Worksheets

Daniel Hillerström

20th July 2015

Contents

Chapter 1

Introduction

1.1 Problem analysis

Used car-dealers got a notorious reputation for being dishonest. Dishonest, because, they are reluctant to disclose any unfortunate effects that their cars may have. On the other hand, if they *did* disclose all the effects then we probably would not buy them anyway as it appears too much effort to handle those effects.

Correspondingly, some programming languages do not disclose the potential run-time effects of code execution, e.g. the ML-family of languages. For example consider the signature `readFile : string -> [string]` for a function in SML, its suggestive name hints that given a file name the function reads the file and return the contents line by line. In order to read a file the function must inevitably perform a side-effecting action, namely, accessing a storage media. But this information is not conveyed in the function signature.

Other languages disclose effects, albeit with varying degree. For example the Java programming language requires programmers to be explicit about potential unhandled checked exceptions that may occur during run-time, e.g. `String[] readFile(String f) throws IOException`. But programmers can circumvent this requirement by raising unchecked exceptions. Critics argue that Java's checked exceptions suffer versionability and scalability issues [2], and therefore it is better not to have explicit `throws` declarations.

The Haskell programming language is also explicit about effects, but, in contrast to Java, it offers no escape hatch to be implicit. Haskell insists that every effectful computation is encapsulated inside an appropriate monad. In Haskell the file reading function would be typed as `readFile :: String -> IO [String]`, where the `IO`-annotation signifies that the function might perform an input/output side-effect. A consequence of enforcing effectful computations to be wrapped inside a monad is that the function signature conveys additional information about the computation which the programmer and compiler can rely on.

1.1.1 The problem with monads

Monads provide a remarkably powerful way for structuring computations, because they integrate effectful and pure computations in an elegant and flexible manner. Sadly, monads do not compose well [3], and accordingly it is difficult to give a monadic description of computations that might perform multiple effects.

1.1.2 Composing monads with Monad Transformers

Monad transformers allow two monads to be combined by stacking one on top of the other. Furthermore, a monad transformer is itself a monad, and thus we can create arbitrarily complex compositions. Incidentally, monad transformers can capture computations that may cause several different effects. However they are no silver bullet as they impose an ordering on effects.

1.2 Problem statement

In the previous section we argued that programming with *explicit* effect is desirable, but we pointed out that it is not painless to program with explicit effects. In particular, we demonstrated that the monadic approach imposes an ordering on effects which impedes compositionality and modularity. Two key properties in programming which we ideally would like to retain along with explicit effects. This observation leads us to the following problem statement:

> *How may we achieve a programming model for modular, composable and unordered effects?*

1.3 Proposed solution

Some words here.

1.3.1 Project scope

A few words about the scope.

1.4 Related work

Existing solutions.

1.5 Row polymorphism

Records are unordered collections of fields, e.g. $\langle l_1 : t_1, \ldots, l_n : t_n\rangle$ denotes a record type with n fields where l_i and t_i denote the name and type, respectively, of the ith field. Records are particularly great for structuring related data. For example the record instance $\langle name = "Daniel", age = 25\rangle$ could act as simple description of a person. A possible type for the record would be $\langle name : \texttt{string}, age : \texttt{int}\rangle$. The implicit ordering of fields in records is not important as the following two types are considered equivalent:

$$\langle name : \texttt{string}, age : \texttt{int}\rangle \equiv \langle age : \texttt{int}, name : \texttt{string}\rangle.$$

The equivalence captures what is meant by *unordered* collection of fields.

We can define functions that work on records, for example, occasionally we might find it useful to retrieve the name field in an instance of the person-record:

$$\text{name}_1 = \lambda x.(x.name).$$

We can apply name_1 to the above record instance, e.g.

$$\text{name}_1(\langle name = "Daniel", age = 25\rangle) = "Daniel"$$

yields the expected value. The question remains which type the function name_1 has. For this particular example the type $\langle name : \texttt{string}\rangle \rightarrow \texttt{string}$ seems reasonable.

Now, consider the following function takes a record and returns a pair where the first component contains the value of the name field and the second component contains the input record itself:

$$\text{name}_2 = \lambda x.(x.name, x)$$

Again, we ask ourselves what is the type of name_2? The function must have type $\langle name : \texttt{string}\rangle \rightarrow \texttt{string} \times \langle name : \texttt{string}\rangle$. This type appears innocuous, but consider the consequence of

$$\text{let } (n, r) = \text{name}_2(\langle name = "Daniel", age = 25\rangle).$$

Now, n has type $\texttt{string}$ as desired, but r has type $\langle name : \texttt{string}\rangle$. In other words, we have lost the field "age". This example is a particular instance of the *loss of information* problem.

Row polymorphism was conceived to address this problem. Row polymorphism is powerful a typing discipline for typing records [1]. The principal idea is to extend record types with a *polymorphic row variable*, ρ, which can be instantiated with additional fields. For example with row polymorphism our function name_2 would have type $\langle name : \texttt{string} \mid \rho\rangle$, and r in the previous example would be assigned the type $\langle name : \texttt{string}, age : \texttt{int} \mid \rho\rangle$. Now, we have no longer lose information.

1.5.1 Type theoretic row polymorphism

1.5.2 Row polymorphism is not subtyping

Bibliography

[1] Didier Rémy. “Type Inference for Records in a Natural Extension of ML”. In: *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design.* Ed. by Carl A. Gunter and John C. Mitchell. MIT Press, 1993.

[2] Bill Venners and Bruce Eckel. *The Trouble with Checked Exceptions: A Conversation with Anders Hejlsberg, Part II.* `http://www.artima.com/intv/handcuffs.html`. 2003.

[3] Ohad Kammar, Sam Lindley and Nicolas Oury. “Handlers in Action”. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming.* ICFP ’13. Boston, Massachusetts, USA: ACM, 2013, pp. 145–158. ISBN: 978-1-4503-2326-0. DOI: `10.1145/2500365.2500590`. URL: `http://doi.acm.org/10.1145/2500365.2500590`.