A Statically Typed Functional Language With Programmable Binders

Florian Lorenzen

TU Berlin

Binding structures are ubiquitous elements in functional (but also other) programming languages. Here are some examples taken from various statically typed languages:

```
do { x ← getLine; putStrLn x }                    Haskell: do-notation
for (i ← 1 until 3) println(2*i)                  Scala: for expression
with address do begin writeln(city) end           Pascal: record opening
let val x = 7 in x-4 end                          SML: let expression
```

All these expressions are "syntactic sugar" [4] for more primitive expressions. For example, a Haskell implementation will "desugar" the first line into

```
getLine >>= (\x -> putStrLn x)
```

using a λ-abstraction as a more primitive binder and implementing the operational behaviour by the monadic bind operator `>>=`. Similar transformations are used for the other examples.

All these transformations are hardwired into the respective compilers and applied after the type checking phase. As a consequence, type errors are described relative to the original input, which makes diagnostics much more useful. Moreover, the input of the transformation is type correct with respect to the typing rules of the input language. It is important that the input's type is preserved during desugaring to obtain a type-safe implementation.

We observe that binder support in programming languages is restricted to a few "chosen ones". But many fields and concepts bring their own specialized binders like arrows [7], "comprehensive comprehensions" [8], monadic value recursion [2], embedded grammars [1], comonads [6], or functional hybrid modelling [3]. These are either dependent on language extensions or on embeddings using techniques like quasiquoting [5]. The former is only available to the language implementor, the latter often requires a significant implementation effort and since the expansion is performed prior to type checking error messages are often hard to decipher.

Instead, we advocate to equip a programming language with an abstraction mechanism to define new binders on top of existing ones. We demand that such a mechanism satisfies the following requirements:

1. Type error detection on the original input, not the desugared form.
2. Detection of ill-formed binder definitions which would break type-safety.
3. Mixfix syntax and overloading of symbols.

From these, the second requirement is the most important one: Suppose we define a new binder by a desugaring transformation. Then, the expansion of a

binder might raise a type error because either its arguments have the wrong type or the desugaring transformation is errorneous. This is an unfortunate situation because it complicates debugging and also exposes implementation details. We would rather check the definition of the binder and verify once and forall that, if a binder is applied to arguments of the correct types, its expansion is also free from type errors. This situation is enforced by the second requirement, since it establishes an abstraction principle between binder definition and binder application: if a binder definition is well-formed, an application of this binder can be type checked without referring to the code of the implementation but only its type information.

In this presentation, we propose an approach to tackle the first two aspects: We extend System F^ω with two additional forms to define and use sequential and parallel binders. A sequential binder like `LET*` in Scheme binds variables in sequence such that later bindees have earlier variables in scope. We extend the well-known desugaring

$$\texttt{LET*}\ x_1\texttt{=}t_1\texttt{;}\ \ldots\texttt{;}\ x_n\texttt{=}t_n\ \texttt{IN}\ t \quad\Longrightarrow\quad (\lambda x_1 \ldots.(\lambda x_n.\ t)\ t_n \ldots)\ t_1$$

to include programmer supplied terms to implement specific operational behaviour.

A parallel binder like `LET` in Scheme binds several variables simultaneously. We include it in a similar fashion like the sequential one.

We use System F^ω as basis for our extension since many binding structures are related to polymorphic data structures (e. g. lists or monads). System F^ω formally captures these as type operators and gives a good foundation for our purposes.

The desugaring transformation is defined on typing derivations rather than plain terms to infer the types of bound variables. The desugaring transformation takes a term of the extended language as input and returns a plain System F^ω term as result. A type preservation theorems guarantees that input and output have the same types if the input is well-typed by the rules of the extended language.

References

1. Baars, A.I., Swierstra, S.D.: Type-safe, self inspecting code. In: Haskell '04 (2004)
2. Erkök, L., Launchbury, J.: A recursive do for Haskell. In: Haskell '02 (2002)
3. Giorgidze, G., Nilsson, H.: Embedding a functional hybrid modelling language in Haskell. In: IFL '08 (2008)
4. Landin, P.J.: The mechanical evaluation of expressions. Comp. J. 6(4) (1964)
5. Mainland, G.: Why it's nice to be quoted: quasiquoting for Haskell. In: Haskell '07 (2007)
6. Orchard, D., Mycroft, A.: A notation for comonads. Submitted to Haskell 2012 (2012)
7. Paterson, R.: A new notation for arrows. In: ICFP '01 (2001)
8. Peyton Jones, S.L., Wadler, P.: Comprehensive comprehensions – comprehensions with 'order by' and 'group by'. In: Haskell '07 (2007)