}{\text{unitIntro}}
\]
\[
\frac{\Gamma \vdash \text{True} : \text{bool}}{\text{type-true}}
\]
\[
\frac{\Gamma \vdash \text{False} : \text{bool}}{\text{type-false}}
\]
\[
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash (+ \ e_1 \ e_2) : \text{int}} \quad \text{type-add}
\]
\[
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash (- \ e_1 \ e_2) : \text{int}} \quad \text{type-sub}
\]
\[
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash (= \ e_1 \ e_2) : \text{bool}} \quad \text{type-equals}
\]
\[
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash (< \ e_1 \ e_2) : \text{bool}} \quad \text{type-lt}
\]
\[
\frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash \text{then} : T} \quad \text{type-then}
\]
\[
\frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash \text{else} : T} \quad \text{type-else}
\]
\[
\frac{\Gamma \vdash e_1 : S_1 \quad \Gamma \vdash e_2 : S_2}{\Gamma \vdash (\text{Case } e \ (x_1 \Rightarrow e_1) \ (x_2 \Rightarrow e_2)) : T} \quad \text{type-case}
\]
\[
\frac{\Gamma \vdash e_1 : S_1 \quad \Gamma \vdash e_2 : S_2}{\Gamma \vdash (\text{Inj}_1 \ e_1) : (S_1 + S_2)} \quad \text{Intro1}
\]
\[
\frac{\Gamma \vdash e_2 : S_2}{\Gamma \vdash (\text{Inj}_2 \ e_2) : (S_1 + S_2)} \quad \text{Intro2}
\]
\[
\frac{\Gamma \vdash e : (S_1 + S_2) \quad \Gamma, x_1 : S_1 \vdash e_1 : T \quad \Gamma, x_2 : S_2 \vdash e_2 : T}{\Gamma \vdash (\text{Case} \ e \ (x_1 \Rightarrow e_1) \ (x_2 \Rightarrow e_2)) : T} \quad \text{type-case}
\]
\[
\frac{\Gamma \vdash e_1 : S_1 \quad \Gamma \vdash e_2 : S_2}{\Gamma \vdash (\text{Pair } e_1 \ e_2) : (S_1 \times S_2)} \quad \text{Intro}
\]
\[
\frac{\Gamma \vdash e : (S_1 \times S_2) \quad \Gamma \vdash (\text{Proj}_1 \ e) : S_1}{\Gamma \vdash (\text{Proj}_1 \ e) : S_1} \quad \text{Elim1}
\]
\[
\frac{\Gamma \vdash e : (S_1 \times S_2) \quad \Gamma \vdash (\text{Proj}_2 \ e) : S_2}{\Gamma \vdash (\text{Proj}_2 \ e) : S_2} \quad \text{Elim2}
\]

Figure 1 Typing with functions, integers, booleans, sums, and pairs
§1 Typing

**Question 1(a).** Complete the following typing derivation. You can write $\Gamma$ instead of $y : \text{bool}$.

\[
\frac{\quad}{y : \text{bool} \vdash (\text{Ite } y \ 1 \ 0) : \text{int}} \quad \text{type-ite}
\]

**Question 1(b).** Complete the following typing derivation. You can write $\Gamma$ instead of $z : (\text{int} + \text{bool})$.

\[
\frac{(z : (\text{int} + \text{bool})) \in \Gamma}{\Gamma \vdash z : (\text{int} + \text{bool})} \quad \text{type-assum}
\]

\[
\frac{\Gamma, x : \text{int} \vdash \quad \Gamma, y : \text{bool} \vdash}{\Gamma, x : \text{int} \vdash \quad \Gamma, y : \text{bool} \vdash}{\frac{\quad}{z : (\text{int} + \text{bool}) \vdash (\text{Case } z (x \Rightarrow \text{True}) (y \Rightarrow y)) : \text{bool}}} \quad \text{+Elim}
\]

**Question 1(c)+.** Booleans are not really necessary, because we can write $(\text{Inj}_1 ())$ instead of True, and $(\text{Inj}_2 ())$ instead of False, and use Case instead of Ite. Translate the expression from 1(a), $(\text{Ite } y \ 1 \ 0)$, into an expression that has type int under the typing context $y : (\text{unit} + \text{unit})$

**Hint:** think about the derivation in 1(b).
Consider the sequent calculus rules in Figure 2.

\[\begin{array}{c}
\Gamma \vdash A \text{ true} \\
\hline
\text{true-Intro}
\end{array}\]

Under assumptions \(\Gamma\), formula \(A\) is true

\[\begin{array}{c}
x[A \text{ true}] \in \Gamma \\
\hline
\Gamma \vdash A \text{ true} \\
\text{sc-assum}
\end{array}\]

\[\begin{array}{c}
\Gamma \vdash (A \supset B) \text{ true} \\
\hline
\Gamma \vdash B \text{ true} \\
\text{sc-\supset-Elim}
\end{array}\]

\[\begin{array}{c}
\Gamma \vdash A \supset B \text{ true} \\
\hline
\Gamma \vdash A \text{ true} \\
\text{sc-\supset-Intro}
\end{array}\]

\[\begin{array}{c}
\Gamma \vdash A \text{ true} \\
\hline
\Gamma \vdash \text{true} \\
\text{sc-TrueIntro}
\end{array}\]

\[\begin{array}{c}
\Gamma \vdash A \text{ true} \\
\hline
\Gamma \vdash A \lor B \text{ true} \\
\text{sc-\lor-Intro1}
\end{array}\]

\[\begin{array}{c}
\Gamma \vdash B \text{ true} \\
\hline
\Gamma \vdash A \lor B \text{ true} \\
\text{sc-\lor-Intro2}
\end{array}\]

\[\begin{array}{c}
\Gamma \vdash A \lor B \text{ true} \\
\Gamma, x[A \text{ true}] \vdash C \text{ true} \\
\hline
\Gamma, y[B \text{ true}] \vdash C \text{ true} \\
\text{sc-\lor-Elim}
\end{array}\]

\[\begin{array}{c}
\Gamma \vdash A \text{ true} \\
\hline
\Gamma \vdash A \land B \text{ true} \\
\text{sc-\land-Intro}
\end{array}\]

\[\begin{array}{c}
\Gamma \vdash A \text{ true} \\
\hline
\Gamma \vdash B \text{ true} \\
\text{sc-\land-Elim1}
\end{array}\]

\[\begin{array}{c}
\Gamma \vdash A \land B \text{ true} \\
\Gamma \vdash A \text{ true} \\
\hline
\Gamma \vdash B \text{ true} \\
\text{sc-\land-Elim2}
\end{array}\]

**Figure 2** Sequent calculus, with \(\supset\), True, \(\lor\), and \(\land\)

Some of the typing rules from Figure 1 have a curious property: if we change some of the metavariables, remove the expression and colon, translate the types, and add the word \text{true}, we get a rule in Figure 2.

For example, the rule \(\times\)Elim1 becomes the rule \(\land\)Elim1:

\[\begin{array}{c}
\Gamma \vdash e : S_1 \times S_2 \\
\hline
\Gamma \vdash \text{proj}_1 e : S_1 \\
\text{sc-\times-Elim1}
\end{array}\]

The types can be translated as follows:

- \text{unit} becomes \text{True}
- \(\times\) becomes \(\land\)
- \(\lor\) becomes \(\lor\)
- \(\rightarrow\) becomes \(\rightarrow\)

Assumptions need some extra work; for example, in translating \(\rightarrow\)Intro, \(x : S\) becomes \(x[S \text{ true}]\):

\[\begin{array}{c}
\Gamma, x : S \vdash e : T \\
\hline
\Gamma \vdash (\text{Lam } x e) : S \rightarrow T \\
\text{sc-\rightarrow-Intro}
\end{array}\]

Not all the typing rules in Figure 1 have meaningful translations. The rules involving arithmetic, among others, do not lead to anything interesting.
**Question 2(a)**. Following the procedure above, translate \( \rightarrow\text{Elim} \). Indicate which rule from Figure 2 was “rediscovered” by this translation. (For example, by translating \( \times\text{Elim1} \), I rediscovered \( \text{sc-}&\text{Elim1} \).)

**Question 2(b)**. In addition to translating rules, we can translate derivations.

\[
(\lambda f (\lambda g (\lambda x (\text{Call } f (\text{Call } g x)))))
\]

This expression is a function that creates a new function, \( (\lambda x (\text{Call } f (\text{Call } g x))) \), which *composes* \( f \) and \( g \): its body, in less Rackety syntax, is

\[
f(g(x))
\]

You may want to experiment with the expression in Racket (\texttt{lambda.rkt} on the course website).

The following is a typing derivation for the expression. Let \( \Gamma_{fgx} = f : (S_2 \rightarrow S_3), g : (S_1 \rightarrow S_2), x : S_1 \).

Using the rule you translated in 2(a), complete the following *sequent calculus* derivation. Above, I wrote the expressions in gray to make it easier to ignore them as you complete the derivation.

You may write \( \Gamma_{fgx} \) instead of \( f[(A_2 \supset A_3) \text{ true}], g[(A_1 \supset A_2) \text{ true}], x[A_1 \text{ true}] \).

\[
\begin{align*}
(f : S_2 \rightarrow S_3) &\in \Gamma_{fgx} : \text{type-assum} &
(g : S_1 \rightarrow S_2) &\in \Gamma_{fgx} : \text{type-assum} &
(x : S_1) &\in \Gamma_{fgx} : \text{type-assum} \\
\Gamma_{fgx} &\vdash f : S_2 \rightarrow S_3 &
\Gamma_{fgx} &\vdash g : S_1 \rightarrow S_2 &
\Gamma_{fgx} &\vdash x : S_1 \\
\Gamma_{fgx} &\vdash (\text{Call } g x) : S_2 &
\Gamma_{fgx} &\vdash \Gamma_{fgx} &\vdash (\text{Call } f (\text{Call } g x)) : S_3 \\
\Gamma_{fgx} &\vdash f : (S_2 \rightarrow S_3) &
\Gamma_{fgx} &\vdash g : (S_1 \rightarrow S_2) &
\Gamma_{fgx} &\vdash (\lambda x (\text{Call } f (\text{Call } g x))) : (S_1 \rightarrow S_2) \rightarrow (S_1 \rightarrow S_3) \\
\emptyset &\vdash (\lambda f (\lambda g (\lambda x (\text{Call } f (\text{Call } g x))) : (S_2 \rightarrow S_3) \rightarrow (S_1 \rightarrow S_2) \rightarrow (S_1 \rightarrow S_3)
\end{align*}
\]

a4, J. Dunfield, CISC 865, Winter 2019

5

2019/3/17
§2 Mirror World

Question 2(c)+.

A single &-elimination: We have two elimination rules for &, which separately extract a sub-formula. Design a single sequent-calculus elimination rule for &. Hint: think about the structure of sc-∨Elim. Second hint: think about the connection between \((P_1 & P_2) \supset Q\) and \(P_1 \supset (P_2 \supset Q)\), and the shape of a derivation of \(\emptyset \vdash P_1 \supset (P_2 \supset Q)\) true: how many assumptions are needed within that derivation?

A single ×-elimination rule: Translate your new &-elimination rule into a typing rule. You will need to extend the grammar of expressions e.

Small-step semantics: Extend the small-step semantics with a reduction rule for your new expression form, and extend the grammar C of contexts as appropriate.
§2 Mirror World

3 Progress

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

Expressions \( e ::= () \)  \hspace{1cm} \text{Contexts } \ C ::= [] \\
\begin{align*}
\text{n} \mid (+ \ e \ e) \mid (- \ e \ e) \mid (\text{Abs} \ e) \\
\text{True} \mid \text{False} \mid (\text{Ite} \ e \ e) \\
(= \ e \ e) \mid (< \ e \ e) \\
\text{x} \mid (\text{Lam} \ x \ S \ e) \mid (\text{Call} \ e \ e) \\
(\text{Pair} \ e \ e) \mid (\text{Proj} _1 \ e) \mid (\text{Proj} _2 \ e)
\end{align*}

Values \( v ::= () \)
\begin{align*}
\text{n} \mid \text{True} \mid \text{False} \\
\text{x} \mid (\text{Lam} \ x \ S \ e) \\
(\text{Pair} \ v \ v)
\end{align*}

\[
e \mapsto R \ e'
\]
Expression \( e \) reduces to \( e' \)

\[
(+ \ n_1 \ n_2) \mapsto_R (n_1 + n_2) \hspace{1cm} (- \ n_1 \ n_2) \mapsto_R (n_1 - n_2) \\
(\text{Abs} \ n) \mapsto_R |n|
\]

\[
(= \ n_1 \ n_2) \mapsto_R (n_1 = n_2) \hspace{1cm} (< \ n_1 \ n_2) \mapsto_R (n_1 < n_2)
\]

\[
(\text{Ite} \ True \ e_{\text{then}} \ e_{\text{else}}) \mapsto_R e_{\text{then}} \\
(\text{Ite} \ True \ e_{\text{then}} \ e_{\text{else}}) \mapsto_R e_{\text{then}}
\]

\[
(\text{Call} \ (\text{Lam} \ x \ S \ e) \ v) \mapsto_R [v/x]e
\]

\[
(\text{Proj} _1 \ (\text{Pair} \ v_1 \ v_2)) \mapsto_R v_1 \hspace{1cm} (\text{Proj} _2 \ (\text{Pair} \ v_1 \ v_2)) \mapsto_R v_2
\]

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

\[
\frac{e \mapsto_R e'}{C[e] \mapsto C[e']}
\]
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 $(+ 1 1)$ and $(- 5 3)$ 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 $v = \text{Lam } x e$ 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, $\times$Elim1 and $\times$Elim2, the expression being typed is not a value.

- In $\times$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 prove the goal for Part 4: “there exist $v_1$ and $v_2$ such that $v = (\text{Pair } v_1 v_2)$ and $\emptyset \vdash v_1 : S_1$ and $\emptyset \vdash v_2 : S_2$.”]

- type-assum:

- $\rightarrow$Intro:

- $\times$Elim:

- $\times$Intro:

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

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

- $\rightarrow$Intro:

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

a4, J. Dunfield, CISC 865, Winter 2019

2019/3/17