Introduction

In this assignment, we will implement System F, the polymorphic lambda calculus. We will demonstrate that we can implement sums, products, and natural numbers in this calculus by elaborating programs written in Gödel’s T into System F. This elaboration will be typed, meaning we will have (1) a translation from types in T to types in F and (2) a translation of terms which preserves type with respect to the type translation.

Submission

We will collect exactly the following files from the /afs/andrew/course/15/312/ directory:

```
handin/<yourandrewid>/assn5/assn5.pdf
handin/<yourandrewid>/assn5/elaborator.sml
handin/<yourandrewid>/assn5/typechecker.sml
handin/<yourandrewid>/assn5/dynamics.sml
```

Make sure that your files have the right names (especially assn5.pdf!) and are in the correct directories.

1 System F

In this section we will develop the semantics of System F, as a target for the elaboration in the next section. As programmers, we often want to define function which can act at any type; for example, we may want an identity function which can act as \( \tau \to \tau \) for any type \( \tau \). In Gödel’s T or PCF, this function is undefinable, and we must define a new identity function for each type. With the untyped lambda calculus, we have the ability to make such definitions – we can define the identity function as \( \lambda x.x \) – but at the cost of losing all type information in our language.

System F, also known as the polymorphic lambda calculus, allows us to define such functions while still retaining type information by introducing universal types \( \forall \alpha.\tau \). An element of \( \forall \alpha.\tau \), where \( \alpha \) may appear in \( \tau \), can behave as a term of type \( [\sigma/\alpha]\tau \) for any type \( \sigma \). For example, we can implement the identity function as an element of \( \forall \alpha.\alpha \to \alpha \). Elements of \( \forall \alpha.\tau \) behave much like functions, but which take types as arguments instead of terms: the introductory form \( \Lambda(\alpha)e \) is type abstraction, and the elimination form \( e[\tau] \) is type application.

The syntax of System F follows. You will implement the statics and dynamics for lazy System F, as defined in Appendix B. We are implementing the lazy versions of T and F because the translation is simpler, and because we will be able to prove a strong correctness result.
Task 1.1 (14%). Implement the structure `TypeChecker` satisfying the signature `TYPECHECKER`.

Task 1.2 (10%). Implement the structure `Dynamics` satisfying the signature `DYNAMICS`.

2 Typed Elaboration

Much like the untyped lambda calculus, System F is powerfully expressive for its size. We can encode products, sums, and natural numbers using similar techniques to our previous Church encodings. For example, we can encode the product \( \tau_1 \times \tau_2 \) as \( \forall \alpha. (\tau_1 \rightarrow \tau_2 \rightarrow \alpha) \rightarrow \alpha \) and the sum \( \tau_1 + \tau_2 \) as \( \forall \alpha. (\tau_1 \rightarrow \alpha) \rightarrow (\tau_2 \rightarrow \alpha) \rightarrow \alpha \). In general, we encode a type as the data necessary to eliminate from it.

In fact, all of the constructs we defined in Gödel’s T are definable using only System F. We can make this precise by defining an elaboration from T into F. Recall the syntax of T (with sums and products):

Note that, in order to simplify the translation, `nat` uses a recursor `iter` which does not supply the predecessor in a recursive call.
For our elaboration, each type in Gödel’s T will correspond to a particular type in System F. We define a judgment \( \tau \rightsquigarrow \tau' \) for the elaboration of types:

\[
\begin{align*}
\text{nat} & \rightsquigarrow \forall \alpha. \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \quad \text{(T0)} \\
\tau_1 & \rightsquigarrow \tau_1' \quad \tau_2 & \rightsquigarrow \tau_2' \quad \text{(T1)} \\
\tau_1 \times \tau_2 & \rightsquigarrow \forall \alpha. (\tau_1' \rightarrow \tau_2' \rightarrow \alpha) \rightarrow \alpha \quad \text{(T3)} \\
\text{void} & \rightsquigarrow \forall \alpha. \alpha \quad \text{(T4)} \\
\tau_1 + \tau_2 & \rightsquigarrow \forall \alpha. (\tau_1' \rightarrow \alpha) \rightarrow (\tau_2' \rightarrow \alpha) \rightarrow \alpha \quad \text{(T5)}
\end{align*}
\]

Now we give a judgment \( \Gamma \vdash e : \tau \rightsquigarrow \tau' \) for elaboration of expressions; this judgment expresses that \( \Gamma \vdash e : \tau \) in Gödel’s T and \( e \) elaborates to \( \tau' \) in System F.

\[
\begin{align*}
\Gamma \vdash z : \text{nat} & \rightsquigarrow \Lambda(\alpha) \lambda(z : \alpha) \lambda(s : \alpha \rightarrow \alpha) z \quad \text{(E0)} \\
\Gamma \vdash e : \text{nat} & \rightsquigarrow e' \quad \text{(E1)} \\
\Gamma \vdash e_0 : \tau & \rightsquigarrow \epsilon_0 \quad \Gamma, y : \tau \vdash e_1 : \tau & \rightsquigarrow \epsilon_1' \quad \Gamma \vdash \text{iter} e \{ z \Rightarrow e_0 \mid y \Rightarrow e_1 \} : \tau \rightarrow \epsilon'(\tau)e_0'(\lambda(y : \tau)e_1') \quad \text{(E2)} \\
\Gamma \vdash \text{fn}(x : \tau_1)e : (\tau_1 \rightarrow \tau_2) & \rightsquigarrow \lambda(x : \tau_1') e' \quad \Gamma \vdash e_1 : (\tau_2 \rightarrow \tau_1) & \rightsquigarrow \epsilon_1' \quad \Gamma \vdash e_2 : \tau_2 & \rightsquigarrow \epsilon_2' \quad \Gamma \vdash e_1 e_2 : \tau_1 & \rightsquigarrow \epsilon_1' \epsilon_2' \quad \text{(E3)} \\
\Gamma \vdash \langle \rangle : \text{unit} & \rightsquigarrow \Lambda(\alpha) \lambda(x : \alpha).x \quad \text{(E5)} \\
\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 & \rightsquigarrow \Lambda(\alpha) \lambda(f : \tau_1' \rightarrow \tau_2' \rightarrow \alpha). f e_1 e_2' \quad \text{(E6)} \\
\Gamma \vdash e_1 : \tau_1 & \rightsquigarrow \epsilon_1' \quad \Gamma \vdash e_2 : \tau_2 & \rightsquigarrow \epsilon_2' \quad \Gamma \vdash e : \tau_1 \times \tau_2 & \rightsquigarrow e' \quad \Gamma \vdash e : \tau & \rightarrow \epsilon'(\tau') \quad \Gamma \vdash \text{abort} \epsilon(\tau)(e) : \tau & \rightsquigarrow e'(\tau') \quad \text{(E7)} \\
\Gamma \vdash e : \tau_1 \times \tau_2 & \rightsquigarrow e'[\tau_1](\lambda(x : \tau_1') \lambda(y : \tau_2') y) \quad \text{(E8)} \\
\Gamma \vdash \text{inl} \langle \tau_1, \tau_2 \rangle e : \tau_1 + \tau_2 & \rightsquigarrow \Lambda(\alpha) \lambda(f_1 : \tau_1' \rightarrow \alpha). \lambda(f_2 : \tau_2' \rightarrow \alpha). f_1 e' \quad \text{(E9)} \\
\Gamma \vdash \text{inr} \langle \tau_1, \tau_2 \rangle e : \tau_1 + \tau_2 & \rightsquigarrow \Lambda(\alpha) \lambda(f_1 : \tau_1' \rightarrow \alpha). \lambda(f_2 : \tau_2' \rightarrow \alpha). f_2 e' \quad \text{(E10)} \\
\end{align*}
\]
It can be difficult to test a System F interpreter, since checking equality of functions is hard in general. For example, although parametricity guarantees that every element of \(\forall \alpha. \alpha \to \alpha\) is contextually equivalent to \(\Lambda(\alpha)(\lambda(x: \alpha)x)\), not all are \(\alpha\)-equivalent. Luckily, Theorem [1] gives us a useful fact: if \(\emptyset \vdash e : \text{unit + unit} \leadsto e'\), then \(e'\) reduces exactly (up to \(\alpha\)-equivalence) to either the translation of \(\text{inl}[\text{unit + unit}](\langle \rangle)\) or \(\text{inr}[\text{unit + unit}](\langle \rangle)\). We recommend you take advantage of this for testing.
3 Parametricity

In class we defined logical equivalence (\(\sim\)) and contextual equivalence (\(\simeq\)) for System F. We then stated Reynolds’s abstraction theorem, which says that logical equivalence is reflexive. As a corollary, we claimed (without proof) that logical equivalence and contextual equivalence are the same. We then showed how to apply these results to prove that two implementations of an abstract data type were contextually equivalent. In recitation, we used Reynolds’s theorem to prove “free theorems” about programs just by looking at their types. In this section you will practice applying these ideas.

For this section, assume we are working with System F extended with base types \(\text{bool}\) and \(\text{nat}\). You can assume that all of the standard constants and operations for these types such as \(\text{tt}\), \(\text{ff}\), \(\text{ifz}\), \(\text{ifb}\), etc. are available.

Task 3.1 (12%). For each of the claims below, state whether it is true or not. If not, give a counter-example.

1. If \(f : \forall \beta. (\text{bool} \rightarrow \beta) \rightarrow \beta\) then either
   - (a) for all \(g\), \(f [\text{nat}] g \simeq g \text{ff} : \text{nat}\), or
   - (b) for all \(g\), \(f [\text{nat}] g \simeq g \text{tt} : \text{nat}\).

2. If \(f : \forall \alpha. \forall \beta. (\alpha \rightarrow \beta \rightarrow \text{bool}) \rightarrow (\beta \rightarrow \alpha \rightarrow \text{bool})\) then for all \(g, f [\tau_1] [\tau_2] g b a \simeq g a b : \text{bool}\).

3. If \(f : \forall \beta. (\text{bool} \rightarrow \beta) \rightarrow (\beta \rightarrow \beta \rightarrow \text{bool}) \rightarrow \beta\) then either
   - (a) for all \(g\) and \(h\), \(f [\text{nat}] g h \simeq g \text{ff} : \text{nat}\), or
   - (b) for all \(g\) and \(h\), \(f [\text{nat}] g h \simeq g \text{tt} : \text{nat}\).

4. If \(f : \forall \beta. \beta \rightarrow (\beta \rightarrow \beta) \rightarrow \beta\) then there exists \(n\) such that for all \(g\) and \(x\), \(f [\text{nat}] x g \simeq g^n x : \text{nat}\), where \(g^0 = \lambda x : \text{nat}. x\) and \(g^{n+1} = \lambda x : \text{nat}. g (g^n x)\).

Task 3.2 (10%). Recall that in order to show that two implementations of some abstract datatype, say \(\text{pack} \tau_1\) with \(e_1\) as \(\exists \alpha. \tau\) and \(\text{pack} \tau_2\) with \(e_2\) as \(\exists \alpha. \tau\) were contextually equivalent, it suffices to construct some relation \(R\) between terms of type \(\tau_1\) and \(\tau_2\) such that:

1. \(R\) is closed under equivalence, and
2. \(e_1 \sim e_2 : [R/\alpha]\tau\)

The second condition guarantees that all of the operations in the interface of the abstract datatype must preserve the relation \(R\).

Below, we give two implementations of integral counters as an abstract datatype. An integral counter is a counter that stores a (possibly negative) integer with operations that allow you to increment, decrement, negate, and check whether the counter is \(\geq 0\).

In the first implementation, we represent the value stored in the counter as elements of the type \(\text{bool} \times \text{nat}\), where \((b, n)\) represents \(n\) if \(b = \text{tt}\), and \(-n\) if \(b = \text{ff}\). In the second, we use elements of the type \(\text{nat} \times \text{nat}\), where \((n_1, n_2)\) represents the integer \(n_1 - n_2\). We assume we have a function \(\text{gte}\) : \(\text{nat} \rightarrow \text{nat}\).
nat → bool which returns true if its first argument is greater than or equal to its second argument. Note that ifb(e, e_f, e_t) steps to e_f if e = ff and e_t if e = tt.

pack bool × nat with
⟨tt, 0⟩,
λx : bool × nat. ifb(x · l, ifz(x · r, ⟨tt, 1⟩), n. ⟨ff, n⟩), ⟨tt, s(x · r)⟩),
λx : bool × nat. ifb(x · l, ⟨ff, s(x · r)⟩), ifz(x · r, ⟨ff, 1⟩), n. ⟨tt, n⟩)),
λx : bool × nat. not(x · l), x · r),
λx : bool × nat. ifz(x · r, tt, x · l))
as ∃α. α × (α → α) × (α → α) × (α → α) × (α → bool)

pack nat × nat with
⟨0, 0⟩,
λx : nat × nat. s(x · l), x · r),
λx : nat × nat. x · l, s(x · r)),
λx : nat × nat. x · r, x · l),
λx : bool × nat. gte(x · l)(x · r))
as ∃α. α × (α → α) × (α → α) × (α → α) × (α → bool)

Give a relation R on bool × nat and nat × nat that has properties (1) and (2) from above. You do not need to prove that your relation has these properties.
A Putting The Code Together

You can compile your files using `CM.make "sources.cm"`

A.1 Interpreter

As usual, to run the interpreter, execute `TopLevel.repl();`. The TopLevel takes a term from Gödel’s T, uses your elaborator to convert it to a System F term, and then either steps or evaluates it.

The syntax for each term construct is as close as possible to the concrete syntax mentioned for it. This is the second column in the table in which introduce the syntax for a language. We provide below the grammar that the interpreter accepts, as well as a sample session of the interpreter. We also list the reserved words of the language, which are a strict super set of those actually used by the language because we will use the same parser for multiple languages.

directive ::=  
  step <exp>;  
  | step;  
  | eval <exp>;  
  | eval;  
  | load <exp>;  
  | use <filename>;

ty ::= nat | <ty> -> <ty> | unit | <ty> * <ty> | void | <ty> + <ty>

exp ::=  
  <ident>  
  | z  
  | s <exp>  
  | iter <exp> { z => <exp> | s <ident> => <exp> }  
  | fn (<ident> : <ty>) <exp>  
  | <>  
  | <<exp>, <exp>>  
  | <exp>.l  
  | <exp>.r  
  | abort[<ty>] <exp>  
  | inl[<ty>,<ty>] <exp>  
  | inr[<ty>,<ty>] <exp>  
  | case e { inl <ident> => exp | inr <ident> => exp }  
  | (<exp>)

ident ::= (* a letter followed by any number of letters and numbers *)

reserved words:
abort bnd case cmd dcl do else end fix fn fold if ifnil ifz in inl inr is l let 
list match mu nat num r ref ret s then top unfold unit val void z

Example interpreter session:

- `TopLevel.repl();`
- `->step (fn (m : nat) fn (n : nat) iter n { z => m | s y => s y}) (1) (1);`
- Elaborator: `term : nat """" (((fn (m@22 : (all t@24. (t@24 -> ((t@24 -> t@24) ..."
A.2 Testing

It is somewhat difficult to read the output of the elaborator. Another way to test your code is by using the default tests in tests.sml. You should also consider adding your own. There are two sets of tests, one for the dynamics of System F and one for the elaborator. Running TestHarness.runftests(v), TestHarness.runetests(v), or TestHarness.runalltests(v) (where v is a bool indicating whether you want verbose output), will run the dynamics tests, elaborator tests, or both. This is mostly just a framework set up for you, in tests.sml, with a few simple test cases. You are responsible for handing in a working solution. Although not sufficient, this means handing in a well-tested implementation. You need to come up with test cases to exercise your code. In order to generate a comprehensive suite of tests, you are encouraged to share test cases with your classmates.

A.3 Reference Implementation

We have also included the solution to this assignment as a binary heap image, ref_impl. You can load it into SML by passing in the @SMLload=ref_impl flag. Your solution should behave just like ours.
B System F

B.1 Type Formation

\[
\begin{align*}
\Delta, \tau \vdash \tau \text{ type} & \quad \text{(var)} \\
\Delta \vdash \tau_1 \rightarrow \tau_2 \text{ type} & \quad \text{($\to$-F)} \\
\Delta, \alpha \vdash \tau \text{ type} & \quad \text{($\forall$-F)} \\
\end{align*}
\]

B.2 Statics

\[
\begin{align*}
x : \tau \in \Gamma & \quad \text{(var)} \\
\Delta \Gamma \vdash x : \tau & \quad \text{($\var$)} \\
\Delta \Gamma \vdash \lambda (x : \tau_1) e : \tau_1 \rightarrow \tau_2 & \quad \text{($\to$-I)} \\
\Delta \Gamma \vdash e_1 e_2 : \tau_1 & \quad \text{($\to$-E)} \\
\Delta \Gamma \vdash \Lambda (\alpha) e \vdash \forall \alpha. \tau & \quad \text{($\forall$-I)} \\
\Delta \Gamma \vdash e' : \tau' & \quad \text{($\forall$-E)}
\end{align*}
\]

B.3 Dynamics

Premises and rules marked in brackets are omitted in the lazy dynamics.

\[
\begin{align*}
\lambda (x : \tau) e \text{ val} & \quad \text{(FD}_0\text{)} \\
e_1 \mapsto e_1' & \quad \text{(FD}_1\text{)} \\
e_1 e_2 \mapsto e_1'e_2 & \quad \text{(FD}_2\text{)} \\
\end{align*}
\]

\[
\begin{align*}
\begin{bmatrix} e_1 \text{ val} \\
e_2 \text{ val} \end{bmatrix} & \quad \text{($\lambda (x : \tau_1) e_1) e_2 \mapsto [e_2/x]e_1$ (FD}_3\text{)} \\
\Lambda (\alpha) e \text{ val} & \quad \text{(FD}_4\text{)} \\
e \mapsto e' & \quad \text{(FD}_5\text{)} \\
\end{align*}
\]

\[
\begin{align*}
(\Lambda (\alpha) e)[\tau] & \mapsto [\tau/\alpha]e \quad \text{(FD}_6\text{)}
\end{align*}
\]
C Gődel’s T

C.1 Statics

\[ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad (\text{var}) \]

\[ \frac{\vdash z : \text{nat}}{\Gamma \vdash z : \text{nat}} \quad (\text{nat-I}_1) \]

\[ \frac{\Gamma \vdash e : \text{nat} \quad \Gamma \vdash e_0 : \tau}{\Gamma \vdash \text{iter}(e; e_0; y; e_1) : \tau} \quad (\text{nat-E}) \]

\[ \frac{\Gamma \vdash e_1 : \tau \rightarrow \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \cdot e_2 : \tau_1} \quad (\rightarrow \text{-E}) \]

\[ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad (\times \text{-I}) \]

\[ \frac{\Gamma \vdash e : \text{void}}{\Gamma \vdash \text{abort}[\tau](e) : \tau} \quad (\text{void-E}) \]

\[ \frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash e : \tau_2}{\Gamma \vdash e : \tau_1 \times \tau_2} \quad (\times \text{-E}_1) \]

\[ \frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot 1 : \tau_1} \quad (\times \text{-E}_2) \]

\[ \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{inl} \{\tau_1 + \tau_2\}(e) : \tau_1 + \tau_2} \quad (+\text{-I}_1) \]

\[ \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{inr}[\tau_1 + \tau_2](e) : \tau_1 + \tau_2} \quad (+\text{-I}_2) \]

\[ \frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau \quad \Gamma, x : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{case } e \{\text{inl } x_1 \Rightarrow e_1 \mid \text{inr } x_2 \Rightarrow e_2\} : \tau} \quad (+\text{-E}) \]
C.2  Dynamics

Premises and rules marked in brackets are omitted in the lazy dynamics.

\[
\begin{align*}
\quad & z \text{ val} \quad (\text{TD}_0) & & [e \mapsto e'] \quad (\text{TD}_1) & & [e \text{ val}] \quad (\text{TD}_2) \\
& e \mapsto e' \quad (\text{TD}_3) & & \text{iter}(e; e_0; y. e_1) \mapsto \text{iter}(e'; e_0; y. e_1) \quad (\text{TD}_4) \\
& \text{iter}(s(e); e_0; y. e_1) \mapsto \text{iter}(e; e_0; y. e_1)/y \quad (\text{TD}_5) & & \lambda(x : \tau_1) e \text{ val} \quad (\text{TD}_6) & & e_1 \mapsto e_1' \quad (\text{TD}_7) \\
& \begin{cases} e_1 \text{ val} & e_2 \mapsto e_2' \\
& e_1 e_2 \mapsto e_1 e_2' \end{cases} \quad (\text{TD}_8) & & \begin{cases} e_1 \text{ val} & e_2 \mapsto e_2' \\
& (e_1, e_2) \mapsto (e_1', e_2') \end{cases} \quad (\text{TD}_9) & & \begin{cases} e_1 \text{ val} & e_2 \mapsto e_2' \\
& (e_1, e_2) \mapsto (e_1', e_2') \end{cases} \quad (\text{TD}_{13}) \\
& e \mapsto e' & & \begin{cases} e \text{ val} & [e_1 \text{ val}] \\
& (e_1, e_2) \cdot 1 \mapsto e_1 \end{cases} \quad (\text{TD}_{15}) & & e \mapsto e' \quad (\text{TD}_{16}) \\
& \begin{cases} e_1 \text{ val} & [e_2 \text{ val}] \\
& (e_1, e_2) \cdot r \mapsto e_2 \end{cases} \quad (\text{TD}_{17}) & & e \mapsto e' \quad (\text{TD}_{18}) & & [e \text{ val}] \quad (\text{TD}_{19}) \\
& \text{abort}[\tau](e) \mapsto \text{abort}[\tau](e') \quad (\text{TD}_{20}) & & e \mapsto e' \quad (\text{TD}_{21}) \\
& \begin{cases} e \text{ val} & [e \text{ val}] \\
& \begin{cases} \text{inl}[\tau_1 + \tau_2](e) & \mapsto \text{inl}[\tau_1 + \tau_2](e') \end{cases} \end{cases} \quad (\text{TD}_{22}) \\
& e \mapsto e' \quad (\text{TD}_{23}) \\
& \begin{cases} e \text{ val} \\
& \text{case e } \{ \text{inl } x_1 \mapsto e_1 | \text{inr } x_2 \mapsto e_2 \} \mapsto \text{case e' } \{ \text{inl } x_1 \mapsto e_1 | \text{inr } x_2 \mapsto e_2 \} \end{cases} \quad (\text{TD}_{24}) \\
& e \text{ val} \quad (\text{TD}_{25}) \\
& \begin{cases} e \text{ val} \\
& \text{case inl}[\tau_1 + \tau_2](e) \{ \text{inl } x_1 \mapsto e_1 | \text{inr } x_2 \mapsto e_2 \} \mapsto [e/x] e_1 \end{cases} \quad (\text{TD}_{24}) \\
& e \text{ val} \quad (\text{TD}_{25}) \\
& \begin{cases} e \text{ val} \\
& \text{case inr}[\tau_1 + \tau_2](e) \{ \text{inl } x_1 \mapsto e_1 | \text{inr } x_2 \mapsto e_2 \} \mapsto [e/x] e_2 \end{cases} \quad (\text{TD}_{25})
\end{align*}
\]