Assignment 3

J. Dunfield

due: Tuesday, 10 March 2020

If you wish, you may work in a group of 2 on this assignment.

ID number(s):

Name(s) (optional):

Estimated time spent:
(does not affect your mark)

Please fill in the “Estimated time spent” (above). This helps me to design reasonable assignments.

Late policy

Assignments may be submitted up to 48 hours late with a 20% penalty (that is, I multiply your mark by 0.8).

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

1  Typing

Types

| S, T ::= unit type |
|---|---|
| int type of integers |
| bool type of booleans |
| S → T type of functions on S that produce T |
| S × T type of pairs of one S and one T |
| S + T disjoint union or sum type: contains either an S or a T |

Typing contexts

| Γ ::= ∅ empty context |
|---|---|
| Γ, x : S x has type S |

Now consider the typing rules in Figure 1.

[Γ ⊢ e : T] Under assumptions Γ, expression e has type T

- (x : S) ∈ Γ, type-assum
- Γ, x : S ⊢ e : T, unitIntro
- Γ ⊢ (Lam x e) : (S → T), Intro
- Γ ⊢ (Call e₁ e₂) : T, Elim

- Γ ⊢ () : unit, unitIntro
- Γ ⊢ True : bool, type-true
- Γ ⊢ False : bool, type-false

- Γ ⊢ e₁ : int, Γ ⊢ e₂ : int, type-add
- Γ ⊢ (+ e₁ e₂) : int, type-add
- Γ ⊢ (- e₁ e₂) : int, type-sub

- Γ ⊢ e₁ : int, Γ ⊢ e₂ : int, type-equals
- Γ ⊢ (= e₁ e₂) : bool, type-equals
- Γ ⊢ (< e₁ e₂) : bool, type-lt

- Γ ⊢ e : bool, Γ ⊢ e₁ : T then : T, Γ ⊢ e₂ : T else : T, type-ite

- Γ ⊢ e₁ : S₁
- Γ ⊢ (Inj₁ e₁) : (S₁ + S₂), Intro1
- Γ ⊢ e₂ : S₂
- Γ ⊢ (Inj₂ e₂) : (S₁ + S₂), Intro2

- Γ ⊢ e : (S₁ + S₂), Γ, x₁ : S₁ ⊢ e₁ : T, Γ, x₂ : S₂ ⊢ e₂ : T, +Elim

- Γ ⊢ e₁ : S₁
- Γ ⊢ e₂ : S₂, Γ ⊢ (Pair e₁ e₂) : (S₁ × S₂), Intro
- Γ ⊢ e : (S₁ × S₂), Γ ⊢ (Proj₁ e) : S₁, ×Elim1
- Γ ⊢ e : (S₁ × S₂), Γ ⊢ (Proj₂ e) : S₂, ×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 $x : \text{bool}$.

\[ \begin{array}{c}
    x : \text{bool} \vdash (\text{Ite } y 3 4) : \text{int} \\
    \text{type-ite}
\end{array} \]

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

\[ \begin{array}{c}
    (z : (\text{bool} + \text{int})) \in \Gamma \\
    \Gamma \vdash z : (\text{bool} + \text{int}) \quad \text{type-assum} \\
    \Gamma, y : \text{bool} \vdash \\
    \Gamma, x : \text{int} \vdash \\
    z : (\text{bool} + \text{int}) \vdash (\text{Case } z (y \Rightarrow y) (x \Rightarrow \text{True})) : \text{bool} \\
    \text{+Elim}
\end{array} \]

**Question 1(c).** Booleans are not really necessary, because we can write $\langle \text{Inj}_1 () \rangle$ instead of True, $\langle \text{Inj}_2 () \rangle$ instead of False, and Case instead of Ite. Translate the expression from 1(a), $(\text{Ite } y 3 4)$, into an expression that has type int under the typing context

\[ y : (\text{unit + unit}) \]

and which would step to 3 if $y$ were replaced with $\langle \text{Inj}_1 () \rangle$, and to 4 if $y$ were replaced with $\langle \text{Inj}_2 () \rangle$. **Hint:** think about the derivation in 1(b).
2 Mirror World

Consider the sequent calculus rules in Figure 2.

![Sequent calculus rules](image)

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

For example, the rule \(\times\)Elim1 becomes the rule sc-\(\wedge\)Elim1:

\[
\begin{align*}
\Gamma &\vdash \text{proj}_1 e : S_1 \\
\Gamma &\vdash e : S_1 \times S_2 \\
\Gamma &\vdash e : A_1 \times A_2 \\
\Gamma &\vdash A_1 \times A_2 \\
\Gamma &\vdash A_1 \wedge A_2 \\
\Gamma &\vdash A_1 \text{ true} \\
\end{align*}
\]

Here we translated the type \(S_1 \times S_2\) to \(A_1 \wedge A_2\) by replacing \(\times\) with \(\wedge\). The types can be translated as follows:

- unit becomes True
- \(\times\) becomes \(\wedge\)
- \(+\) becomes \(\vee\)
- \(\rightarrow\) becomes \(\supset\)

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

\[
\begin{align*}
\Gamma, x : S &\vdash e : \top \\
\Gamma &\vdash (\text{Lam } x \ e) : S \to \top \\
\Gamma &\vdash (\text{Lam } x \ e) : A \to B \\
\Gamma &\vdash (A \supset B) \text{ true} \\
\end{align*}
\]

Not all the typing rules in Figure 1 have meaningful translations. The rules involving arithmetic, among others, do not lead to anything interesting.
§2 Mirror World

Question 2(a). Following the procedure above, translate $\rightarrow$Elim. Indicate which rule from Figure 2 was “rediscovered” by this translation. (For example, by translating $\times$Elim1, I rediscovered sc-$\land$Elim1.)

Question 2(b) + (bonus).

A single $\land$-elimination: We have two elimination rules for $\land$, which separately extract a sub-formula. Design a single sequent-calculus elimination rule for $\land$. **Hint:** think about the structure of sc-$\lor$Elim. **Second hint:** think about the connection between $(P_1 \land P_2) \supset Q$ 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 $\times$-elimination rule: Translate your new $\land$-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.
3 Progress

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

Expressions $e ::= (\varepsilon)$

| $n$ | ($+ e e$) | ($- e e$) | (Square $e$) |
| True | False | (Ite $e e e$) |
| ($e e$) | ($< e e$) |
| $x$ | (Lam $x S e$) | (Call $e e$) |
| (Pair $e e$) | (Proj$_1 e$) | (Proj$_2 e$) |
| (Inj$_1 e$) | (Inj$_2 e$) |
| (Case $e (x \Rightarrow e) (x \Rightarrow e)$) |

Values $v ::= (\varepsilon)$

| $n$ |
| True | False |
| $x$ | (Lam $x e$) |
| (Pair $v v$) |
| (Inj$_1 v$) | (Inj$_2 v$) |

Expression $e \mapsto_R e'$: Expression $e$ reduces to $e'$

\[ (+ n_1 n_2) \mapsto_R (n_1 + n_2) \quad \text{red-add} \quad (Suqare n) \mapsto_R n^2 \quad \text{red-sq} \]

\[ (- n_1 n_2) \mapsto_R (n_1 - n_2) \quad \text{red-sub} \]

\[ (= n_1 n_2) \mapsto_R (n_1 = n_2) \quad \text{red-equals} \]

\[ (< n_1 n_2) \mapsto_R (n_1 < n_2) \quad \text{red-lessthan} \]

\[ (Ite \ True \ e_{\text{then}} \ e_{\text{else}}) \mapsto_R e_{\text{then}} \quad \text{red-ite-then} \]

\[ (Ite \ True \ e_{\text{then}} \ e_{\text{else}}) \mapsto_R e_{\text{then}} \quad \text{red-ite-then} \]

\[ (Call \ (Lam \ x \ e) \ v) \mapsto_R [v/x] e \quad \text{red-beta} \]

\[ (Proj$_1 (\text{Pair} \ v_1 v_2)) \mapsto_R v_1 \quad \text{red-proj1} \]

\[ (Proj$_2 (\text{Pair} \ v_1 v_2)) \mapsto_R v_2 \quad \text{red-proj2} \]

\[ (Case \ (\text{Inj$_1$} v_1) \ (x_1 \Rightarrow e_1) \ (x_2 \Rightarrow e_2)) \mapsto_R [v_1/x_1] e_1 \quad \text{red-case1} \]

\[ (Case \ (\text{Inj$_2$} v_2) \ (x_1 \Rightarrow e_1) \ (x_2 \Rightarrow e_2)) \mapsto_R [v_2/x_2] e_2 \quad \text{red-case2} \]

Expression $e \mapsto e'$: Expression $e$ takes one step to $e'$

\[ e \mapsto_R e' \]

\[ C[e] \mapsto C[e'] \quad \text{step-context} \]
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 $e \rightarrow e'$.

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 $(- 5 3)$ are both 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 $\rightarrow\text{Elim}$. 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 $x : S_1 \vdash e : S_2$.
6. If $\emptyset \vdash v : (S_1 + S_2)$ then either (1) there exists $v_1$ such that $v = (\text{Inj}_1 \ v_1)$ and $\emptyset \vdash v_1 : S_1$ or (2) there exists $v_2$ such that $v = (\text{Inj}_2 \ v_2)$ and $\emptyset \vdash v_2 : S_2$.

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

Part 1: The only rule whose conclusion can be $\emptyset \vdash () : \text{unit}$ is $\text{unitIntro}$. By inversion on $\text{unitIntro}$, 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 $\rightarrow\text{Intro}$, 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.
§3 Progress

- In \( \rightarrow \text{Elim} \), 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.
- In type-add, type-sub, type-abs, type-equals, type-lt, type-ite, \( \times \text{Elim1} \) and \( \times \text{Elim2} \), the expression being typed is not a value.
- In \( \times \text{Intro} \), 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 6: Question 3(a). Fill in the four listed cases.
Consider cases of the rule concluding \( \emptyset \vdash v : (S_1 + S_2) \). [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 6: “either (1) there exists \( v_1 \) such that \( v = (\text{Inj}_1 v_1) \) and \( \emptyset \vdash v_1 : S_1 \) or (2) there exists \( v_2 \) such that \( v = (\text{Inj}_2 v_2) \) and \( \emptyset \vdash v_2 : S_2 \).”]

- type-assum:

- \( \rightarrow \text{Intro} \):

- \( + \text{Elim} \):

- \( + \text{Intro1} \):

- \( + \text{Intro2} \): Similar to the \( + \text{Intro1} \) case.

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