Comp 311
Principles of Programming Languages
Lecture 23
Extending the Typed Lambda Calculus

Corky Cartwright

November 1, 2010

Accomodating Standard Ground Types

Any realistic statically typed function language includes ground types like `bool` and `int` (and many more such as `char` and `float`).

Hence, the definition of types (type expressions) τ looks like:

$$\tau \; ::= \; \texttt{int} \mid \texttt{bool} \mid \ldots \mid \tau \to \tau$$

and the base environment (which is empty in the simply typed lambda calculus) contains types for all of the primitive functions and operators. (We interpret operators as abbreviated syntax for conventional function applications.)

How Do We Type Functional Constructs?

Examples:

Conditional expressions

New algebraic (inductively defined) data types

Recursive let

General answer:

(i) for each new syntactic construct, we introduce a new rule;

(ii) for each new form of data (which only requires adding new contants, [including functions] to the languge), we simply augment the set of type constructors by the new type constructor (*e.g*. int-list) and augment the contents of the base type environment by the new constants and their types.

Let's look at each example in our list above.

Typing Conditional Expressions

Note: if conditional expressions are simply written as applications of a ternary `if` operator, then all we need to do is add the type of `if` to the base type environment (assuming our typing framework is polymorphic which we will explain later). In fact, our new rule for the `if` construct simply codifies the same typing constraints.

Our conditional expressions presume the existence of a bool type

$$\frac{\Gamma \mid \texttt{B:bool};\ \Gamma \mid \texttt{M}:\tau;\ \Gamma \mid \texttt{N}:\tau}{\Gamma \mid \texttt{if B then M else N}:\tau}$$

Typing New Forms of Data

Assume we define some new form of data in a program. In structurally typed programming languages, all data values have a unique type. Hence, when a new union type is introduced, all values of that type must be disjoint from all existing types. This invariant is maintained by forcing all of the components of a new union type to be tagged. This data construction is called a discriminated union.

Example: binary trees

```
 BT :: = leaf(int) | make-BT(BT, BT)
```

In ML-like languages, new forms of data are introduced in `datatype` declarations which have an abstract syntax similar to our example. (In ML, the names of accessors (selectors) are implicit because pattern-matching notation is used to extract the fields of constructed data objects.

Typing New Forms of Data cont.

Given

```
BT :: = make-leaf(int) | make-BT(BT, BT)
```

we augment the set of type expressions by the type `BT` and the base typing environment by the declarations:

```
make-leaf: int → BT
make-BT: BT x BT → BT
leaf-1: BT → int
BT-1: BT → BT
BT-2: BT → BT
```

assuming that we use the name `leaf-1` for the accessor for `make-leaf` and the names `BT-1` and `BT-2` for the accessors for `make-BT`.

In ML-like languages, data type definitions are typically lexically scoped so `datatype` statements have an abstract syntax like `let` with an explicit body which is the scope of the definition. We will ignore scoping for data definitions and fix the data types for any program that we consider. (In Java, data definitions, do not have scope. Visibility is an administrative notion not a semantic one.)

Typing Let and Recursive Let

The typing rule for let is simply an abbreviation for appropriately combining the abstraction and application rules:

$$\frac{\Gamma \,|\, M : \sigma \;;\quad \Gamma, x{:}\sigma \,|\, N : \tau}{\Gamma \,|\, \texttt{let}\; x{:}\sigma := M \;\texttt{in}\; N : \tau} \qquad \text{(let rule)}$$

The corresponding rule for recursive let:

$$\frac{\Gamma, x{:}\sigma \,|\, M : \sigma \;;\quad \Gamma, x{:}\sigma \,|\, N : \tau}{\Gamma \,|\, \texttt{reclet}\; x{:}\sigma := M \;\texttt{in}\; N : \tau} \qquad \text{(reclet rule)}$$

differs only in one small (but important!) detail.