A functional language for specifying business reports

Bahr, Patrick

Published in:
Proceedings of the 23rd Nordic Workshop on Programming Theory

Publication date:
2011

Document version
Peer reviewed version

Citation for published version (APA):
Bahr, P. (2011). A functional language for specifying business reports. In P. Pettersson, & C. Seceleanu (Eds.), *Proceedings of the 23rd Nordic Workshop on Programming Theory* (pp. 24-26). Mälardalen University. Mrtc technical report, No. 254, Vol.. 2011

A Functional Language for Specifying Business Reports

Patrick Bahr
Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen, Denmark
paba@diku.dk

Abstract

We describe our work on developing a functional domain specific language for specifying business reports. The report specification language is part of a novel enterprise resource planing system based on the idea of a providing a lean core system that is highly customisable via a variety of domain specific languages.

1 Introduction

Process-oriented event-driven transaction systems (POETS) is a novel software architecture for enterprise resource planning (ERP) systems, introduced by Henglein et al. [1]. Rather than storing both transactional data and implicit process state in a database, POETS employs a pragmatic separation between (a) transactional data, that is what has happened; (b) reports, that is what can be derived from the transactional data; and (c) contracts, that is which transactions are expected in the future. Moreover, rather than using general purpose programming languages to specify business processes, POETS utilises declarative domain-specific languages (DSLs) to customise the different aspects of a system. The use of DSLs not only enables explicit formalisation of business processes, it also minimises the gap between requirements and a running system.

A simplified overview over the POETS architecture is presented in Figure 1. At the heart of the system is the event log, which is an append-only list of transactions. Transactions represent relevant events that may occur, such as a payment by a customer, a delivery of goods by a shipping agency, or a movement of items into an inventory. This does not only satisfies the legal requirement for ERP systems to archive all transaction data that is relevant for auditing but also makes it possible to compute reports incrementally as shown by Nissen and Larsen [3].

2 The Report Language

The purpose of the report engine is to provide a structured view of the data base that is constituted by the system's event log. This structured view of the data in the event log comes in the form of a *report*, a collection of condensed structured information compiled from the event log. Conceptually, a report is compiled from the event log by a function of type $\mathsf{EventLog} \rightarrow \mathsf{Report}$, a *report function*. The *report language* provides a means to specify such a report function in a declarative manner.

The report language is – much like the query fragment of *SQL* – a functional language *without side effects*. It only provides operations to non-destructively manipulate and combine values. Since the system's storage is based on a shallow event log – basically a list of event representations – the report language has to provide operations to relate, filter and aggregate pieces of information. Moreover, as the data stored in the event log is inherently heterogeneous – containing data of different kind – the report language needs to offer a comprehensive type system that allows to safely operate in this setting.

The entire system is based on a common basis of base types consisting of strings, Booleans, integers, lists etc. Apart from that the system offers user-defined record types with an inheritance system based on *nominal subtyping*. To this end, POETS also offers an *ontology language* that is used to describe record types, their fields and their interdependence. The central record type is Event, which represents the events that are registered in the event log. In fact, as far as the report language is concerned, the event log is a value of type [Event].

Every interaction with the running system is reflected with a corresponding value of (a subtype of) type Event in the event log. The simplest example is the interface to the report engine itself. It allows to add, modify and remove reports. Each such operation is reflected by an event of type

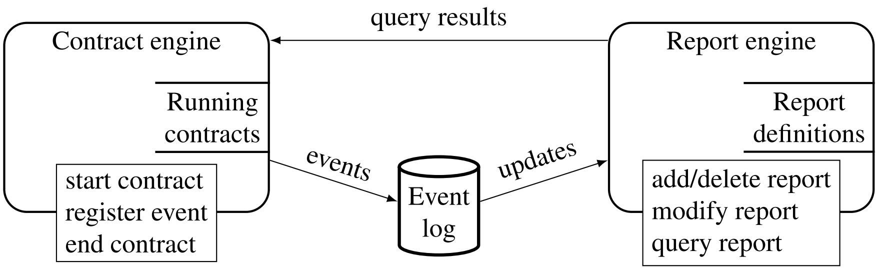

Figure 1: Bird's-eye view of the POETS architecture [1].

CreateReport, UpdateReport, and DeleteReport, respectively. The former two are subtypes of PutReport, which in turn is – like DeleteReport – a subtype of ReportEvent.

This allows us to write the following simple report function that creates the report which lists the names of all active (i.e. not deleted) reports:

reportNames : [**String**]
reportNames = [*pr.name* |
  cr : CreateReport ← **events**,
  pr : PutReport = *head* [*ur* |
    ur : ReportEvent ← **events**,
    ur.name ≡ *cr.name*]]

Every report function implicitly has as its first argument the event log of type [Event] – a list of events – bound to the name **events**. The syntax of the report language – and to large parts also its semantics – is based on Haskell [2]. The central data structure is that of lists. In order to formulate operations on lists concisely, we use list comprehensions [4] as seen in the above example. A list comprehension of the form [e | c] denotes a list containing elements of the form e generated by c, where c is a sequence of *generators* and *filters*.

As we have mentioned, access to type information and its propagation to subsequent computations is essential due to the fact that the event log is a list of heterogeneously typed elements – events of different kinds. The generator *cr* : CreateReport ← **events** iterates through elements of the list **events** binding each element to the variable *cr*. The typing *cr* : CreateReport restricts this iteration to elements of type CreateReport. This type information is propagated through the subsequent generators and filters of the list comprehension. In the filter *ur.name* ≡ *cr.name*, we use the fact that elements of type ReportEvents have a field *name* of type **String**. When binding the first element of the result of the nested list comprehension to the variable *pr* it is also checked whether this element is in fact of type PutReport. Thus we ignore reports that are marked as deleted via a DeleteReport event.

The report language is based on the simply typed lambda calculus extended with a polymorphic (non-recursive) let expression and a type case expression. The core language is given by the following grammar:

$$e ::= x \mid c \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{let}\ x = e\ \textbf{in}\ e' \\ \mid \textbf{type}\ x = e\ \textbf{of}\ \{r \to e_1; \_ \to e_2\}$$

where x ranges over variables, and c over constants which includes integers, Booleans, tuple and list constructors as well as operations on them like $+$, *if-then-else* etc. In particular, we have a fold operation **fold** of type $(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$. This is the only operation of the report language that permits recursive computations on lists. List comprehensions are mere syntactic sugar and can be reduced to **fold** and let expressions as for example in Haskell [2].

The extended list comprehension of the report language that allow filtering according to run time type information depend on type case expressions of the form $\textbf{type}\ x = e\ \textbf{of}\ \{r \to e_1; \_ \to e_2\}$. In such a type case expression, an expression e of some record type r_e gets evaluated to record value v which is then bound to a variable x. The record type r that the record value v is matched against can be any subtype of r_e. The further evaluation of the type case expression depends on the type r_v of the record value v. This type can be any subtype of r_e.

If $r_v \leq r$, the evaluation proceeds with e_1, otherwise with e_2. Binding e to a variable x allows to use the stricter type r in the expression e_1.

Although, the subtyping discipline that we use is nominal, the type system also allows the programmer to use record types as if the subtyping was purely structural. This is needed in order to allow the sharing of field names between distinct record types. To this end, we use *type constraints* of the form $\alpha.f : \tau$ which intuitively states that α is a record type with a field f of type τ. Field selectors are merely postfix operators. For example the *.name* field selector in the example is of type $\alpha.\mathsf{name} : \beta \Rightarrow \alpha \rightarrow \beta$.

Another important aspect of POETS in general and the report language in particular is the maintaining of references and the access of the data they refer to. This becomes necessary as certain pieces of information, e.g. customer information, are attached to a unique entity with lifecycle, e.g. a customer. To this end, POETS allow to create an entity with a unique id – a *reference*. Subsequently, information attached to this entity can be updated and eventually, the entity can be removed altogether. All these changes are, of course, reflected in the event log and can thus be examined by a report function. Nevertheless, due to the importance of references, the report language offers dedicated dereferencing operations that allow quick and typesafe access to the data associated with entities.

While the type system is important in order to avoid obvious specification errors, it is also important to ensure a fast execution of the thus obtained functional specifications. This is, of course, a general issue for querying systems. In our system, it is, however, of even greater importance since shifting the structure of the data – from the data store to the domain of queries – means that queries operate on the complete data set of the data base and thus each report has to be recomputed after each transaction. In other words, if treated naïvely, the conceptual simplification provided by the flat event log has to be paid via much more expensive computations.

This issue can be addressed by transforming a given report function f into an incremental function f' which updates a previously computed report according to the changes that have occurred since the report was computed before. That is, given an event log l and an update to it $l \oplus e$, we require that $f(l \oplus e) = f'(f(l), e)$. The new report $f(l \oplus e)$ is obtained by updating the previous report $f(l)$ according to the changes e. In the case of the event log, we have a list structure. Changes only occur *monotonically*, by adding new elements to it: Given an event log l and a new event e, the new event log is $e\#l$, where # is the list constructor of type $\alpha \rightarrow [\alpha] \rightarrow [\alpha]$.

Here it is crucial that we have restricted the report language such that operations on lists are limited to the higher-order function **fold**. The fundamental idea of incrementalising report functions is based on the following equation:

$$\textbf{fold}\ f\ e\ (x\#xs) = f\ x\ (\textbf{fold}\ f\ e\ (xs))$$

Based on this idea, we are able to make the computation of most reports independent of the size of the event log but only dependent of the changes to the event log and the previous report [3]. Unfortunately, if we have for example list comprehensions containing more than one generator, we get report functions with nested folds. In order to properly incrementalise such functions, we need to move from list structures to multisets. This is, however, only rarely a practical restriction since most aggregation functions are based on commutative binary operations and are thus oblivious to ordering.

References

[1] Fritz Henglein, Ken Friis Larsen, Jakob Grue Simonsen, and Christian Stefansen. POETS: Process-oriented event-driven transaction systems. *Journal of Logic and Algebraic Programming*, 78(5):381–401, May 2009.

[2] Simon Marlow. *Haskell 2010 Language Report*, 2010.

[3] Michael Nissen and Ken Friis Larsen. FunSETL — Functional Reporting for ERP Systems. In Olaf Chitil, editor, *IFL '07*, pages 268–289, 2007.

[4] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(04):461–493, 1992.