Abstract [submitted for presentation at the NWPT'01]

If values are nodes, then references are arrows

Nils Andersen (DIKU, Denmark)

Two desirable properties in a programming language are the ability to work with self-referring or cyclic data structures, and a polymorphic typing system, so that programs can be written in their most general form. Unfortunately, these two features are difficult to combine without losing the protection offered by a strong typing system, such as "subject reduction" (preservation of type under evaluation), or freedom from type errors under evaluation.

The programming language Standard ML designates a compromise. In the revised (1997) definition of the language, a distinction is made between *expansive* and *non-expansive* expressions (also known as *syntactic values*). In the construction

$$\texttt{let val } v \texttt{ = } e' \texttt{ in } e \texttt{ end}$$

v is only (in e) bound to a general type scheme if e' is non-expansive. Otherwise v has a fixed proper type, and the construction is completely equivalent to `(fn` v `=>` e`)` e'.

This presentation suggests that since the restriction is present anyway (whether or not a program actually uses references), we might as well consider every data constructor field to be a reference or pointer.

In addition to the *node variables*, the usual variables of lambda calculus, introduced via lambda abstractions or `let` expressions, we now also consider *arrow variables.* An arrow variable y_j contains a field of a data constructor and is instroduced via a new piece of syntax:

$$\texttt{open } e \texttt{ as } \ldots \ \| \ C_i \ y_1 \ \ldots \ y_k \texttt{ => } e_i \ \| \ \ldots$$

As for pattern matching, the scope of $y_1, \ldots, y_k$ is the expression e_i, and inside e_i assignments to these arrow variables are permitted.

Constructed values $C_i \ e_1 \ \ldots \ e_k$ must now also be considered expansive, but it is possible, for instance, to define in the proposed calculus a strongly typed version of the LISP function `rplaca` (in SCHEME `set-car!`, assign a new value to the head of a list).

The advantages of strong typing are preserved: Each closed expression has a principal type, evaluation preserves types (subject reduction) and cannot lead to type errors (well typed expressions can't go wrong).

Compared to Standard ML some of the features of the suggested calculus are that

- potentially, each data constructor field is a reference; in a cyclic data structure it is unnecessary to select a particular link to be the "backwards thread" or give it any special declaration
- notation and typing rules are (in the author's opinion) simpler