CS520 Problems for Assignment 11 (Part I)

Posted: 3 December 2004
Due: 13 December 2004

Throughout this problem set, use sums and sum types according to their presentation in Handout 24. In particular, we want you to use "`case`", "`inl`" and "`inr`" as prim ops not as constructors (in contrast to their presentation in [Pierce]).

Problem 1: We want you to show that in the presence of "`Unit`" and sum types, we can eliminate "`Bool`" as a base type (and therefore "`true`", "`false`" and "`if - then - else`" as primitives). For this, we first make "`Bool`" syntactic sugar, i.e., we define it using the remaining primitives (in the syntax of types). Specifically, we define

$$\texttt{Bool} \triangleq \texttt{Unit} + \texttt{Unit}$$

1. Given the preceding definition of "`Bool`", define appropriately "`true`" and "`false`" using other primitives (in the syntax of terms).
2. Given the preceding definition of "`Bool`", "`true`" and "`false`", define appropriately "`if - then - else`" using other primitives (in the syntax of terms).
3. Verify your definitions, i.e., show that for arbitrary terms t_1 and t_2:

   $$\texttt{if true then } t_1 \texttt{ else } t_2 \;\Rightarrow^*\; t_1$$
   $$\texttt{if false then } t_1 \texttt{ else } t_2 \;\Rightarrow^*\; t_2$$

   after "`if - then - else`", "`true`" and "`false`" are replaced by their definitions.

Problem 2: We want you to show that, in the presence of sum types, we can take *variants* and *variant types* as sugar (rather than as primitive features of the programming language, according to [Pierce]). Moreover, we want to handle variants with appropriate generalizations of "`case`", "`inl`" and "`inr`" as operators (not as constructors) in the style of Handout 24. Intuitively, an element of the variant type $< \ell_1 : T_1, \ldots, \ell_n : T_n >$ is an element of one of the types T_i for some $1 \leqslant i \leqslant n$, tagged with label ℓ_i. More precisely, if $t_i : T_i$ then the expression

$$\ell_i^{<\ell_1 : T_1, \ldots, \ell_n : T_n>} \; t_i$$

is a term of type $< \ell_1 : T_1, \ldots, \ell_n : T_n >$. The corresponding "`case`" operator is

$$\texttt{case}^{<\ell_1 : T_1, \ldots, \ell_n : T_n>, U} \;:$$
$$< \ell_1 : T_1, \ldots, \ell_n : T_n > \to (T_1 \to U) \to (T_2 \to U) \to \cdots \to (T_n \to U) \to U$$

As in Handout 24, we omit the type superscripts whenever convenient without ambiguity. The corresponding evaluation rule is:

$$(\texttt{case}^{<\ell_1:T_1,\ldots,\ell_n:T_n>}\ \ (\ell_i\, v)\ \ t_1 \cdots t_n)\ \Rightarrow\ (t_i\ \ v)$$

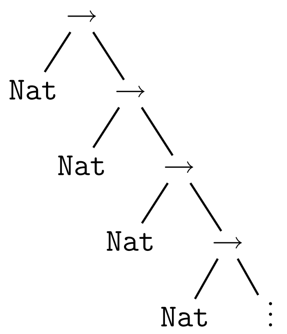
Show carefully how to treat variants as syntactic sugar using sums.

Problem 3: This continues the preceding problem. We want you to show that, in the presence of variant types (and, therefore, in the presence of sum types according to the result of Problem 2), we can take *enumeration types* as sugar. We want to write an enumeration in the form $(\ell_1, \ldots, \ell_n)$ where the literals $\ell_1, \ldots, \ell_n$ are the elements of the enumeration and are considered distinct values not equal to any values of any other type. Show how to regard the enumeration $(\ell_1, \ldots, \ell_n)$ as syntactic sugar for the variant $< \ell_1 : \texttt{Unit}, \ldots, \ell_n : \texttt{Unit} >$ and illustrate your technique by showing how to translate a term containing enumerations and literals into a term with "`Unit`" and variants.

Problem 4: In lecture we explained how a recursive type is a finite representation of an infinite type (without recursion). For example, $\mu\texttt{X}.\texttt{Nat} \to \texttt{X}$ is a finite representation of the infinite type:

$$\texttt{Nat} \to \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat} \to \cdots$$

The parse tree of which is:



Show carefully that, if we restrict attention to the pure lambda-calculus (i.e., the language built up from term variables, applications and abstractions only), then ***every*** term is typable in a system of recursive types.