1 Strong normalization

Show that all expressions in the language $\lambda^{\rightarrow+\times}$ (without the rec expression) are strongly normalizing by extending the proof of strong normalization for $\lambda^{\rightarrow}$.

2 Recursive types

Once we add recursive types to the simply-typed lambda calculus, we see that the untyped lambda calculus is a subset of the simply-typed lambda calculus in which every variable is declared to have the type $\mu D.D \rightarrow D$. An arbitrary lambda calculus expression can be desugared to the typed lambda calculus with recursive types as follows:

$$[\![\lambda x\ .\ e]\!] = (\mathsf{up}\ (\lambda x : \mu D.D \rightarrow D\ .\ [\![e]\!]))$$

$$[\![e_0\ e_1]\!] = ((\mathsf{down}\ [\![e_0]\!])\ [\![e_1]\!])$$

In the untyped lambda calculus, the rec operator is not necessary because we can write a closed expression Y that has the same effect. Using the desugaring above, we can obtain a version of Y for the typed lambda calculus with recursive types. For this problem, we will that the language has lazy (normal order) evaluation, so either form of the Y operator that we saw earlier in lecture will work.

However, the desugared Y operator is not generally useful in the typed lambda calculus because it takes fixed points only for functions from $\mu D.D \rightarrow D$ to the same type. For a given type τ, we can define an operator Y_τ that takes fixed points over τ. Write the appropriate typed lambda expression for Y_τ. Make sure that your expression can be proved to have the correct type using the typing rules. To make this straightforward, assume that the up and down operators are annotated with the type of the value that they produce, and have the following typing rules:

$$\frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \mathsf{up}^{\mu\alpha.\tau} e : \mu\alpha.\tau} \qquad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \mathsf{down}^{\mu\alpha.\tau} e : \tau[\mu\alpha.\tau/\alpha]}$$

3 Type reconstruction

a. What type scheme will the Hindley-Milner type reconstruction algorithm $\mathcal{W}$ find for the lambda expressions representing Church numerals?

b. Show the steps performed by type reconstruction on the following expression:

```
(let (id (fn x x))
  (let (true (fn x (fn y x)))
    ((id id) (id (true 2 id)))))
```

c. Consider a typed version of the module construct from Homework 3, but without the select or extend operations. In the module expression, each expression e_i may refer to any of the module variables x_i. The type system is extended to include a corresponding new moduleof type.

$$e ::= \ldots \mid (\mathsf{module}\ (x_1\ e_1) \ldots (x_n\ e_n)) \mid (\mathsf{with}\ e_m\ e_v)$$

$$\tau ::= \ldots \mid (\mathsf{moduleof}\ x_1 : \tau_1 \ldots x_n : \tau_n)$$

Write the appropriate typing rules for the new expressions.

d. Algorithm $\mathcal{W}$ cannot be extended to reconstruct types for the extended language including these constructs; however, it can if the with expression is modified to state which variables e_m overrides with new definitions:

$$e ::= \ldots \mid (\mathsf{with}\ (x_1\ \ldots\ x_n)\ e_m\ e_v)$$

The identifiers bound by e_m must be the set $\{x_1, \ldots, x_n\}$; however, their types are not mentioned and must be reconstructed.

Provide an example of an expression using the original with expression that will create difficulties for a type reconstruction algorithm, and show how to extend algorithm $\mathcal{W}$ to reconstruct types for the language with the with expression with explicitly-named identifiers.

We will also accept an answer to the original part (d) that was posed. However, you should characterize when your type reconstruction algorithm works and when it does not.

4 The Power of the Polymorphic Lambda Calculus

We saw how the untyped lambda calculus is able to encode many datatypes such as pairs, booleans, natural numbers, lists, *etc.* In this problem, we will see that parametric polymorphism is powerful enough to encode many of these same same datatypes as well. Thus we consider a restricted language which has only a base type, b, containing a single value (u), functions, and first-class polymorphic types. The grammar for this language, the polymorphic (or second-order) lambda calculus is given below:

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \mathsf{b} \mid \forall\alpha.\tau \mid \tau \to \tau \\ e & ::= & x \mid \mathsf{u} \mid \lambda x\!:\!\tau.\, e \mid (e\ e) \mid \Lambda\alpha.\, e \mid (e\ [\tau]) \end{array}$$

Recall that in the untyped lambda calculus, we encoded the pair $\langle e_1, e_2\rangle$ via the term $\lambda f.\,(f\ e_1\ e_2)$. Thus, we could write the function pair that takes e_1 and e_2 and produces $\langle e_1, e_2\rangle$ like this: $\lambda(x, y, f).\,(f\ x\ y)$. We can't quite use the same trick in the simply-typed lambda calculus without polymorphism because we are forced to write down a type for f, but we don't know what the return type should be. That is, even if e_1 has type τ_1 and e_2 has type τ_2, there is no way to give a type for the "?" below such that term will have the properties needed to encode pairs (why?):

$$\lambda(x\!:\!\tau_1, y\!:\!\tau_2, f\!:\!\tau_1 \to \tau_2 \to ?).\,(f\ x\ y)$$

Polymorphic types, however, give us a reasonable way to give a type for "?". We can encode the type $\tau_1 * \tau_2$ using the polymorphic lambda calculus as follows:

$$\tau_1 * \tau_2 \equiv \forall\gamma.(\tau_1 \to \tau_2 \to \gamma) \to \gamma$$

The introduction form for products can be encoded via a function $\mathsf{pair}^{\tau_1,\tau_2}\!:\!\tau_1 \to \tau_2 \to \tau_1 * \tau_2$ given by:

$$\mathsf{pair}^{\tau_1,\tau_2} \equiv \lambda x\!:\!\tau_1.\,\lambda y\!:\!\tau_2.\,\Lambda\gamma.\,\lambda f\!:\!\tau_1 \to \tau_2 \to \gamma.\,(f\ x\ y)$$

We can similarly encode the elimination forms, $\pi_1^{\tau_1,\tau_2}\!:\!\tau_1 * \tau_2 \to \tau_1$ and $\pi_2^{\tau_1,\tau_2}\!:\!\tau_1 * \tau_2 \to \tau_2$, as follows:

$$\mathsf{pi}_1^{\tau_1,\tau_2} \equiv \lambda p\!:\!\tau_1 * \tau_2.\,(p\ [\tau_1]\ (\lambda x\!:\!\tau_1.\,\lambda y\!:\!\tau_2.\,x))$$

$$\mathsf{pi}_2^{\tau_1,\tau_2} \equiv \lambda p\!:\!\tau_1 * \tau_2.\,(p\ [\tau_2]\ (\lambda x\!:\!\tau_1.\,\lambda y\!:\!\tau_2.\,y))$$

In fact, we can do even better. Since the pair function and the projections pi_1 and pi_2 are parameterized by types τ_1 and τ_2, we can create polymorphic versions that take τ_1 and τ_2 as type arguments. That is, pair now has the type $\forall\alpha.\forall\beta.\alpha \to \beta \to \forall\gamma.(\alpha \to \beta \to \gamma) \to \gamma$.

$$\mathsf{pair} \equiv \Lambda\alpha.\,\Lambda\beta.\,\lambda x\!:\!\alpha.\,\lambda y\!:\!\beta.\,\Lambda\gamma.\,\lambda f\!:\!\alpha \to \beta \to \gamma.\,f\ x\ y$$

$$\mathsf{pi}_1 \equiv \Lambda\alpha.\,\Lambda\beta.\,\lambda p\!:\!\forall\gamma.(\alpha \to \beta \to \gamma) \to \gamma.\,(p\ [\alpha]\ (\lambda x\!:\!\alpha.\,\lambda y\!:\!\beta.\,x))$$

$$\mathsf{pi}_2 \equiv \Lambda\alpha.\,\Lambda\beta.\,\lambda p\!:\!\forall\gamma.(\alpha \to \beta \to \gamma) \to \gamma.\,(p\ [\beta]\ (\lambda x\!:\!\alpha.\,\lambda y\!:\!\beta.\,y))$$

a. Define the type bool using the types available in the polymorphic lambda calculus. Give closed terms true and false such that $\emptyset; \emptyset \vdash \mathsf{true} : \mathsf{bool}$ and $\emptyset; \emptyset \vdash \mathsf{false} : \mathsf{bool}$. Also give a closed term if of type $\forall \alpha.\mathsf{bool} \to \alpha \to \alpha \to \alpha$ such that ($\mathsf{if}\ [\beta]\ \mathsf{true}\ x\ y$) reduces to x and ($\mathsf{if}\ [\beta]\ \mathsf{false}\ x\ y$) reduces to y. (Hint: look at the encoding of booleans in the untyped-lambda calculus.)

b. Just as we can encode product types using just polymorphism and functions, we can do the same thing for sum types. Show how to encode the sum type $\tau_1 + \tau_2$ using this language. Give translations for polymorphic versions of the operators inl, inr, and case.

c. We know that the untyped lambda calculus has divergent terms such as $((\lambda x.\, x\ x)\ (\lambda x.\, x\ x))$. It is a theorem (beyond the scope of this course) that programs written in the polymorphic lambda calculus studied here are strongly normalizing, and so we have lost some expressiveness by moving to a typed setting. Nevertheless, it is natural to wonder how "close" to the untyped lambda calculus we can get. In this problem, we will see that any primitive recursive function from the natural numbers to the natural numbers can be computed in the polymorphic lambda calculus.

   The type nat of natural numbers represented as Church numerals can be defined as in Problem 3. With explicit parameterization over types, the expressions for 0 and succ are as follows:

$$\mathsf{0} \equiv \Lambda\alpha.\, \lambda f\!:\!\alpha \to \alpha.\, \lambda x\!:\!\alpha.\, x$$

$$\mathsf{succ} \equiv \lambda n\!:\!\mathsf{nat}.\, \Lambda\alpha.\, \lambda f\!:\!\alpha \to \alpha.\, \lambda x\!:\!\alpha.\, f\ (n\ [\alpha]\ f\ x)$$

   We wish to define a recursion operator, R, of the type $\mathsf{nat} \to (\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat} \to \mathsf{nat}$ with the property that $(\mathsf{R}\ n\ f\ \mathsf{0})$ reduces to n, and $(\mathsf{R}\ n\ f\ (\mathsf{succ}\ m))$ reduces to $(f\ (R\ n\ f\ m)\ (\mathsf{succ}\ m))$.

   As an example of how to use the R operator, assume that the terms $\mathsf{1} : \mathsf{nat}$ and $\mathsf{times} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ are defined appropriately. Then the factorial function can be encoded as: $\lambda x\!:\!\mathsf{nat}.\, (\mathsf{R}\ \mathsf{1}\ \mathsf{times}\ x)$.

   Give a closed, polymorphic lambda calculus term for R as described above. (Hint: You may find it helpful to create an auxiliary function $\mathsf{step} : (\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} * \mathsf{nat}) \to (\mathsf{nat} * \mathsf{nat})$ that takes a function f and maps the pair $\langle(\mathsf{R}\ n\ f\ m), m\rangle$ to the pair $\langle(\mathsf{R}\ n\ f\ (\mathsf{succ}\ m)), (\mathsf{succ}\ m)\rangle$. Feel free to use the terms pair and pi_i.)