IIT CS595: Topics & Applications in Programming Languages

Homework 1: Curry-Howard, Expressions continued

Your name here

October 7, 2021

Logistics

The submission and collaboration instructions (as well as the preference for submissions typeset in LaTeX) are the same as for HW0.

In particular: Submit your answers as a single .pdf or .doc file on Blackboard under the correct assignment.

1 The Curry-Howard Correspondence

We discussed the Curry-Howard Correspondence in class as a correspondence between STLC and Intuitionistic Propositional Logic (IPL). We can also add universal quantification to both. On the logic side, this gives us intuitionistic second-order logic, where you can express things like $\forall A.\forall B.A \land B \Rightarrow B \land A$ ("for all propositions A and B, A and B implies B and A"). On the PL side, it gives us STLC with "forall" types (or, equivalently, System F with notations for sums and products defined). The type corresponding to the above logic statement would be $\forall A.\forall B.(A \times B) \to (B \times A)$— the logical $\forall$ and its type equivalent even conveniently use the same symbol.

In Classical Logic, we have the bi-implication $\forall A.\forall B.(A \Rightarrow B) \Leftrightarrow (\neg A \lor B)$. Only one direction of this implication is true constructively (i.e., in intuitionistic second-order logic).

Task 1. *Write the STLC (with $\forall$) type corresponding to the logical implication $\forall A.\forall B.(\neg A \lor B) \Rightarrow (A \Rightarrow B)$.*

Answer: $\forall\alpha.\forall\beta.((\alpha \to \mathsf{void}) + \beta) \to (\alpha \to \beta)$

Task 2. *Prove the implication above by giving an STLC term of the type you wrote in Task 1.*

Answer: $\Lambda\alpha.\Lambda\beta.$
  $\lambda b : \beta.$
    $\lambda l : (\alpha \to \mathsf{void}) + \beta.$
      $\lambda a : \alpha.$
        $\mathsf{case}\ l\ \mathsf{of}\ \{x.b; y.y\}$

When invoked with a value of type β, the above term will have the type specified in task 1.

It turns out that, for any classically true proposition P, $\neg\neg P$ is true constructively (even if P is not)[1].

Task 3. *We've discussed that the Law of the Excluded Middle (now restated with $\forall$), $\forall A.A \lor \neg A$, is not true constructively. Show that $\neg\neg(\forall A.A \lor \neg A)$ is true constructively by giving an STLC term of type*

$$\forall\alpha.((\alpha + (\alpha \to \mathsf{void})) \to \mathsf{void}) \to \mathsf{void}$$

Hint: *The term will be of the form*

$$\Lambda\alpha.\lambda f : ((\alpha + (\alpha \to \mathsf{void})) \to \mathsf{void}).\ e$$

[1] If you're wondering whether this fact itself is meaningful in programming languages through the Curry-Howard correspondence, Google "continuation passing transformation".

for some e such that $\alpha; \Gamma \vdash e : \mathsf{void}$, where $\Gamma = f : ((\alpha + (\alpha \to \mathsf{void})) \to \mathsf{void})$. Can you construct terms e_1 and e_2 such that $\alpha; \Gamma \vdash e_1 : (\alpha \to \mathsf{void}) \to \mathsf{void}$ and $\alpha; \Gamma \vdash e_2 : \alpha \to \mathsf{void}$? How can you then use e_1 and e_2 to construct the function body, e?

Answer: $\Lambda\alpha.\lambda f : (\alpha + (\alpha \to \mathsf{void})) \to \mathsf{void}.f\ (\lambda a : \alpha + (\alpha \to \mathsf{void}).\mathsf{case}\ a\ \mathsf{of}\ \{x.f\ x; y.f\ y\})$

2 Division by zero

In the past, we've used type systems and the ideas of type safety (progress and preservation) to prevent run-time type errors. Here, we'll use a type system to prevent another run-time error: division by zero. Consider a language we'll call EZ, whose syntax and dynamics are below.

$$\begin{array}{rcl} \tau & ::= & \mathsf{nz} \mid \mathsf{mz} \\ e & ::= & \overline{n} \mid e + e \mid e - e \mid e * e \mid e / e \end{array}$$

$$\frac{}{\overline{n}\ \mathsf{val}}\ (\text{V-1}) \qquad \frac{e_1 \mapsto e_1' \qquad op \in \{+, -, *, /\}}{e_1\ op\ e_2 \mapsto e_1'\ op\ e_2}\ (\text{S-1}) \qquad \frac{e_2 \mapsto e_2' \qquad op \in \{+, -, *, /\}}{\overline{n_1}\ op\ e_2 \mapsto \overline{n_1}\ op\ e_2'}\ (\text{S-2})$$

$$\frac{}{\overline{n_1} + \overline{n_2} \mapsto \overline{n_1 + n_2}}\ (\text{S-3}) \qquad \frac{}{\overline{n_1} - \overline{n_2} \mapsto \overline{n_1 - n_2}}\ (\text{S-4}) \qquad \frac{}{\overline{n_1} * \overline{n_2} \mapsto \overline{n_1 n_2}}\ (\text{S-5}) \qquad \frac{n_2 \neq 0}{\overline{n_1}/\overline{n_2} \mapsto \overline{n_1/n_2}}\ (\text{S-6})$$

This is basically our expression language E from class (without let), but with only integers and with subtraction, multiplication and (integer) division added. To save space and time, we have only two search rules: rules (S-1) and (S-2) apply for all 4 arithmetic operations in place of *op*. Note that rule (S-6), as we'd hope, allows division only if $\overline{n_2}$ is nonzero. That means that a division by zero, e.g., $\overline{1}/\overline{0}$, is a "stuck" expression: it's not a value and it can't step. So in order to make Progress and Preservation hold, we need a type system that rules out division by zero. The EZ language has only integers, so we don't need types to distinguish integers from strings, but we still have two types: nz ("not zero") and mz ("maybe zero"), which we'll use to check whether an expression *might* evaluate to zero: the idea will be that if $e : \mathsf{nz}$, then e definitely does not evaluate to zero (that is, if $e \mapsto^* \overline{n}$, then $n \neq 0$). We'll only allow division by expressions that definitely won't evaluate to zero.

Task 4. *Fill in the blanks in the typing rules for EZ. The first two are done for you: any integer can have type* mz*, but it can only have* nz *if it's not 0. There are two rules for multiplication, depending on whether or not both operands are nonzero (think about why).*

Hint: *Think about, for each operation (addition, subtraction, multiplication, division), whether performing the operation on two integers can result in zero, depending on whether the two operands can be zero.* Integers can be negative.

$$\frac{}{\overline{n} : \mathsf{mz}}\ (\text{T-1}) \qquad \frac{n \neq 0}{\overline{n} : \mathsf{nz}}\ (\text{T-2}) \qquad \frac{e_1 : \mathsf{mz} \qquad e_2 : \mathsf{mz}}{e_1 + e_2 : \mathsf{mz}}\ (\text{T-3}) \qquad \frac{e_1 : \mathsf{mz} \qquad e_2 : \mathsf{mz}}{e_1 - e_2 : \mathsf{mz}}\ (\text{T-4})$$

$$\frac{e_1 : \mathsf{nz} \qquad e_2 : \mathsf{nz}}{e_1 * e_2 : \mathsf{nz}}\ (\text{T-5}) \qquad \frac{e_1 : \mathsf{mz} \qquad e_2 : \mathsf{mz}}{e_1 * e_2 : \mathsf{mz}}\ (\text{T-6}) \qquad \frac{e_1 : \mathsf{mz} \qquad e_2 : \mathsf{nz}}{e_1 / e_2 : \mathsf{mz}}\ (\text{T-7})$$

You may find it concerning that, for example, there's no rule for $e_1 + e_2$ when e_1 has type nz and e_2 has type mz. This actually isn't a problem, because[2]:

[2]This is, in fact, a form of *subtyping*, which you may be familiar with from languages like Java, where whenever something has one type, it also has a more general type. We'll study this in more detail later in the course.

Theorem 1. *For any expression e, if $e : \mathsf{nz}$ then $e : \mathsf{mz}$.*

Task 5. *Prove Theorem 1 by induction on the derivation of $e : \mathsf{nz}$.*

Hint: *If you've set up the typing rules correctly, there should only be two cases. If a typing rule has a conclusion of the form $e : \mathsf{mz}$, we don't need to consider it because it can't be used to derive our assumption $e : \mathsf{nz}$.*

Answer:

- Rule (T-1): already of type mz
- Rule (T-2): $e = \overline{n}$ thus apply (T-1)
- Rule (T-3): already of type mz
- Rule (T-4): already of type mz
- Rule (T-5): $e = e_1 * e_2$
  by inversion $e_1 : \mathsf{nz}$ and $e_2 : \mathsf{nz}$
  by induction $e_1 : \mathsf{mz}$ and $e_2 : \mathsf{mz}$
  apply rule (T-6) giving $e : \mathsf{mz}$
- Rule (T-6): already of type mz
- Rule (T-7): already of type mz

The Canonical Forms lemma is similar to before, but now includes the fact that values of type nz are not zero.

Lemma 1 (Canonical Forms)**.** *If $e : \tau$ and e val, then:*

1. *If $\tau = \mathsf{nz}$, then $e = \overline{n}$ for some $n \neq 0$.*
2. *If $\tau = \mathsf{mz}$, then $e = \overline{n}$ for some n.*

Here are the statements of Progress and Preservation for EZ.

Theorem 2 (Progress)**.** *If $e : \tau$ then e val or there exists e' such that $e \mapsto e'$.*

Theorem 3 (Preservation)**.** *If $e : \tau$ and $e \mapsto e'$ then $e' : \tau$.*

Task 6. *Prove the Progress and Preservation theorems for EZ.*

Hints/Comments – you'll want to read these.

- *For Progress, you only need to prove the cases for (T-1), (T-2) and (T-7)*[3].
- *For Preservation, you only need to prove the cases for (S-3), (S-4), (S-5) and (S-6)*[4].
- *In the (S-5) case for Preservation, if you do inversion on the typing rules, there will actually be* two *rules you need to consider in your inversion and thus, two cases (think about why).*
- *You can use without proof any (true!) facts of basic arithmetic.*

Answer:
Progress:

- (T-1): $e = \overline{n}$ and $\tau = \mathsf{mz}$
  according to (V-1) e val.

[3]This is *not* for the same reason that you only need to prove two cases for Theorem 1; the other cases for Progress need to be proven too, this is just me being nice and not making you do them :)

[4]Same.

- (T-2): $e = \overline{n}$ and $\tau = \mathsf{nz}$
  according to (V-1) e val.

- (T-7): $e = e_1/e_2$ and $\tau = \mathsf{mz}$
  by inversion $e_2 : \mathsf{nz}$
  by inversion on (T-2) $e_2 \neq 0$
  by induction e_1 and e_2 either step or are values
  if they are values we can use the canonical forms lemma to show that e steps via (S-6)
  if they step then the expression steps, proving the proposition.

Preservation:

- (S-3): $e = e_1 + e_2$ and $\tau = \mathsf{mz}$
  by inversion on (T-3) $e_1 : \mathsf{mz}$ and $e_2 : \mathsf{mz}$ (note that theorem 1 says this includes nz)
  thus by induction on rule (T-3) $e' : \mathsf{mz}$

- (S-4): $e = e_1 - e_2$ and $\tau = \mathsf{mz}$
  by inversion on (T-3) $e_1 : \mathsf{mz}$ and $e_2 : \mathsf{mz}$ (note that theorem 1 says this includes nz)
  thus by induction on rule (T-3) $e' : \mathsf{mz}$

- (S-5): $e = e_1 * e_2$ there are two cases to consider

  1. when $e_1 : \mathsf{nz}$ and $e_2 : \mathsf{nz}$, $\tau = \mathsf{nz}$
     by inversion on (T-5) $e_1 : \mathsf{nz}$ and $e_2 : \mathsf{nz}$
     thus by induction $e : \mathsf{nz}$
  2. when $e_1 : \mathsf{mz}$ or $e_2 : \mathsf{mz}$, $\tau = \mathsf{mz}$
     by inversion on (T-6) $e_1 : \mathsf{mz}$ and $e_2 : \mathsf{mz}$ (note that theorem 1 says this includes nz)
     thus by induction $e' : \mathsf{mz}$

- (S-6): $e = e_1/e_2$ and $\tau = \mathsf{mz}$
  by inversion on (T-7) $e_1 : \mathsf{mz}$ and $e_2 : \mathsf{nz}$
  thus by induction $e' : \mathsf{mz}$

Task 7. *Give an expression in EZ that* does not *perform a division by zero, but is still* not well-typed *by the above typing rules. Suggest a change, addition, refinement, etc., to the type system that could make your expression well-typed (but would still rule out expressions that divide by zero). You don't need to actually implement your suggestion, or even argue that it's possible or practical, and it's fine if your solution still rules out some "OK" programs (programs that don't divide by zero). This is just to get you to start thinking about how to get what you want with type systems.*

Answer:
Example Expression: $555/(1 + 4)$
We could extend the type system, allowing in addition to tracking zero-ness we also track the signedness of numbers. So we could have some new addition rules like the following shorthand for the type rules where u* means unsigned (≥ 0) and s* means signed ($\mathbb{Z}$)

1. umz + umz : umz
2. umz + unz : unz
3. unz + umz : unz
4. unz + unz : unz

5. smz + smz : smz
6. smz + snz : smz
7. snz + smz : smz

8. snz + snz : smz

9. depending on implementation, it might also make sense to add negative signed types

So in the example we could have used rule 4 to verify that the denominator would not be zero. This would still not work with all valid expressions but at least this could potentially be done at compile time.

Also partial evaluation could potentially help.