1 Subtype and parametric polymorphism

Subtype and parametric polymorphism can co-exist in a programming language. Suppose we extend the polymorphic lambda calculus with the notion of subtyping. We already have all the necessary typing rules except that we need a rule to relate polymorphic types.

a. Define an expressive subtyping rule for the types $\forall\alpha.\tau$ and $\forall\alpha'.\tau'$. The rule should not require that the types be equivalent; however, if the subtype relation holds in both directions, the rule should enforce equivalence of the two types as defined in class.

b. Provide the corresponding coercion function $\Theta(\forall\alpha.\tau \leq \forall\alpha'.\tau')$.

2 Soundness with subtypes

Suppose we take the simply-typed lambda calculus with the ground types int, bool, and 1 (unit), and augment it with all of the subtype relations and the subsumption rule given in Lecture 35. We then have subtype relations $\mathsf{int} \leq 1$, $\mathsf{bool} \leq 1$, and $\tau \to \tau' \leq 1$ for all types τ, as well as all the subtype relations derived using the rule for subtyping on function types.

This language is more permissive than the simply-typed lambda calculus; for example, we can give the expression $(\lambda \mathsf{x}.\, (\mathsf{x}\ \mathsf{x}))$ the type $(1 \to 1) \to 1$ by typing the argument x suitably: $(\lambda \mathsf{x} : 1 \to 1.\, (\mathsf{x}\ \mathsf{x}))$. This typing relies on the fact that $(1 \to 1) \leq 1$. Recall that before we introduced subtyping, we had to introduce recursive types to describe the type of this expression. Now, we would like to prove that this type system is sound.

a. Is this subtype relation a partial order or merely a pre-order?

b. We would like to prove soundness by using induction on the size of the type derivation. One difficulty with this approach is that a given expression may have many possible type derivations. Show that for every expression we can construct a canonical type derivation in which every other step in the derivation is a use of the subsumption rule. (This use of subsumption corresponds to type-checking implicitly inserted applications of suitable coercion functions.)

c. We can demonstrate that this type system is sound, which can be proved by demonstrating the usual preservation and progress lemmas. Show that the preservation lemma holds, assuming a substitution lemma analogous to that from Homework 4, problem 1(b):

$$\Gamma[x \mapsto \tau_2'] \vdash e_1 : \tau_1' \wedge \Gamma \vdash e_2 : \tau_2 \wedge \tau_2 \leq \tau_2' \wedge \tau_1' \leq \tau_1 \quad \Rightarrow \quad \Gamma \vdash e_1[e_2/x] : \tau_1$$

d. By adapting the proof from Homework 4, show that this substitution lemma holds.

3 Implementation

We saw in class how the Curry-Howard isomorphism allows us to treat types as propositions in logic. The type 0 corresponds to the logical proposition false, and we claimed that a continuation which accepts a value of type α can be given the type $\alpha \to 0$. In this problem, we explore the duality between sums and products by seeing how DeMorgan's laws let us encode sum using products and continuations, and products using sums and continuations.

In SML, the module SMLofNJ.Cont provides functions for manipulating continuations as first-class values. The type of α-accepting continuations is given by α cont (which you should think of as $\alpha \to 0$). Two functions for manipulating continuations are: $\mathsf{callcc} : (\alpha\ \mathsf{cont} \to \alpha) \to \alpha$ and $\mathsf{throw} : \alpha\ \mathsf{cont} \to \alpha \to \beta$. The expression callcc(fn k => e) evaluates e with k bound to the continuation of the entire callcc expression. Inside of e, you can use throw k v to invoke the continuation k on the value v.

The file hw6.sml contains the code needed for this assignment; mail your completed version to zdance@cs.cornell.edu.

1. The signature for an ML module implementing product types is:

```
signature PRODUCT = sig
   (* Datatype *)
   type ('a, 'b) product
   (* Elimination *)
   val pi1 : ('a, 'b) product -> 'a
   val pi2 : ('a, 'b) product -> 'b
   (* Introduction *)
   val pair : ('c -> 'a) -> ('c -> 'b) -> ('c -> ('a, 'b) product)
end
```

(We've made the type of pair take functions from 'c to 'a and 'b to make the duality between products and sums more explicit. . . see how the types are just "backwards" from those in the signature for SUM below.) Implement the structure product with the above signature using only SML's (non-recursive) datatype mechanism (for sums), functions, and continutations. (The type you define is not allowed to use *, and the code you write can't use tupling or #1, #2, etc. in any form.) Hint: Apply DeMorgan's law to the type 'a * 'b.

2. Dually, we can implement sums using products and continuations. The appropriate signature is:

```
signature SUM = sig
   (* Datatype *)
   type ('a, 'b) sum
   (* Introduction *)
   val inl : 'a -> ('a, 'b) sum
   val inr : 'b -> ('a, 'b) sum
   (* Elimination *)
   val mycase : ('a -> 'c) -> ('b -> 'c) ->  ('a, 'b) sum -> 'c
end
```

Implement the structure sum with the above signature. The type you define cannot mention the SML keyword datatype, nor can you use case or (non-trivial) pattern-matching in the implementation. (In SML, the projection functions, π_1 and π_2, are written #1 and #2.)