Higher-Order Abstract Syntax

Carsten Schürmann

September 28, 2000

In this lecture we are concerned with the question how to represent Mini-ML [Pfe00] expressions and how to program the related algorithms which we have encoded as deductive systems. The programming language of choice for this lecture is SML.

The possibilities to encode Mini-ML expressions are manifold, and each has its advantages and disadvantages. In essence, the choice of how an *expression* is represented is driven by the way how *variables* are represented. In this lecture we consider strings, integers, and meta-variables.

1 Variables As Strings

In the previous lectures, we defined the language of expressions to be defined as follows:

$$\begin{array}{lrcl} \textit{Expressions}: & e & ::= & \mathbf{z} \mid \mathbf{s}\ e \mid (\mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2) \\ & & & \mid x \mid \mathbf{lam}\ x.\, e \mid \mathbf{app}\ e_1\ e_2 \\ & & & \mid \langle e_1, e_2 \rangle \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e \\ & & & \mid \mathbf{fix}\ x.\, e \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \end{array}$$

Following the judgments as types paradigm $\ulcorner \text{is exp} \urcorner = \text{exp} : \text{type}$, where $\ulcorner \cdot \urcorner$ is the representation function, as discussed in [Pfe00]. The way expressions are encoded in SML is by the means of datatype definitions. Each constructor of Mini-ML is turned into a constructor for the datatype exp. For example:

$$\begin{array}{rcl} \ulcorner \mathbf{z} \urcorner & = & \text{z} : \text{exp} \\ \ulcorner \mathbf{s}\ e \urcorner & = & \text{s}\ \ulcorner e \urcorner : \text{exp} \end{array}$$

yields the first two constructors of the datatype:

$$\begin{array}{l}\textbf{datatype}\ \mathrm{exp}\\ \quad = \mathrm{z}\\ \quad\ \ \mid \mathrm{s}\ \textbf{of}\ \mathrm{exp}\end{array}$$

The representation of the case construct is slightly more complicated because it involves the representation of variables as strings. Therefore,

$$\ulcorner \textbf{case}\ e\ \textbf{of}\ \textbf{z}\ \Rightarrow e_1 \mid \textbf{s}\ x \Rightarrow e_2 \urcorner \quad = \quad \mathrm{case}\ (\ulcorner e \urcorner, \ulcorner e_1 \urcorner, (\text{“}x\text{”}, \ulcorner e_2 \urcorner))$$

where

$$\ulcorner x \urcorner \quad = \quad \text{“}x\text{”}$$

which leads to the definition of the following two constants for the datatype exp:

$$\begin{array}{lcl}\mathrm{case} & : & \mathrm{exp} * \mathrm{exp} * (\mathrm{string} * \mathrm{exp}) \to \mathrm{exp}\\ \mathrm{var} & : & \mathrm{string}\end{array}$$

Similarly,

$$\begin{array}{lcl}\ulcorner \textbf{lam}\ x.\, e \urcorner & = & \mathrm{lam}\ (\text{“}x\text{”}, \ulcorner e \urcorner)\\ \ulcorner \textbf{app}\ e_1\ e_2 \urcorner & = & \mathrm{app}\ \ulcorner e_1 \urcorner\, \ulcorner e_2 \urcorner\\ \ulcorner \langle e_1, e_2 \rangle \urcorner & = & \mathrm{pair}\ \ulcorner e_1 \urcorner\, \ulcorner e_2 \urcorner\\ \ulcorner \textbf{fst}\ e \urcorner & = & \mathrm{fst}\ \ulcorner e \urcorner\\ \ulcorner \textbf{snd}\ e \urcorner & = & \mathrm{snd}\ \ulcorner e \urcorner\\ \ulcorner \textbf{fix}\ x.\, e \urcorner & = & \mathrm{fix}\ (\text{“}x\text{”}, \ulcorner e \urcorner)\\ \ulcorner \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \urcorner & = & \mathrm{let}\ \ulcorner e_1 \urcorner\ (\text{“}x\text{”}, \ulcorner e_2 \urcorner)\end{array}$$

Putting it altogether, we arrive at the following SML datatype declaration for Mini-ML expression.

$$\begin{array}{l}\textbf{datatype}\ \mathrm{exp}\\ \quad = \mathrm{z}\\ \quad\ \ \mid \mathrm{s}\ \textbf{of}\ \mathrm{exp}\\ \quad\ \ \mid \mathrm{case}\ \textbf{of}\ \mathrm{exp} * \mathrm{exp} * (\mathrm{string} * \mathrm{exp})\\ \quad\ \ \mid \mathrm{var}\ \textbf{of}\ \mathrm{string}\\ \quad\ \ \mid \mathrm{lam}\ \textbf{of}\ \mathrm{string} * \mathrm{exp}\\ \quad\ \ \mid \mathrm{app}\ \textbf{of}\ \mathrm{exp} * \mathrm{exp}\\ \quad\ \ \mid \mathrm{pair}\ \textbf{of}\ \mathrm{exp} * \mathrm{exp}\\ \quad\ \ \mid \mathrm{fst}\ \textbf{of}\ \mathrm{exp}\\ \quad\ \ \mid \mathrm{snd}\ \textbf{of}\ \mathrm{exp}\\ \quad\ \ \mid \mathrm{fix}\ \textbf{of}\ \mathrm{string} * \mathrm{exp}\\ \quad\ \ \mid \mathrm{let}\ \textbf{of}\ \mathrm{exp} * (\mathrm{string} * \mathrm{exp})\end{array}$$

Example 1.1 (Double function)

$$\begin{array}{ll} & \ulcorner \mathbf{fix}\ f.\ \mathbf{lam}\ x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathbf{z}\ \Rightarrow \mathbf{z} \mid \mathbf{s}\ x' \Rightarrow \mathbf{s}\ (\mathbf{s}\ (\mathbf{app}\ \ f\ x'))\urcorner \\ = & \text{fix}\ (\text{“f”}, \text{lam}\ (\text{“x”}, \\ & \qquad \text{case}\ (\text{var “x”}, \text{z}, (\text{“x' ”}, \text{s s}\ (\text{app}\ (\text{var “f”})\ (\text{var “x' ”})))))) \end{array}$$

This encoding is certainly very direct but it possesses a few drawbacks, which we will discuss now. Consider an implementation of the operational semantics for Mini-ML as presented in one of the previous lectures in SML as a function mapping expression to expressions. Conceptually, such a function is not very difficult to program. However, the implementation will be rather long and tedious. Why? Because it must treat variables, renamings, and substitutions explicitly. This is commonly done by defining the appropriate helper functions, which are typically tedious to code, error-prone, and difficult to debug.

2 Variables As Integers

A second possibility is to use integers instead of strings. One technique based on this idea goes back to de Bruijn [dB72], where each variable is represented by a so called *de Bruijn index*. The face value of each index denotes the number of nested variable binders and points therefore to the binder that introduces the variable.

Differently from above, each construct of Mini-ML which binds variables need not to bind a name anymore — a corresponding variable binder can be directly inferred from the structure of the term.

$$\ulcorner \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z}\ \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \urcorner \quad = \quad \text{case}\ (\ulcorner e \urcorner, \ulcorner e_1 \urcorner, \ulcorner e_2 \urcorner)$$

where

$$\ulcorner x \urcorner \quad = \quad \text{depth of the variable with respect to the term}$$

which leads to the definition of the following two constants for the datatype exp:

$$\begin{array}{lcl} \text{case} & : & \text{exp} * \text{exp} * \text{exp} \to \text{exp} \\ \text{var} & : & \text{int} \end{array}$$

This technique has the advantage that variables are disambiguated from their names. Renaming of variables is hence not an external operation any more, it is taken care of by the representation itself.

In summary, with the variables as de Bruijn index representation, we obtain the following datatype declaration in SML:

$$
\begin{array}{l}
\textbf{datatype}\ \mathrm{exp} \\
\quad = \mathrm{z} \\
\quad\ \ \mid \mathrm{s}\ \textbf{of}\ \mathrm{exp} \\
\quad\ \ \mid \mathrm{case}\ \textbf{of}\ \mathrm{exp} * \mathrm{exp} * \mathrm{exp} \\
\quad\ \ \mid \mathrm{var}\ \textbf{of}\ \mathrm{int} \\
\quad\ \ \mid \mathrm{lam}\ \textbf{of}\ \mathrm{exp} \\
\quad\ \ \mid \mathrm{app}\ \textbf{of}\ \mathrm{exp} * \mathrm{exp} \\
\quad\ \ \mid \mathrm{pair}\ \textbf{of}\ \mathrm{exp} * \mathrm{exp} \\
\quad\ \ \mid \mathrm{fst}\ \textbf{of}\ \mathrm{exp} \\
\quad\ \ \mid \mathrm{snd}\ \textbf{of}\ \mathrm{exp} \\
\quad\ \ \mid \mathrm{fix}\ \textbf{of}\ \mathrm{exp} \\
\quad\ \ \mid \mathrm{let}\ \textbf{of}\ \mathrm{exp} * \mathrm{exp}
\end{array}
$$

Example 2.1 (Double function)

$$
\begin{array}{rl}
& \ulcorner \textbf{fix}\ f.\ \textbf{lam}\ x.\ \textbf{case}\ x\ \textbf{of}\ \textbf{z}\ \Rightarrow \textbf{z} \mid \textbf{s}\ x' \Rightarrow \textbf{s}\ (\textbf{s}\ (\textbf{app}\ \ f\ x'))\urcorner \\
= & \mathrm{fix}\ (\mathrm{lam}\ (\mathrm{case}\ (\mathrm{var}\ 1, \mathrm{z}, \mathrm{s}\ \mathrm{s}\ (\mathrm{app}\ (\mathrm{var}\ 3)\ (\mathrm{var}\ 1)))))
\end{array}
$$

Nevertheless, the notion of substitution is still not part of the representation. On the contrary, they have to be defined explicitly. A correct implementation of a substitution mechanism with strings for names or integers remains challenging.

3 Variables As Meta-Variables

Any implementation of Standard ML of New Jersey also implements several concepts related to variables and substitutions. For example, a function "fn x => x" in Standard ML notation, defines a binder and a variable, and applying this function to the integer "(fn x => x) 6" evaluates to 6.

When using SML as a representation language we can take advantage of this property by representing variables of the object language, here Mini-ML, as SML variables. In this sense.

$$
\begin{array}{rcl}
\ulcorner \textbf{case}\ e\ \textbf{of}\ \textbf{z}\ \Rightarrow e_1 \mid \textbf{s}\ x \Rightarrow e_2 \urcorner & = & \mathrm{case}\ (\ulcorner e \urcorner, \ulcorner e_1 \urcorner, \mathrm{fn}\ x \Rightarrow \ulcorner e_2 \urcorner) \\
& & \mathrm{where}\ \ulcorner x \urcorner = x
\end{array}
$$

which leads to the definition of the following constants for the datatype exp:

$$\text{case} \quad : \quad \text{exp} * \text{exp} * (\text{exp} \to \text{exp}) \to \text{exp}$$

where we do not need an explicit constructors for variables because they are represented by the typical LF variables. The same technique can be applied for the representation of functions, the let construct, and the fixed point construct:

$$\begin{array}{lcl}
\ulcorner \mathbf{lam}\, x.\, e \urcorner & = & \text{lam (fn } x \text{ => } \ulcorner e \urcorner) \\
 & & \text{where } \ulcorner x \urcorner = x \\[1ex]
\ulcorner \mathbf{fix}\, x.\, e \urcorner & = & \text{fix (fn } x \text{ => } \ulcorner e \urcorner) \\
 & & \text{where } \ulcorner x \urcorner = x \\[1ex]
\ulcorner \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \urcorner & = & \text{let } \ulcorner e_1 \urcorner \text{ (fn } x \text{ => } \ulcorner e_2 \urcorner) \\
 & & \text{where } \ulcorner x \urcorner = x
\end{array}$$

Putting it altogether, we arrive at a new datatype declaration for expressions in SML.

$$\begin{array}{l}
\mathbf{datatype}\ \text{exp} \\
\quad = \text{z} \\
\quad\ \ \mid \text{s } \mathbf{of}\ \text{exp} \\
\quad\ \ \mid \text{case } \mathbf{of}\ \text{exp} * \text{exp} * (\text{exp} \to \text{exp}) \\
\quad\ \ \mid \text{lam } \mathbf{of}\ \text{exp} \to \text{exp} \\
\quad\ \ \mid \text{app } \mathbf{of}\ \text{exp} * \text{exp} \\
\quad\ \ \mid \text{pair } \mathbf{of}\ \text{exp} * \text{exp} \\
\quad\ \ \mid \text{fst } \mathbf{of}\ \text{exp} \\
\quad\ \ \mid \text{snd } \mathbf{of}\ \text{exp} \\
\quad\ \ \mid \text{fix } \mathbf{of}\ \text{exp} \to \text{exp} \\
\quad\ \ \mid \text{let } \mathbf{of}\ \text{exp} * (\text{exp} \to \text{exp})
\end{array}$$

Example 3.1 (Double function)

$$\begin{array}{l}
\ulcorner \mathbf{fix}\, f.\, \mathbf{lam}\, x.\, \mathbf{case}\ x\ \mathbf{of}\ \mathbf{z}\ \Rightarrow \mathbf{z} \mid \mathbf{s}\ x' \Rightarrow \mathbf{s}\ (\mathbf{s}\ (\mathbf{app}\ \ f\ x')) \urcorner \\
\quad = \ \ \text{fix (fn } f \text{ => lam (fn } x \text{ => case } (x, \text{z}, \text{fn } x' \text{ => s s (app } f\ x'))))
\end{array}$$

These datatype declarations have to be used with care as we will discuss in the next lecture.

References

[dB72] N.G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[Pfe00] Frank Pfenning. *Computation and Deduction*. Cambridge University Press, 2000. In preparation. Draft from April 1997 available electronically.