Syntax of a Typed Lambda Calculus and Variants

This note introduces the syntax of the pure typed lambda calculus, and some variants we will study this term.

1 Foundation Languages

1.1 The Pure Typed Lambda Calculus

The pure typed lambda calculus is the foundation for the following developments. As is standard, the pure typed lambda calculus uses call by name.

$$
\begin{aligned}
\langle \texttt{expr} \rangle &::= \langle \texttt{identifier} \rangle \\
&\mid \texttt{(}\ \langle \texttt{expr} \rangle\ \langle \texttt{expr} \rangle\ \texttt{)} \\
&\mid \langle \texttt{function abstract} \rangle
\end{aligned}
$$

$$\langle \texttt{function abstract} \rangle ::= \texttt{(}\ \lambda\ \langle \texttt{identifier} \rangle\ \texttt{:}\ \ \langle \texttt{type} \rangle\ \texttt{.}\ \ \langle \texttt{expr} \rangle\ \texttt{)}$$

$$
\begin{aligned}
\langle \texttt{type} \rangle &::= \langle \texttt{identifier} \rangle \\
&\mid \texttt{(}\ \langle \texttt{type} \rangle \rightarrow \langle \texttt{type} \rangle\ \texttt{)}
\end{aligned}
$$

There need only be one basic type identifier (e.g., $\texttt{o}$), but we will asssume at times as many as covenient.

1.2 Pairs

Following the "Curry-Howard isomorphism," it is useful to add a cartesian product to the pure typed lambda calculus. Pairs cannot be introduced simply in the environment, since one needs a type for pairs.

$$
\begin{aligned}
\langle \texttt{expr} \rangle &::= \langle\ \langle \texttt{expr} \rangle\ \texttt{,}\ \langle \texttt{expr} \rangle\ \rangle \\
&\mid \pi^1 \mid \pi^2
\end{aligned}
$$

$$\langle \texttt{type} \rangle ::= \langle \texttt{type} \rangle \times \langle \texttt{type} \rangle$$

1.3 Uncurry

The language "Uncurry" is semantically similar to the pure typed lambda calculus with pairs. However, some notational changes are made to make it more "fun" and function abstracts may take several arguments. The new definition of $\langle$expr$\rangle$ is as follows. The definition of $\langle$type$\rangle$ is as before.

```
⟨expr⟩ ::= ⟨identifier⟩
    | ⟨expr⟩ ( ⟨exprs⟩ )
    | ⟨function abstract⟩
    | cons | fst | snd

⟨exprs⟩ ::= ⟨expr list⟩ | ⟨empty⟩
⟨expr list⟩ ::= ⟨exprs⟩ | ⟨expr list⟩ , ⟨expr⟩

⟨empty⟩ ::=

⟨function abstract⟩ ::= ( fun ⟨formals⟩ = ⟨expr⟩ )

⟨formals⟩ ::= ( ⟨decls⟩ ) : ⟨type⟩

⟨decl⟩ ::= ⟨identifier⟩ : ⟨type⟩

⟨decls⟩ ::= ⟨decl list⟩ | ⟨empty⟩
⟨decl list⟩ ::= ⟨decl⟩ | ⟨decl list⟩ , ⟨decl⟩
```

1.4 Recursive Definitions

The langauge "Uncurry+RecDef" adds a primitive **let** that allows one to make recursive defintions (as in ML's **letrec**). The following productions are added to the grammar of "Uncurry."

```
⟨expr⟩ ::= let ⟨def list⟩ in expr
    | ( ⟨expr⟩ )

⟨def⟩ ::= ⟨value def⟩
    | ⟨type def⟩

⟨value def⟩ ::= value ⟨decl⟩ = ⟨expr⟩

⟨type def⟩ ::= type ⟨identifier⟩ = ⟨type⟩

⟨defs⟩ ::= ⟨def list⟩ | ⟨empty⟩
⟨def list⟩ ::= ⟨def⟩ | ⟨def list⟩ ; ⟨def⟩
```

Note that this change is semantic as well as syntactic, as now nonterminating computations can be expressed.

1.5 If Expressions

If one wants to consider a call-by-value variant of "Uncurry+RecDef," one needs a built-in **if**-expression. This language is "Uncurry+RedDef+If" or URDI for short.

⟨expr⟩ ::= **if** ⟨expr⟩ **then** ⟨expr⟩ **else** ⟨expr⟩ **fi**

2 Subtypes

The language "URDI+Subtypes" adds the following declaration to one to declare one type as a subtype of another.

⟨type def⟩ ::= **type** ⟨identifier⟩ **subtype of** ⟨type⟩

⟨type list⟩ ::= ⟨type⟩ | ⟨type list⟩ , ⟨type⟩

3 Type Constructions

⟨type⟩ ::= **data** ⟨identifier⟩ ⟨signature⟩ ⟨axioms⟩

⟨signature⟩ ::= **is** ⟨decl list⟩

⟨axioms⟩ ::= ⟨empty⟩
    | ⟨equations⟩
    | **for each** ⟨decl list⟩ ⟨equations⟩

⟨equations⟩ ::= ⟨empty⟩ | ⟨equation list⟩
⟨equation list⟩ ::= ⟨equation⟩
    | ⟨equation list⟩ ⟨equation⟩

⟨equation⟩ ::= **eq** ⟨expr⟩ = ⟨expr⟩

4 System F

4.1 Pure form

Take the pure typed lambda calculus and add to it the following to get Girard's system **F**.

⟨type⟩ ::= (Π ⟨type identifier⟩ . ⟨type⟩)

⟨expr⟩ ::= (Λ ⟨type identifier⟩ . ⟨expr⟩)
    | (⟨expr⟩ ⟨type⟩)

4.2 Fun form

The following may be added to URDI to give the language "URDI+F". When this is done the built-in functions `cons`, `fst`, and `snd` become unnecessary, as they can be defined in "URDI+F".

⟨`type`⟩ ::= **forall** `[` ⟨`identifier list`⟩ `]` ⟨`type`⟩

⟨`expr`⟩ ::= **poly** `[` ⟨`identifier list`⟩ `]` ⟨`expr`⟩
    | ⟨`expr`⟩ `[` ⟨`type list`⟩ `]`