Data Analysis

Michael Nissen & Ken Friis Larsen

October 21, 2007

Abstract

We present the language FunSETL. FunSETL is a small functional language for capturing the essence of report generation programs in Enterprise Resource Planning (ERP) systems. The purpose of FunSETL is twofold: First, we use it as a mean to understand how to optimize report generation functions with *automatic incrementalization* in ERP systems. Second, we hope to gain insights about how to design a domain specific language for report generation. In this paper we describe the current status of our project.

1 Introduction

Todays ERP systems like Microsoft Dynamics AX and Navision use *multi purpose* programming languages and/or SQL queries to express *reports*, where reporting essentially means computing (simple) functions on large amount of data. In Dynamics AX the multi purpose language X++ (see [1][p.91-118]) is used to express reports and in Navision C/AL (see [2]) is used. Both X++ and C/AL are state based (imperative) programming languages, which makes them even more unfit to express reports on data, since a *declarative* approach seams more intuitive. Increasing competition in the business field and rapidly growing amount of data in ERP systems has dictated the need for faster computation of reporting functions and *real time* access to the results of all reports, ie. real time access to *business intelligence*.
General purpose languages are used in ERP systems because sometimes SQL is not expressive enough to support a specific report. Therefore we suggest that ERP systems should contain a *domain specific* language in which to express reports, ie. a language that has the power of SQL and furthermore it should also be able to express the reports that SQL can not.
In order to increase the computation speed of reports, we would like to be able to make *automatic incrementalization* of reporting functions, ie. new versions of the reporting functions, that does not need to traverse all data every time they are computed, which should be transperant to the user. Therefore it becomes crucial to limit the reporting language to only the necessary constructs.
We have defined a small functional language called FunSETL with the following properties:

1 It is a *functional* language.

2 It is *strongly normalizing*, ie. every program always terminates.

3 It is possible to represent and iterative over large amount of data.

ad 1: The language is made functional (declarative), since this seems more intuitive when expressing reports.
ad 2: This is a reasonable requirement, since all reporting programs should terminate with a result.
ad 3: This property is very central, when we need to compute functions over large collections of data.
The above criteria only apply to the *pure* FunSETL language, ie. it is possible for the programmer to call external methods and programs, and thereby bypass the FunSETL properties. In order to ensure usability of the language we have integrated it with .NET by compiling FunSETL code to C# code.

2 Data Analysis

In an ERP system we need a module that can perform computational *tasks* e.g. data analysis on the data stored by the ERP system (ie. computing *reports*).
This section will describe how our data analysis module will look like, both from a users perspective and "under the hood" (internally). A very simplified conceptual model of the architecture of our ERP system is shown in figure 1, page 3.
The following sections will describe the architecture and how the data analysis module should interact with the users and database, and how it works internally.

2.1 Architecture

In the heart of the architecture we have a database of *events*, ie. a log of *everything* that has happened. Events can be committed by users (note that in this context users also can be other automated systems) and they are filtered by a module called *Filtering & Decoration*, which will not be described here, but the module either accepts an event and commits it to the database or rejects the event and informs the user.
When an event is committed to the database it is also passed to the data analysis module, ie. every newly committed event is passed to the analysis module when it is committed by the Filtering & Decoration module.
This means that the data analysis module has access to *all information* at commit time. Later we will get back to why this is important.
A user can interact in two ways with the Data analysis module. Either the user can commit a report to the module, which is then stored in the *report repository*, or the user can request the result of a report computed on the current event database.
In our context a report is a program written in a *report language* called *FunSETL*, which is a *declarative* programming language. Section 3, page 4 will give a description of FunSETL.
Furthermore it should be noted that conceptually current ERP systems could be considered as the event database and the filtering and decoration module, and where we have a *monotone* growing event log. This means that the data analysis module can be considered as a *plug-in* for current ERP systems.

2.2 Data Analysis "under the hood"

The data analysis module contains a report repository where all reports that needs to be computed are stored.
When a report is committed it is then *automatically incrementalized*, ie. it is transformed into a new report, that hopefully will not need to traverse all the data in the event database every time it is computed. In Section 4, page 7 there will be an explanation of incrementalization and the advantages it gives us. The analysis module then *incrementally* maintains in *real time* the results of the reports in the repository based on the events that are continuously added to the event database, and handed to the analysis module. This means that we have *real time* access to the results of all the reports.
Furthermore all the advanced incrementalization stuff is transparent to the user, since the user only sees the declarative specification of the report that he/she writes, and then the result of the report, when it is requested. The automatically incrementalized versions of the reports are only used internally in the Data analysis module to provide real time access to

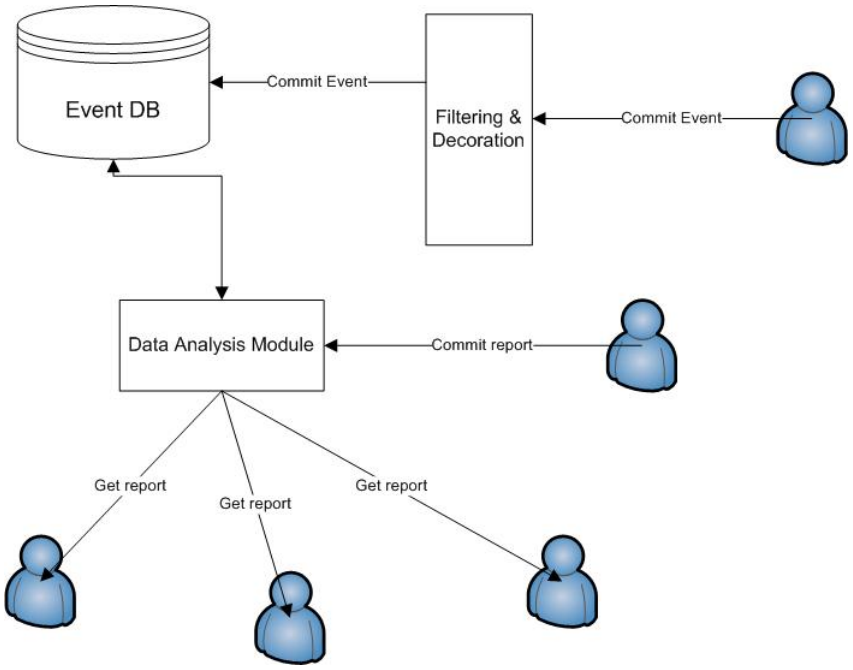

Figure 1: Overview of Architecture

the results of the reports in the repository.

3 FunSETL

As said in the previous section FunSETL is a declarative (functional) language, which means that the language does not contain *statements* but only *expressions*. The syntax of FunSETL is described by grammars in *Backus Naur Form* (BNF). Before we can define the syntax of expressions we need to define the syntax of types, since types are used in expressions.

Definition 3.1 *Type syntax.*
Let $TVar$ be an infinite set of type names. Then the syntax of FunSETL types can be expressed by the following grammar written in BNF:

$$\tau ::= id \mid \textbf{bool} \mid \textbf{int} \mid \textbf{real} \mid \textbf{date} \mid \tau_1 + \tau_2 \mid \{lab_1 : \tau_1, \ldots, lab_k : \tau_k\} \mid \textbf{map}(\tau_1, \tau_2) \mid \textbf{mset}(\tau)$$

where $k \geq 1$ and $id \in TVar$.

As usual *id* are type identifiers and **int**, **real** and **date** are the types for integers, real numbers and dates. $\tau_1 + \tau_2$ is the sum-type and $\{lab_1 : \tau_1, \ldots, lab_k : \tau_k\}$ is the record type. Elements of $\textbf{map}(\tau_1, \tau_2)$ are *finite* maps that map elements of type τ_1 to elements of τ_2, and $\textbf{mset}(\tau)$ is the type for *multi-sets* of elements of type τ.
The typeidentifiers are only included, such that it is possible to make shorthands for types, ie. before writing any FunSETL code one can make typedeclarations in the form **type** $id = \tau$.
Now we are able to define the syntax of FunSETL expressions.

Definition 3.2 *Expression Syntax.*
Assume Var is a set of identifiers, $FVar$ is a set of function identifiers, N is the set of (syntactic) integers and R is the set of (syntactic) reals. The syntax of the FunSETL expressions (production e) is described by the following grammar written in BNF:

$$\begin{aligned}
c &::= n \mid r \mid yyyy - mm - dd \mid \textbf{true} \mid \textbf{false} \\
binop &::= + \mid - \mid * \mid / \mid = \mid <= \mid < \mid \textbf{and} \mid \textbf{or} \mid \textbf{with} \mid \textbf{inter} \mid \textbf{union} \mid \textbf{diff} \mid \textbf{in} \mid \textbf{subset} \\
unop &::= \textbf{not} \mid \textbf{dom} \mid \textbf{toSet} \\
e &::= x \mid e_1\ binop\ e_2 \mid unop\ e \mid \textbf{inL}(e)\ \textbf{as}\ \tau \mid \textbf{inR}(e)\ \textbf{as}\ \tau \mid \\
&\quad \textbf{valL}(e) \mid \textbf{valR}(e) \mid \{lab_1 := e_1, \ldots, lab_k := e_k\} \mid \#lab(e) \mid f(e_1, \ldots, e_m) \mid \\
&\quad [\,]\ \textbf{as}\ \tau \mid e[e'] \mid e[e_1 \rightarrow e_1'] \mid \\
&\quad \{\}\ \textbf{as}\ \tau \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid \textbf{foreach}\ (a, b \rightarrow e_1)\ e_2\ e_3 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ \textbf{end}
\end{aligned}$$

where $k \geq 1$, $m \geq 0$, $n \in N$, $r \in R$, $x \in Var$ and $f \in FVar$.

Before we continue to the type-system and the semantics, let us make an *informal* description of the language constructs.
Simple Constants: n denotes integers, r denotes reals, $yyyy - mm - dd$ denotes dates and **true** and **false** are the boolean values.
Arithmetic operators: $e_1 + e_2, e_1 - e_2, e_1/e_2$ and $e_1 * e_2$ are the usual arithmetic expressions the operators and can only be applied to integers and reals.
Logical operators: e_1 **and** e_2, e_1 **or** e_2 and **not** e has the usual semantics and these operators can only be applied to boolean expressions.
Comparison operators: $e_1 = e_2$ denotes equality of e_1 and e_2 and equality can only be applied to expressions of type integer, real, boolean or date. $e_1 <= e_2$ and $e_1 < e_2$ are the

"less than equal" and "less than" operators and they can only be applied to integers and reals.
Sum-type construction/destruction: The **inL**(e) **as** τ and **inR**(e) **as** τ are the sum-type constructors. τ should have the form $\tau_1 + \tau_2$ and **inL**(e) **as** $\tau_1 + \tau_2$ constructs something of type $\tau_1 + \tau_2$ if e has type τ_1. Symmetrically for **inR**(e). **valL**(e) returns the value v if e evaluates to **inL**(v). Symmetrically for **valR**(e).
Record construction/destruction: As usual $\{lab_1 := e_1, \ldots, lab_k := e_k\}$ denotes the construction of a record with fields $lab_1, \ldots, lab_k$ and $\#lab(e)$ returns the lab field of e if e evaluates to a record with a field named lab.
Function Application: As usual $f(e_1, \ldots, e_n)$ denotes the application of function f on arguments $e_1, \ldots, e_n$.
Multi-sets: $\{\}$ **as mset**(τ) denotes the construction of an empty multi-set, where added elements should have type τ. e_1 **with** e_2 means the resulting multi-set of adding element e_2 to the multi-set e_1. e_1 **inter** e_2, e_1 **union** e_2 and e_1 **diff** e_2 means the intersection, union and difference of multi-sets e_1 and e_2. e_1 **in** e_2 returns whether or not element e_1 is in multi-set e_2 and e_1 **subset** e_2 return whether or not multi-set e_1 is a subset of multi-set e_2.
Finite Map: [] **as map**(τ_1, τ_2) denotes the construction of an empty finite map from elements of type τ_1 to elements of type τ_2. $e[e']$ is the lookup operation on a finite map, ie. if e' evaluates to v' and e evaluates to a finite map with a binding on v then the binding of v is returned. $e[e_1 \rightarrow e_2]$ denotes the update operation on finite maps, ie. the finite map e is updated (overwritten if a binding already exists) with the binding of e_1 to e_1'. **dom**(e) returns a multi-set consisting of all elements in the domain of the finite map e and **toSet**(e) returns a multi-set of pairs, where each pair denotes a binding from argument to value in the finite map e.
Conditional: **if** e_1 **then** e_2 **else** e_3 denotes the usual conditional expression.
Iteration: **foreach** $(a, b \rightarrow e_1)$ e_2 e_3 is like the usual *fold-left* from SML (see [3][p. 145-148]). Hence $a, b \rightarrow e_1$ should be viewed as a *lambda* expression, ie. an anonymous function with arguments a and b and function body e_1. b is the accumulating parameter with starting value e_2 and the anonymous function is then folded over the multi-set e_3.
Let: **let** $x = e_1$ **in** e_2 **end** denotes the computation of e_1, where the result is bound to x and x can then be used in e_2.
Fixme: Sig at man skal bruge kommutative funktioner idet der anvendes multisets. Fortæl hvorfor sproget ikke indeholder lister i stedet (lettere inkrementalisering).
Furthermore we define some extra constructs on finite maps, which are just syntactic sugar on some of the existing constructs:
Syntactic sugar: $[e_1 \rightarrow e_1', \ldots e_n \rightarrow e_n']$ **as map**(τ_1, τ_2) is just syntactic sugar for ([] **as map**$(\tau_1, \tau_2))[e_1 \rightarrow e_1'] \ldots [e_n \rightarrow e_n']$ and $e[e_1 \rightarrow e_1', \ldots, e_n \rightarrow e_n']$ is syntactic sugar for $e[e_1 \rightarrow e_1'] \ldots [e_n \rightarrow e_n']$.
Now we can define the syntax for FunSETL programs.

Definition 3.3 *Program Syntax.*
The production fdecl describes FunSETL function declarations and p describes FunSETL programs.

$$\begin{aligned} fdecl &::= \textbf{fun}\ id(x_1 : t_1, \ldots, x_k : t_k) = e \\ p &::= fdecl_1 \ldots fdecl_n \end{aligned}$$

where $k \geq 0$ *and* $n \geq 1$.

A FunSETL program is a series of *function declarations*. In reality we would like a program to be a series of function declarations together with an expression, that can make use of the declared functions. So typically we will refer to a series of declarations and an expression to be a program, but in order to handle theoretical issues better the definition of a program is just a series of function declarations.

4 Incremental functions

In this section we will see what is meant by a functions *incremental counterpart*. This will be described together with a few examples on incremental programs.
Furthermore it will described why incremental programs are interesting in relation to ERP systems.

4.1 Definition of incremental functions

Let us first define what is meant by an incremental function:

Definition 4.1 *Incremental function.*
Assume f is a function and $\oplus$ an update operation then a program f' that computes $f(x \oplus y)$ by making use of the value of $f(x)$ is called an incremental function (incremental version of f with respect to the update operation $\oplus$). Furthermore we also allow that the incremental version makes use of the original input. Hence we want the following implication to hold for all x, all y and all r:

$$r = f(x) \quad \Rightarrow \quad f'(x, y, r) = f(x \oplus y)$$

The definition above may be extended, because in some cases the intermediate results of the computation of $f(x)$ can be used in an incremental computation of $f(x \oplus y)$, but for now it is sufficient with the above definition.
Furthermore the definition says nothing about the running time of the functions and their incremental counterparts, but the idea is that by incrementalizing a function we should gain an *asymptotic speedup* in the computation time when we try to compute $f(x \oplus y)$ from $f(x)$.
Let us now see an example of function and its incremental counterpart.

Example 4.2 *Below we see a function sum which computes the sum of the elements of a multi-set of integers.*

1: **fun** $sum(numbers : \mathbf{mset}(\mathbf{int})) = \mathbf{foreach}\ (a, b \rightarrow a + b)\ 0\ numbers$

We want to incrementalize this function such that, if we add one element to the multi-set of integers then we can compute the sum of all the elements in the new multi-set, based on the new element and the sum of all the old elements.
Ie. we incrementalize with respect to the update operation **with** *on multi-sets and get the following function*

1: **fun** $sum'(numbers : \mathbf{mset}(\mathbf{int}), y : \mathbf{int}, r : \mathbf{int}) = r + y$

Let us now discuss the example *informally*.
Using any reasonable measure of running time of FunSETL programs, we would expect that the *sum* function has running time proportional to the number of elements in the multi-set. Furthermore we would expect that the sum' function can be computed in constant time, since we only have one arithmetic operation, ie. we have gained an asymptotic speedup in computation time when trying to compute $sum(x\ \mathbf{with}\ y)$ if we already know the value of $sum(x)$ (otherwise we have gained nothing).

4.2 Automatic incrementalization

As we saw above, we could hand code an incremental version of a function with respect to some update operation, but it would be really interesting, if we could *automatize* the process

of incrementalization of FunSETL functions.
There has been made a lot of work in the field of incrementalization of programs, where *semantics preserving transformations* are used to construct the incremental counterpart of a function. Among the most interesting work can be mentioned [4] and [5].
In [4] there is a description of an incrementalization process for a functional language with mutual recursion, ie. a language which is more general that FunSETL. The incrementalization process is split in three steps:

i Caching all intermediate results.

ii Incrementalization

iii Pruning

The incrementalization process incrementalizes a program with respect to a given operation.
ad i: In the hand coded example we saw earlier, we did not use any intermediate results, but they can often be used when trying to make a more efficient version of a program, fx. when computing the average of a multi-set of integers. When adding an element to the multi-set we are not able to compute the new average from the old average and the recently added element. Hence this part of the process produces from a function f a new function $\hat{f}$, which returns the return value of f together with the intermediate results computed by f.
ad ii: This step incrementalizes the function $\hat{f}$ from the previous step, with respect to an update operation $\oplus$. This step is performed by doing semantics preserving program transformations and by using the intermediate results. The result of this stage is called $\hat{f}'$
ad iii: In this stage the intermediate results, which are not used by $\hat{f}'$ will be removed producing a new function f', which is a incremental version of f returning only the necessary intermediate results in order to compute efficiently.

Steps i and iii can be made fully automatic, but step ii is a bit more involved. Depending on how powerful the incrementalizer engine will be made it will vary from automatic to semi-automatic. Our hope is that because we have chosen a language without full recursion then it will be possible to make an incrementalizer which is very powerful and fully automatic.

4.3 Incremental functions in ERP systems

In this section we will discuss why incremental computations is interesting in relation to ERP systems.
ERP systems contains a lot of reports that presents the computation of some analytical/financial function. Typically these functions are computed by iterating over large collections of data (usually all the rows from a big database table), and it is suitable to make incremental versions of functions like these.
In section 5 we will see an implementation of the Microsoft Dynamics AX financial statement in FunSETL. There are many advantages when using incremental computations and automatic incrementalization. Among the most important we have:

- Efficiency improvement on programs.
- Simpler looking programs.

- Reduce the number of errors induced by humans.
- Gain reduction in programming time.

As we have seen in the small example, a program running in linear time have an incremental counterpart, which run in constant time. This seems to be the case with many computations in ERP systems, since many reports does not have internal dependencies between the data when a report is computed. This means that, if the incremental counterpart is used we can get *real time* computations of many reports in ERP systems, which today is usually computed by a nightly bachrun. Hence if the incremental programs are used, one will also be able to set *alerts* and put *triggers* on reports, since they are computed in *real time* and *on the fly*. Furthermore if automatic incrementalization techniques are used, another side-effect will be that we get simpler looking programs, because the *incrementalizing* software can be used as a front-end to the compiler and hence we do not need to look at the incrementalized code. Automatic incrementalization will probably also lead to fewer programming errors and a reduction in programming time, since the non-incremental programs are often easier to write.

5 Application of FunSETL

This section will show how the Financial statement from Microsoft Dynamics AX can be implemented in FunSETL.

5.1 Microsoft Dynamics AX Financial Statement

The Microsoft Dynamics AX ERP system contains a database with a lot of tables, and the Financial statement makes use of only a couple of these tables.
The Financial statement is in principle an aggregate function, which aggregates information on different financial accounts. Dynamics AX has an accounting system where accounts are numbered from 0 and up. The financial statement computes the following information

- Sumclass computations (balance of account intervals $X000-X999$, where $X = 1, 2, 3, 4, 5, 6, 7, 8$ and from 9000 and up. These summation data are named $SumclassX$ for each X.
- Assets and liabilities
- Other summation data.

All these numbers have in common that they are aggregate information on the Microsoft Dynamics AX table, which contains all transactional data on the different accounts in the system.

5.2 Implementation of Financial Statement in FunSETL

In Section B, page 13 we see an implementation of the *financial statement function* which is computed by Microsoft Dynamics AX 4.0.
The FunSETL implementation is based on the code which can be viewed from *Microsoft Dynamics Application Object Tree*.
However we have made a couple of simplifications, since the database table containing all the transactional data contains more columns than necessary when computing the Financial statement.
Therefore we have made a projection on the transaction table, such that each transaction only contains $accountnr, amount$ and $date$. $accountnr$ is the number of the account where there has been made a transaction of $amount$ on $date$.

5.3 Data used in testing

We have retried a real dataset from a company, where the transaction table contains more than 200.000 entries.

6 Summary & Future work

In this paper we have given a rough sketch of the architecture of the next generation of ERP systems.
Particularly we have been interested in the *Data analysis* aspect of the ERP system, and how it should work. Therefore this paper proposes the next generation of ERP systems should have the following properties:

i There system should contain a domain specific language to express reports, and the language should be powerful enough to eliminate the need for general purpose languages.

ii The system should support real time access to all business intelligence.

ad i: This paper proposes the language FunSETL described in Section 3, which is thought to have the desired properties. In section 5 we made an empirical study and implemented an existing ERP system report using FunSETL and used the implementation of a real dataset.
ad ii: This is the most important property of the system. Microsoft Navision contains some aspects of this, since some reports are maintained using the SIFT technology. We have proposed a generalization of the SIFT technology.

Currently there exist an implementation of a compiler from FunSETL to C#, ie. FunSETL runs on .NET. The next steps in the development of the next generation ERP system will be

- Implement a prototype of the architecture.
- Develop incrementalization tools.

A Literature

References

[1] [GPO+06] Inside Microsoft Dynamics AX 4.0
Arthur Greef, Michael Fruergaard Pontoppidan, Lars Dragheim Olsen, and experts from the Microsoft Dynamics AX team
ISBN: 0-7356-2257-4
Microsoft Press

[2] http://www.consultec.es/DocTutoriales/Introduction_to_CAL_Programming.pdf

[3] [HR99] Introduction to Programming Using SML
Michael R. Hansen & Hans Rischel
ISBN: 0-20139820-6
Biddles Ltd., Guildford and King's Lynn

[4] [LST98] Static Caching for Incremental Computation
Yanhong A. Liu, Scott D. Stoller & Tim Teitelbaum
ISSN: 0164-0925
ACM Press

[5] [RP81] Formal Differentiation: A Program Synthesis Technique
Robert Paige
ISBN: 0-83571213-3
UNI Research Press

B Financial statement

```
type event = {accountnr : int, amount : real, time : date}
type eventset = mset(event)
type interval = {startdate : date, enddate : date}
type classmap = map(int,real)

fun inInterval(e : event, I : interval) =
        #startdate(I) <= #time(e) and #time(e) <= #enddate(I)

fun aktiv(e : event, I : interval) =
        let
                accnr = #accountnr(e)
        in
                if ((0 <= accnr and accnr <= 449) or
                    (451 <= accnr and accnr <= 539) or
                    (541 <= accnr and accnr <= 679) or
                    (681 <= accnr and accnr <= 2999)) and
                   inInterval(e,I)
                then #amount(e)
                else 0.0
        end

fun aktiver(E : eventset, I : interval) =
        foreach (a,b => b + aktiv(a,I)) 0.0 E


fun passiv(e : event, I : interval) =
        let
                accnr = #accountnr(e)
        in
                if ((3000 <= accnr and accnr <= 3999) or
                    (9000 <= accnr and accnr <= 999999)) and
                   inInterval(e,I)
                then #amount(e)
                else 0.0
        end


fun passiver(E : eventset, I : interval) =
        foreach (a,b => b + passiv(a,I)) 0.0 E

fun aufwandSub(e : event, I : interval) =
        let
                accnr = #accountnr(e)
        in
                if (accnr = 450 or
                    accnr = 540 or
                    accnr = 680 or
                    (5000 <= accnr and accnr <= 7999) or
                    (8110 <= accnr and accnr <= 8299) or
                    (8351 <= accnr and accnr <= 8357) or
                    accnr = 8500) and
                   inInterval(e,I)
                then #amount(e)
                else 0.0
        end

fun aufwand(E : eventset, I : interval) =
        foreach (a,b => b + aufwandSub(a,I)) 0.0 E

fun ertragSub(e : event, I : interval) =
        let
                accnr = #accountnr(e)
        in
                if ((4000 <= accnr and accnr <= 4999) or
                    (8010 <= accnr and accnr <= 8100) or
                     accnr = 8300 or
                     accnr = 8400 or
                     accnr = 8450 or
                    (8600 <= accnr and accnr <= 8970)) and
                   inInterval(e,I)
                then #amount(e)
                else 0.0
        end

fun ertrag(E : eventset, I : interval) =
```

```
        foreach (a,b => b + ertragSub(a,I)) 0.0 E


fun bestandsKontenSub(e : event, I : interval) =
        let
                accnr = #accountnr(e)
        in
                if ((0 <= accnr and accnr <= 449) or
                    (451 <= accnr and accnr <= 539) or
                    (541 <= accnr and accnr <= 679) or
                    (681 <= accnr and accnr <= 3999)) and
                   inInterval(e,I)
                then #amount(e)
                else 0.0
        end

fun bestandsKonten(E : eventset, I : interval) =
        foreach (a,b => b + bestandsKontenSub(a,I)) 0.0 E

fun erfolgsKontenSub(e : event, I : interval) =
        let
                accnr = #accountnr(e)
        in
                if ( accnr = 450 or
                     accnr = 540 or
                     accnr = 680 or
                    (4000 <= accnr and accnr <= 8999)) and
                   inInterval(e,I)
                then #amount(e)
                else 0.0
        end

fun erfolgsKonten(E : eventset, I : interval) =
        foreach (a,b => b + erfolgsKontenSub(a,I)) 0.0 E



fun getClass(e : event) =
        let
                accnr = #accountnr(e) / 1000
        in
                if accnr <= 9
                then accnr
                else 9
        end

fun SumClassesSub(e : event, I : interval, f : classmap) =
        if inInterval(e,I)
        then let
                c = getClass(e)
             in
                f[c -> f[c] + #amount(e)]
             end
        else f

fun SumClasses(E : eventset,I : interval) =
        foreach (e,f => SumClassesSub(e,I,f))  ([0 -> 0.0, 1 -> 0.0, 2 -> 0.0,
                                                 3 -> 0.0, 4 -> 0.0, 5 -> 0.0,
                                                 6 -> 0.0, 7 -> 0.0, 8 -> 0.0,
                                                 9 -> 0.0] as classmap) E



fun sumAmountsSub(e : event, I : interval) =
        if inInterval(e,I)
        then #amount(e)
        else 0.0

fun sumAmounts(E : eventset, I : interval) =
        foreach (e,s => s + sumAmountsSub(e,I)) 0.0 E

fun nrOfEventsSub(e : event, I : interval) =
        if inInterval(e,I)
        then 1.0
        else 0.0

fun nrOfEvents(E : eventset, I : interval ) =
        foreach (e,c => c + nrOfEventsSub(e,I)) 0.0 E
```

```
    fun getReturn(aa : real, ab : real) =
            if ab = 0.0
            then 0.0
161         else aa / ab

    fun averageAmount(E : eventset, I : interval) =
            let suma = sumAmounts(E,I) in
            let nre = nrOfEvents(E,I) in
166                 getReturn(suma,nre)
            end end


    fun FinancialStatement(E : eventset, I : interval) =
171         let f = SumClasses(E,I) in
            let aktiver = aktiver(E,I) in
            let passiver = passiver(E,I) in
            let aufwand = aufwand(E,I) in
            let ertrag = ertrag(E,I) in
176         let bestandskonten = bestandsKonten(E,I) in
            let erfolgskonten = erfolgsKonten(E,I) in
            let average = averageAmount(E,I) in
            {SumClass0 := f[0],
             SumClass1 := f[1],
181          SumClass2 := f[2],
             SumClass3 := f[3],
             SumClass4 := f[4],
             SumClass5 := f[5],
             SumClass6 := f[6],
186          SumClass7 := f[7],
             SumClass8 := f[8],
             SumClass9 := f[9],
             Aktiver := aktiver,
             Passiver := passiver,
191          Aufwand := aufwand,
             Ertrag := ertrag,
             BestandsKonten := bestandskonten,
             ErfolgsKonten := erfolgskonten,
             Average := average}
196         end end end end
            end end end end


    let sdate = {startdate := 2007-05-05, enddate := 2007-06-05} in
201 let eset = {{accountnr := 12, time := 2010-05-07, amount := 15.0},
                           {accountnr := 1200, amount := 17.0, time := 2007-05-07},
                           {amount := 19.0, accountnr := 2212, time := 2007-05-07},
                           {amount := 150.0, time := 2007-05-07, accountnr := 3000},
                           {time := 2007-05-07, accountnr := 4000, amount := 15.0},
206                        {time := 2007-05-07, amount := 15.0, accountnr := 5000},
                           {accountnr := 6000, amount := 15.0, time := 2007-05-07},
                           {accountnr := 7000, amount := 15.0, time := 2007-05-07},
                           {accountnr := 8000, amount := 15.0, time := 2007-05-07},
                           {accountnr := 9000, amount := 15.0, time := 2007-05-07},
211                        {accountnr := 1201, amount := 7.5, time := 2007-05-07}} as eventset in
        FinancialStatement(eset, sdate)
    end
    end
```