COMP 321 Homework 7: Function, Product, and Sum Types

Dan Licata

Fall 2026

This semester, we've spent a lot of time on imperative languages. In this homework, we will switch gears and implement a simple purely functional language with function, product, and sum types. The goal is for you to understand functions that return values, and to get more practice with sum types.

The concrete syntax we will use is

```
T ::= X | unit | (T1 * T2) | (T1 + T2) | (T1 -> T2)

e ::= x
    | let x be e1 in e2 end
    | <>
    | <e1,e2>
    | first e
    | second e
    | left[T] e
    | right[T] e
    | case e { left x => e1 | right y => e2}
    | lam x : T. e
    | e1 @ e2
```

with parentheses used for grouping. As in SML, product types $T1 \times T2$ are ASCII-ified as `T1 * T2`. Function application $e_1(e_2)$ is written `e1 @ e2` and is left-associative, so `e1 @ e2 @ e3` means `(e1 @ e2) @ e3`. Pairs are written with angle-brackets `<e1 , e2>` so that parentheses can be used for grouping. The `<>` (with abstract syntax Triv()) is the one value of the unit type. The constructors for the sum type, `left` and `right`, and the function value `lam x : T.e` are annotated with a type to make it easy to infer a type from an expression.

In addition to product, sum, and higher-order function types, we allow any string to be treated as a unknown base type with no known values; these are useful for writing generic/polymorphic examples. For example, `lam x:X.x` is the identity function on an unknown type `X`.

Task 1 (34 %). In `check.sml`, implement a type checker corresponding to the rules we've covered in Lectures 11, 13, and 15:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{Let}(e_1, x.e_2) : \tau_2}$$

$$\frac{}{\Gamma \vdash \mathsf{Triv}() : \mathsf{unit}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{Pair}(e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{First}(e) : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{Second}(e) : \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{Left}(\tau_2, e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{Right}(\tau_1, e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau \quad \Gamma, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{Case}(e, x.e_1, y.e_2) : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathsf{Lam}(\tau_1, x.e) : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{App}(e_1, e_2) : \tau}$$

Task 2 (33 %). In `step.sml`, implement an operational semantics corresponding to the rules we've cov-

ered in Lectures 11, 13, and 15:

$$\frac{e_1 \mapsto e_1'}{\mathsf{Let}(e_1, x.e_2) \mapsto \mathsf{Let}(e_1', x.e_2)}\ \texttt{Let-search} \qquad \frac{e_1\ \mathsf{done}}{\mathsf{Let}(e_1, x.e_2) \mapsto e_2[e_1/x]}\ \texttt{Let-instr}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{Pair}(e_1, e_2) \mapsto \mathsf{Pair}(e_1', e_2)}\ \texttt{pair-search-1} \qquad \frac{e_2 \mapsto e_2' \quad e_1\ \mathsf{done}}{\mathsf{Pair}(e_1, e_2) \mapsto \mathsf{Pair}(e_1, e_2')}\ \texttt{pair-search-2}$$

$$\frac{e_1\ \mathsf{done} \quad e_2\ \mathsf{done}}{\mathsf{Pair}(e_1, e_2)\ \mathsf{done}}\ \texttt{pair-done}$$

$$\frac{e \mapsto e'}{\mathsf{First}(e) \mapsto \mathsf{First}(e')}\ \texttt{first-search} \qquad \frac{\mathsf{Pair}(e_1, e_2)\ \mathsf{done}}{\mathsf{First}(\mathsf{Pair}(e_1, e_2)) \mapsto e_1}\ \texttt{first-instr}$$

$$\frac{e \mapsto e'}{\mathsf{Second}(e) \mapsto \mathsf{Second}(e')}\ \texttt{second-search} \qquad \frac{\mathsf{Pair}(e_1, e_2)\ \mathsf{done}}{\mathsf{Second}(\mathsf{Pair}(e_1, e_2)) \mapsto e_2}\ \texttt{second-instr}$$

$$\frac{e \mapsto e'}{\mathsf{Left}(\tau_2, e) \mapsto \mathsf{Left}(\tau_2, e')}\ \texttt{left-search} \qquad \frac{e\ \mathsf{done}}{\mathsf{Left}(\tau_2, e)\ \mathsf{done}}\ \texttt{left-done}$$

$$\frac{e \mapsto e'}{\mathsf{Right}(\tau_1, e) \mapsto \mathsf{Right}(\tau_1, e')}\ \texttt{right-search} \qquad \frac{e\ \mathsf{done}}{\mathsf{Right}(\tau_1, e)\ \mathsf{done}}\ \texttt{right-done}$$

$$\frac{e \mapsto e'}{\mathsf{Case}(e, x.e_1, y.e_2) \mapsto \mathsf{Case}(e', x.e_1, y.e_2)}\ \texttt{case-search}$$

$$\frac{\mathsf{Left}(\tau, e)\ \mathsf{done}}{\mathsf{Case}(\mathsf{Left}(\tau, e), x.e_1, y.e_2) \mapsto e_1[e/x]}\ \texttt{case-instr-left}$$

$$\frac{\mathsf{Right}(\tau, e)\ \mathsf{done}}{\mathsf{Case}(\mathsf{Right}(\tau, e), x.e_1, y.e_2) \mapsto e_2[e/y]}\ \texttt{case-instr-right}$$

$$\frac{}{\mathsf{Lam}(\tau, x.e)\ \mathsf{done}}\ \texttt{lam-done}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{App}(e_1, e_2) \mapsto \mathsf{App}(e_1', e_2)}\ \texttt{app-search-1} \qquad \frac{e_2 \mapsto e_2' \quad e_1\ \mathsf{done}}{\mathsf{App}(e_1, e_2) \mapsto \mathsf{App}(e_1, e_2')}\ \texttt{app-search-2}$$

$$\frac{e_2\ \mathsf{done}}{\mathsf{App}(\mathsf{Lam}(\tau, x.e), e_2) \mapsto e[e_2/x]}\ \texttt{app-instr}$$

Task 3 (33 %). In `properties.exp`, write functions of the following types

- Left-association of products `(X * (Y * Z)) -> ((X * Y) * Z)`
- Distribution of products over sums `(X * (Y + Z)) -> ((X * Y) + (X * Z))`
- Composition of functions `(Y -> Z) -> ((X -> Y) -> (X -> Z))`

- Distribution of functions over products `(X -> (Y * Z)) -> ((X -> Y) * (X -> Z))`
- Functions out of sums `((X -> Z) * (Y -> Z)) -> ((X + Y) -> Z)`

There are a couple of examples (identity function, commutativity of products, commutativity of sums) in the file to use as a model.

Testing

The testing harness is slightly different from previous homeworks. A `.exp` file contains a sequence of expressions, with a `;` after each expression. There are no top-level declarations, so each expression must be completely self-contained.

- `Top.file_print(filename)` prints the parsed versions of every expression in the file.
- `Top.file_type(filename)` shows the type (or any type errors) of every expression in the file. Use this to test your type checker.
- `Top.file_step(filename)` and `Top.file_eval(filename)` show each step / the final value of every expression in the file, **type checking each intermediate expression**. As soon as any one expression has a type or step error, the rest of the file is not run.
- `Top.file_eval_no_typechecker(filenmae)` and `Top.file_step(filename)` are like the previous two, except they do not type check at all; use these if you want to test your `progress` function before writing `check_exp`.

Some tests are in `examples.exp`. You should write additional tests as needed to convince yourself that your type checker and operational semantics are correct.