# Reporting for Enterprise Systems

Michael Nissen

October 30, 2008

# Contents

**Preface** **iv**

**Resumé** **v**

**Abstract** **vi**

**Acknowledgments** **vii**

**1 Introduction** **1**
1.1 Background and Motivation . . . . . 1
1.2 Goal . . . . . 3
1.3 Overview . . . . . 4

**2 Survey: Reporting Technologies** **7**
2.1 Introduction and Background . . . . . 8
2.2 Relational Databases . . . . . 9
2.2.1 Relational Algebra . . . . . 10
2.2.2 SQL . . . . . 12
2.2.3 Relation to Report Function Computation . . . . . 13
2.3 Relational Database Extensions . . . . . 14
2.3.1 Materialized Views . . . . . 14
2.3.2 OLAP . . . . . 15
2.3.3 Data Warehouses . . . . . 20
2.3.4 Microsoft NAV SIFT . . . . . 22
2.4 Alternative Computational Models . . . . . 24
2.4.1 Streaming Databases . . . . . 24
2.4.2 Map-Reduce and Map-Reduce-Merge . . . . . 28
2.5 Specification of Report Functions . . . . . 31
2.5.1 Microsoft NAV C/AL . . . . . 32
2.5.2 Microsoft AX X++ . . . . . 32

2.5.3 LINQ . . . 33
2.5.4 Relation to Specification of Report Functions . . . 34
2.6 Summary & Future Work . . . 35

**3 Paper: Functional Reporting For ERP Systems 37**
3.1 Motivation . . . 37
3.2 Introduction To FunSETL . . . 39
3.3 Incrementalization . . . 41
3.3.1 Limitations of Our Approach . . . 45
3.4 Implementation . . . 45
3.4.1 Incrementalizer . . . 45
3.4.2 Compiler . . . 46
3.5 Experimental Results . . . 47
3.5.1 Benchmark Suite . . . 47
3.5.2 Financial statement . . . 47
3.5.3 The Data-set . . . 48
3.5.4 Setting up the experiment and expectations . . . 49
3.5.5 Performing and analyzing the experiment . . . 50
3.6 Related Work . . . 53
3.7 Conclusion . . . 53
3.8 Financial statement . . . 54

**4 Technical Report: Finite Differencing 55**
4.1 Finite Differencing of Polynomials . . . 55
4.1.1 Polynomials and Their Degree . . . 55
4.1.2 The Difference Operator . . . 56
4.1.3 Incremental Computation of Polynomial Values . . . 59
4.2 Difference Calculus . . . 62
4.2.1 Incremental Preorders . . . 62
4.2.2 Preorder on Value Sets . . . 66
4.2.3 Difference . . . 66
4.2.4 Integrals . . . 70
4.2.5 Chain Rule . . . 74
4.2.6 Taylor Expansion . . . 77
4.2.7 Incremental Computation Using Differences . . . 79
4.2.8 Comparison to Finite Differencing . . . 82
4.3 Summary and Future Work . . . 83

**5 Sketch: Concrete Difference Calculus** **86**
5.1 Introduction . . . 86
5.2 Syntax of FunDIF . . . 88
5.3 Semantics of FunDIF . . . 90
5.4 Type System of FunDIF . . . 92
5.5 Differencing FunDIF Functions . . . 93
5.6 Incremental Computation Using Differences . . . 99
5.7 Correctness Of The Difference Map . . . 99
5.8 Related Work . . . 100
5.8.1 Deforestation and Dynamic Programming . . . 100
5.8.2 FunSETL . . . 101
5.8.3 Finite Differencing . . . 101
5.8.4 Memoization . . . 106
5.8.5 Magic Sets Transformation . . . 108
5.8.6 Incremental View Maintenance . . . 109

**6 Summary and Future Work** **110**

**Bibliography** **112**

# Preface

This document is my *progress report*, which concludes part A of my 4+4 year Ph.D. program. A progress report counts as a Master Thesis (30 ECTS), but it is not a requirement that it should have the same form as a Master Thesis. The purpose of a progress report is to present the current status of the research. Thus, a progress report is not required to be a finished piece of work, to have a conclusion or to even to be self contained.

Since, a Ph.D. student is a "trainee" with respect to becoming a researcher, I have chosen to write up my research as papers and technical reports. Thus, there are open questions that I hope answer in part B of my 4+4 year Ph.D. program (or at least answer some of them).

This report contains a survey, a paper and a technical report. Furthermore, it contains a sketch of the work I am involved in right now, which is heavily based on the technical report. The survey, the paper and the technical report are finished pieces of work and can be read independently of each other, but this is not the case with the sketch work. Since, the different parts of this report can be read independently of each other there is some overlap in the content.

The introduction explains, what the overall goal of my research is and thereby sets up a context for the different parts of the report.

Understanding the content of this document requires that the reader is familiar with functional programming, databases, incremental computation, program transformation, lambda calculus, type systems, operational semantics, complexity theory, algebra, induction and finite calculus. That is, it is recommended that the reader has knowledge corresponding to a Masters degree in computer science and a bachelors degree in math from the University of Copenhagen.

# Resumé

Denne rapport omhandler rapportering i Enterprise Systemer. For yderligere information henvises til det engelske resumé.

# Abstract

One of the essential features of Enterprise Systems is the ability to provide businesses with reports on, how the business is running and to assist businesses in providing the authorities with the reports required by law.

Over the last couple of decades there has been an increase in the amount of information registered by Enterprise Systems and thus reports typically depend on a huge amount of data. However, businesses still need the reports in a timely manner, preferably in real-time (or near-real-time) to help monitor the business and to satisfy the legal requirements.

This document contains a survey, a paper, a technical report and some sketch work (four parts). Each part addresses a practical or theoretical issue on the path to solving the overall problem of real-time computation of report functions.

A summary of the contributions of the four parts are:

- We give a survey of the existing technologies used for expressing report functions and doing report function computations. Furthermore, we explain why these technologies does not have good support for real-time computation.
- We show that automatic incrementalization of report functions written in the programming language FunSETL, can give an asymptotic improvement, as well as a practical speedup in the computation time.
- We propose a generalization of *finite differencing* to list functions, and show how this can be used to construct incremental functions, which can lead to real-time computation.
- We sketch how the generalization of finite differencing to list functions, can be used to automatically incrementalize functions in a programming language called FunDIF.

# Acknowledgments

I would like to thank my advisers Professor Fritz Henglein and Associate Professor Ken Friis Larsen for all the good ideas, comments and valuable discussions we have had during this project. Furthermore I would like to thank my wife Sidsel Helle Boesen for her moral support.

# Chapter 1

# Introduction

## 1.1 Background and Motivation

Computer assistance in the business world has been used for several decades to manage and analyze businesses. Increasing competition in the business field has dictated the need for better tools to complete these tasks.

The purpose of this collection of documents is to look into a single area of computer assistance for businesses called *reporting*. Reporting falls under the category of analyzing businesses, which means computing derived information based on the available data. Examples are: Financial statement, inventory value or maybe something more advanced to predict future market trends.

Within the last couple of decades the amount of information stored by businesses has exploded. Thus, it can be time consuming to compute report functions, even the conceptually simple ones. However, businesses often need the results of the report functions fast or preferably in real-time (or near-real-time). The time consumption stems from the fact that we often need to run through all the stored data to compute a report function. However, this is a naive approach, because businesses primarily accumulate data, that is, when data is stored, it will not be altered or deleted. This means that one often traverses the same data when computing the same report function at different points in time.

Custom report functions take a lot of time to implement, because typically non-domain experts of the business (programmers) implement them and therefore it also becomes error-prone. Thus, there is a substantial cost involved for businesses, when implementing custom report functions in their Enterprise Systems. It is of vital importance that these report functions

becomes easier to implement (correctly) and that they compute efficiently.

ERP is an acronym for *Enterprise Resource planning* and ERP systems are used by businesses to manage and analyze their business. Examples of ERP system are the Microsoft Dynamics systems and SAP. Common to all ERP systems is that they contain some technology to do *reporting*. We use the term ERP system instead of Enterprise System, even though this document is only concerned with reporting and not with resource planning.

The term reporting is used in many different contexts and has different semantics depending on the context. However, we are only interested in using a specific definition of reporting. First we define, what we mean by *report function*, and it is this definition we use throughout the document.

**Definition 1.1.1 (Report Function)** *A report function is a side effect free, PTIME computable function on transactional data.*

Let us briefly discuss the components of the definition. Transactional data describes *events*, that is, changes as a result of a transaction in a business (representation of a "real world" event). However, we also include changes to *Master data*, that is, changes to customer records, personal records etc. as transactional data. Furthermore, we do not restrict our work to *only* functions on transactional data. Report functions usually take a large collection of events as input. Thus, we have included the PTIME (polynomial time) computability limitation, because we might as well limit our attention to this class of functions, because report functions should be feasible to compute. The requirement that a report function be side effect free, that is, the function only returns a result of the function computed, is because running a report function should not change the system data.

We have chosen this rather general definition of report function, because we have seen other definitions including also the graphical presentation of the result of a report function, which implicitly assumes that we have a system that can print numbers and other information on the screen. In our opinion, including stuff like graphical presentation of results makes the definition to specific.

*Reporting* is the discipline of

- *Applying* report functions, that is, executing their specification on actual data.
- *Expressing* report functions. By expressing report functions we mean describe them in a specification- or programming language.

That is, reporting consists of both expressing report functions in a formalism and computing the result of report functions applied to data. Today most ERP systems support a variety of built in report functions and features for expressing new report functions. However, report functions are typically expressed in an imperative programming language and for very simple report functions in a graphical user interface. Typically, report functions are computed from scratch, that is, the report function specifications are executed against the data in the ERP system database.

Companies use a substantial part of the total cost of ownership (TCO) on expressing report functions, which includes ad-hoc implementations of new report functions and getting existing report functions to compute faster[1]. Since performance is critical and because the amount of data has grown very large, we need to address the following issues:

- Computation time of report functions should be decreased as many report functions have become very time consuming to compute and others have become practically infeasible to compute.
- Expressing report functions should be easier and it should not be the programmers responsibility that the report functions are computed efficiently.

## 1.2 Goal

Today, report functions are typically expressed in an imperative programming language that computes the report function from scratch when it is applied. Our hypothesis is:

**Hypothesis 1** *Using a strongly normalizing (all programs terminate) declarative programming language for expressing report functions, where automatic incrementalization is used to optimize these, can in many cases yield real-time (or near real-time) computation of report functions.*

Since many report functions are time consuming to compute, it can be hard to guess if the computation of a report function loops indefinitely or it is about to terminate. Thus, the strongly normalizing (termination) criteria of the programming language is a desirable practical property, because then we know for sure that the computation terminates.

[1] 25 % of TCO for reporting was mentioned by Niels Bjørn Andersen, Professor at CBS on an internal 3gERP meeting.

The formalism for expressing report functions does not need to be imperative, because then the formal specification of a report function will be cluttered with, *how* the report function is computed. The idea is that the programmer only specifies, *what* should be computed and not *how* it is done. That is, we remove the responsibility that the report function is computed efficiently away from the programmer.

Many of the computations in a report function can be reused in the next computation of the same function, because report functions often are simple functions computing summarizing information on a large dataset, where only data is added. Furthermore the size of the result is typically asymptotically smaller or constant in the size of the input. Thus, we would like to automatically create incremental specifications of report functions from non-incremental specifications (automatic incrementalization). Using the incremental specifications, we want to maintain the result of the report functions, when new data is added to the ERP system database. Thus

- Maintaining the result of the report function incrementally should in many cases yield up to date, real-time (or near-real-time) access to the results of the report functions.
- Using automatic incrementalization should remove the responsibility of expressing efficient report functions away from the programmer, which also should decrease the expenses for businesses for doing reporting.

## 1.3 Overview

We have chosen to start building an argument for Hypothesis 1 using both a *top down* and *bottom up* approach. Top down approach means that we look at existing technologies for computing report functions and see, what their reporting capabilities are. The bottom up approach means that we look at the theory for incremental computation and try to see, how this can be applied to give more efficient computations.

This document contains four components, and we have listed them on a scale from practical to theoretical, which can be seen in Figure 1.1. Each box in Figure 1.1 refers to a component in this document and every component is covered in its own chapter.

Chapter 2 (Reporting Technologies) contains a survey of reporting technologies and a description of, what their shortcomings are with respect to real-time computation of report functions. Furthermore it contains a

summary of the theory of some technologies which could be applied in the ERP system field, and a description of how report functions are declared today. The survey has been accepted for the 2nd 3gERP workshop, which is held November 17-18, 2008, at Copenhagen Business School (http://www.3gerp.org/workshop/default.aspx).

Chapter 3 (Functional Reporting for ERP Systems) uses an implementation of FunSETL, which is a small functional programming language, to show that automatic incrementalization can yield asymptotically and practically faster versions of report functions. The paper was presented at the TFP 2008 conference, which was held 26-28 May, 2008, in Nijmegen, The Netherlands (see http://www.st.cs.ru.nl/AFP_TFP_2008/) and is appearing in this document with minor changes. The paper is joined work with my adviser Ken Friis Larsen.

Chapter 4 (Finite Differencing) is a technical report on Finite Differencing, which shows that we can create incremental versions of a class of *monotonic* functions by using *difference* functions. This is joined work with my adviser Fritz Henglein.

Chapter 5 (Concrete Difference Calculus) is ongoing work on creating a calculus that uses the theory of Chapter 4 to *automatically* create incremental functions from non-incremental specifications. This is joined work with my adviser Fritz Henglein.

Thus, the first two chapters show that the technologies used today, are not suited for real-time computation of report functions, and that automatic incrementalization of declarative specifications looks promising with respect to real-time computation of report functions. However, the FunSETL programming language lacks ordered data structures and the power of the automatic incrementalization is limited. Therefore, we try a new approach for constructing incremental functions, which is covered by the last two chapters.

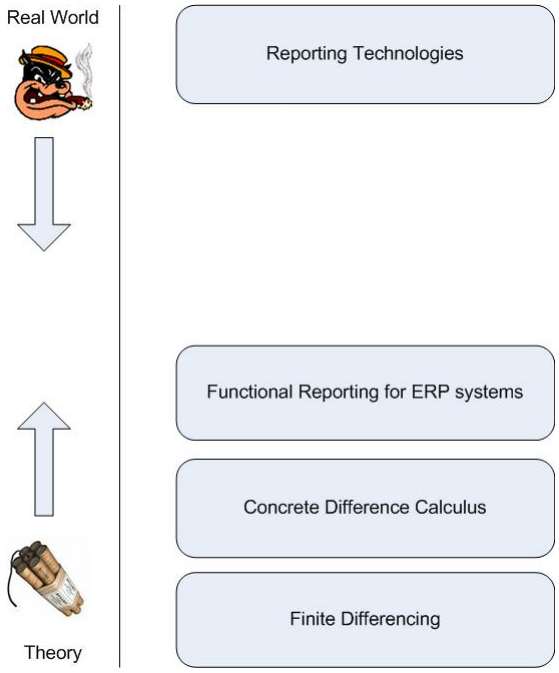


Figure 1.1: Components in this document

# Chapter 2

# Survey: Reporting Technologies

**Abstract:** Report functions are functions computed on the transactional/historical data of an ERP (Enterprise Resource Planning) system database. The purpose of computing report functions is to get information about, how a business is running. Performance of the computation is an important issue, because faster computation of report functions can yield a competitive advantage. In many cases *real-time* computation is preferable, because this enables *dashboard* (real-time) monitoring of the business. Todays ERP systems store data in relational databases and uses some of the modern capabilities of relational databases like OLAP, when computing report functions. However, relational databases does not provide good support for real-time computation of queries.

The first contribution of this paper is to present some of the theory for relational databases together with Materialized Views, OLAP, Data warehousing and SIFT, and explain why these technologies are interesting in relation to reporting in ERP systems, but also what their shortcomings are with respect to real-time computation.

The second contribution is that we present other kinds of *realized* technologies and explain, how they differ from relational databases, their strengths and weaknesses compared to relational databases, why they could be used to do reporting and in which degree they can be used to compute report functions in real-time.

The third contribution is to describe the trend in technologies for report function *declaration*, that is, how are report functions expressed and why this is a good way.

## 2.1 Introduction and Background

First of all it should be noted that this survey is *not* a product survey. That is, this is not a description of the available products for doing reporting, but an explanation of the technologies that the mainstream products rely on. However, we include some examples of products implementing the technologies we describe. Furthermore it should be noted that when we use the term report function or reporting, it has nothing to do with the presentation of the result of a computation, but only the computation itself and the specification of the computation.

The theory of relational databases has existed for many years and it is a well explored area of research. There exist numerous implementations of relational databases, which are highly optimized with respect to both storing and querying data. However, the relational model on which relational databases are build, does not contain any notion of time. That is, the temporal aspect of the change of data versus querying data is not natively build into the model. This means that to some degree data is *static* and queries are *dynamic*, because a query in basic SQL is always applied to a relation instance at a certain point in time.

ERP systems have also existed for many years, but it is within the last couple of decades that the amount of information recorded by ERP systems has exploded. Information is added every day, every hour, every minute and every second depending of the size and kind of the business. However, businesses often know in advance many of the queries they want answers of, like a financial statement or inventory value computation. Thus, the idea that data is static and querying is dynamic does not entirely fit todays ERP systems. Of course businesses also need the possibility of ad hoc querying (dynamic querying) the data stored in the ERP system, but not all queries need to be computed in this way. OLAP is in practice a widely used technology to do reporting in businesses. However, it is in the authors opinion that OLAP is not, what businesses really want, but it is the solution that has been provided through the relational database community, because it was a *best effort* attempt to accommodate the ERP systems with extra tools for doing reporting, which did not contradict the underlying model of relational databases.

In this paper we explain the basic theory of relational databases (Section 2.2). Then we explain, the technologies *Materialized Views*, *OLAP*, *Data Warehousing* and *Microsoft NAV SIFT*, which can be considered relational database extensions and we explain, how they are related to ERP systems. Furthermore we present other *realized* technologies like *Streaming*

*Databases* and Google's *Map Reduce* and explain, how they can be used to do report function computations and how the technologies differ from the relational database technologies. Concretely we answer the following questions for all technologies:

1 How does it differ from relational databases.

2 Which problems and weaknesses does the technology address and introduce.

3 Why is the technology interesting with respect to report function computation.

4 Can the technology be utilized for real-time computation of report functions.

Each of the above questions are answered in a section called "Relation to Computing Report Functions", which is included for each technology (Section 2.3 and Section 2.4).

In the last part of the paper we switch context and purely consider the specification of the report functions. We explain the Microsoft NAV C/AL programming language, the Microsoft AX X++ programming language and finally LINQ from the .NET framework. Furthermore we explain, how these specification formalisms are related and argue that declarative specification of report functions is the way to go.

Before we continue the discussion of reporting technologies we would like to give an example of information recorded by a business, and then use this example in the following sections to explain the different technologies: Assume we have a chain of bicycle stores that for a sale, registers in which branch the sale has happened, which color the bicycle had, at what time the bicycle was sold and at what price. Figure 2.1 shows an example of information registered by the business. How this registration of information can be described more formally is explained in the next section.

## 2.2 Relational Databases

Relational databases are build on the relational model, which has mathematical relations as its corner stone. Queries are written in a query programming language called *SQL*, which is based on *relational algebra*. Silberschatz et al. [2002] and Lewis et al. [2002] provide comprehensive descriptions of relational algebra and SQL. In the following we give a short presentation

| Branch | Color | Time_Id | Price |
|---|---|---|---|
| Valby | Red | T1 | 1599 |
| Frederiksberg | Red | T1 | 1799 |
| Valby | Red | T2 | 1399 |
| Frederiksberg | Blue | T2 | 2199 |
| Valby | Red | T3 | 1299 |
| Frederiksberg | Blue | T3 | 1299 |
| Frederiksberg | Blue | T3 | 2399 |

| Time_Id | Month | Quarter |
|---|---|---|
| T1 | January | 1 |
| T2 | February | 1 |
| T3 | April | 2 |

Figure 2.1: Bicycle business

of both and explain, how relational databases are used in relation to report function computation. Examples of database systems are Oracle, IBM DB2 and Microsoft SQL Server.

### 2.2.1 Relational Algebra

Relational algebra consists of a number of operations on relations. In figure 2.1 we see two examples of relations, which we name *sale* (table to the left) and *time* (table to the right). The header of the table is called the *schema*, that is, the schema of the sale relation is $(Branch, Color, Time\_id, Price)$ and the components of the schema are called *attributes*. Furthermore each attribute has a domain of values. That is, the attribute $Branch$ has as domain the set of branches for the bicycle business, attribute $Color$ has as domain the set of colors, attribute $Time\_id$ has as domain the set of references to elements of the *time* relation and attribute $Price$ has as domain the set of possible prices for the bicycles.

A relational algebra operation takes as input one or more relations, depending on its arity and returns a relation, which means that relational algebra operations can be composed. Assume that $R_1$ and $R_2$ are relations with schemas $(A_{11}, \ldots, A_{1n_1})$ and $(A_{21}, \ldots, A_{2n_2})$, that is, $R_1$ and $R_2$ can either be constant relations or relations generated using relational algebra operations. The basic operations can be seen in Figure 2.2. The predicate language used by selections is not explained here, but the examples include predicates, which are legal relational algebra predicates and where the semantics is as expected. Let us see an example.

**Example 2.2.1** *Find all sales of blue bicycles sold in the second quarter and show the branch name, month of sale and price.*

- *Union*: $R_1 \cup R_2 = \{x \mid x \in R_1 \vee x \in R_2\}$.
- *Difference*: $R_1 - R_2 = \{x \mid x \in R_1 \wedge x \notin R_2\}$.
- *Product*:

  $R_1 \times R_2 = \{(x_{11}, \ldots, x_{1n_1}, x_{21}, \ldots, x_{2n_2}) \mid (x_{11}, \ldots, x_{1n_1}) \in R_1 \wedge (x_{21}, \ldots, x_{2n_2}) \in R_2\}$

- *Selection*: $\sigma_P(R_1) = \{x \mid x \in R_1 \wedge P(x)\}$, where $P$ is a predicate using comparisons $(=, \neq, <, \leq, >, \geq)$ and logical connectives $(\wedge, \vee, \neg)$.
- *Projection*: $\Pi_{A_{i_1}, \ldots, A_{i_k}}(R_1) = \{(x_{i_1}, \ldots, x_{i_k}) \mid (x_1, \ldots, x_{n_1}) \in R_1\}$, where $1 \leq i_j \leq n_1$ for all $1 \leq j \leq k$.

Figure 2.2: Basic relational algebra operations

The relational algebra expression is written as

$$\Pi_{sale.Branch, time.Month, sale.Price}(\sigma_{sale.Time\_id=time.Time\_id \wedge time.Quarter=2 \wedge sale.Color=blue}(sale \times time))$$

The resulting relation is

| Branch | Month | Price |
|---|---|---|
| Frederiksberg | April | 1299 |
| Frederiksberg | April | 2399 |

■

Note, that the sale and time relation both have a column called $Time\_id$, which should be considered a reference from the sale relation to the time relation. There exists a relational algebra operation called *join*, which can generate a new relation from the sale and time relation, where equality on the $Time\_id$ from the relations is performed and one of the columns is projected away. Since the join operation can expressed by the operations in Figure 2.2 it is not included here.

Until now, we have only been working with set relations which can not contain duplicates. However, in databases it is often necessary to be able to handle duplicates. In the above we have been "lucky" that the elements of the relations have staid unique even though we have selected and projected, but this is often not the case. Fortunately the above operations generalizes straightforward to multi-set relations. From now on the term *relation* means multi-set relation.

Furthermore there are some additional (extended) relational algebra operations called *aggregates*, which also apply to multi-sets. In extended relational algebra an aggregate on $R$ is expressed by

$$_{A_{i_1},\ldots,A_{i_{k_1}}}\mathcal{G}_{F_{j_1}(A_{j_1}),\ldots,F_{j_{k_2}}(A_{j_{k_2}})}(R)$$

where $R$ is a relation and $F_{j_h}$ are *aggregate functions*, that is, either **min**, **max**, **count**, **sum** or **avg** and where $1 \leq i_t \leq n_1$ and $1 \leq j_h \leq n_1$. The aggregate says that the elements of the relation $R$ should be split in different equivalence classes depending on the values of the attributes $A_{i_1}, \ldots, A_{k_1}$, where equality on the attributes decides in which equivalence class an element belongs. For each equivalence class the functions $F_{j_1}(A_{j_1}), \ldots, F_{j_{k_2}}(A_{j_{k_2}})$ are computed on the elements of each equivalence class for the given attribute. Let us see an example.

**Example 2.2.2** *Find the sum of the sales for each branch.*

This can be expressed in relational algebra by

$$_{Branch}\mathcal{G}_{\mathbf{sum}(Price)}(sale)$$

The resulting relation is

| Branch | sum(Price) |
|---|---|
| Frederiksberg | 7696 |
| Valby | 4297 |

■

### 2.2.2 SQL

If we think of a database as a collection of multi-set relations then SQL (Structured Query Language) *queries* are programs used to extract information from the database. A basic query has the following structure:

**select** $A_1, \ldots, A_n$ **from** $R_1, \ldots, R_k$ **where** $P$

which is equivalent to the relational algebra expression

$$\Pi_{A_1,\ldots,A_n}(\sigma_P(R_1 \times \ldots \times R_k))$$

Furthermore SQL has expressions corresponding to the other basic relational algebra expressions from Section 2.2.1 (the same holds for expressing predicates). Let us see an example.

**Example 2.2.3** *Write the relational algebra expression from Example 2.2.1 as a SQL query.*

The SQL query can be written as

**select** $sale.Branch, time.Month, sale.Price$
**from** $sale, time$
**where** $sale.Time\_id = time.Time\_id$ **and** $time.Quarter = 2$ **and** $sale.Color =$ *"blue"*

■

It is also possible to express aggregates using SQL. The SQL expression

**select** $A_{i_1}, \ldots, A_{i_{k_1}}, F_{j_1}(A_{j_1}), \ldots, F_{j_{k_2}}(A_{j_{k_2}})$
**from** $R$
**group by** $A_{i_1}, \ldots, A_{i_{k_1}}$

is equivalent to

$$_{A_{i_1},\ldots,A_{i_{k_1}}}\mathcal{G}_{F_{j_1}(A_{j_1}),\ldots,F_{j_{k_2}}(A_{j_{k_2}})}(R)$$

Let us see an example of a SQL aggregate.

**Example 2.2.4** *Write the relational algebra expression from Example 2.2.2 as an SQL query.*

The SQL query can be written as

**select** $Branch, \mathbf{sum}(Price)$
**from** $sale$
**group by** $Branch$

■

Furthermore SQL has operations for inserting, updating and deleting entries in a database relation together with the possibility to express, how the result of a query should be presented (e.g. sorted with respect to a given attribute).

### 2.2.3 Relation to Report Function Computation

Relational databases are used by many ERP systems. Typically, an ERP systems queries the database to fetch data needed for report function computation, or it queries the database directly for the result of some computation needed for a report function. If the computation requires that we traverse a big part of the database, there can be a long delay from calling a method in the ERP system that computes a report function, to getting the result of the computation, no matter which of these approaches we choose. For

ad hoc querying (dynamic querying) this is acceptable, but if we in advance know many of the queries we want to compute (static queries) this is not acceptable.

Note that we only argue that using the relational database queries for report functions is not optimal. However, relational databases can still be used to store information.

Basic SQL does not support any kind of real-time computation.

## 2.3 Relational Database Extensions

This section contains a description of add ons that accommodate faster computation of a special class of queries, which often are relevant for businesses. That is, we describe Materialized views and OLAP, which are extensions of SQL, and data-warehouses which are relational databases optimized for the OLAP extension. Oracle, IBM DB2 and Microsoft SQL Server all have support for materialized views and OLAP. Finally we describe the SIFT technology from the Microsoft NAV ERP system.

### 2.3.1 Materialized Views

A view is a virtual relation declared on existing data in a database, which typically is computed on demand, when needed. Let us first see an example of a view.

**Example 2.3.1** *Declare a view to create a virtual relation, which uses the query from Example 2.2.4.*

**create view** $totalsales(branch, amount)$ **as**
**select** $Branch, \mathbf{sum}(Price)$
**from** $sale$
**group by** $Branch$

This creates a virtual relation called *totalsales* with attributes *branch* and *amount*. ■

However, if we depend on a view when doing queries, it can become a performance issue to compute the view every time the virtual relation is accessed. To avoid the performance penalty of computing a view on the fly, we can use *materialized views*. The term materialized view is used, when the virtual relation is stored as an actual relation, enabling the possibility of querying it directly without the need for computing it (note that the view

is no longer a virtual relation). The problem with materialized views is that they can become inconsistent, when the underlying data changes.

Keeping the materialized view consistent with the underlying data is referred to as *view maintenance*. The most naive approach to view maintenance, is to recompute the query describing the view, if there is a modification of the underlying data. However, this is not a very good choice, if the underlying data changes a lot. Thus *incremental view maintenance* is introduced. Incremental view maintenance means that instead of recomputing the view relation from scratch every time there is a change to the underlying data, we modify the view incrementally with respect to the changes on the underlying data.

#### Relation to Report Function Computation

The idea of virtual relations is not in conflict with the relational model. However, the idea of maintaining a view over time, that is, update the materialized view, when the underlying data is changed, is something that is outside the relational model.

In the computation of report functions, materialized views can improve the performance of computing report functions, because the views can contain aggregate information. However, because ERP system data changes so rapidly and because ERP systems typically contains massive amounts of data, a materialized view needs to be updated all the time. Thus, the updates need to be done incrementally, if we want a chance for it to be feasible. Since the views need to be computed incrementally, we need to know in advance, which views that could be relevant for report function computations, and thus in some sense the queries become static.

Thus materialized views that are updated incrementally, can be used to give real-time computations of some aggregate functions (count, sum, avg) which are used in many report functions. However, this still requires that we in advance know, what we need to maintain incrementally and thus the view is a static query. Furthermore, updating materialized views incrementally can also be slow, if we for example use the aggregate functions max and min, because there is no good way to maintain these values incrementally, when elements are deleted or modified.

### 2.3.2 OLAP

OLAP is an acronym for *OnLine Analytical Processing* (Silberschatz et al. [2002][p. 818-829], Garcia-Molina et al. [2002][p.1070-1078] and Lewis et al.

[2002][p. 643-657] give descriptions).

### Fact and Dimension tables

Assume we have a database relation with schema $(dimension_1, \ldots, dimension_n, value)$ That is, each row consists of a *value* tagged with references to a number of different *dimensions*. A value is something, where the domain supports aggregate operations, and a dimension is something that describes the value. A table with a schema like this is called a *fact* table. The entries in the dimension columns of a fact table refer to entries in *dimension tables*, which contains the dimension information.

The table relations from Figure 2.1 can be considered fact- (the left table) and dimension tables (the right table). The relation between the fact and dimension tables can be drawn as a *star*, because the fact table refers to the dimension tables, but the dimension tables do not refer to each other or to the fact table, and thus it is called a *star schema*. For simplicity we have not included dimension tables for colors and branches, because in the running example colors and branches do not contain more related information. Time entries however, contain which month and quarter it belongs to. The reason for using references to dimension tables is to keep the data somewhat normalized, that is, no information must be repeated. However, it is not a strict demand to keep the data normalized, if it comes at a cost of using extra time to query the database. Often we are only interested in some of the dimension information, thus pushing information to dimension tables has the advantage that we do not need to fetch the information that is not used.

### OLAP Cubes

An *OLAP cube* relation is a relation containing aggregate information based on the fact and dimension tables. Assume we have a fact table referring to some dimension tables. Select a number of attributes from the dimension tables and name them $A_1, \ldots, A_n$ for some $n \in \mathbb{N}_0$. Then we create $2^n$ equivalence relations by considering all $2^n$ subsets of $\{A_1, \ldots, A_n\}$ (note, that this also includes the empty subset): For each subset $\{A_{i_1}, \ldots, A_{i_j}\}$ the elements from the fact table are put in the same equivalence class, if they have the same values on the dimension attributes. An aggregate function is chosen and computed on all elements from the same (non-empty) equivalence class for each equivalence relation and stored as an OLAP cube relation. Let us see an example.

**Example 2.3.2 (OLAP Cube Relation)**

Below we see an example of an OLAP cube relation, where we as attributes for a sale have chosen Color and Quarter and as aggregate *sum*. Note, that the table represents the computation of the sum function on each equivalence class in four equivalence relations, where "−" in the table is used to denote that this particular attribute is not included in the equivalence relation, which can be represented by a *null* value in a database relation.

| Color | Quarter | sum(Price) |
|---|---|---|
| Red | 1 | 4797 |
| Blue | 1 | 2199 |
| Red | 2 | 1299 |
| Blue | 2 | 3698 |
| Blue | - | 5897 |
| Red | - | 6096 |
| - | 1 | 6996 |
| - | 2 | 4997 |
| - | - | 11993 |

■

The name OLAP *cube* stems from the case, where the number of dimensions is three, because then the OLAP cube relation can be visualized as a cube. If $\mathcal{A}$ denotes the set of attributes, then the size (number of entries) of the OLAP cube relation $M$ is

$$|M| \in \mathcal{O}\left(\sum_{\{A_{i_1},\ldots,A_{i_j}\}\subseteq\mathcal{A}} |A_{i_1}| \times \ldots \times |A_{i_j}|\right)$$

where $|A|$ denotes the size of an attribute domain. This means that OLAP cube relations can be large, if the attribute domains are big. In principle the attribute domains can be infinite, but this still results in a finite OLAP cube, because we only look at the non-empty equivalence classes and for all practical purposes in databases the domains are finite.

An OLAP cube relation can be substantially bigger than the fact table it is constructed from, which the following argument shows: Assume we have a fact table with $k$ entries and assume we choose the attributes $A_1, \ldots, A_n$ from some of the dimension tables referred to by the fact table. Furthermore assume that all entries have different values on attribute $A_1$ and the same values in the rest of the attributes. This means that each non-empty

equivalence class in each equivalence relation, where $A_1$ is included as an attribute, contains exactly one element. There are $2^{n-1}$ different equivalence relations, which includes $A_1$ and thus each of these relations contains exactly $k$ non-empty equivalence classes, which means that the OLAP cube table contains at least $k2^{n-1}$ entries and thus the size of the OLAP cube relation can be exponential in the number of attributes chosen. This means that the dimensions should be chosen carefully.

OLAP cube operations have been introduced in SQL 99' standard and without further introduction the OLAP cube relation from Example 2.3.2 can be constructed by:

**select** $sale.Color, time.Quarter, \mathbf{sum}(sale.Price)$
**from** $sale, time$
**where** $sale.Time\_id = time.Time\_id$
**group by cube**$(sale.Color, time.Quarter)$

### Drilling Down and Rolling Up

Assume $A_1$ and $A_2$ are attributes with domains $D_1$ and $D_2$ of some dimension table $Dim$. $Dim$ has an aggregation *hierarchy* on $A_1$ and $A_2$ denoted $A_1 \rightarrow A_2$, if $A_1$ can considered "a part of" $A_2$. That is, if there exists a function $f : D_1 \rightarrow D_2$, where $r[A_2] = f(r[A_1])$ for all rows $r$ in $Dim$, where $r[A]$ denotes the value of attribute $A$ in row $r$.

So assume we have a hierarchy $A_1 \rightarrow A_2$ and we have an OLAP cube, where $A_2$ is included as a attribute. Then we can *drill down* on $A_2$ to $A_1$, which means that we create a new OLAP cube, where attribute $A_1$ is included instead of $A_2$. Let us see an example.

### Example 2.3.3 (Drilling Down)

In the time dimension from Figure 2.1 we have the hierarchy $Month \rightarrow Quarter$, where we as $f$ let the months be mapped to the quarter they belong too. Then $r[Quarter] = f(r[Month])$ for all rows $r$ in $Time$. If we take the OLAP cube relation from Example 2.3.2 and drill down from Quarters to Months we get the following relation:

| Color | Quarter | sum(Price) |
|---|---|---|
| Red | January | 3398 |
| Red | February | 1399 |
| Blue | February | 2199 |
| Red | April | 1299 |
| Blue | April | 3698 |
| Blue | - | 5897 |
| Red | - | 6096 |
| - | January | 3398 |
| - | February | 3598 |
| - | April | 4997 |
| - | - | 11993 |

■

*roll up* is the converse of drilling down. That is, if we have an OLAP cube where $A_1$ is included as an attribute, then we can construct a new OLAP cube where attribute $A_2$ is included instead of $A_1$. Note, that when rolling up, we can compute the new OLAP cube from the old OLAP cube by using $f^{-1}$ to see, which equivalence classes that should be collapsed.

### Relation to Report Function Computation

As explained above OLAP is an extension of SQL, which produces new relations from other relations. That is, OLAP does not change the relational model. It is the authors opinion that OLAP can be considered a special kind of materialized view.

The idea of OLAP is that we want faster computation of a special kind of queries, which is achieved by constructing OLAP cube relations. After an OLAP cube relation has been created, we can issue a large number of different queries on the relation, which in many cases can be computed faster than issuing equivalent queries on the original database. Thus, we can consider OLAP cubes as a way of formatting the information in the database, which generally improves the performance of the type of queries supported by OLAP cubes, compared to equivalent queries on the original data.

If we have a big database, it can take a long time to materialize an OLAP cube, because we need to traverse all the data from the tables that the cube is build from, and because the OLAP cube relation can contain many entries (even more than the origin tables as explained in Section 2.3.2). Thus, there exists execution plans for when to materialize the different parts of

an OLAP cube. However, if the entire OLAP cube relation is not materialized in advance, it decreases the performance of querying the cube, if non-materialized information is needed to comply to the query.

Since OLAP is a part of relational databases it inherits the properties that data is to some degree static, that is, when we create an OLAP cube, we take a snapshot of the database from which the OLAP cube is constructed and then create the cube. Thus, if OLAP is used in an ERP system report function computation context, then the answer to the queries on an OLAP cube can be obsolete, even before we are able to query the cube.

Incremental OLAP has been developed to incrementally update an already existing OLAP cube, when the underlying data of the cube is changed. In the authors opinion this shows that the idea that data is static is invalid in the OLAP area. Incremental OLAP is therefore a hack to circumvent the static nature of the data with respect to querying in relational databases. Querying in some sense becomes static when using incremental OLAP, because we in advance need to know, which cube we are interested in building and maintaining incrementally.

OLAP also has the problem that if we compute the entire OLAP cube to get good performance of the OLAP queries, then there is a good chance that we compute a lot of irrelevant stuff. That is, when we generate an OLAP cube we actually need to know, which kind of queries we are interested in doing and that we actually do them, not to waste time constructing the cube.

Incremental OLAP can in some cases be used to update OLAP cube relations in real-time (or near-real-time). Whether or not the cube can be queried in real-time is a question of, how the cube is declared. To benefit from incremental computation of OLAP cubes, we need to know in advance, which values we need to maintain incrementally (like in incremental view maintenance), thus the queries become static. However, because OLAP cubes often contain redundant information, we can sometimes "get lucky" that information we did not see was relevant from the beginning, can be extracted from the OLAP cube. This of course, comes at a cost of maintaining the redundant information until it is actually needed.

### 2.3.3 Data Warehouses

Silberschatz et al. [2002][p. 843], Garcia-Molina et al. [2002][p. 1051] and Lewis et al. [2002][p. 663] give descriptions of data warehouses, which we explain here. A data warehouse is a relational database containing information from many other relational databases, such that it becomes easier

to construct OLAP cubes based on all the data from all the databases. Constructing a data warehouse introduces problems like:

1 The tables of the underlying databases may have different schemas.

2 The data is not stored in the same granularity.

3 The data warehouse should be synchronized with the underlying databases.

Problems 1 and 2 are related, because different schemas can lead to different granularities of the data. This induces a design question of the schemas in the data warehouse, because we would like to merge the information from different databases into one database. Thus, it can be a non-trivial task to decide, which database relation schemas that should be used in the data warehouse.

The information in a data warehouse is stored as fact and dimension tables, such that a data warehouse can support OLAP queries. This also impacts the design of the schemas for the data warehouse.

Problem 3 addresses the issue of when the data warehouse should be updated. Typically changes to the data warehouse is not made every time there is a change on one of the underlying databases. Therefore updates to the data warehouse are typically performed as a batch-run. This imposes the problem that the data warehouse is out of date all the time.

### Relation to Report Function Computation

A data warehouse is a relational database and therefore it in theory does not differ from relational databases.

Data warehouses are created, because we want to analyze all the data from the underlying databases.

Updating the data warehouse in batch runs (maybe once a day) has the advantage that the data is static for relative long periods of time, which typically is not the case for an ERP system database. Thus OLAP cubes can be created after the most recent data has been committed to the data warehouse. Furthermore since businesses can be distributed over more than one location, they often have more than one database system (maybe one per store/location), which can contain tables with different schemas. Even though the databases essentially are used to record the same information, it makes sense to store the data in a uniform way to be able to analyze it.

Since data warehouses typically are updated using batch runs, the data is unsynchronized with respect to the underlying databases almost all the time.

Thus, the report functions computed on behalf of the information in a data warehouse, does not reflect the latest changes in an ERP systems databases. Furthermore, real-time computations are not an option unless, all changes to the databases of all the locations are transmitted to the data warehouse, when they happen. However, if the changes are transmitted when they occur, it might be possible to update an OLAP cube in real-time. Whether of not this leads to real-time computation of report functions depends on the OLAP cube relation.

### 2.3.4 Microsoft NAV SIFT

Microsoft NAV is an ERP system developed by Microsoft. Studebaker [2007] has given a comprehensive description of Microsoft NAV from a developers points of view and Hvitved [2008] has created a shorter description from a computer scientists point of view. The explanation here is primarily based on the description given by Hvitved [2008].

Microsoft NAV includes something called *Sum Index Field Technology* (SIFT). Using SIFT it is possible to create virtual values called *FlowFields* in the tables in the Microsoft NAV database. A FlowField is a virtual attribute of a table, which values can depend on the table, where the FlowField is declared and entries in one other table. Let us see a simple example of a FlowField formula and what the result looks like.

**Example 2.3.4** *Assume that Microsoft NAV contains the sale table from Figure 2.1 and the location table seen in Figure 2.3 on the left. We declare a FlowField formula that can be used to add the virtual attribute Avg to the location table, which values are average Price per sale for each Branch mentioned in the location table.*

$$\phi = \text{Average}(sales.Price \text{ WHERE } (Name = FIELD(Branch)))$$

When declaring a *FlowField* called *Avg* on the the *location* table, the table appears as the right table of Figure 2.3 everywhere in the Microsoft NAV system. Thus, from the developers and users point of view the *Avg* attribute is part of the *Location* table. ■

The Microsoft NAV system keeps the virtual values of the the FlowFields materialized. Therefore it is possible to fetch the virtual values at any time, without the need for computing the FlowField formula at the time of fetch. Thus, every time an entry is inserted, deleted or modified in the NAV database, the FlowFields depending on this entry are updated. FlowFields allows us to compute the virtual values given by the functions Exist, Count,

| Name |
|---|
| Frederiksberg |
| Valby |

| Name | Avg |
|---|---|
| Frederiksberg | 1924.00 |
| Valby | 1432.33 |

Figure 2.3: *Location* table on the left and *Location* table with virtual field *Avg* on the right.

SUM, Average, Min, Max and Lookup, which are explained by Hvitved [2008]. SIFT can be used, when we know in advance that some kind of information is relevant to maintain, when new information is added to the database.

Strictly speaking SIFT is not an extension to relational databases, but it is build on top of a relational database. That is, when entries are inserted, deleted or modified through Microsoft NAV, the Microsoft NAV systems ensures that the virtual values of the FlowFields are maintained.

#### Relation to Report Function Computation

FlowFields are used to circumvent the idea that data is static, because certain values are maintained when data is inserted, deleted or modified. Thus, we can declare FlowFields, which essentially are static queries. That is, SIFT tries to switch around on the ideas of data and query, thus querying become static and data dynamic.

SIFT can be used to compute simple functions, which values are maintained and does not need to be computed when fetched. However, as explained above only a very limited number of functions are supported by SIFT.

Another disadvantage of SIFT is that the developers need to manually declare FlowFields and manually clean up in the FlowField declarations. This means that if a declared FlowField is not needed anymore, then it is the developers responsibility to delete the FlowField. If the FlowField is not deleted it is still updated every time entries that affect the FlowField are inserted, deleted or modified, which can substantially decrease the performance of Microsoft NAV.

Thus the concept of SIFT may support real-time computation of some functions, which can be expressed as FlowField values, because some FlowFields can in theory be maintained incrementally in real-time. However, like all the other technologies seen so far, we need to know in advance, which val-

ues we are interested in maintaining and thus the real-time report functions become static.

## 2.4 Alternative Computational Models

Until now, we have seen the theory of relational databases and some of the extensions that have been created to accommodate the need for computation of report functions. This section explains, what *streaming databases* and *MapReduce* are and how they can be viewed as systems for computing report functions. We also explain, how these technologies differ from the report function computation technologies of the relational database systems.

### 2.4.1 Streaming Databases

*Streaming databases* is a relative new area of research (Golap and Özsu [2003] give an overview). The idea is that we want to add temporality in the database model.

#### CQL

Arasu et al. [2006] give a description of CQL (continuous query language) and there exists a prototype implementation called STREAM that supports CQL queries. CQL is a mix of relational operations written using SQL and operations used to transform relations to *streams* and vice versa. Arasu et al. [2006] define a stream as:

**Definition 2.4.1 (Stream)** *A stream $S$ is a (possibly infinite) multi-set of elements $\langle s, \tau \rangle$, where $s$ is an element belonging to the domain of the schema of $S$ and $\tau \in \mathcal{T}$ is the timestamp of the element.*

Thus, as the definition implies, a stream also has a schema and the elements in a stream are contained in the Cartesian product of the domains of the attributes from the schema and a timestamp domain. Arasu et al. [2006] do not give a description of the requirements on a domain of timestamps $\mathcal{T}$, but at least it should a have total ordering and finite intervals. As Arasu et al. [2006] we use $\mathbb{N}_0$ equipped with the standard ordering as the time domain. It is required that a stream only contains a finite number of elements with the same timestamp (even though it is not included in the definition), and that there exists a "clock" telling the "current time" with respect to the time domain. Furthermore, there is a notion of *time advancement* in the streaming database model, which means that elements in a stream arrive in

order with respect to their timestamps and it is not possible to *see* stream elements with timestamp greater than the current time given by the clock.

Besides the definition of stream, the following definition of relation is used by Arasu et al. [2006].

**Definition 2.4.2 (Relation)** *A relation $R$ is a mapping from each time instant in $\mathcal{T}$ (the time domain) to a finite multi-set of elements belonging to the domain of the schema of $R$.*

That is, for each $\tau \in \mathcal{T}$ then $R(\tau)$ is a finite multi-set relation called a *base relation*, which is a relation in the relational model sense. Even though it is not included in the definition, $R(\tau)$ should be read as *the elements of the relation $R$ at time $\tau$*, that is, how the relation looks like at time $\tau$.

Let us now take a look at the different mappings between streams and relations and vice versa. CQL includes the following three kinds of stream to relation constructs:

1 *Time based sliding windows*: $S$ [**range** $T$] is a relation $R$, with schema $S$, where precisely the elements from $S$ with timestamps described by the *range* $T$ is included. That is, at a given point in time $\tau$ the relation $R(\tau)$ contains the elements of the stream $S$ with timestamps between $\tau$ and $\tau - T$. The definition given by Arasu et al. [2006] is

$$R(\tau) = \left\{ s \mid \left\langle s, \tau' \right\rangle \in S \wedge \tau' \leq \tau \wedge \tau' \geq \max\left\{\tau - T, 0\right\} \right\}$$

How $T$ is specified varies depending on the timestamp domain $\mathcal{T}$.

2 *Element based windows*: $S$ [**rows** $n$] is a relation $R$ with the same schema as $S$, which includes the last $n$ elements of $S$ with respect to the current time. Since there can be many elements with the same timestamp this is not a deterministic operator, because one could for example write $S$ [**rows** 1] and then if the stream $S$ contains more than one element with the current timestamp then it is random, which element is chosen.

3 *Partitioned windows*: Similar to a *group by* in SQL, but it is not be explained here (Arasu et al. [2006] give details).

Furthermore CQL contains the following relation to stream operations, which we only describe very informally. Assume that $R$ is a relation, then

1 *Istream*: Shorthand for *insert stream*. When the clock moves from $\tau - 1$ to $\tau$ (for all $\tau \in \mathcal{T}$) then the stream $Istream(R)$ is extended with

elements $(R(\tau)\backslash R(\tau-1)) \times \{\tau\}$. The extension of the stream at each point in time is finite, because the definition of relation ensures that $R(\tau)$ is finite for all $\tau \in \mathcal{T}$. The reason this operation is called *insert stream* is because it describes, what has been added to the relation $R$ from time $\tau - 1$ to $\tau$. The definition given by Arasu et al. [2006] is

$$Istream(R) = \bigcup_{\tau \geq 0}((R(\tau)\backslash R(\tau-1)) \times \{\tau\})$$

2 *Dstream*: Shorthand for *delete stream*. The same as above, except that the stream $Dstream(R)$ is extended with elements $(R(\tau-1)\backslash R(\tau)) \times \{\tau\}$, which are the elements that are removed from the relation from time $\tau - 1$ to $\tau$ and thus the name *delete stream*. The definition given by Arasu et al. [2006] is

$$Dstream(R) = \bigcup_{\tau \geq 0}((R(\tau-1)\backslash R(\tau)) \times \{\tau\})$$

3 *Rstream*: Shorthand for *relation stream*. When the clock moves from $\tau - 1$ to $\tau$ then $R(\tau)$ is appended to the stream $Rstream(R)$. The definition given by Arasu et al. [2006] is

$$Rstream(R) = \bigcup_{\tau \geq 0}(R(\tau) \times \{\tau\})$$

Let us see a couple of examples of CQL queries.

**Example 2.4.3** *Assume we have a stream* sale *with schema* $(sale\_id, item\_id)$. *We construct a stream of* $sale\_id$ *consisting of only the sales, where the* $item\_id$ *is less than* 100.

The following CQL query creates the desired stream.

**select** $Istream(sale\_id)$
**from** *sale* [**range** *unbounded*]
**where** $item\_id \leq 100$

The *unbounded* keyword used as a range description means all stream elements up to the current point in time. That is, *sale*[**range** *unbounded*] is a relation $R$, where $R(\tau)$ contains all the elements from stream sale up to time $\tau$. Note, that this is a monotonic increasing relation over time. $Istream(sale\_id)$ produces a stream from $R$, where the $Istream(sale\_id)$ is the union of all increases of the relation $R$ since time 0, where only $sales\_id$

with *item_id* less than 100 are included. Thus, the Istream operator in principle just propagates the relation when it is monotonically increasing over time. That is, the resulting stream is

$$\bigcup_{0\leq\tau} \Pi_{sale\_id}(\sigma_{item\_id\leq 100}(R(\tau)\backslash R(\tau-1))) \times \{\tau\}$$

where $\sigma$ denotes filtering and $\Pi$ denotes projection. At a given point in time $\tau_0$, the above query, which we denote $Q$, returns a stream that can be described as

$$Q(\tau_0) = \bigcup_{0\leq\tau\leq\tau_0} \Pi_{sale\_id}(\sigma_{item\_id\leq 100}(R(\tau)\backslash R(\tau-1))) \times \{\tau\}$$

■

Note, that even though the query returns a stream, we talk about the result of a query at a given point in time.

**Example 2.4.4** *Assume we have the sale stream from Example 2.4.3 and assume we have a relation* item *with schema* $(item\_id, price)$*, which has* $item\_id$ *as a primary key. The resulting stream from the query below contains the price of the sales.*

**select** $Rstream(R.sale\_id, I.price)$
**from** $sale$ [**range** $Now$] **as** $R, item$ **as** $I$
**where** $R.item\_id = I.item\_id$

The range descriptor $Now$ means the elements from the sale stream with timestamp "current time". That is, $sale$[**range** $Now$] is a relation $R$, where $R(\tau)$ is a base relation containing the elements from the stream sale with timestamp $\tau$. At time $\tau$ the base relation $I(\tau)$ contains the prices for the items at this point in time. The stream $Rstream(S.sale\_id, I.price)$ contains at time $\tau_0$ the union of the *sale_id* and *price* from $R(\tau) \times I(\tau)$ for $0 \leq \tau \leq \tau_0$ which satisfies the filter condition. Thus, the stream is given by

$$\bigcup_{0\leq\tau} \Pi_{R.sale\_id,I.price}(\sigma_{R.item\_id=I.item\_id}(R(\tau) \times I(\tau))) \times \{\tau\}$$

If the above query is executed at time $\tau_0$, we get a stream of elements consisting of *sale_id* and *price* for each sale. That is, if we name the above query $Q$ then the result of executing the query at time $\tau_0$ is given by

$$Q(\tau_0) = \bigcup_{0\leq\tau\leq\tau_0} \Pi_{R.sale\_id,I.price}(\sigma_{R.item\_id=I.item\_id}(R(\tau) \times I(\tau))) \times \{\tau\}$$

■

#### Relation to Report Function Computation

The major difference of the relational model and the model behind streaming databases is that temporality is included in the model, and that the model includes infinite streams. Thus, the model tries to fix the problem of relations changing over time, by defining them as maps from points in time to multi-sets of elements, which represent the relations at that point in time.

It is unclear how it would affect the performance and what kind of storage capacity would needed to implement streaming databases and use them in a ERP system context. Furthermore, this technology is only at an experimental stage. However, as Example 2.4.4 shows it is fairly simple to express a query, where we do not need to be concerned with the temporal changes on the *item* relation, because this is included in the model and the semantics of CQL. This could be beneficial in an ERP system context.

Furthermore, many queries over streams and relations can be identified as being monotonic, which means we can imagine that the result of queries can be maintained incrementally over time, thus giving fast replies to many static (or in CQL terminology continuous) queries. This can possibly be utilized for real-time computation of some report functions.

### 2.4.2 Map-Reduce and Map-Reduce-Merge

Dean and Ghemawat [2008] and chih Yang et al. [2007] give a description of Map-Reduce and its extension Map-Reduce-Merge. Map-Reduce and Map-Reduce-Merge are C++ libraries developed by Google for internal use to abstract from explicit parallelization of large distributed computations.

#### Map-Reduce

Using the library requires that the user declares two functions called *map* and *reduce*, which should have the types (written as a functional programming language types)

$$\begin{aligned} \textbf{map} &: \alpha \times \beta \to (\alpha' \times \beta')\ list \\ \textbf{reduce} &: \alpha' \times \beta'\ list \to \beta'\ list \end{aligned}$$

Furthermore the user should choose an input and output file, where the input file should contain a list of elements of type $\alpha \times \beta$, which are called *key* and *value* pairs.

The above information is then passed to the Map-Reduce library and then all the entries in the input file is feed to the **map** function, which then

```
1  fun insert ((a,b),[]) = [(a,[b])]
2    | insert ((a,b),(a',bs) :: group) =
3    if a = a'
4    then (a',b :: bs) :: group
5    else (a',bs) :: insert ((a,b),group)
6
7  val partition = foldl insert []
8
9  fun MapReduce f reduce xs =
10 let
11   val xsmapped = map f xs
12   val xsgrouped = partition (foldl (op @) [] xsmapped)
13   val xsreduced = foldl (fn (x,acc) => reduce x @ acc) [] xsgrouped
14 in
15   xsreduced
16 end
```

Figure 2.4: SML declaration of the MapReduce library function

for each entry in the input file produces zero or more intermediate results of key and value pairs of type $\alpha' \times \beta'$. This is called the *map phase*.

Before invoking the **reduce** function the Map-Reduce library groups together all intermediate values with the same key. Then for each intermediate key the **reduce** function is called with the key and a collection of intermediate values with this key.

In Figure 2.4 we see a SML declaration of the *MapReduce* library function, where we have not restricted the output elements of the reduce function to be of the same type as the input elements, that is, **reduce** : $\alpha' \times \beta'\ list \rightarrow \beta''\ list$. The first argument to the *MapReduce* function is the **map** function, which we here call $f$. The second argument is the **reduce** function and the third argument is a list of pairs (which is used to simulate the content of a file). However, the terminology used by Google is conflicting with the standard terminology from functional programming, because in functional programming *map* is the function applying another function to the input. However, the designers of the MapReduce library, have chosen to call the function that is being applied the *map* function. The intuition of, why the computation can be parallelized is as follows: As the code from Figure 2.4 shows, the MapReduce library function can be implemented using the map and fold functions of SML. Since it does not matter in which order $f$ is applied to the elements of the input list, and because the associative operator

append is used to combine the result of applying the reduce function, at least the map and reduce phase can be parallelized. However, it is not a priori clear, how the grouping phase can be parallelized.

Let us see a very simple example using the information from Figure 2.1.

**Example 2.4.5** *Compute the total number of bicycles sold of each color.*

Assume that the Branch and Color information from the left table in Figure 2.1 is written into a file as strings. Then we can define the **map** and **reduce** functions as (written in pseudo code).

```
1: map (String branch, String color) :
2:    EmitIntermediate(color, 1);
3:
4: reduce (String color, Iterator values) :
5:    int result = 0;
6:    foreach v in values :
7:       result += v;
8:    Emit(result);
```

The **EmitIntermediate** and **Emit** means constructing intermediate- and final values, which means that in this example **map** and **reduce** only produce *singletons* whenever they are called. A more complicated example could include several **EmitIntermediate** and **Emit** every time **map** and **reduce** are called respectively.

The **map** function produces for each entry in the input a pair consisting of the color of the bicycle and the number 1. When the input is processed, all the values (a list of 1's) for each intermediate key (the color) are passed to the **reduce** function, which then computes the total number of bicycles sold for each color. ■

### Map-Reduce-Merge

Google has created an extension to Map-Reduce called Map-Reduce-Merge, which allows the merging of the result of two Map-Reduce results. First of all the type of the **reduce** function is changed to

$$\textbf{reduce} : \alpha' * \beta' \; list \rightarrow \alpha' * \beta' \; list$$

Map-Reduce-Merge needs the keys in the merge phase and therefore the type of **reduce** has been changed. The type of **merge** is

$$\textbf{merge} : (\alpha' * \beta' \; list) * (\alpha'' * \beta'' \; list) \rightarrow (\alpha''' * \beta''') \; list$$

Furthermore the user also need to declare some other functions that specify in which order the input to the merger should be given. chih Yang et al. [2007] explains why the Map-Reduce-Merge framework is sufficiently strong to express the relational algebra operations from Section 2.2.1.

#### Relation to Report Function Computation

The Map-Reduce library has been constructed to abstract away parallelization of computations and there is no abstract model for data, other than it resides in a file.

Map-Reduce and Map-Reduce-Merge were not constructed with the intention of doing reporting. However, since Map-Reduce-Merge is sufficiently strong to express relational algebra, it would be possible to construct a SQL skin for Map-Reduce-Merge. This means that we could write SQL queries and then let the Map-Reduce-Merge library parallelize the computation of the query.

A priori the MapReduce library can not be used for computing report functions in real-time, because when the underlying data changes, the computation has to be performed from scratch again. However, one could imagine that a library called MapReduceIncrement can be implemented, which incrementally maintains the result of a MapReduce computation. The idea is as follows: If entries are inserted into the underlying data we apply the **map** function to these new elements. The newly added elements produces a set of key and value pairs. To update the result of MapReduce function we need to run the reduce phase again for all the keys appearing in the result of the map phase of the newly added elements (deletions can be handled correspondingly). This does not at all lead to real-time computation, because if we for example have very few keys and a lot of values for each key, then the reduce phase need to traverse a lot of information, when doing the incremental update of the reduce phase. However, if the reduce function can be computed incrementally then it might be possible to achieve real-time computation.

## 2.5 Specification of Report Functions

The technologies explained in the preceding sections all have their own language for specifying, what we want to compute. In this section we see, how data can be retrieved from a database into a general purpose programming language. We give a short explanation of, how data is retrieved from the

database into both the Microsoft NAV and Microsoft AX. Furthermore we give a description of the LINQ library for .NET.

### 2.5.1 Microsoft NAV C/AL

Microsoft NAV uses its own programming language called C/AL to express the functionality of the system. Hvitved [2008] and Studebaker [2007] give a description of the C/AL programming language. C/AL is an imperative Pascal like programming language and it includes a table type, which gives direct access to the underlying database tables.

The integration of the database and C/AL is an important property of Microsoft NAV, because the whole system is centered around the database. Thus, one can directly fetch the data from the database and perform any computation on the data. However, this means that the report function specification in NAV is primarily performed in C/AL code, which at the authors opinion is to low level.

### 2.5.2 Microsoft AX X++

Microsoft AX uses its own programming language called X++ to express the functionality of the system. Greef et al. [2006][p. 91-118] give a description of the syntax of X++ together with examples indicating the semantics of the different syntactic constructs. X++ is an object oriented, C++ like programming language. Like C/AL, X++ has a table type, which can be used to access the tables of the underlying database of Microsoft AX directly. X++ also has something called *data-aware* statements, which essentially is SQL syntax for querying tables. Let us see an example.

**Example 2.5.1** *Assume the sale table of Figure 2.1 (left table) is a table in Microsoft AX. Write a X++ method that prints the branch name of the sales, where the price is greater than 2000.*

```
1  static void myPrint()
2  {
3    Sale sale;
4    while select *
5          from sale
6          where sale.Price >= 2000
7    {
8      print sale.Branch;
9    }
10 }
```

■

Example 2.5.1 is a very simple example of iterating over the rows of a table, which satisfies a certain predicate. However, the data-aware statements also supports joining tables and doing aggregation. Thus, the data-aware statements are not only used to fetch information, but also to compute functions.

### 2.5.3 LINQ

LINQ is an acronym for *Language INtegrated Query*, which is a .NET library and some extra syntax for the .NET programming languages. The idea of LINQ is to bring interaction with external data sources closer to the programming language by enabling the programmer to write SQL like syntax (LINQ syntax) to query data sources like SQL databases, XML- or .NET data structures.

Torgersen [2007] gives a short description of LINQ. In this section we see, how LINQ can be used to express the SQL queries from Section 2.2.2. Furthermore, the examples have been executed using the .NET 3.5 Framework and implemented using $C\#$, where arrays have been used as data sources. Let us see an example.

**Example 2.5.2** *Write the SQL query from Example 2.2.3 as a LINQ query.*

**var** $query =$
**from** $s$ **in** $sales$
**from** $t$ **in** $times$
**where** $s.Time\_Id == t.Time\_Id$ && $t.Quarter == 2$ && $s.Color ==$ "$Blue$"
**select new** $Salesummary\{Branch = s.Branch, Month = t.Month, Price = s.Price\}$;

■

Example 2.5.2 uses two data sources called *sales* and *times*. For the LINQ query to typecheck, both of the data source variables need to implement an interface that enables us to query the data source using LINQ. The LINQ syntax is similar to SQL, but instead of referring to attributes, we refer to the elements of the data source. That is, **from** $s$ **in** *sale* means "for all elements s in sales". Thus, the first two lines of the query results in the Cartesian product of the elements in *sales* and *times*, where we in the following lines use $s$ and $t$ to refer to the elements. The **where** clause is like in SQL a filter on the elements $(s, t)$ and the **select** creates an element in

the output for each pair $(s, t)$ that passes the filter condition. *query* is then a collection of objects of type *Salesummary*. However, the LINQ query is not executed, before the result of the query is used, that is, the query is executed on the data source, when we iterate over the result. This means that iterating over the same query variable can produce different results, if the data source has changed between the iterations.

Let us see the other query example from Section 2.2.2

**Example 2.5.3** *Write the SQL from Example 2.2.4 as a LINQ query.*

**var** $query =$
**from** $s$ **in** $sales$
**group** $s$ **by** $s.Branch$ **into** $h$
**select new** $Branchsale\{Branch = h.Key, SumPrice = h.Sum(x => x.Price)\}$;

■

The example uses the **group by** construct of LINQ, which produces a collection of collections of elements from the sales data source, where each collection is tagged by the branch name its elements belongs too. The **select** then produces one element in the output for each entry in the group collection, where the sum of all prices for each branch is computed using the aggregate *sum*. The C# expression $x => x.Price$ is a *lambda expression*, that is, an anonymous function that maps each entry to the value of the *Price* field.

LINQ contains the same aggregate operators and group by capabilities as SQL, and furthermore it supports that we can supply our own "aggregate functions", which essentially results in a *fold left* (using SML terminology) over the information in the data source.

SQL integration is done by describing, how tables and columns of tables are mapped to objects and fields. This is done using some special syntax referred to as *LINQ to SQL*. However, LINQ does not have special SQL like syntax for manipulating existing entries or inserting new entries.

### 2.5.4 Relation to Specification of Report Functions

First C/AL was developed, then X++ and finally LINQ. Thus, one can see that the trend of the programming languages is: Closer integration of data sources and programming language, using SQL like syntax to retrieve data. Furthermore, it should be noted that the SQL syntax is not only used for fetching data, but also to compute simple functions. However, the functions

expressible in the SQL syntax is also computable by the native programming language, which in the authors opinion indicates that the declarative specification style of SQL is preferable over the imperative style of the native programming languages.

Thus, when expressing report functions, it is preferable to use a declarative approach over an imperative. However, SQL is not satisfactory in itself, because it does not have sufficient expressive power. Thus there is a need to bring information from the database into a programming language more expressive than SQL.

## 2.6 Summary & Future Work

We have presented the theory of relational databases together with technologies from other computational models, and explained how they can be used to compute report functions. Furthermore we have explained, if we think it is possible to achieve real-time computation of report functions using the different technologies.

For all the technologies presented here the following holds: If real-time computation of a report function is possible, then the report function is static, because the result of the report function needs to maintained incrementally, when information is added, changed or deleted. This is not surprising, because if a report function depends on a massive amount of data, we can not traverse all the data, when the result of the report function is requested (because then it is not real-time) and therefore we need to maintain the result of the report function.

Furthermore, we have argued that relational databases are probably not the way to go for ERP systems, because the data in ERP systems changes so rapidly. ERP systems primarily accumulate data and the relational model does not have any notion of static querying over time. Thus, a computational model, which includes some aspect of temporality and static querying would be preferable. For dynamic querying, ideas like massive parallelization could be used to give faster computation of report functions. Therefore, one could imagine a computational model, which supports static queries that are maintained in real-time (or near-real-time) expressed in one formalism, and dynamic queries which could be massively parallelized expressed in another.

The current trends in reporting technology indicates that a declarative approach of specifying report functions is preferable.

Thus, for future work it could be interesting to investigate:

- Compare the expressiveness of the different reporting technologies by

giving a denotational semantics for each language and investigate if the languages can be reduced to one another.

- Which kind of computations can be maintained incrementally in real-time.
- Which kind of computations can be computed efficiently by using massive parallelization.
- Construct a declarative formalism for expressing static and dynamic queries and compare to existing technologies.

# Chapter 3

# FunSETL—Functional Reporting For ERP Systems

**Abstract:** One of the essential features of enterprise resource planning systems is the ability to provide the users and decision makers with reports on how the enterprise is running, and to enable the enterprise to provide the authorities the required legal reports. By their nature these reports needs to operate on large amounts of data and the decision makers need the reports in a timely manner. To achieve acceptable performance of the programs that generates these reports, the data, the full transaction log that the programs operate on is kept in denormalized form. What we propose instead is to write the programs as they are operating on the full amount of data and then use *automatic incrementalization* for achieving acceptable performance. To study whether automatic incrementalization is practically feasible we introduce the reporting language FunSETL, which is a restricted ML dialect, a compiler for FunSETL that can perform automatic incrementalization, and we have collected a small suite of report functions written in FunSETL containing a real life report function. We show that using incrementalization on our suite we obtain an *asymptotic improvement* of a *linear factor* in the running time compared to the non-incrementalized original programs.

## 3.1 Motivation

Enterprise Resource Planning (ERP) systems are widely used, because all businesses have at least an account system or a financial system. One of the main motivations for having an ERP system is to get computer assistance for keeping track of how your business is running. That is, to keep track

of accounts, inventory and the use of various resources. Likewise, we want computer assistance for trying to forecast future opportunities. Also, all businesses are required by law to provide certain reports for the authorities, such as an income statement. These different kinds of information range from a simple number on the screen specifying what is in stock to complex business intelligence reports. We call all of these kinds of information for *reports*. That is, in our terminology a report can be both a single number or a complex data-set.

Common to all ERP systems is that they need to provide reports for their users and most of the report functions often rely on giga-bytes or maybe even tera-bytes of data. Thus, there has been an increasing demand for faster report function computation, because it can help companies take better decisions and thereby getting an advantage over their competitors.

What these report functions have in common is that they often need to operate on large amounts of data, such as the complete log of transactions, for instance. The users and decision makers need the reports in a timely manner, preferably in real-time. Thus, report functions are somewhat performance critical. Hence, to get adequate performance people often keep the data in denormalized form, which makes the programs that operates on the data more complex and more prone to errors. For example, if you want to compute the average price of a certain kind of items you sell. You either run through the whole transaction log, find all the relevant transactions, sum all the relevant prices, and finally divide by the number of items sold. If you denormalize the data by hand you need to introduce and maintain the total sum of all the items and the number of items.

Our hypothesis is that report functions are easier to write and analyze if you write them in a declarative manner and they should be written as close as possible to the specification of the report function, similar to what is normally done using SQL. However, it is not all report functions that are easy to express in SQL, especially those that need some amount of calculations, like business intelligence for instance, takes some SQL expertise to express. Thus, we introduce our own ML dialect which we believe makes it easier to express computations and still keep the report functions declarative.

Declarative report functions that are written to operate on the full log of transactions, however, can be prohibitively slow. Thus, to regain performance, we use the insight that report functions are run over the same data again and again, and thus it should be possible to cache and reuse some computations and do incremental updates to the cached result. This is called *incrementalization*. In this paper we show that it is possible to *automatically* transform some report functions into incremental versions and

that these incremental functions have adequate performance.

The paper is structured as follows: Section 3.2 contains an informal description of the programming language FunSETL. Section 3.3 introduces *incrementalization* and how *automatic incrementalization* can be performed on FunSETL programs. Section 3.4 contains a description of the current implementation of a FunSETL compiler and FunSETL incrementalizer and Section 3.5 shows the benchmark suite together with some experimental results on a *real life* report function from an ERP system. The related work and conclusion can be found in Section 3.6 and Section 3.7. The work presented in this paper was presented in an earlier version at the International Symposium on Implementation and Application of Functional Languages (Nissen and Larsen [2007]).

## 3.2 Introduction To FunSETL

Figure 3.1 shows the syntax of the language (we also allow some extra syntactic sugar for finite maps). As it can be seen in Figure 3.1, FunSETL is a variant of explicitly typed Mini-ML extended with support for data types useful for writing report functions. That is, dates, finite mappings, and multi-sets. We use multi-sets rather than ordinary sets because they correspond better to tables in relational database systems, and one of our plans is eventually to experiment with compilation to SQL.

What Figure 3.1 does not show is that recursion is not allowed, which is enforced by the type-system (you are only allowed to call functions defined before the current function). The only kind of repetition allowed is iteration over multi-sets, by using the **foreach** expression, which is similar to the SML function **fold** . Thus, it can be shown that every FunSETL program terminates. Hence, FunSETL is not Turing complete.

Figure 3.2 shows a FunSETL program that computes the average of the elements in a multi-set. The average is computed by summing all the elements and dividing the sum by the number of elements in the multi-set. In Figure 3.2 this is done by declaring two functions: *count* that count the number of elements in a multi-set, and *average* that compute the sum with **foreach** and call *count* for getting the number of elements in the multi-set.

It is not the intention to promote FunSETL as a new language, but rather a as a tool to understand the problem domain. FunSETL is constructed to show that it is possible to automatically incrementalize programs, when recursion is disallowed and replaced by iteration over multi-sets. FunSETL can in some sense be compared to an intermediate representation in a com-

$$
\begin{array}{rcl}
\tau & ::= & id \mid \textbf{bool} \mid \textbf{int} \mid \textbf{real} \mid \textbf{date} \mid \tau_1 + \tau_2 \mid \{lab_1 : \tau_1, \ldots, lab_k : \tau_k\} \mid \\
 & & \textbf{map}\ (\tau_1, \tau_2) \mid \textbf{mset}(\tau) \\
c & ::= & n \mid r \mid yyyy - mm - dd \mid \textbf{true} \mid \textbf{false} \\
binop & ::= & + \mid - \mid * \mid / \mid = \mid <= \mid < \mid \textbf{and} \mid \textbf{or} \mid \textbf{with} \mid \textbf{inter} \mid \textbf{union} \mid \\
 & & \textbf{diff} \mid \textbf{in} \mid \textbf{subset} \\
unop & ::= & \textbf{not} \mid \textbf{dom} \\
e & ::= & c \mid x \mid e_1\ binop\ e_2 \mid unop\ e \mid \textbf{inL}(e)\ \textbf{as}\ \tau \mid \textbf{inR}(e)\ \textbf{as}\ \tau \mid \textbf{valL}(e) \mid \\
 & & \textbf{valR}(e) \mid \{lab_1 := e_1, \ldots, lab_k := e_k\} \mid \#lab(e) \mid f(e_1, \ldots, e_m) \mid \\
 & & [] \ \textbf{as}\ \tau \mid e[e'] \mid e[e_1 \rightarrow e_1'] \mid \{\} \ \textbf{as}\ \tau \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid \\
 & & \textbf{foreach}(a, b \rightarrow e_1)\ e_2\ e_3 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\
fdecl & ::= & \textbf{fun}\ id(x_1 : t_1, \ldots, x_m : t_m) = e \\
p & ::= & fdecl_1 \ldots fdecl_k
\end{array}
$$

Figure 3.1: FunSETL syntax

```
1: fun count(s : mset(int)) = foreach(x, sum → sum + 1) 0 s
2:
3: fun average(s : mset(int)) =
4:     foreach(x, sum → x + sum) 0 s/count(s)
```

Figure 3.2: Function for computing the average of a multi-set of integers, written in FunSETL

piler. Sources where FunSETL trees could stem from are, for example: quoted F# syntax trees, LINQ queries described by Torgersen [2007], X++ data-statements, or perhaps SQL queries.

## 3.3 Incrementalization

Assume $f$ is a function and assume $f(x)$ has been computed for some $x$. Also assume that $y$ is a "small change" to $x$, such that $x \oplus y$ does not differ much from $x$, where $\oplus$ is some operation on $x$ and $y$. Furthermore assume that we would like to compute $f(x \oplus y)$. We say that $f'(x, y, r)$ is an *incremental version* or *incremental counterpart* of $f$ with respect to the *update operation* $\oplus$ (using the same terminology as Liu et al. [1998]), if

$$r = f(x) \quad \Rightarrow \quad f'(x, y, r) = f(x \oplus y) \tag{3.1}$$

for all $x$ and $y$.

In ERP systems many report functions takes as argument a multi-set of records (corresponding to a database relation). Therefore we are interested in incrementalizing these functions with respect to the update operation **with** (that is addition of an element to a multi-set).

Below we see a description of the transformations we need to do to incrementalize a function $f$ with respect to an operation $\oplus$ (that is, we wish to compute $f(x \oplus y)$ from $f(x)$). The description is assisted by doing the transformations on the *average function* seen in Figure 3.2, which is incrementalized with respect to the operation **with**, that is, addition of an element to the multi-set.

Below we describe the incrementalization process proposed by Brixen [2005], which is inspired by the work done by Liu et al. [1998] and Paige [1981], but where we have included a *normalization* step in the beginning to ease later transformations. A short overview of the incrementalization process is

- Normalize the program into a form similar to A normal form.
- Cache all intermediate results and include them in the return value.
- Incrementalize, that is, use the cache to replace computations by cache lookups.
- Prune, that is, clean up unused computations.

Below we elaborate on the above steps.

### Normalization

The program is normalized *into normal* form similar to A normal form (described by Flanagan et al. [1993]). That is, all intermediate results are **let** bound, **let** bindings are not nested, and functions are only called with identifiers or values as arguments. Applying normalization to the average example yields:

1: **fun** $count(s : \mathbf{mset}(\mathbf{int})) =$
2: $\quad$ **let** $tmp_1 = \mathbf{foreach}(x, sum \rightarrow sum + 1)\ 0\ s$ **in**
3: $\quad$ $tmp_1$
4:
5: **fun** $average(s : \mathbf{mset}(\mathbf{int})) =$
6: $\quad$ **let** $tmp_2 = \mathbf{foreach}(x, sum \rightarrow x + sum)\ 0\ s$ **in**
7: $\quad$ **let** $tmp_3 = count(s)$ **in**
8: $\quad$ **let** $tmp_4 = tmp_2/tmp_3$ **in**
9: $\quad$ $tmp_4$

### Cache all intermediate results

In the short explanation of incrementalization above, we did not mention any use of intermediate results of a computation, but they can often be used when trying to construct a more efficient version of a program. This part of the process produces from $f$ a function $\hat{f}$, which returns the value of $f$ together, with the intermediate results computed by $f$, as a record. Caching intermediate results of the normalized average function yields:

1: **fun** $count(s : \mathbf{mset}(\mathbf{int})) =$
2: $\quad$ **let** $tmp_1 = \mathbf{foreach}(x, sum \rightarrow sum + 1)\ 0\ s$ **in**
3: $\quad$ $\{1 := tmp_1; 2 := tmp_1\}$
4:
5: **fun** $average(s : \mathbf{mset}(\mathbf{int})) =$
6: $\quad$ **let** $tmp_2 = \mathbf{foreach}(x, sum \rightarrow x + sum)\ 0\ s$ **in**
7: $\quad$ **let** $tmp_3 = count(s)$ **in**
8: $\quad$ **let** $tmp_4 = tmp_2/\#1(tmp_3)$ **in**
9: $\quad$ $\{1 := tmp_4; 2 := tmp_2; 3 := tmp_3; 4 := tmp_4\}$

### Incrementalize

This step incrementalizes the function $\hat{f}$ from the previous step, by using the old return value and the intermediate results. The result of this step is called $\hat{f}'$. The incrementalization consists of the following sub-steps:

1. We do a complete syntactic *unfold* of the expression $\hat{f}(x)$. That is, $f(x)$ is unfolded to its body and then the let expressions are unfolded using the equivalence

$$\textbf{let } y = e_1 \textbf{ in } e_2 \;\equiv\; e_2[e_1/y] \tag{3.2}$$

   where $e_2[e_1/y]$ denotes capture avoiding substitution of $e_1$ for $y$ in $e_2$. This gives us an expression consisting only of a record, where each field in the record contains an expression that computes the value of the field, when $x$ is given as an argument to $\hat{f}$. We construct a cache, consisting of a mapping from the expressions that computes the fields in the record, to the field name which they compute (the cache contains the return value of $f(x)$ and the intermediate results). Applying this step to our example yields:

   1: $\{1 := \textbf{foreach}(x, sum \rightarrow x + sum)\ 0\ x\ /\#1(count(x));$
   2: $\ 2 := \textbf{foreach}(x, sum \rightarrow x + sum)\ 0\ x;$
   3: $\ 3 := count(x);$
   4: $\ 4 := \textbf{foreach}(x, sum \rightarrow x + sum)\ 0\ x\ /\#1(count(x))\}$

   This record is our *cache* $r$.

2. $\hat{f}(x \oplus y)$ is unfolded once to the body of $\hat{f}$, where $x \oplus y$ is inserted as parameters. Then we *simplify* this expression, which means that we use the semantic equivalence

$$\begin{aligned} & \textbf{foreach } (a, b \rightarrow e1)\ e2\ (S \textbf{ with } e) \\ \equiv\ & e1[e/a, \textbf{foreach } (a, b \rightarrow e1)\ e2\ S/b] \end{aligned}$$

   where it is possible. After this we try to replace expressions by cache lookups, where we use *syntactic equality* to determine, whether we can replace an expression by a cache lookup. This equivalence is what in some cases can produce an asymptotic speedup of the program, if the cache contains **foreach** $(a, b \rightarrow e1)\ e2\ S$. The resulting expression is named *body*. Applying this to the average function yields:

   1: $\quad \textbf{let } tmp_2 = y + \#2(r) \textbf{in}$
   2: $\quad \textbf{let } tmp_3 = count(x \textbf{ with } y) \textbf{ in}$
   3: $\quad \textbf{let } tmp_4 = tmp_2/\#1(tmp_3); \textbf{in}$
   4: $\quad \{1 := tmp_4; 2 := tmp_2; 3 := tmp_3; 4 := tmp_4\}$

3. We declare $\hat{f}'(x, y, r) = body$. When calling $\hat{f}'$ one should supply the previous result of calling $\hat{f}'$ as the argument $r$.

If $\hat{f}$ contains function calls they are also incrementalized during this process, but explaining how this can be done is a bit more involved, and is therefore not be presented here. Incrementalizing the average function including the call to the count function yields:

```
1: type count_rt = {1 : int; 2 : int}
2: fun  count(x : mset(int), y : int, r : count_rt) =
3:    let tmp1 = #1(r) + 1 in
4:    {1 := tmp1; 2 := tmp1}
5:
6: type average_rt = {1 : int; 2 : int; 3 : count_rt; 4 : int}
7: fun  average(x : mset(int), y : int, r : average_rt) =
8:    let tmp2 = y + #2(r) in
9:    let tmp3 = count(x, y, #3(r)) in
10:   let tmp4 = tmp2/#1(tmp3) in
11:   {1 := tmp4; 2 := tmp2; 3 := tmp3; 4 := tmp4}
```

### Prune

In this step the intermediate results, which are not used by $\hat{f}'$ is removed producing a new function $f'$, which is an incremental version of $f$ returning only the necessary intermediate results to compute efficiently. Pruning the incrementalized average function yields:

```
1: type count_rt = {1 : int}
2: fun  count(r : count_rt) =
3:    let tmp1 = #1(r) + 1 in
4:    {1 := tmp1}
5:
6: type average_rt = {1 : int; 2 : int; 3 : count_rt}
7: fun  average(y : int, r : average_rt) =
8:    let tmp2 = y + #2(r) in
9:    let tmp3 = count(#3(r)) in
10:   let tmp4 = tmp2/#1(tmp3) in
11:   {1 := tmp4; 2 := tmp2; 3 := tmp3}
```

Currently our implementation has implemented all the above steps except *pruning*.

Brixen [2005] describes, how the above method can be can be made fully automatic, by providing an algorithm to perform automatic incrementalization. The work presented by Liu et al. [1998] can however not be made fully automatic, because the incrementalization method has a problem, when in-

crementalizing sub-calls, since the language contains recursion. The method explained by Brixen [2005] could also be improved, since we use syntactic equality to check, if it is possible to replace code with a cache retrieval. Instead we could use a theorem prover to check, if code from the cache is equivalent to code in the program we wish to incrementalize, thereby making it possible to do more retrievals.

### 3.3.1 Limitations of Our Approach

FunSETL has limited expressive power and currently it does not have good support for ordered data structures like trees or lists. One could simulate ordered data structures using multi-sets with records containing auxiliary information describing, how the elements are ordered. However, there is no good way of iterating through the ordered data, since the elements of a multi-set is not stored in a particular order, and since we can only iterate through the multi-set using a **foreach** expression.

Some of the report functions in ERP systems uses ordered data structures, for example when computing *ranking functions*, which essentially is breaking a set into disjoint subsets, and then ordering the elements in each of the subsets.

## 3.4 Implementation

To test whether simple iteration (in the form of **foreach**-loops) is adequate for writing report functions, and to test that the incrementalization algorithm can handle the report functions we want, we have implemented a simple test incrementalizer and compiler for FunSETL and have assembled a suite of small report functions.

### 3.4.1 Incrementalizer

The *incrementalizer* is a program that transforms a FunSETL program into its incremental counterpart. Incrementalization is an automatic version of the FunSETL program transformation described in Section 3.3. The *incrementalizer* takes three arguments

1 A File containing a *specification of an update operation.*

2 A File containing the FunSETL program to be incrementalized.

3 A File name to write the incrementalized program to.

As an example of how a specification of an update operation looks like, we use the average function from Figure 3.2. The average function takes as argument a multi-set of integers, and it should be incrementalized with respect to adding one integer to the multi-set. This can be specified in a file, which should contain (this is an actual specification from a file):

```
[x : mset(int)]
[y : int]
[x with y]
```

The first line tells the incrementalizer what the argument to the "earlier" call of the function was. The second line defines the "change" and the last line defines, how the old argument should be combined with the change (that is $x \oplus y$).

One could imagine that since incremental versions of programs often are *harder* (more complex) to write than their non-incremental counterparts, some of the *hard* work could have been pushed to the specification of an update operation. The above example should illustrate that this is not case.

The implementation of the incrementalizer is done in F#, a programming language similar to O'Caml (explained by Syme et al. [2007]).

### 3.4.2 Compiler

The compiler is also implemented using F# and it compiles a FunSETL program into a C# program. C# is the target language, because we are also working on integrating and testing FunSETL with Microsoft NAV, which is a Microsoft developed ERP system that supports communication through .NET.

The compiler takes a FunSETL program and a file name for which to write the compiled output to. Furthermore the compiler also takes an optional flag, which enables counting of the number of FunSETL operations used, when running the program. We have categorized FunSETL operations as follows:

- **Arithmetic**: Standard arithmetic operations $+, -, *$ and $/$, but also all boolean operations are put into this category.
- **Record**: Construction and destruction of records.
- **Control**: Conditionals (if-then-else).
- **Sum**: Construction and destruction of sums.

| Program | Description |
|---|---|
| sum | Sums a multi-set of integers |
| max | Maximum of a multi-set of integers |
| min | Minimum of a multi-set of integers |
| count | Counts the number of elements in a multi-set of integers |
| average | Takes the average of a multi-set of integers |
| reverse-index | Generates a reverse index of an index |
| groupbysum | Constructs a group by on an attribute |
| financial-statement | Company accounting information |

Figure 3.3: Overview of the benchmark suite programs

- **FinMap**: All operations on finite maps including construction and lookup.
- **Mset**: All operations in multi-sets including construction.

## 3.5 Experimental Results

To test the compiler we have assembled a suite of small report functions. Most of these are micro-benchmarks to test a specific syntactic form or certain idioms. Furthermore we have ported a report function, Financial statement, from Microsoft Dynamics AX to FunSETL. The Financial statement report function is treated in detail in Section 3.5.2.

### 3.5.1 Benchmark Suite

In Figure 3.3 there is a short description of which programs our benchmark suite contains. All the programs in the benchmark suite have been automatically incrementalized. The incremental version gives a linear factor asymptotic speedup in the computation.

### 3.5.2 Financial statement

The Microsoft Dynamics AX ERP system contains a database with many tables, and the Financial statement uses of only a couple of these. To simplify the testing we have projected only the necessary columns from the

tables that are used when computing the Financial statement. The data can in FunSETL be specified by:

1: **type** $event = \{accountnr : \mathbf{int}, amount : \mathbf{real}, time : \mathbf{date}\}$
2: **type** $dimensions = \mathbf{map}\ (\mathbf{string}, \mathbf{mset}(\{start : \mathbf{int}, stop : \mathbf{int}\})$
3: **type** $interval = \{start : \mathbf{date}, stop : \mathbf{date}\}$

An *event* is a change of *balance* by amount *amount* on an account *accountnr* at time *time*. The *dimensions* type describes mappings from *result-names* to multi-sets of account intervals and the interval type describes time intervals.

The Financial statement takes as input a multi-set of events, a dimensions mapping and a time interval and returns a finite map. The dimensions mapping describes the output of the financial statement in the following way: The result is a map from strings to reals and the domain of the dimensions map is used as the domain of the resulting map. Each element in the domain of the result is mapped to the sum of the balance changes of all the events, that lie in one of the account-intervals for the given element specified by the dimensions map. The implementation of the financial statement can be seen in Appendix 3.8.

The financial statement is incrementalized with respect to adding an event to the multi-set of events and the dimensions and interval arguments are held constant. The dimensions argument describes the accounting system of the company and can be assumed not to change (if it changes, then none of the computed results make sense anymore, and everything needs to be computed from scratch). For space reasons the incrementalized code has not been included.

The Financial statement is an important, since it is required by law. The current implementation of the Financial statement in Microsoft Dynamics AX is a non-incremental version.

### 3.5.3 The Data-set

We have retrieved a data-set from a real company, where the dimensions table describes that the following is computed:

- Sumclass computations (balance of account intervals $X000 - X999$, where $X = 1, 2, 3, 4, 5, 6, 7, 8$ and from $9,000$ up to $999,999$. These summation data are named *sumclassX* for each $X$. That is, the dimensions map contains the mapping "sumclass0" to {{start := 0000, stop :=0999}} and correspondingly for the other sum-classes.
- Assets, liabilities, "Aufwand", "Ertrag", "Bestandskonten" and "Erfolgskonten", which all can be described by the dimensions map, where

the resulting multi-set contains at most 6 account-intervals for any of these.

The use of the German names is due to the accounting system in the German company, which has delivered the data-set.

### 3.5.4 Setting up the experiment and expectations

The experiment contains two parts

1. An experiment suggesting that there is an asymptotic decrease in the number of operations used, when computing the financial statement using the incremental version instead of the non-incremental version.

2. An experiment suggesting that we get a performance increase in the time usage, when using the incremental version instead of the non-incremental.

The idea is that both experiments should simulate a growing database of events in a company. Both experiments should measure, how many resources (be that operations or time) that is needed to get the result of the financial statement, when the database contains 10,000 events, 20,000 events, ... , 100,000 events. This means, that when using the non-incremental version we compute the financial statement on a multi-set containing 10,000 events, then again on the multi-set containing 20,000 events and so on. Thus, the non-incremental financial statement is computed 10 times. The incremental version is computed by starting with an empty multi-set and then adding 1 event at the time and thereby computing the financial statement incrementally. This means that the result of the financial statement is available after adding any number of events, whereas it is only available after 10,000 events, 20,000 events, ..., and 100,000 events when using the non-incremental version.

As we expect that the resource usage of the non-incremental version is proportional to the number of events (running it once), we expect that the resource usage of maintaining the result of the financial statement is (that be operation count or time usage)

$$r = c_{non-inc} \cdot c_{available} \cdot \sum_{i=1}^{t} i = c_{non-inc} \cdot c_{available} \cdot \sum_{i=1}^{t} i \in \mathcal{O}(n^2) \qquad (3.3)$$

for some constant $c_{non-inc}$, where $c_{available}$ is a constant describing how often the financial statement should be recomputed and where the total number of

events added is $n = t \cdot c_{available}$ (in our experiment $c_{available} = 10,000$). That is, if we want the result of the financial statement to be available between each addition of $c_{available}$ events then the resource usage of the financial statement is $\mathcal{O}(n^2)$.

Furthermore we expect that the incremental version of the financial statement can be recomputed in $\mathcal{O}(1)$ resources (that be operations or time), when adding one event to the multi-set of events. That is, we expect that the resource usage of the incremental version

$$r = c_{inc} \cdot n \in \mathcal{O}(n) \tag{3.4}$$

where $c_{inc}$ is some constant and where $n$ is the number of events added.

### 3.5.5 Performing and analyzing the experiment

In this section we perform and discuss the experiments of the Financial statement described above.

For the Financial statement all operations used can be performed in constant time, because the only FunSETL operations that could take non-constant time are some of the multi-set and map-operations, which are not used here (even though we update finite maps, the size of the maps does not chance and therefore we can assume that these operations are in constant time).

In Figure 3.4 the second and third column contains the accumulated number of operations used, when computing the financial statement on a multi-set containing from 10,000-100,000 events. That is, the i'th row contains the number of operations used, when computing the financial statement on a multi-set with $i \cdot 10,000$ events plus the number of operations in the i-1'th column. The operations are accumulated to simulate, how many operations are needed, when the result of the financial statement should be available every time 10,000 events are added.

From the first row of Figure 3.4 we see that incrementalizing the Financial statement introduces some overhead, because the number of operations are larger than in the non-incremental case, when running the financial statement on 10,000 events only. But as the other rows indicate the number of operations needed by the non-incremental version exceeds the number of operations used by the incremental version already after about 40,000-50,000 events, that is, 4-5 re-computations of the non-incremental version.

If we plot (events,operation count) from Figure 3.4 from the non-incremental version in a coordinate system, we would see that a quadratic degree poly-

| Events | Accumulated operation count | | Accumulated running time (sec) | |
|---|---|---|---|---|
| | Non-incremental | Incremental | Non-incremental | Incremental |
| 10,000 | 3,330,036 | 8,170,021 | 0.23 | 0.50 |
| 20,000 | 9,990,054 | 16,340,021 | 0.67 | 0.98 |
| 30,000 | 19,980,072 | 24,510,021 | 1.34 | 1.47 |
| 40,000 | 33,300,090 | 32,680,021 | 2.23 | 1.95 |
| 50,000 | 49,950,108 | 40,850,021 | 3.32 | 2.42 |
| 60,000 | 69,930,126 | 49,020,021 | 4.65 | 2.90 |
| 70,000 | 93,240,144 | 57,190,021 | 6.22 | 3.39 |
| 80,000 | 119,880,162 | 65,360,021 | 8.11 | 3.87 |
| 90,000 | 149,850,180 | 73,530,021 | 10.20 | 4.34 |
| 100,000 | 183,150,198 | 81,700,021 | 12.50 | 4.82 |

Figure 3.4: Accumulated resource usage for the non-incremental and incremental version of the financial statement

nomial would fit nicely through these points. For the incremental version a first degree polynomial would fit, thus confirming our expectations.
ad 2: In Figure 3.4 column four and five we see the time usage of running the incremental and non-incremental version of the financial statement on multi-sets containing 10,000 - 100,000 events. Like above the time usage is also accumulated to simulate the time usage for maintaining the results.

In Figure 3.5 we see a plot of the "Accumulated Running time" numbers from Figure 3.4. The "crosses" describe the time usage of the non-incremental version and the gray curve is the best fitting polynomial of degree two through these points. The "circles" describe the time usage of the incremental version and the black line is the best fitting polynomial of degree one through these points. That is, the experiment confirms that the time usage of the non-incremental version is $(\mathcal{O}(n^2))$ and for the incremental version it is $(\mathcal{O}(n))$ as expected. Using any reasonable model of time and resource usage of FunSETL, we would arrive at the same conclusion as above, when inspecting the FunSETL code for both the non-incremental and incremental program.

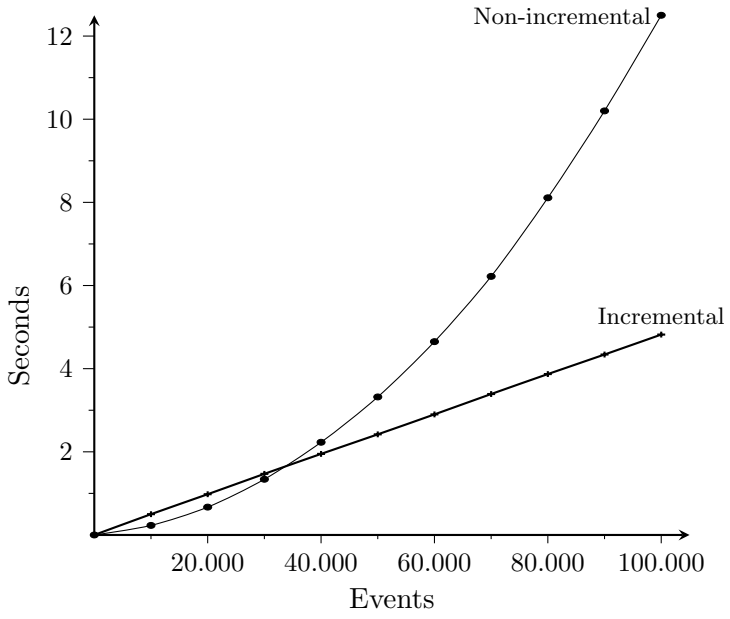


Figure 3.5: Accumulated time usage for the financial statement

## 3.6 Related Work

The language FunSETL and the automatic incrementalization technique presented in this article is closely related to the work by Liu et al. [1998] and Paige and Koenig [1982]. As described in Section 3.3 we are basically using the same algorithm as Liu et al. [1998] adapted by Brixen [2005] to with a simple iteration construct rather than general recursion. Our contribution is to simplify the description to make the incrementalization algorithm more accessible, and we have more thoroughly tested whether it is feasible to use a FunSETL like language for business reporting by collecting a benchmark of representative report functions.

To aide report function generation some programming languages have build-in support for database manipulation. A new example on this is the Language Integrated Query (LINQ) framework explained by Torgersen [2007] or *data-statements* in X++ (described by Greef et al. [2006]). FunSETL is currently not a competitor to these general-purpose languages. It is rather an optimization technique we hope can be adapted to be used in these languages.

MapReduce (presented by Dean and Ghemawat [2008]) is based on a computational model with similar restrictions as FunSETL. Also to make it easy to express report functions and to enable automatic optimization. But the optimization done by MapReduce is *automatic parallelization* rather than incrementalization. The two approaches are somewhat related and we plan to investigate how they can be combined in the future.

OLAP (OnLine Analytical Processing) explained by Lewis et al. [2002] is a way to specify, on which *dimensions* you would like to do aggregate computations. Hence, OLAP enables fast report function queries on the aggregated data in an OLAP cube, but since a cube maintains much aggregate information you often maintain more than what is needed. This is not the case for incrementalized report functions, since only necessary information is maintained after the pruning phase, but OLAP has the advantage that you sometimes can compute report functions that you could not foresee that you would need.

## 3.7 Conclusion

Summary of contributions: We have made a benchmark of representative report functions, and have thus demonstrated that it seems feasible and practical to use automatic incrementalization in the domain of business re-

port functions. We have shown that we gain an asymptotic speedup by a linear factor on a real life report function when applying automatic incrementalization. Furthermore, we have simplified the presentation of the incrementalization algorithm presented by Liu et al. [1998] and Brixen [2005]. We have revealed some weaknesses in current design of FunSETL, such as the bad support for ordered data-structures and operations on these, which are used in some report functions in ERP systems.

## 3.8 Financial statement

```
type event = {accountnr : int, amount : real, time : date}
type eventset = mset(event)
type range = {start : int, stop : int}
type dimensions = map(string,mset(range))
type interval = {start : date, stop : date}


fun inInterval(e : event, I : interval) =
    #start(I) <= #time(e) and #time(e) <= #stop(I)

fun inRanges(x : event, r : mset(range)) =
    foreach (a,b => b or (#start(a) <= #accountnr(x) and
                    #accountnr(x) <= #stop(a))) false r

fun financial_statement_sub (e : event, m : map(string,real),
                             dims : dimensions, I : interval) =
    if inInterval(e,I)
    then let dimnames = dom(dims)
         in  foreach(name,b => if inRanges(e,(dims)[name])
                               then (b)[name -> (b)[name] + #amount(e)]
                               else b) m dimnames
    else m

fun initmap (dims : dimensions) =
    let dimnames = dom(dims) in
        foreach(a,m => (m)[a -> 0.0]) ([] as map(string,real)) dimnames

fun financial_statement (log : eventset, dims : dimensions, I : interval) =
    foreach (e,acc => financial_statement_sub(e,acc,dims,I)) initmap(dims) log
```

# Chapter 4

# Technical Report: Finite Differencing

If we have a polynomial $p$ and have computed the values $p(n), \Delta p(n), \Delta^2 p(n), \ldots, \Delta^{(deg(p))} p(n)$ for some $n \in \mathbb{N}$ then there is an easy and fast way to compute the value $p(n+1)$ incrementally from the before mentioned values. This is an application of the finite differencing theory on polynomials.

We propose a generalization of the theory for finite differencing that applies to functions, which as argument takes elements from abstract structures called *incremental preorders*. Furthermore, we show a version of the *chain rule*, *integrals* and *Taylor expansion* for this theory and show how all this can be applied to compute functions incrementally.

First we introduce the finite differencing theory by using it on polynomials and then we explain the generalized theory of finite differencing, keeping the finite differencing theory on polynomials as an analogy. Throughout this chapter we include numerous examples to show how the theory can be applied.

## 4.1 Finite Differencing of Polynomials

The theory of difference functions is described by Levy and Lessman [1992][chapter 1] and by Graham et al. [2004][chapter 2].

### 4.1.1 Polynomials and Their Degree

First we define what we mean by a polynomial.

**Definition 4.1.1 (Polynomials)** *Assume $p : \mathbb{Z} \to \mathbb{Z}$ is a function, which has the form*

$$p(n) = \sum_{i=0}^{k} a_i n^i$$

*for some $k \in \mathbb{N}_0$ and where $a_i \in \mathbb{Z}$ for all $0 \leq i \leq k$. $a_1, \ldots, a_k$ are called the* coefficients *and $a_k \neq 0$. The polynomial $p(n) = 0$ is called the* zero polynomial*. Furthermore the set of polynomials is denoted $\mathbb{Z}[X]$.*

Throughout this chapter $p(n)$ denotes a polynomial.

The theory explained in this section works equally well on polynomials with coefficients from $\mathbb{R}$ and considered as functions on $\mathbb{R}$. However, since we only study discrete math in the rest of this report, we thought that the intuition might as well be kept in the discrete world.

**Definition 4.1.2 (Degree of a polynomial)** *The degree of a polynomial is denoted as* $\deg(p(n))$ *and it is defined as*

$$\begin{aligned} \deg(\sum_{i=0}^{k} a_i n^i) &= k \\ \deg(0) &= -\infty \end{aligned}$$

*($k$ is the power of the highest degree term of the polynomial with a non-zero coefficient.) If all coefficients of the polynomial are zero we say that the degree is $-\infty$.*

Note, that from the definition of degree, the zero polynomial has lower degree than any other polynomial.

**Example 4.1.3** *An example of a polynomial and its degree.*

Define

$$p(n) = n^3 + 3n - 2 \tag{4.1}$$

Then $\deg(p(n)) = 3$ ■

### 4.1.2 The Difference Operator

The difference operator on polynomials is a discrete analogue to the differential operator from Calculus.

**Definition 4.1.4 (Difference operator)** *The difference operator* $\Delta : \mathbb{Z}[X] \rightarrow \mathbb{Z}[X]$ *is defined as*

$$(\Delta(p))(n) = p(n+1) - p(n)$$

That is, the difference operator is a higher order function, because it takes a polynomial function as argument and returns a polynomial function. We write $\Delta p(n)$ for $(\Delta(p))(n)$. Since, we often apply the difference operator more than once, we introduce the following notation.

**Notation 4.1.5 ($\Delta^i$)** *Several applications of the difference operator is notated as:*

$$\begin{aligned} \Delta^0 p(n) &= p(n) \\ \Delta^i p(n) &= \Delta(\Delta^{i-1} p)(n) \end{aligned}$$

*for all* $i \in \mathbb{N}$.

Let us see an example.

**Example 4.1.6** *Applying the difference operator to the polynomial* $p(n) = n^3 + 3n - 2$ *from Example 4.1.3.*

$$\begin{aligned} \Delta p(n) &= ((n+1)^3 + 3(n+1) - 2) - (n^3 + 3n - 2) \\ &= ((n^3 + 3n^2 + 3n + 1) + (3n + 3) - 2) - (n^3 + 3n - 2) \\ &= 3n^2 + 3n + 4 \end{aligned}$$

Furthermore

$$\begin{aligned} \Delta^2 p(n) &= 6n + 6 \\ \Delta^3 p(n) &= 6 \\ \Delta^4 p(n) &= 0 \end{aligned}$$

■

Note, that the difference polynomial of a polynomial has lower degree than the polynomial itself. This holds in general, which the following lemma shows.

**Lemma 4.1.7 (Difference has lower degree)** *Simple properties of the difference operator*

*1* $\Delta(p_1 + p_2)(n) = \Delta p_1(n) + \Delta p_2(n)$.

*2 Assume $k \in \mathbb{Z}$ and assume that for all $n \in \mathbb{Z}$ we have $p_1(n) = p_2(n+k)$. Then*

$$\Delta p_1(n) = \Delta p_2(n+k)$$

*3 If $p(n) \neq 0$ then $\deg(\Delta p(n)) < \deg(p(n))$.*

Proof:
ad 1: Since

$$\begin{aligned}
\Delta(p_1 + p_2)(n) &= (p_1 + p_2)(n+1) - (p_1 + p_2)(n) \\
&= p_1(n+1) + p_2(n+1) - (p_1(n) + p_2(n)) \\
&= p_1(n+1) - p_1(n) + p_2(n+1) - p_2(n) \\
&= \Delta p_1(n) + \Delta p_2(n)
\end{aligned}$$

we have completed this part of the proof.
ad 2: Since

$$\begin{aligned}
\Delta p_1(n) &= p_1(n+1) - p_1(n) \\
&= p_2(n+1+k) - p_2(n+k) \\
&= p_2((n+k)+1) - p_2(n+k) \\
&= \Delta p_2(n+k)
\end{aligned}$$

we have completed this part of the proof.
ad 3: Since $p(n)$ is not the zero polynomial, at least one of its coefficients must be non-zero. Thus, there exists $k \in \mathbb{N}_0$ such that $\deg(p(n)) = k$ and coefficients $a_0, \ldots, a_k \in \mathbb{Z}$ such that

$$p(n) = \sum_{i=0}^{k} a_i n^i$$

where $a_k \neq 0$. Since

$$\begin{aligned}
\Delta p(n) &= p(n+1) - p(n) && (4.2) \\
&= \sum_{i=0}^{k} a_i((n+1)^i - n^i) && (4.3) \\
&= \sum_{i=0}^{k} a_i \left( \sum_{j=0}^{i} \binom{i}{j} n^j - n^i \right) && (4.4) \\
&= \sum_{i=0}^{k} a_i \left( \sum_{j=0}^{i-1} \binom{i}{j} n^j \right) && (4.5)
\end{aligned}$$

(4.4): Follows from the binomial formula.
Inspecting (4.5) we see that there is no term of degree $k$ anymore and no terms of higher degree, and if $k = 0$ then $\Delta p(n) = 0$ thus proving that $\deg(\Delta p(n)) < \deg(p(n))$. ■

This leads to the corollary.

**Corollary 4.1.8** *Assume $p(n)$ is a non-zero polynomial and $\deg(p(n)) = k$ then $\Delta^k p(n) = a$ for some $a \in \mathbb{Z}$*

Proof: Induction on $k$ and using Lemma 4.1.7 part 3. ■

**Corollary 4.1.9** *For all $i \in \mathbb{N}_0$ we have*

$$\Delta^i p(n+1) = \Delta^{i+1} p(n) + \Delta^i p(n)$$

Proof: Induction on $i$.
$i = 0$: By definition we have $\Delta^1 p(n) = \Delta^0 p(n+1) - \Delta^0 p(n)$ and thus $\Delta^0 p(n+1) = \Delta^1 p(n) + \Delta^0 p(n)$, which completes the proof of the basic case.
$i > 0$: Assume the theorem holds for smaller $i$. That is, assume (induction hypothesis)

$$\begin{aligned} \Delta^{i-1} p(n+1) &= \Delta^i p(n) + \Delta^{i-1} p(n) \\ &= (\Delta^i p + \Delta^{i-1} p)(n) \end{aligned}$$

Thus

$$\begin{aligned} \Delta^i p(n+1) &= \Delta\Delta^{i-1} p(n+1) && (4.6) \\ &= \Delta(\Delta^i p + \Delta^{i-1} p)(n) && (4.7) \\ &= \Delta\Delta^i p(n) + \Delta\Delta^{i-1} p(n) && (4.8) \\ &= \Delta^{i+1} p(n) + \Delta^i p(n) && (4.9) \end{aligned}$$

(4.7): Induction hypothesis and Lemma 4.1.7 part 2.
(4.8): Lemma 4.1.7 part 1.
This completes the proof. ■

### 4.1.3 Incremental Computation of Polynomial Values

Let us see, how we can use the preceding theory to compute the values of a polynomial incrementally. That is, assume we are given a polynomial $p(n)$ of degree $k$ and we wish to compute the values $p(0), p(1), p(2), \ldots$. By Corollary 4.1.8 we know that $\Delta^k p(n) = a$ for some $a \in \mathbb{Z}$. This means that

$$\Delta^{k+1} p(n) = p(n+1) - p(n) = a - a = 0$$

By Corollary 4.1.9 we get that

$$\Delta^k p(n+1) = \Delta^{k+1} p(n) + \Delta^k p(n) = \Delta^k p(n)$$

Thus, in general we have

$$\begin{pmatrix} \Delta^0 p(n+1) \\ \Delta^1 p(n+1) \\ \vdots \\ \Delta^{k-1} p(n+1) \\ \Delta^k p(n+1) \end{pmatrix} = \begin{pmatrix} \Delta^1 p(n) + \Delta^0 p(n) \\ \Delta^2 p(n) + \Delta^1 p(n) \\ \vdots \\ \Delta^k p(n) + \Delta^{k-1} p(n) \\ \Delta^k p(n) \end{pmatrix}$$

This means that, if we know $\Delta^0 p(n), \Delta^1 p(n), \ldots, \Delta^k p(n)$ then we can compute $\Delta^0 p(n+1), \Delta^1 p(n+1), \ldots, \Delta^k p(n+1)$ by doing only $k$ additions. That is, we can compute the $\Delta^i p(n+1)$ values from the $\Delta^i p(n)$ values incrementally. Let us apply this technique to our example polynomial.

**Example 4.1.10** *Incremental computations of the values of* $p(n) = n^3 + 3n - 2$.

From Example 4.1.6 we know that (by definition $\Delta^0 p(n) = p(n)$)

$$\begin{aligned} \Delta^0 p(n) &= n^3 + 3n - 2 \\ \Delta^1 p(n) &= 3n^2 + 3n + 4 \\ \Delta^2 p(n) &= 6n + 6 \\ \Delta^3 p(n) &= 6 \\ \Delta^4 p(n) &= 0 \end{aligned}$$

Now we evaluate all the difference polynomials for $n = 0$, that is, we extract the constant coefficient from each polynomial. Thus $\Delta^0 p(0) = -2, \Delta^1 p(0) = 4, \Delta^2 p(0) = 6, \Delta^3 p(0) = 6$. Therefore

$$\begin{pmatrix} \Delta^0 p(1) \\ \Delta^1 p(1) \\ \Delta^2 p(1) \\ \Delta^3 p(1) \end{pmatrix} = \begin{pmatrix} \Delta^1 p(0) + \Delta^0 p(0) \\ \Delta^2 p(0) + \Delta^1 p(0) \\ \Delta^3 p(0) + \Delta^2 p(0) \\ \Delta^3 p(0) \end{pmatrix} = \begin{pmatrix} 4 + (-2) \\ 6 + 4 \\ 6 + 6 \\ 6 \end{pmatrix} = \begin{pmatrix} 2 \\ 10 \\ 12 \\ 6 \end{pmatrix}$$

Thus $p(1) = 2$. Furthermore

$$\begin{pmatrix} \Delta^0 p(2) \\ \Delta^1 p(2) \\ \Delta^2 p(2) \\ \Delta^3 p(2) \end{pmatrix} = \begin{pmatrix} \Delta^1 p(1) + \Delta^0 p(1) \\ \Delta^2 p(1) + \Delta^1 p(1) \\ \Delta^3 p(1) + \Delta^2 p(1) \\ \Delta^3 p(1) \end{pmatrix} = \begin{pmatrix} 10 + 2 \\ 12 + 10 \\ 6 + 12 \\ 6 \end{pmatrix} = \begin{pmatrix} 12 \\ 22 \\ 18 \\ 6 \end{pmatrix}$$

Thus is $p(2) = 12$. ■

The example above shows how we can compute $p(n+1)$ by storing a sufficient set of intermediate values $\Delta^i p(n)$ (pieces of information). The idea is that we want to generalize the difference operator to functions which uses other kinds of update to the input than $+1$.

## 4.2 Difference Calculus

In this section we develop the basic theory of a difference calculus on functions, working on the elements of abstract structures called incremental preorders. The ideas in this section are based on the ideas of finite differencing of polynomials, which was explained in Section 4.1. Finite differencing as described by Graham et al. [2004] and Levy and Lessman [1992] are based on functions working on integers or real numbers, where we have hardcoded the update operation into the difference operator, that is, $\Delta f(x) = f(x+1) - f(x)$ ($+1$ is hardcoded into the right hand side). We generalize this idea, such that we can find the difference of functions working on lists, where there is no canonical update to $x$ like $+1$ in the case of integers and reals.

Therefore we first introduce the notion of incremental preorders, which essentially are monoids with some extra requirements.

### 4.2.1 Incremental Preorders

**Definition 4.2.1 (Monoid)** *A monoid $M = (S, \oplus)$ consists of a set $S$, a function $\oplus : S \times S \to S$ and a neutral element $0 \in S$ such that*

$$\begin{aligned} x \oplus (y \oplus z) &= (x \oplus y) \oplus z \\ x \oplus 0 = x &= 0 \oplus x \end{aligned}$$

*for all $x, y, z \in S$.*

The neutral element of a monoid has the following property:

**Proposition 4.2.2 (Neutral element of a monoid is unique)**

Proof: Let $0$ and $0'$ be neutral elements. Then

$$0 = 0' \oplus 0 = 0'$$

■

Thus, we write $0_S$ to denote the unique zero element of the monoid $(S, \oplus)$, and if it does not lead to confusion we just write 0.

Before we introduce incremental preorders we need to have a notion of factorization of elements in a monoid with respect to a set $D$.

**Definition 4.2.3 (Factorization)** *Assume $(S, \oplus)$ is a monoid and assume $D \subseteq S$. We say that $S$ can be factorized with respect to $D$ if:*

1 *For all* $x \in S$ *then* $x$ *can be factorized with respect to* $D$*, which is the case, if one of the following conditions hold:*

   a. $x = 0$

   b. $x = d' \oplus x'$ *for some* $x' \in S$ *and* $d' \in D$*, where* $x'$ *also can be factorized with respect to* $D$*.*

2 $0 \neq d \oplus x$ *for all* $x \in S$ *and* $d \in D$*.*

Note, that the definition of factorization of elements is inductive, and that requirement 2 ensures that only rule 1.a can be applied to show that 0 can be factorized. Now we are ready to introduce incremental preorders.

**Definition 4.2.4 (Incremental preorder)** *Assume* $(S, \oplus)$ *is a monoid and assume that* $D$ *is a set (which we call* increments*). Then we say that* $(S, \oplus)_D$ *is an* incremental preorder *(also denoted* ipo*), if*

1 $S$ *can be factorized with respect to* $D$*.*

2 $\forall x, x' \in S\, \forall d, d' \in D.\ (d \oplus x = d' \oplus x' \Rightarrow d = d' \wedge x = x')$

*Requirement 1 is referred to as the "existence of factorization" and requirement 2 as the "uniqueness of factorization". Furthermore,* $S$ *is referred to as the* value set*.*

Let us see some examples of ipo's.

**Example 4.2.5 (ipo's)** *Examples of ipo's are natural numbers, lists of natural numbers, lists of lists of natural numbers and singleton lists of natural numbers.*

- $(\mathbb{N}_0, +)_{\{1\}}$: The natural numbers, where 0 is the zero element, can be factorized uniquely with + and 1, which corresponds to the unary representation of the natural numbers.
- $(\mathbb{N}_0^*, @)_{[\mathbb{N}_0]}$: $\mathbb{N}_0^*$ is the set of lists of numbers from $\mathbb{N}_0$. @ denotes the append operator, [] (zero element) is the empty list and the set $[\mathbb{N}_0]$ of singleton lists is defined as $[\mathbb{N}_0] = \{[n] \mid n \in \mathbb{N}_0\}$. Thus @ provide a unique factorization of $\mathbb{N}_0^*$.
- $((\mathbb{N}_0^*)^*, @)_{[\mathbb{N}_0^*]}$: $((\mathbb{N}_0^*)^*, @)$ is the set of lists of lists of natural numbers and $[\mathbb{N}_0^*]$ is the set of singleton lists of natural numbers.

- $([\mathbb{N}_0], +@)_{[1]}$: $+@$ is defined as $[n_1] +@ [n_2] = [n_1 + n_2]$ and $[0]$ is the zero element. Since $(\mathbb{N}_0, +)_{\{1\}}$ is an ipo and since $+@$ inherits the factorization property from $+$ then $([\mathbb{N}_0], +@)_{[1]}$ is an ipo.

In general we can write $(D^*, @)_{[D]}$ for the ipo consisting of lists of elements from the set $D$. Note, that if we have an ipo $(S, \oplus)_D$ we can construct a singleton list ipo $([S], \oplus@)_{[D]}$ from this ipo, where the sets, operations and zero element are defined like in the last ipo example from above.

Since we have unique factorizations of elements in ipo's, we can define the size of elements.

**Definition 4.2.6 (Size)** *Assume $(S, \oplus)_D$ is an ipo. Assume $x \in S$ and $d \in D$ then we define $|\cdot| : S \to \mathbb{N}_0$*

$$\begin{aligned} |0_S| &= 0 \\ |d \oplus x| &= 1 + |x| \end{aligned}$$

The size function makes sense since factorizations are unique, which the following lemma shows.

**Lemma 4.2.7 (Size)** *The definition of the size function makes sense.*

Proof: By rule induction on the definition of factorization of element.
rule a: $x = 0$ can be factorized by this rule and since we require that $0 \neq d \oplus x$ for all $x \in S$ and $d \in D$, we get that $|0_S| = 0$ is well-defined.
rule b: Assume that $x = d' \oplus x' = d'' \oplus x''$. By induction we get that $|x'|$ and $|x''|$ are well-defined. Furthermore, by requirement 2 of ipo's we get that $x' = x''$ and thus $|x'| = |x''|$. Therefore

$$|d' \oplus x'| = 1 + |x'| = 1 + |x''| = |d'' \oplus x''|$$

which proves that $|x|$ is well-defined. ■

**Lemma 4.2.8** *Assume $(S, \oplus)_D$ is an ipo. Then for all $x, y \in S$ we have*

$$|x \oplus y| = |x| + |y|$$

Proof: By induction on $|x|$.
$|x| = 0$: Thus $x = 0$ and therefore

$$|x \oplus y| = |0 \oplus y| = |y| = 0 + |y| = |x| + |y|$$

which completes the basic case of the lemma.
$|x| > 0$: That is, there exist $x' \in S$ and $d' \in D$ such that $x = d' \oplus x'$ and thus

$$\begin{aligned} |x \oplus y| &= |d' \oplus x' \oplus y| \\ &= 1 + |x' \oplus y| \\ &= 1 + |x'| + |y| \\ &= |d' \oplus x'| + |y| \\ &= |x| + |y| \end{aligned}$$

where the third equality follows by induction because $|x| = 1 + |x'| > |x'|$. This proves the lemma. ■

**Lemma 4.2.9 (Factorizations in ipo's)** *Assume $(S, \oplus)_D$ is an ipo. Then*

$$\forall x_1, x_2, y \in S. \quad (x_1 \oplus y = x_2 \oplus y \;\Rightarrow\; x_1 = x_2)$$

Proof: Induction on $|x_1|$.
$|x_1| = 0$: This means that

$$\begin{aligned} |x_2| + |y| &= |x_2 \oplus y| \\ &= |x_1 \oplus y| \\ &= |x_1| + |y| \\ &= |y| \end{aligned}$$

where the first and third equality follows from Lemma 4.2.8. Thus $|x_2| = 0$ and therefore $x_2 = 0$, which completes the proof in the basic case.
$|x_1| > 0$: First we note that it must be the case that $|x_2| > 0$, because if $|x_2| = 0$ then

$$|y| < |x_1| + |y| = |x_1 \oplus y| = |x_2 \oplus y| = |x_2| + |y| = |y|$$

which gives a contradiction. This means that there exists $x', x'' \in S$ and $d', d'' \in D$ such that $x_1 = d' \oplus x'$ and $x_2 = d'' \oplus x''$ where $d' \oplus x' \oplus y = d'' \oplus x'' \oplus y$. By requirement 2 of an ipo we get that $d' = d''$ and $x' \oplus y = x'' \oplus y$. Since $|x_1| = 1 + |x'| > |x'|$ we get by induction that $x' = x''$ and because $d' = d''$ we get that

$$x_1 = d' \oplus x' = d'' \oplus x'' = x_2$$

which completes the proof. ■

### 4.2.2 Preorder on Value Sets

Next we define an ordering on the value set $S$ in an ipo and show that it is a preorder.

**Definition 4.2.10 (Relation $\sqsubseteq$ on $S$)** *Assume $(S, \oplus)_D$ is a ipo. Define the relation $\sqsubseteq_S \subseteq S \times S$ as*

$$x \sqsubseteq_S y \quad \textit{if} \quad \exists x' \in S.\ y = x' \oplus x$$

If the value set $S$ is understood from the context we write $\sqsubseteq$ instead of $\sqsubseteq_S$.

**Lemma 4.2.11 ($\sqsubseteq_S$ is a preorder)** *Assume $(S, \oplus)_D$ is an ipo. Then $\sqsubseteq_S$ is a preorder on $S$.*

Proof: We need to show:

1 $\forall x \in S. \quad x \sqsubseteq x$

2 $\forall x, y, z \in S. \quad (x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z)$

ad 1: Assume $x \in S$. Choose $x' = 0$. Since $x = 0 \oplus x$ then by definition we are done.
ad 2: Assume $x, y, z \in S$, $x \sqsubseteq y$ and $y \sqsubseteq z$. That is, there exists $x', x'' \in S$ such that

$$y = x' \oplus x$$
$$z = x'' \oplus y$$

Composing the above equations we get that

$$z = x'' \oplus (x' \oplus x) = (x'' \oplus x') \oplus x$$

Thus $x \sqsubseteq z$ which completes the proof. ■

### 4.2.3 Difference

Below we see the definition of *difference*. The intuition of the difference is that given a monotonic function $f : S \to S'$ on a value set $S$ to a value set $S'$ (monotonic with respect to the induced preorder on $S$ and $S'$), then the difference of $f$ denoted $f'$ describes the change from $f(x)$ to $f(d \oplus x)$.

**Definition 4.2.12 (Difference)** *Assume that $(S, \oplus)_D$ and $(S', \oplus')_{D'}$ are ipo's and assume that $f : S \to S'$ is a monotonic function, that is, $f(x) \sqsubseteq_{S'} f(y)$ whenever $x \sqsubseteq_S y$. Difference of $f$ is defined as a function $f' : S \to S'$, which satisfies*

$$\begin{aligned} f(d \oplus x) &= f'(d \oplus x) \oplus' f(x) \\ f'(0_S) &= 0_{S'} \end{aligned}$$

*for all $x \in S$ and $d \in D$.*

The difference corresponds to the difference as explained in Section 4.1. However, the difference function from Section 4.1 satisfies $p(x+1) = \Delta p(x) + p(x)$. Thus, our difference function is generalized in the sense that we allow any kind of update $d \in D$ to be added to $x$ and then the equation still holds.

Before we continue with some examples we show that the difference always exist and that it is unique.

**Theorem 4.2.13 (Existence and uniqueness of difference)** *Assume $(S, \oplus)_D$ and $(S', \oplus')_{D'}$ are ipo's and assume that $f : S \to S'$ is a monotonic function. Then there exists a unique difference function $f' : S \to S'$ of $f$.*

Proof:
Existence: We show that we can find $y'_{d,x} \in S'$ for all $x \in S$ and $d \in D$ such that $f(d \oplus x) = y'_{d,x} \oplus' f(x)$. That is, if we can find such an element for each $x$ and $d$, then we have found a function $g : S \to S'$ such that $f(d \oplus x) = g(d \oplus x) \oplus' f(x)$, since $d \oplus x$ identifies $x$ and $d$ uniquely (requirement 2 of a an ipo).
Assume $x \in S$ and $d \in D$. By definition we get that $x \sqsubseteq_S d \oplus x$. Because $f$ is monotonic we get that $f(x) \sqsubseteq_{S'} f(d \oplus x)$. By definition there exists $y'_{d,x} \in S'$ such that

$$f(d \oplus x) = y'_{x,d} \oplus' f(x)$$

which completes the existence part of the proof.
Uniqueness: Assume there exists $g_1 : S \to S'$ and $g_2 : S \to S'$, which both are differences of $f$ and where $g_1 \neq g_2$. This means that

$$\begin{aligned} f(d \oplus x) &= g_1(d \oplus x) \oplus' f(x) \\ f(d \oplus x) &= g_2(d \oplus x) \oplus' f(x) \\ g_1(0_S) &= 0_{S'} = g_2(0_S) \end{aligned}$$

for all $x \in S$ and $d \in D$. Since $g_1 \neq g_2$ there exists $x' \in S'$ such that $y'_1 = g_1(x') \neq g_2(x') = y'_2$. Since $g_1(0_S) = 0_{S'} = g_2(0_S)$, we get that $x' \neq 0_S$.

That is, there exists a factorization of $x'$ which has the form $x' = d \oplus x$ for some $x \in S$ and $d \in D$. This means that

$$\begin{aligned} y_1' \oplus' f(x) &= g_1(d \oplus x) \oplus' f(x) \\ &= f(d \oplus x) \\ &= g_2(d \oplus x) \oplus' f(x) \\ &= y_2' \oplus' f(x) \end{aligned}$$

By Lemma 4.2.9 we get that $y_1' = y_2'$, which is a contradiction. This completes the proof. ■

Let us see a couple of examples of functions and their differences. We will see the definition of the function sum, which sums the elements of a list of natural numbers. The map function, which takes a function and a list as arguments and applies the function to all the elements in the list. Furthermore, we will see suffixsum, which computes the sum of all suffixes of a list of natural numbers and a function called filter, which takes a predicate and a list as arguments and filters out all the elements not satisfying the predicate.

**Example 4.2.14 (Functions and their differences)**

- The definition of $\mathbf{sum} : \mathbb{N}_0^* \to \mathbb{N}_0$ is $\mathbf{sum}([x_n, \ldots, x_1]) = \sum_{i=1}^{n} x_i$. That is, **sum** can be considered to have $(\mathbb{N}_0^*, @)_{[\mathbb{N}_0]}$ as source ipo and $(\mathbb{N}_0, +)_{\{1\}}$ as target ipo. Since

$$\begin{aligned} \mathbf{sum}([x_{n+1}]@[x_n, \ldots, x_1]) &= \mathbf{sum}([x_{n+1}, x_n, \ldots, x_1]) \\ &= \sum_{i=1}^{n+1} x_i \\ &= x_{n+1} + \sum_{i=1}^{n} x_i \\ &= x_{n+1} + \mathbf{sum}([x_n, \ldots, x_1]) \end{aligned}$$

  we get that $\mathbf{sum}'([x_{n+1}]@[x_n, \ldots, x_1]) = x_{n+1}$.

- Assume that $f : \mathbb{N}_0 \to \mathbb{N}_0$. The definition of $(\mathbf{map}\ f) : \mathbb{N}_0^* \to \mathbb{N}_0^*$ is $(\mathbf{map}\ f)\ ([x_n, \ldots, x_1]) = [f(x_n), \ldots, f(x_1)]$. We can consider $(\mathbf{map}\ f)$ to have $(\mathbb{N}_0^*, @)_{[\mathbb{N}_0]}$ as source and target ipo. Since

$$\begin{aligned} (\mathbf{map}\ f)\ ([x_{n+1}]@[x_n, \ldots, x_1]) &= (\mathbf{map}\ f)\ ([x_{n+1}, x_n, \ldots, x_1]) \\ &= [f(x_{n+1}), f(x_n), \ldots, f(x_1)] \\ &= [f(x_{n+1})]@[f(x_n), \ldots, f(x_1)] \\ &= [f(x_{n+1})]@(\mathbf{map}\ f)\ ([x_n, \ldots, x_1]) \end{aligned}$$

we get that $(\mathbf{map}\ f)'\ [x_{n+1}, x_n, \ldots, x_1] = [f(x_{n+1})]$.

- The definition of suffixsum is

$$\mathbf{suffixsum}([x_n, \ldots, x_1]) = [\sum_{i=1}^{n} x_i, \sum_{i=1}^{n-1} x_i, \ldots, \sum_{i=1}^{2} x_i, \sum_{i=1}^{1} x_i]$$

We can consider **suffixsum** to have $(\mathbb{N}_0^*, @)_{[\mathbb{N}_0]}$ as source and target ipo. Since

$$\begin{aligned}
\mathbf{suffixsum}([x_{n+1}]@[x_n, \ldots, x_1]) &= \mathbf{suffixsum}([x_{n+1}, x_n, \ldots, x_1]) \\
&= [\sum_{i=1}^{n+1} x_i, \sum_{i=1}^{n} x_i, \ldots, \sum_{i=1}^{2} x_i, \sum_{i=1}^{1} x_i] \\
&= [\sum_{i=1}^{n+1} x_i]@[\sum_{i=1}^{n} x_i, \ldots, \sum_{i=1}^{2} x_i, \sum_{i=1}^{1} x_i] \\
&= [\sum_{i=1}^{n+1} x_i]@\mathbf{suffixsum}([x_n, \ldots, x_1])
\end{aligned}$$

we get that $\mathbf{suffixsum}'([x_{n+1}]@[x_n, \ldots x_1]) = [\sum_{i=1}^{n+1} x_i]$

- Assume that $p : \mathbb{N}_0 \to \{\mathbf{true}, \mathbf{false}\}$. The definition of $(\mathbf{filter}\ p) : \mathbb{N}_0^* \to \mathbb{N}_0^*$ is

$$(\mathbf{filter}\ p)\ xs = \begin{cases} [] & xs = [] \\ (\mathbf{if}\ p(x_n)\ \mathbf{then}\ [x_n]\ \mathbf{else}\ []) @ & \\ \quad (\mathbf{filter}\ p)\ [x_{n-1}, \ldots, x_1] & xs = [x_n, \ldots, x_1] \wedge n \in \mathbb{N} \end{cases}$$

$(\mathbf{filter}\ p)$ can be considered to have $(\mathbb{N}_0^*, @)_{[\mathbb{N}_0]}$ as source and target ipo. Since

$$\begin{aligned}
& (\mathbf{filter}\ p)\ ([x_{n+1}]@[x_n, \ldots, x_1]) \\
= \ & (\mathbf{filter}\ p)\ [x_{n+1}, \ldots, x_1] \\
= \ & (\mathbf{if}\ p(x_{n+1})\ \mathbf{then}\ [x_{n+1}]\ \mathbf{else}\ []) @ (\mathbf{filter}\ p)\ [x_n, \ldots, x_1]
\end{aligned}$$

Thus $(\mathbf{filter}\ p)'\ ([x_{n+1}]@[x_n, \ldots, x_0]) = \mathbf{if}\ p(x_{n+1})\ \mathbf{then}\ [x_{n+1}]\ \mathbf{else}\ []$

- The Fibonacci function $fib : \mathbb{N}_0 \to \mathbb{N}_0$ is defined as:

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-2) + fib(n-1) & n > 1 \end{cases}$$

We can consider the Fibonacci function to have $(\mathbb{N}_0, +)_{\{1\}}$ as source and target ipo. Since the difference should satisfy $fib(1) = fib'(1) + fib(0)$ we get that $fib'(1) = 1$ and since

$$\begin{aligned} fib(n+1) &= fib(n+1-2) + fib(n+1-1) \\ &= fib(n-1) + fib(n) \end{aligned}$$

we get that $fib'(n+1) = fib(n-1)$ for $n > 1$. Thus

$$fib'(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-2) & n > 1 \end{cases}$$

■

### 4.2.4 Integrals

Like in differential calculus there exists an inverse operation to the difference operator, which we define in this section. First we need to define a partial subtraction operator on the elements of ipo's.

**Definition 4.2.15 ($\ominus$)** *Assume $(S, \oplus)_D$ is an ipo, assume that $a, b \in S$ and $a \sqsubseteq b$. That is, there exists $x \in S$ such that $b = x \oplus a$. Define*

$$b \ominus a = x$$

It follows from Lemma 4.2.9 that $\ominus$ is well-defined.

**Lemma 4.2.16** *Assume $(S, \oplus)_D$ is an ipo and assume that $a, b, c \in S$ where $a \sqsubseteq b$ and $b \sqsubseteq c$. Then*

1. $(c \ominus b) \oplus (b \ominus a) = c \ominus a$
2. $(b \oplus a) \ominus a = b$

Proof:
ad 1: There exist $x, x' \in S$ such that

$$\begin{aligned} b &= x \oplus a \\ c &= x' \oplus b \end{aligned}$$

Composing the equations above gives

$$c = x' \oplus (x \oplus a) = (x' \oplus x) \oplus a$$

Thus

$$c \ominus a = x' \oplus x$$

Therefore

$$\begin{aligned}(c \ominus b) \oplus (b \ominus a) &= x' \oplus x \\ &= c \ominus a\end{aligned}$$

which proves the first part of the lemma.
ad 2: This is a special case of the first part of the lemma. Since $0 \sqsubseteq a$ and since $a = a \ominus 0$ for all $a \in S$ we get that

$$(b \oplus a) \ominus a = (b \oplus a) \ominus (a \ominus 0) = b \ominus 0 = b$$

which proves the lemma. ■

We can now define *integrals* and show that the definition makes sense:

**Definition 4.2.17 (Integral)** *Assume $(S, \oplus)_D$ and $(S', \oplus')_{D'}$ are ipo's and assume that $f : S \to S'$ is a monotonic function. Furthermore, assume that $a, b \in S$ and $a \sqsubseteq b$. Define for all $d \in D$*

$$\begin{aligned}\sum_a^{d \oplus b} f(x)\ \delta x &= f(d \oplus b) \oplus' \sum_a^b f(x)\ \delta x \\ \sum_a^a f(x)\ \delta x &= 0_{S'}\end{aligned}$$

We need to show that the definition of integral makes sense.

**Lemma 4.2.18** *The definition of integral makes sense.*

Proof: Assume $(S, \oplus)_D$ and $(S', \oplus')_{D'}$ are ipo's, assume that $a, b \in S$ and $a \sqsubseteq b$. Furthermore, assume that $f : S \to S'$ is a monotonic function. We show by induction on $|b \ominus a|$ that the definition of $\sum_a^b f(x)\ \delta x$ makes sense.
$|b \ominus a| = 0$: This means that $b \ominus a = 0$ and thus $b = a$. In this case $\sum_a^b f(x)\ \delta x = 0_{S'}$ and thus the definition makes sense.
$|b \ominus a| > 0$: That is, there exists $x \in S$, where $x \neq 0$ such that $b = x \oplus a$. This means that $x = d' \oplus x'$ for some $x' \in S$ and $d' \in D$. Furthermore, since $|(x' \oplus a) \ominus a| = |x'| < |x| = |b \ominus a|$ we get by induction that $\sum_a^{x' \oplus a} f(x)\ \delta x$ makes sense and since

$$\sum_a^b f(x)\ \delta x = f(b) \oplus' \sum_a^{x' \oplus a} f(x)\ \delta x$$

the definition of $\sum_a^b f(x)\ \delta x$ makes sense. This completes the proof. ■.

Before we can prove the fundamental theorem of the integral operator, we need to show a lemma relating subtraction to the difference.

**Lemma 4.2.19** *Assume that $(S, \oplus)_D$ and $(S', \oplus')$ are ipo's and assume that $f : S \to S'$ is a monotonic function. Then*

$$f'(d \oplus x) = f(d \oplus x) \ominus f(x)$$

*for all $d \in D$ and all $x \in S$*

Proof: By Theorem 4.2.13 we know that there exists a unique difference $f'$ of $f$ such that

$$f(d \oplus x) = f'(d \oplus x) \oplus' f(x)$$

Since $f$ is monotonic we get that $f(x) \sqsubseteq f(d \oplus x)$. This means that

$$\begin{aligned} f(d \oplus x) \ominus' f(x) &= (f'(d \oplus x) \oplus' f(x)) \ominus' f(x) \\ &= f'(d \oplus x) \end{aligned}$$

where the last equality follows from Lemma 4.2.16, part 2. This completes the proof of the lemma. ■

**Theorem 4.2.20 (Fundamental Theorem of Difference Calculus)** *Assume $(S, \oplus)_D$ and $(S', \oplus')_{D'}$ are ipo's. Assume $f : S \to S'$ is monotonic. For all $a, b \in S$ where $a \sqsubseteq b$ and for all $d \in D$ we have*

$$\sum_a^b f'(x)\ \delta x = f(b) \ominus' f(a) \tag{4.10}$$

Proof: Let us first see why both sides of the equation (4.10) makes sense. Since $f$ is monotonic we know by Theorem 4.2.13 that there exists a unique difference $f'$ of $f$, thus the left hand side of the equation (4.10) makes sense. Furthermore, since $a \sqsubseteq b$ we get that $f(a) \sqsubseteq f(b)$, which means that $f(b) \ominus' f(a)$ makes sense. The proof of the theorem proceeds by induction on $|b \ominus a|$.
$|b \ominus a| = 0$: This means that $b \ominus a = 0$. By the definition of $\ominus$ we see that $a = b$. From the definition of integral and $\ominus$ we get that

$$\sum_a^a f'(x)\ \delta x = 0_{S'} = f(a) \ominus' f(a)$$

which completes the proof of the basic case of the theorem.
$|b \ominus a| > 0$: Assume by induction that the theorem holds for smaller $|b \ominus a|$. By definition of $\ominus$ we get that there exist $x \in S$, where $x \neq 0$ such that $b = x \oplus a$. This means that there exist $x' \in S$ and $d' \in D$ such that $x = d' \oplus x'$ and $b = d' \oplus x' \oplus a$. We can now rewrite:

$$\sum_{a}^{b} f'(x)\ \delta x \tag{4.11}$$

$$= \quad f'(b) \oplus' \sum_{a}^{x' \oplus a} f'(x)\ \delta x \tag{4.12}$$

$$= \quad f'(b) \oplus' (f(x' \oplus a) \ominus' f(a)) \tag{4.13}$$

$$= \quad (f(b) \ominus' f(x' \ominus a)) \oplus' (f(x' \oplus a) \ominus' f(a)) \tag{4.14}$$

$$= \quad f(b) \ominus' f(a) \tag{4.15}$$

(4.12): By the definition of integral.
(4.13): Because $|(x' \oplus a) \ominus a| = |x'| < |d' \oplus x'| = |x| = |b \ominus a|$ this step follows by induction.
(4.14): Follows by Lemma 4.2.19.
(4.15): Follows from Lemma 4.2.16 part 1.
This completes the proof of the theorem. ■

Let us see an example on, how the theorem can be applied.

**Example 4.2.21** *Integrating the* (**map** $f$) *function.*

By definition (**map** $f$) $[x_n, \ldots, x_1] = [f(x_n), \ldots, f(x_1)]$ and in Example 4.2.14 we saw that (**map** $f$)$'$ $([x_{n+1}]@[x_n, \ldots, x_1]) = [f(x_{n+1})]$. Using the definition of integral we can compute

$$\sum_{[]}^{[x_n,\ldots,x_1]} (\mathbf{map}\ f)'\ (x)\ \delta x$$

$$= \quad (\mathbf{map}\ f)'\ [x_n, \ldots, x_1]\ @\ (\mathbf{map}\ f)'\ [x_{n-1}, \ldots x_1]\ @\ \ldots\ @\ (\mathbf{map}\ f)'\ [x_1]\ @\ []$$

$$= \quad [f(x_n)]\ @\ [f(x_{n-1})]\ @\ \ldots\ @\ [f(x_1)]\ @\ []$$

$$= \quad [f(x_n), \ldots, f(x_1)]$$

By Theorem 4.2.20 the integral can also be computed as

$$\sum_{[]}^{[x_n,\ldots,x_1]} (\mathbf{map}\ f)'(x)\ \delta x \quad = \quad (\mathbf{map}\ f)[x_n, \ldots, x_1] \ominus (\mathbf{map}\ f)\ []$$

$$= \quad [f(x_n), \ldots, f(x_1)] \ominus []$$

$$= \quad [f(x_n), \ldots, f(x_1)]$$

■

Furthermore, the integral operator has the following property.

**Theorem 4.2.22** *Assume $(S, \oplus)_D$ and $(S', \oplus')_{D'}$ are ipo's and assume that $f : S \to S'$ is monotonic. Then $\forall a, b, c \in S$ where $a \sqsubseteq b \wedge b \sqsubseteq c$ we have*

$$\sum_b^c f(x)\ \delta x \oplus' \sum_a^b f(x)\ \delta x = \sum_a^c f(x)\ \delta x$$

Proof: Induction on $|c \ominus b|$.
$|c \ominus b| = 0$: This means that $c = b$ and therefore

$$\begin{aligned}\sum_b^c f(x)\ \delta x \oplus' \sum_a^b f(x)\ \delta x &= \sum_c^c f(x)\ \delta x \oplus' \sum_a^c f(x)\ \delta x \\ &= 0_{S'} \oplus' \sum_a^c f(x)\ \delta x \\ &= \sum_a^c f(x)\ \delta x\end{aligned}$$

which completes the basic case.
$|c \ominus b| > 0$: Assume the theorem holds for smaller $|c \ominus b|$. Since $|c \ominus b| > 0$ there exist $x \in S$, where $x \neq 0$ such that $c = x \oplus b$ and $x = d' \oplus x'$ for some $x' \in S$ and $d' \in D$. Since $b \sqsubseteq x' \oplus b$ and $|(x' \oplus b) \ominus b| = |x'| < |x| = |c \ominus b|$ we get that

$$\begin{aligned}\sum_b^c f(x)\ \delta x \oplus \sum_a^b f(x)\ \delta x &= f(c) \oplus \sum_b^{x' \oplus b} f(x)\ \delta x \oplus \sum_a^b f(x)\ \delta x \\ &= f(c) \oplus \sum_a^{x' \oplus b} f(x)\ \delta x \\ &= \sum_a^c f(x) \delta x\end{aligned}$$

where the first and last equality follows from the definition of integral and the middle equation follows by induction. This completes the proof. ■

### 4.2.5 Chain Rule

In this section we show, how to find the difference of the composition of functions. First we need a lemma that shows that the composition of monotonic functions yields a monotonic function.

**Lemma 4.2.23** *Assume $f : S \to S'$ and $g : S' \to S''$ are monotonic functions. Then $(g \circ f) : S \to S''$ is monotonic.*

Proof: Assume $x \sqsubseteq y$. Since $f$ is monotonic we get that $f(x) \sqsubseteq f(y)$ and since $g$ is monotonic we get that $g(f(x)) \sqsubseteq g(f(y))$. Thus proving $(g \circ f)(x) \sqsubseteq (g \circ f)(y)$, which proves that $g \circ f$ is monotonic. ■

Now we are ready to prove the chain rule.

**Theorem 4.2.24 (Chain Rule)** *Assume $(S, \oplus)_D$, $(S', \oplus')_{D'}$ and $(S'', \oplus'')_{D''}$ are ipo's and $f : S \to S'$ and $g : S' \to S''$ are monotonic functions. Then*

$$(g \circ f)'(d \oplus x) = \sum_{f(x)}^{f'(d \oplus x) \oplus' f(x)} g'(t)\ \delta t$$

Proof: By Lemma 4.2.23 we get that $g \circ f$ is monotonic and Theorem 4.2.13 says that there exists a unique difference $(g \circ f)'$ of $(g \circ f)$. Thus we need to show that the difference can be expressed as claimed above. We do the following rewritings:

$$\begin{aligned}
(g \circ f)'(d \oplus x) &= (g \circ f)(d \oplus x) \ominus'' (g \circ f)(x) && (4.16)\\
&= g(f(d \oplus x)) \ominus'' g(f(x)) && (4.17)\\
&= \sum_{f(x)}^{f(d \oplus x)} g'(t)\ \delta t && (4.18)\\
&= \sum_{f(x)}^{f'(d \oplus x) \oplus' f(x)} g'(t)\ \delta t && (4.19)
\end{aligned}$$

(4.16): Lemma 4.2.19
(4.17): Definition of function composition.
(4.18): Since $f$ is monotonic we get that $f(x) \sqsubseteq f(d \oplus x)$ and Theorem 4.2.20.
(4.19): Theorem 4.2.13.
This completes the proof. ■

The formula in the chain rule could have been written in many different ways, but we have chosen to write it like this, because then $(g \circ f)'(d \oplus x)$ only depends on $g'$, $f'$ and $f(x)$, which is practical when we want to compute functions incrementally.

**Example 4.2.25 (Application of Chain Rule)** *Let us try and use the chain rule to find the difference of* **suffixsum** $\circ$ (**map** $f$) *and* (**map** $f$) $\circ$ (**filter** $p$)

- In example 4.2.14 we saw that

$$\begin{aligned}(\mathbf{map}\ f)'([x_{n+1}]@[x_n,\ldots,x_1]) &= [f(x_{n+1})]\\ \mathbf{suffixsum}'([x_{n+1}]@[x_n,\ldots,x_1]) &= [\sum_{i=1}^{n+1} x_i]\end{aligned}$$

  By the chain rule we get that

$$\begin{aligned}&(\mathbf{suffixsum}\circ(\mathbf{map}\ f))'([x_{n+1}]@[x_n,\ldots,x_1])\\ =&\sum_{(\mathbf{map}\ f)[x_n,\ldots,x_1]}^{(\mathbf{map}\ f)'([x_{n+1}]@[x_n,\ldots,x_1])\ @\ (\mathbf{map}\ f)[x_n,\ldots,x_1]} \mathbf{suffixsum}'(t)\ \delta t\\ =&\sum_{[f(x_n),\ldots,f(x_1)]}^{[f(x_{n+1})]\ @\ [f(x_n),\ldots,f(x_1)]} \mathbf{suffixsum}'(t)\ \delta t\\ =&\ \mathbf{suffixsum}'([f(x_{n+1})]@[f(x_n),\ldots,f(x_1)])\ @\ \sum_{[f(x_n),\ldots,f(x_1)]}^{[f(x_n),\ldots,f(x_1)]} \mathbf{suffixsum}'(t)\ \delta t\\ =&\ \mathbf{suffixsum}'([f(x_{n+1}),f(x_n),\ldots,f(x_1)])\ @\ []\\ =&\ [\sum_{i=1}^{n+1} f(x_i)]\end{aligned}$$

- In example 4.2.14 we saw that

$$(\mathbf{filter}\ p)'\ ([x_{n+1}]@[x_n,\ldots,x_0]) = \mathbf{if}\ p(x_{n+1})\ \mathbf{then}\ [x_{n+1}]\ \mathbf{else}\ []$$

  Since the factorization of the result of $(\mathbf{filter}\ p)'$ can have different sizes we need to split the difference in two cases depending on the size of the result.
  $p(x_{n+1}) = \mathbf{false}$: That is, $(\mathbf{filter}\ p)'\ ([x_{n+1}]@[x_n,\ldots,x_0]) = []$, which means that the factorization consists of only the zero element. By Theorem 4.2.24 we get that

$$\begin{aligned}&((\mathbf{map}\ f)\circ(\mathbf{filter}\ p))'\ ([x_{n+1}]@[x_n,\ldots,x_0])\\ =&\sum_{\mathbf{filter}\ p\ ([x_n,\ldots,x_0])}^{\mathbf{filter}\ p\ ([x_n,\ldots,x_0])} (\mathbf{map}\ f)'(t)\ \delta t\\ =&\ []\end{aligned}$$

$p(x_{n+1}) =$ **true**: That is, (**filter** $p)'$ $([x_{n+1}]@[x_n, \ldots, x_0]) = [x_{n+1}]$. By Theorem 4.2.24 we get that

$$\begin{aligned} & ((\textbf{map } f) \circ (\textbf{filter } p))\ ([x_{n+1}]@[x_n, \ldots, x_0]) \\ = & \sum_{\substack{[x_{x+1}]@(\textbf{filter } p\ ([x_n,\ldots,x_0])) \\ \textbf{filter } p\ ([x_n,\ldots,x_0])}} (\textbf{map } f)'(t)\ \delta t \\ = & \ (\textbf{map } f)'\ ([x_{n+1}]@\textbf{filter } p\ [x_n, \ldots, x_1]) \\ = & \ [f(x_{n+1})] \end{aligned}$$

Combining the two cases we get that

$((\textbf{map } f) \circ (\textbf{filter } p))'\ ([x_{n+1}]@[x_n, \ldots, x_1]) = \textbf{if } p(x_{n+1}) \textbf{ then } [f(x_{n+1})] \textbf{ else } []$

■

### 4.2.6 Taylor Expansion

Let us see, how we can difference a function more than once, and how this can be used to express the original function. First let us see an example of a second order difference.

**Example 4.2.26 (Second order difference)**

In example 4.2.14 we saw that $\textbf{suffixsum}' : \mathbb{N}_0^* \to \mathbb{N}_0^*$ has the difference

$$\textbf{suffixsum}'([x_{n+1}]@[x_n, \ldots x_1]) = [\sum_{i=1}^{n+1} x_i]$$

and by Theorem 4.2.13 we know that **suffixsum**$'$ should be considered a function from source and target ipo $(\mathbb{N}_0^*, @)_{[\mathbb{N}_0]}$, because this is the source and target ipo of **suffixsum**. But as the above definition of **suffixsum**$'$ shows, only singleton lists are used as targets for **suffixsum**$'$. Let us now consider **suffixsum**$'$ a function with source ipo $(\mathbb{N}_0^*, @)_{[\mathbb{N}_0]}$ and target ipo $([\mathbb{N}_0], +@)_{[1]}$. Since

$$\begin{aligned} \textbf{suffixsum}'([x_{n+1}]@[x_n, \ldots, x_1]) &= \textbf{suffixsum}'([x_{n+1}, x_n, \ldots, x_1]) \\ &= [\sum_{i=1}^{n+1} x_i] \\ &= [x_{n+1}]\ +@\ [\sum_{i=1}^{n} x_i] \\ &= [x_{n+1}]\ +@\ \textbf{suffixsum}'([x_n, \ldots, x_1]) \end{aligned}$$

Thus $\mathbf{suffixsum}''([x_{n+1}]@[x_n,\ldots,x_1]) = [x_{n+1}]$, if we consider $([\mathbb{N}_0], +@)_{[1]}$ to be the target ipo of $\mathbf{suffixsum}'$. ■

In the example above we used the idea that a function could be considered as having another target ipo than first intended. This idea introduces the notion of *sub-ipo*.

**Definition 4.2.27 (sub-ipo)** *Assume $(S,\oplus)_D$ and $(\hat{S},\hat{\oplus})_{\hat{D}}$ are ipo's. Then we say $(\hat{S},\hat{\oplus})_{\hat{D}}$ is a* sub-ipo *of $(S,\oplus)_D$ if $\hat{S} \subseteq S$ and $\hat{D} \subseteq D$. We write this as $(S,\oplus)_D \subseteq (\hat{S},\hat{\oplus})_{\hat{D}}$.*

Now we are ready to prove a form of *Taylor expansion* of functions, which is related to the Taylor expansion from Differential and Integral Calculus.

**Theorem 4.2.28 (Taylor Expansion)** *Assume $(S,\oplus)_D$ and $(S_n,\oplus_n)_{D_n} \subseteq (S_{n-1},\oplus_{n-1})_{D_{n-1}} \subseteq \ldots \subseteq (S_0,\oplus_0)_{D_0}$ are ipo's for some $n \in \mathbb{N}_0$. Assume $f = f_0 : S \to S_0$ and $f_i : S \to S_i$ for $1 \leq i \leq n-1$ are monotone functions and furthermore assume that $f_i$ is the difference of $f_{i-1}$ for $1 \leq i \leq n$. Then*

$$f(d \oplus x) = (\ldots((f_n(d \oplus x) \oplus_{n-1} f_{n-1}(x)) \oplus_{n-2} f_{n-1}) \ldots \oplus_1 f_1(x)) \oplus_0 f_0(x)$$

*for all $d \in D$ and $s \in D$*

Proof: By induction on $n$.
$n = 0$: We need to prove that $f(d \oplus x) = f_0(d \oplus x)$, which is trivially true since $f = f_0$ by assumption.
$n > 0$: Assume that the theorem holds for smaller $n$. By Theorem 4.2.13 we get that

$$f(d \oplus x) = f_1(d \oplus x) \oplus_0 f(x) \tag{4.20}$$

Using induction on $f_1$ and using the assumption that $f_i$ is the difference of $f_{i-1}$ we get that

$$f_1(d \oplus x) = (\ldots((f_n(d \oplus x) \oplus_{n-1} f_{n-1}(x)) \oplus_{n-2} f_{n-1}) \ldots \oplus_2 f_2(x)) \oplus_1 f_1(x) \tag{4.21}$$

Composing (4.20) and (4.21) we get that

$$f(d \oplus x) = ((\ldots((f_n(d \oplus x) \oplus_{n-1} f_{n-1}(x)) \oplus_{n-2} f_{n-1}) \ldots \oplus_2 f_2(x)) \oplus_1 f_1(x)) \oplus_0 f(x)$$

Thus proving the theorem. ■

**Example 4.2.29** *Let us see how the Taylor expansion theorem can used to describe the suffixsum function.*

In Example 4.2.14 and Example 4.2.26 we found that

$$\begin{aligned}
\textbf{suffixsum}([x_n,\ldots,x_1]) &= [\sum_{i=1}^{n} x_i, \sum_{i=1}^{n-1} x_i, \ldots, \sum_{i=1}^{2} x_i, \sum_{i=1}^{1} x_i] \\
\textbf{suffixsum}'([x_{n+1}]@[x_n,\ldots,x_1]) &= [\sum_{i=1}^{n} x_i] \\
\textbf{suffixsum}''([x_{n+1}]@[x_n,\ldots,x_1]) &= [x_{n+1}]
\end{aligned}$$

By Theorem 4.2.28 we get that

$$\begin{aligned}
& \textbf{suffixsum}'([x_{n+1}]@[x_n,\ldots,x_1]) \\
= \;& (\textbf{suffixsum}''([x_{n+1}]@[x_n,\ldots,x_1]) \; +@ \textbf{suffixsum}'([x_n,\ldots,x_1])) \\
& @\, \textbf{suffixsum}([x_n,\ldots,x_1]) \\
= \;& ([x_{n+1}] \; +@ [\sum_{i=1}^{n} x_i]) \; @ \; [\sum_{i=1}^{n} x_i, \sum_{i=1}^{n-1} x_i, \ldots, \sum_{i=1}^{2} x_i, \sum_{i=1}^{1} x_i]
\end{aligned}$$

which is correct. ■

### 4.2.7 Incremental Computation Using Differences

Let us see, how we can use the differences to do incremental computation of functions. The Taylor Expansion theorem (Theorem 4.2.28) tells us, how we can compute $f(d \oplus x)$ using the differences and $f(x)$. The following corollary shows, how we can compute the value of a function incrementally. The corollary corresponds to the technique of computing polynomials incrementally as shown in Section 4.1.

**Corollary 4.2.30 (Incremental computation)** *Assume* $(S,\oplus)_D$ *and* $(S_n,\oplus_n)_{D_n} \subseteq (S_{n-1},\oplus_{n-1})_{D_{n-1}} \subseteq \ldots \subseteq (S_0,\oplus_0)_{D_0}$ *are ipo's for some* $n \in \mathbb{N}_0$. *Assume* $f = f_0 : S \to S_0$ *and* $f_i : S \to S_i$ *for* $1 \leq i \leq n-1$ *are monotone functions and furthermore assume that* $f_i$ *is the difference of* $f_{i-1}$ *for* $1 \leq i \leq n$.

*Define*

$$\overline{f}(x,(r_0,\ldots,r_{n-1})) = \begin{cases} (f(0),f_1(0),\ldots,f_{n-1}(0)) & x = 0_S \\ \textbf{let} & x \neq 0_S \\ \quad r_n^{new} = f_n(x) & \\ \quad r_{n-1}^{new} = r_n^{new} \oplus_{n-1} r_{n-1} & \\ \quad \ldots & \\ \quad r_1^{new} = r_2^{new} \oplus_1 r_1 & \\ \quad r_0^{new} = r_1^{new} \oplus_0 r_0) & \\ \textbf{in} & \\ \quad (r_0^{new},\ldots,r_{n-1}^{new}) & \end{cases}$$

*Then for all $x \in S$ and all $d \in D$ we get that*

$$\overline{f}(d \oplus x,(f(x),f_1(x),\ldots,f_{n-1}(x))) = (f(d \oplus x),f_1(d \oplus x),\ldots,f_{n-1}(d \oplus x))$$

Proof: Follows directly from the Taylor Expansion theorem (Theorem 4.2.28). ■

**Example 4.2.31** *This example shows how the differences of the* **map** $f$*, $suffixsum$ and $fib$ functions from Example 4.2.14 and Example 4.2.26 can be used to create incremental versions of these functions. If the incremental function only uses first order differences we have chosen for simplicity not to return this as a tuple.*

- The **map** $f$ function can be computed incrementally by

$$\begin{aligned} \overline{\textbf{map}\ f}([x_n,\ldots,x_1],r_0) &= \begin{cases} (\textbf{map}\ f)([]) & n = 0 \\ \textbf{let} & n > 0 \\ \quad r_1^{new} = (\textbf{map}\ f)'([x_n,\ldots,x_1]) & \\ \quad r_0^{new} = r_1^{new} @ r_0 & \\ \textbf{in}\quad r_0^{new} & \end{cases} \\ &= \begin{cases} [] & n = 0 \\ [f(x_n)]\,@r_0 & n > 0 \end{cases} \end{aligned}$$

- The $suffixsum$ function can be computed incrementally by:

$$
\begin{array}{ll}
& suffixsum([x_n,\ldots,x_1],(r_0,r_1)) \\
= & \left\{\begin{array}{ll}
(suffixsum([]), suffixsum'([])) & n=0 \\
\textbf{let} & n>0 \\
\quad r_1^{new} = suffixsum''([x_n,\ldots,x_1]) \mathbin{@+} r_1 & \\
\quad r_0^{new} = r_1^{new} @ r_0 & \\
\textbf{in} \quad (r_0^{new}, r_1^{new}) &
\end{array}\right. \\
= & \left\{\begin{array}{ll}
([],[0]) & n=0 \\
\textbf{let} & n>0 \\
\quad r_1^{new} = [x_n] \mathbin{@+} r_1 & \\
\quad r_0^{new} = r_1^{new} @ r_0 & \\
\textbf{in} \quad (r_0^{new}, r_1^{new}) &
\end{array}\right.
\end{array}
$$

- Corollary 4.2.30 says that the result of the last derivative should not be included in the return value of the incremental function, because it is not necessary. This is because we assume that the $n$'th derivative is computed from its definition. However, as seen below the derivative of the Fibonacci function can be compute from the preceding Fibonacci number and the preceding value of the derivative

$$
\begin{array}{rcl}
fib'(n+1) & = & fib(n-1) \\
& = & fib(n) - fib(n-2) \\
& = & fib(n) - fib'(n)
\end{array}
$$

  Thus, we extend the incremental version of the Fibonacci function to also include the value of the last derivative, because it can be used in the next computation.

$$
\begin{array}{rcl}
\overline{fib}(n,(r_0,r_1)) & = & \left\{\begin{array}{ll}
(fib(0), fib'(0)) & n=0 \\
(fib(1), fib'(1)) & n=1 \\
\textbf{let} & n>1 \\
\quad r_1^{new} = fib'(n) & \\
\quad r_0^{new} = r_1^{new} + r_0 & \\
\textbf{in} \quad (r_0^{new}, r_1^{new}) &
\end{array}\right. \\
& = & \left\{\begin{array}{ll}
(0,0) & n=0 \\
(1,1) & n=1 \\
\textbf{let} & n>1 \\
\quad r_1^{new} = r_0 - r_1 & \\
\quad r_0^{new} = r_1^{new} + r_0 & \\
\textbf{in} \quad (r_0^{new}, r_1^{new}) &
\end{array}\right.
\end{array}
$$

■

Note, that using the differences to compute the Fibonacci function yields a function that is very different from the non-incremental version. This is discussed further in the future work section.

### 4.2.8 Comparison to Finite Differencing

In the finite differencing calculus described by Graham et al. [2004] we define $\Delta f(n) = f(n+1) - f(n)$ for a function $f$, where we do not require that $f$ should be monotonic, but in our context we need to require monotonicity or else $f'(d \oplus x) = f(d \oplus x) \ominus f(x)$ does not make sense. Thus, the main differences of finite differencing and the difference calculus presented here is that we require monotonicity, but we allow the update operation on $x$ to be dynamic, that is, the update (increment $d\oplus$) to $x$ can be any element from the set $D$, which generates the elements of the value set of the incremental preorder. The monotonicity requirement restricts the number of functions for which the theory applies, but the idea that the update operation is dynamic extends the kind of functions for which the theory applies. Note, that the finite differencing calculus described by Graham et al. [2004] does not make any sense for functions on lists, but the theory developed here works for monotonic list functions.

Furthermore, we do not hard code the update operation into the difference function, because we use $f'(d \oplus x)$ on the right hand side of the equation, where as in the $\Delta$ operator we do not include the update $d$. Thus we get the slightly different result of integrals:

$$\begin{aligned} \sum_{a}^{b} \Delta f(n)\ \delta n &= \sum_{n=a}^{n<b} \Delta f(n) \\ \sum_{a}^{b} f'(n)\ \delta n &= \bigoplus_{n>a}^{n=b} f'(n) \end{aligned}$$

where $\bigoplus_{n>a}^{n=b} f'(n)$ means the sum with respect to $\oplus$ as defined in the integral definition earlier. This means that we add everything from $a$ to $b$, but where we do not include $a$, but instead include $b$.

However, if we had

1 Defined the difference operator on polynomials to be $\Delta p(n) = p(n) - p(n-1)$ and thereby $\Delta p(n+1) = p(n+1) - p(n)$ (this would not affect the results of the preceding section).

2 Used $\mathbb{N}_0$ as coefficients in the polynomials instead of $\mathbb{Z}$.

3 Restricted the polynomials to be functions $p : \mathbb{N}_0 \to \mathbb{N}_0$ instead of $p : \mathbb{Z} \to \mathbb{Z}$

Then the polynomials could be assumed to have $(\mathbb{N}_0, +)_{\{1\}}$ as source and target ipo and because the coefficients would be from $\mathbb{N}_0$ the polynomials would automatically be monotonic and we would get $\Delta p = p'$. Thus, we could use the finite differencing theory of this section to compute polynomials incrementally. Also, there would not be any difference of the integral operators.

## 4.3 Summary and Future Work

We have presented how the theory of finite differencing works on polynomials, and demonstrated how the values of polynomials can be computed incrementally using difference polynomials. Our contributions are

- A generalization of the finite differencing calculus to incremental preorders.
- A chain rule, an integral operator and a Taylor expansion rule for the generalization.

Using these theorems and operators we have seen, how functions can be rewritten to be declared incrementally. Furthermore, it should be noted that there does not exist a general chain rule in the standard finite calculus presented by Graham et al. [2004].

Below we discuss some of the ideas and thoughts we have encountered for further research:

- The requirements that a function as argument takes elements of a value set of a source ipo, return elements of a value set of a target ipo and that functions should be monotonic are extensional properties. Thus, before applying the differencing theory to a function, we need to determine which source and target ipo's are used, and we also need to show that the function is monotonic. Furthermore the discovery of the differences in the examples are rather ad hoc. As a result, it is interesting to develop a calculus, where syntactical analysis of functions is used to discover their differences (like differentiation in Calculus). Thus, we would like to create a *specification language*, where we can express functions and automatically create the differences of these functions and thereby creating incremental versions of them.

- In the theory of finite differencing of polynomials, the degree of a polynomial tells us, how many times we need to take the difference of a polynomial to reach a constant. Intuitively, we can in general never end up with a constant function using (a function independent of its argument) our difference theory, because the difference is parametrized over the change $d$. However, one could imagine that if we differenced a function sufficiently many times we would end up with a function only on $d$, that is, $f^{(n)}(d \oplus x) = g(d)$ for some $n \in \mathbb{N}_0$. In the current version of the finite differencing theory it can be hard to see, how many times we can or need to difference a given function to end up with a function that only depends on the increment $d$. Thus, we would like to investigate, if there exists something similar to the degree function of polynomials to determine, how many times we need to difference a given function.

- It could be interesting to do a further investigation of, which kind of structures we can consider as ipo's. Furthermore, we would like to *relax* the requirements on ipo's, especially the unique factorization of the elements. In the current version of the finite differencing calculus, we do not support $\mathbb{R}$ as the value set. Furthermore, we have not found a good way to model $\mathbb{Z}$ as an ipo: We could use $(\mathbb{N}, +)_{\{1\}}$ to model $\mathbb{Z}$, by mapping
$$f(n) = \begin{cases} k & n = 2k \\ 0 & n = 0 \\ -k & n = 2k + 1 \end{cases}$$
However, this gives a weird monotonicity constraint which renders the encoding useless.

- Even though the incremental versions of the functions from the examples are asymptotically faster (under any reasonable computational model) than their non-incremental counterparts, the incremental versions of the functions (if we disregard the Fibonacci example) do at most provide a polynomial speedup. However, in the case of the Fibonacci function we go from an exponential function to a constant function (linear, if we need to compute a Fibonacci number from scratch using the incremental version). Thus, one could expect that the theory of differencing also could have some relation to dynamic programming, because the incremental version computes bottom up instead of top down.

- The following is highly speculative: There might be a connection from the differencing of PTIME computable functions and to finite differencing of polynomials.

# Chapter 5

# Sketch: Concrete Difference Calculus

**Abstract:** We introduce a small functional programming language called FunDIF, and show that it is possible to automatically *difference* a subset of programs written in this language. That is, we introduce FunDIF with type system, semantics and a partial map for constructing difference functions. Furthermore we show, how we from the difference function can construct an incremental version of the original function, with a possibly asymptotic speedup in the computation time. The work presented in this report is based on the theory of Section 4.2.

It should be stressed out that this section contains *work in progress*, that is, many of the statements are not proved and may not hold in the form presented here. However, we think that if it is the case that a claim is not true, then there exists a theorem that is similar which is.

## 5.1 Introduction

In Section 4.2 we presented a calculus for differencing monotonic functions, which take elements from value sets of ipo's and return elements of value sets of ipo's. Thus, before looking for the difference of a function, we need to know what the source and target ipo's are and that the function is monotonic with respect to the induced preorders on the ipo value sets. These are extensional properties, but the goal of this report is to do *symbolic* differencing. Thus, from the syntax of a function, we want to determine, what the source and target ipo's are, if the function is monotonic and how the difference function can be declared. It should be noted that in a Turing complete

programming language, it is undecidable to test if a function is monotonic. This can be proven using Rice's Theorem, which says that, if $A$ is an extensional non-trivial program property then $A$ is undecidable. (Jones [1997][p. 76] explains this). Thus, we can never get a Turing complete programming language, where we can identify all monotonic functions.

We introduce the programming language FunDIF, which is a small functional programming language without recursion and only a simple iterative construct called *integral* that corresponds to the integral defined in Section 4.2. Then we introduce the type system, semantics and a partial transformation called $\mathcal{D}_{xs}$ of FunDIF type derivations. $\mathcal{D}_{xs}$ should be considered a partial difference operator of FunDIF expressions, but the reason it takes typing derivations as its argument, is because we need to detect the ipo operators for which we difference with respect to. $\mathcal{D}_{xs}$ returns a typing derivation of the difference function for its input, together with the operator of the target ipo. That is, $\mathcal{D}_{xs}$ returns a FunDIF declaration of the difference function and an operator that tells us, how values are combined in the target ipo. This is sufficient information to create a new function that compute the function we take the difference of, using its difference function and the original function ($f(d \oplus x) = f'(d \oplus x) \oplus' f(x)$). One of the reasons that $\mathcal{D}_{xs}$ is partial, is because we have a conservative estimate of, which FunDIF functions we know to be monotonic by inspecting the syntax. For the subset of FunDIF functions handled by $\mathcal{D}_{xs}$ it is sufficient to inspect their types to determine, which ipo's we can consider as source and target.

The sketch of a calculus presented in this report, should not be considered final and closed work, but just as an appetizer of, what we hope is possible with the difference theory ideas from Section 4.2. Thus, the ideas presented here should be considered preliminary work on how we want to develop and use the theory of Section 4.2. However, we felt that it was necessary, to try and do symbolic differencing using the differencing theory, because we need a solid ground to evaluate the ideas, changes and extensions that we would like to do to the differencing theory to fit it with symbolic differencing. This is also why this section is not concluded with a conclusion and future work, because we do not know, what the conclusion and the future work are. Again we would like to remind that the work presented here is only in *sketch* form as explained in the abstract.

## 5.2 Syntax of FunDIF

In this section we present the syntax of FunDIF together with an informal description of the semantics of the constructs in FunDIF. FunDIF is similar to a small version of SML (described by Hansen and Rischel [1999]) and Figure 5.1 shows the syntactic productions of FunDIF written using BNF grammar. The production $\tau$ are the types in FunDIF, which ranges over lists, natural numbers and booleans. $c$ are the constants in FunDIF, that is, the empty list, natural numbers and truth values. *unop* and *binop* are the unary and binary operators of FunDIF respectively, which are similar to those of SML. The production $f$ is used to express, what is considered as a reference to a function, which is either a function identifier or a composition ($\circ$) of functions. The expressions of FunDIF are given by the production $e$, which says that an expression is either a constant, a variable, an operator application, a let expression, a conditional, a function application (in FunDIF we have for simplicity restricted functions to only having one parameter) or an *integral* expression. All of the expressions mentioned except the *integral* expression should be known from SML. The integral expression could be declared in SML by:

**fun integral** *op f e* [] = *e*
  | **integral** *op f e* (*x* :: *xs*) = *f*(*x* :: *xs*) *op* (**integral** *op f e xs*)

That is, the integral expression applies the argument function $f$ to all suffix lists of the argument list and uses the operator *op* and the value of the expression $e$ to construct the final result. However, when used in FunDIF we only allow $::, @, +$ to be used as *op*. Thus, the integral expression is different from foldright known from SML, because the function $f$ is applied to all suffixes of the input list and because we restrict how the output from the calls to $f$ is combined. It should be noted that the declaration of integral is closely related to the integral operator from Section 4.2. Furthermore, we introduce the following syntactic sugar for lists:

$$e_1 :: \ldots :: e_n :: ([] \textbf{ as } \tau) \equiv [e_1, \ldots, e_n] \textbf{ as } \tau$$

The syntactic production $f_{decl}$ are function declarations in FunDIF and a list of function declarations $p$ is a program.

As we see later our type system on FunDIF prohibits recursive function declarations. This means that the only repetitive construct in FunDIF is the *integral* expression.

Let us see some examples of functions implemented using FunDIF.

$$
\begin{array}{rcl}
\tau & ::= & \textbf{list}\ \tau \mid \textbf{nat} \mid \textbf{bool} \\
c & ::= & [] \ \textbf{as}\ \tau \mid n \mid \textbf{true} \mid \textbf{false} \\
\mathit{unop} & ::= & \textbf{hd} \mid \textbf{tail} \\
\mathit{binop} & ::= & :: \mid @ \mid + \mid = \mid < \mid \textbf{and} \mid \textbf{or} \\
f & ::= & f_{id} \mid f_1 \circ f_2 \\
e & ::= & c \mid x \mid op(e_1, \ldots, e_n) \mid \textbf{let}\ \ x = e_1\ \textbf{in}\ e_2 \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid \\
 & & f(e) \mid \textbf{integral}_{op}\ (xs \Rightarrow e_1)\ e_2\ e_3 \\
f_{decl} & ::= & \textbf{fun}\ f_{id}(x : \tau) = e \\
p & ::= & f^1_{decl} \ldots f^n_{decl}
\end{array}
$$

Figure 5.1: FunDIF syntax

**Example 5.2.1 (FunDIF functions)** *We implement the functions* **map** $f$, **filter** $p$ *and* **suffixsum** *in FunDIF, where the mapping function* $f$ *and filter function* $p$ *are hard coded into the function declaration (FunDIF does not support higher order functions).*

1 *map*: Add two to all the elements of a list of integers.

$\textbf{fun}\ f\ (x : \textbf{nat}) = x + 2$
$\textbf{fun}\ map_f\ (xs : \textbf{list}(\textbf{nat})) = \textbf{integral}_{::}\ (ys \Rightarrow f(\textbf{hd}\ ys))\ ([]\ \textbf{as list nat})\ xs$

2 *filter*: Remove all occurrences of the number five from a list of integers

$\textbf{fun}\ p\ (x : \textbf{nat}) = x = 5$
$\textbf{fun}\ filter_p\ (xs : \textbf{list}(\textbf{nat})) =$
$\textbf{integral}_{@}\ (ys \Rightarrow$
  $\textbf{let}\ x = \textbf{hd}(ys)\ \textbf{in}$
    $\textbf{if}\ p(x)$
     $\textbf{then}\ ([x]\ \textbf{as list nat})$
     $\textbf{else}\ ([]\ \textbf{as list nat})\ ([]\ \textbf{as list nat})\ xs$

3 *suffixsum*:

$\textbf{fun}\ sum\ (xs : \textbf{list}(\textbf{nat})) = \textbf{integral}_{+}\ \textbf{hd}\ 0\ xs$
$\textbf{fun}\ suffixsum\ (xs : \textbf{list}(\textbf{nat})) = \textbf{integral}_{::}\ sum\ ([]\ \textbf{as list nat})\ xs$

■

## 5.3 Semantics of FunDIF

Figure 5.2 contains a big step semantics of FunDIF and it has the following two judgment forms.

1 $p, \delta \vdash e \Downarrow v$ means that expression $e$ evaluates to the value $v$ in the program context $p$ (list of function declarations) and variable environment $\delta$ (a finite map from variable names to values).

2 $p, \delta \vdash^{int} \mathbf{integral}_{op}(xs \Rightarrow e_1)\ v'\ ([v_1, \ldots, v_n]\ \mathbf{as}\ \tau) \Downarrow v$ evaluates the **integral** expression to the value $v$, when the arguments to the **integral** expression already have been evaluated.

Note, that in Figure 5.2 we use $\overline{op}$ to denote the mathematical interpretation of the operator *op*.

**Definition 5.3.1 (Values in FunDIF)** *The values in FunDIF can be described by:*

$$v ::= [v_1, \ldots, v_n]\ \mathbf{as}\ \tau \mid n \mid \mathbf{true} \mid \mathbf{false}$$

■

In the operational semantics we have used the name $v$ as a placeholder for results of evaluations. The following theorem states that, if an expression $e$ evaluates to something then it is a value.

**Theorem 5.3.2 (The result of an evaluation is a value)** *Assume $p$ is a program, $\delta$ is a variable environment and $e$ is an expression. Then*

$$p\ ; \delta \vdash e \Downarrow e' \quad \Rightarrow \quad \exists v.\ e' = v$$

Proof: Induction on the height of the derivation. ■

Furthermore the semantics is deterministic as the following theorem states.

**Theorem 5.3.3 (The semantics of FunDIF is deterministic)** *Assume $p$ is a program, $\delta$ is a variable environment and $e$ is an expression. Then*

$$p, \delta \vdash e \Downarrow v_1 \wedge p, \delta \vdash e \Downarrow v_2 \quad \Rightarrow \quad v_1 = v_2$$

Proof: Induction on the height of one of the derivations. ■

$$ev\_const \quad \frac{}{p, \delta \vdash c \Downarrow c}$$

$$ev\_var \quad \frac{}{p, \delta \vdash x \Downarrow \delta(x)}(x \in Dom(\delta))$$

$$ev\_op \quad \frac{p, \delta \vdash e_1 \Downarrow v_1 \quad \ldots \quad p, \delta \vdash e_n \Downarrow v_n}{p, \delta \vdash op(e_1, \ldots, e_n) \Downarrow \overline{op}(v_1, \ldots, v_n)}$$

$$ev\_if\_true \quad \frac{p, \delta \vdash e_1 \Downarrow \textbf{true} \quad p, \delta \vdash e_2 \Downarrow v}{p, \delta \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Downarrow v}$$

$$ev\_if\_false \quad \frac{p, \delta \vdash e_1 \Downarrow \textbf{false} \quad p, \delta \vdash e_3 \Downarrow v}{p, \delta \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Downarrow v}$$

$$ev\_let \quad \frac{p, \delta \vdash e_1 \Downarrow v' \quad p, \delta[x \to v'] \vdash e_2 \Downarrow v}{p, \delta \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \Downarrow v}$$

$$ev\_call \quad \frac{p, \delta \vdash e \Downarrow v' \quad p, [x \to v'] \vdash e' \Downarrow v}{p, \delta \vdash f_{id}(e) \Downarrow v}(\textbf{fun } f_{id}(x : \tau) = e' \in p)$$

$$ev\_comp \quad \frac{p, \delta \vdash f_1(e) \Downarrow v' \quad p, \delta[x \to v'] \vdash f_2(x) \Downarrow v}{p, \delta \vdash (f_2 \circ f_1)(e) \Downarrow v}$$

$$ev\_integral \quad p, \delta \vdash^{int} \textbf{integral}_{op}(xs \Rightarrow e_1)\ v'\ ([v_1, \ldots, v_n] \textbf{ as } \tau) \Downarrow v$$

$$\frac{p, \delta \vdash e_2 \Downarrow v' \quad p, \delta \vdash e_3 \Downarrow [v_1, \ldots, v_n] \textbf{ as } \tau}{p, \delta \vdash \textbf{integral}_{op}\ (xs \Rightarrow e_1)\ e_2\ e_3 \Downarrow v}$$

$$evint\_base \quad \frac{}{p, \delta \vdash^{int} \textbf{integral}_{op}\ (xs \Rightarrow e_1)\ v\ ([] \textbf{ as } \tau) \Downarrow v}$$

$$evint\_induc \quad \frac{p, \delta \vdash^{int} \textbf{integral}_{op}\ (xs \Rightarrow e_1)\ v'\ ([v_2, \ldots, v_n] \textbf{ as } \tau) \Downarrow v_2' \quad p, \delta[xs \to ([v_1, \ldots, v_n] \textbf{ as } \tau)] \vdash e_1 \Downarrow v_1'}{p, \delta \vdash^{int} \textbf{integral}_{op}\ (xs \Rightarrow e_1)\ v'\ ([v_1, \ldots, v_n] \textbf{ as } \tau) \Downarrow \overline{op}(v_1', v_2')}$$

Figure 5.2: Formal Semantics of FunDIF

## 5.4 Type System of FunDIF

The type system of FunDIF should limit the set of valid FunDIF programs to the non-recursive function declarations. Furthermore we would like that the execution of typeable FunDIF programs are type preserving during execution (Pierce [2002] explains preservation) and that they terminate with a value.

The reason we need non-recursive function declarations is because the algorithm for *differencing* FunDIF functions, which we see later, has this as a sufficient condition for termination.

Figure 5.3 shows the typesystem of FunDIF expressions and it has the judgment form $p\ ;\Delta \vdash e : \tau$, which says that the expression $e$ is typeable with the type $\tau$ in the program $p$ and variable environment $\Delta$ (finite map from variable names to types). $T_{const}$ is mapping from constants to their types and $T_{operator}$ is a mapping from unary and binary operators to their argument and result types. $T_{const}$ and $T_{operator}$ are not presented here, but they behave "appropriately".

Note that the typesystem cannot be used to type recursive programs, because the body of a function is unfolded, when function calls are type-checked. Furthermore note that this typesystem allows non-typeable functions in the program $p$, when building derivations $p\ ;\Delta \vdash e : \tau$, if these functions are not used by $e$ or in any possible subsequent call.

The reason the type system is designed in this way in the case of function calls, is to accommodate the difference map $\mathcal{D}_{xs}$ which is defined later.

The following conjecture states that typeable programs terminate with a value and that the execution of programs is type preserving. However, this conjecture only holds if we assume that $\overline{op}$ is a total function for all operators *op*. Unfortunately this is not the case for the head and tail functions for lists, but one could imagine that exceptions for these functions could be introduced.

**Conjecture 5.4.1 (Typeable programs terminate and are type preserving)** *Assume that $\overline{op}$ is a total function for all operators op. Assume that $p\ ;\Delta \vdash e : \tau$. Then for all $\delta \in Dom(\Delta)$ there exists a value $v$ such that*

$$p, \delta \vdash e \Downarrow v \quad \wedge \quad p, [] \vdash v : \tau$$

Proof idea: Induction on the height of the derivation of $p\ ;\Delta \vdash e : \tau$. ■

Furthermore the typesystem types expressions uniquely:

**Conjecture 5.4.2 (Unique typing)** *Assume $p$ is a program, $e$ is an expression and $\Delta$ is a variable environment. Then*

$$p\ ;\Delta \vdash e : \tau \wedge p\ ;\Delta \vdash e : \tau' \quad \Rightarrow \quad \tau = \tau'$$

$$
\begin{array}{rl}
ty\_const & \dfrac{}{p\,;\Delta \vdash c : T_{const}(c)} \\[2ex]
ty\_var & \dfrac{}{p\,;\Delta \vdash x : \Delta(x)}(x \in Dom(\Delta)) \\[2ex]
ty\_op & \dfrac{p\,;\Delta \vdash e_1 : \tau_1 \ \ldots \ p\,;\Delta \vdash e_n : \tau_n}{p\,;\Delta \vdash op(e_1,\ldots,e_n) : \tau}(T_{operator}(op)(\tau_1,\ldots,\tau_n) = \tau) \\[2ex]
ty\_if & \dfrac{p\,;\Delta \vdash e_1 : \textbf{bool} \quad p\,;\Delta \vdash e_2 : \tau \quad p\,;\Delta \vdash e_3 : \tau}{p\,;\Delta \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau} \\[2ex]
ty\_let & \dfrac{p\,;\Delta \vdash e_1 : \tau' \quad p\,;\Delta[x \to \tau'] \vdash e_2 : \tau}{p\,;\Delta \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau} \\[2ex]
ty\_call & \dfrac{p\,;\Delta \vdash e : \tau' \quad p\,;\Delta[x \to \tau'] \vdash e' : \tau}{p\,;\Delta \vdash f(e) : \tau}(\textbf{fun } f(x : \tau') = e' \in p) \\[2ex]
ty\_comp & \dfrac{p\,;\Delta \vdash e : \tau'' \quad p\,;\Delta[x \to \tau''] \vdash f_1(x) : \tau' \quad p\,;\Delta[y \to \tau'] \vdash f_2(y) : \tau}{p\,;\Delta \vdash (f_2 \circ f_1)(e) : \tau} \\[2ex]
ty\_integral & \dfrac{p\,;\Delta \vdash e_2 : \tau \quad p\,;\Delta \vdash e_3 : \textbf{list } \tau_1 \quad p\,;\Delta[xs \to \textbf{list } \tau_1] \vdash e_1 : \tau_1'}{p\,;\Delta \vdash \textbf{integral}_{op}\ (xs \Rightarrow e_1)\ e_2\ e_3 : \tau}(T_{operator}(op)(\tau_1', \tau) = \tau)
\end{array}
$$

Figure 5.3: Type System of FunDIF

Proof idea: Induction of the height of one of the derivations. ■

The unique typing is an important property of FunDIF, because we need the typing to determine the source and target ipo's of a function.

## 5.5 Differencing FunDIF Functions

Before we introduce the difference of a FunDIF function we need to define free variables and substitution.

**Definition 5.5.1 (Free Variables)** *Assume e is a FunDIF expression. Then $FV(e)$ denotes the free variables of e and it is defined by structural induction on e:*

$$
\begin{array}{rcl}
FV(c) & = & \emptyset \\
FV(x) & = & \{x\} \\
FV(\textbf{let } x = e_1 \textbf{ in } e_2) & = & FV(e_1) \cup (FV(e_2) \setminus \{x\}) \\
FV(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) & = & FV(e_1) \cup FV(e_2) \cup FV(e_3) \\
FV(f(e)) & = & FV(e) \\
FV(\textbf{integral}_{op}\ (xs \Rightarrow e_1)\ e_2\ e_3) & = & (FV(e_1) \setminus \{xs\}) \cup FV(e_2) \cup FV(e_3)
\end{array}
$$

■

**Definition 5.5.2 (Substitution)** *Assume $e$ and $e'$ are FunDIF expressions and assume that $x$ is a variable. Then substitution $e[e'/x]$ is defined by structural induction on $e$:*

$$
\begin{array}{rcl}
c[e'/x] & = & c \\
y[e'/x] & = & \begin{cases} e' & x = y \\ y & x \neq y \end{cases} \\
(\textbf{let } y = e_1 \textbf{ in } e_2)[e'/x] & = & \textbf{let } y = e_1[e'/x] \textbf{ in } e_2[e'/x] \quad y \notin FV(e') \wedge x \neq y \\
(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)[e'/x] & = & \textbf{if } e_1[e'/x] \textbf{ then } e_2[e'/x] \textbf{ else } e_3[e'/x] \\
f(e)[e'/x] & = & f(e[e'/x]) \\
(\textbf{integral}_{op}\ (xs \Rightarrow e_1)\ e_2\ e_3)[e'/x] & = & \textbf{integral}_{op}\ (xs \Rightarrow e_1[e'/x])\ e_2[e'/x]\ e_3[e'/x] \\
 & & xs \notin FV(e') \wedge xs \neq x
\end{array}
$$

■

Note that the definition of substitution is a total map, because we can do alpha conversions of expressions to ensure that the conditions are satisfied in the case of *let* and *integral* expressions.

*The work in the rest of this section is extremely sketchy and vague.*

In Section 4.2 we developed the theory of differences in an abstract difference calculus, that is, we searched for functions $f'$ satisfying $f(d \oplus x) = f'(d \oplus x) \oplus' f(x)$ under suitable conditions. Figure 5.4 defines a partial map $\mathcal{D}_{xs}$ from FunDIF typing derivations to a pair consisting of a FunDIF expression and an operator. $\mathcal{D}_{xs}$ is named the *difference* with respect to the variable $xs$. The intuition is that if we consider an expression to be a function of its free variables then the difference expression corresponds to the difference of the function with respect to the variable $xs$ and the operator returned is the operator used in the target ipo. The reason the difference need typing judgments is because the theory from Section 4.2 is parameterized over the operations on the ipo's and thus we need to point at a specific source and target ipo.

The following examples should help in clarifying how $\mathcal{D}$ works. In the examples we also use some semantic equivalences to end up with a simpler expressions. We do not prove that these equivalences are correct, but it should be evident that they are. $\equiv$ is used, when we do a semantic rewrite.

$$
\begin{array}{ll}
 & \mathcal{D}_n(p\,;\Delta \vdash n : \mathbf{nat}) = (1, +) \\
 & \mathcal{D}_n(p\,;\Delta \vdash e : \mathbf{nat}) = (0, +) \quad n \notin FV(e) \\
 & \mathcal{D}_{xs}(p\,;\Delta \vdash xs : \mathbf{list}\ \tau) = ([\mathbf{hd}\ xs]\ \mathbf{as\ list}\ \tau, @) \\
 & \mathcal{D}_{xs}(p\,;\Delta \vdash e : \mathbf{list}\ \tau) = ([]\ \mathbf{as\ list}\ \tau, @) \quad xs \notin FV(e) \\
 & \mathcal{D}_t\,(p\,;\Delta \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau) = \mathcal{D}_t(p\,;\Delta \vdash e_2[e_1/x] : \tau) \\
 & \mathcal{D}_t\left(\dfrac{p\,;\Delta \vdash e : \tau \quad p\,;\Delta \vdash []\ \mathbf{as\ list}\ \tau : \mathbf{list}\ \tau}{p\,;\Delta \vdash [e]\ \mathbf{as\ list}\ \tau : \mathbf{list}\ \tau}(T_{operator}(::)(\tau, \mathbf{list}\ \tau) = \mathbf{list}\ \tau)\right) \\
= & \ \mathbf{let} \\
 & \quad (e', \oplus) = \mathcal{D}_t(p\,;\Delta \vdash e : \tau) \\
 & \ \mathbf{in} \\
 & \quad ([e']\ \mathbf{as\ list}\ \tau, @\oplus) \\
 & \mathcal{D}_t\left(\dfrac{p\,;\Delta \vdash t : \tau' \quad p\,;[x \to \tau'] \vdash e' : \tau}{p\,;\Delta \vdash f(t) : \tau}(\mathbf{fun}\ \ f(x : \tau') = e' \in p)\right) \\
= & \ \mathcal{D}_t(p\,;[t \to \tau'] \vdash e'[t/x] : \tau) \\
 & \mathcal{D}_{xs}(p\,;\Delta \vdash \mathbf{integral}_{op}\ (ys \Rightarrow e_1)\ e_2\ xs : \tau) = \begin{cases} (e_1[xs/ys], op) & op \in \{+, @\} \\ ([e_1[xs/ys]], @) & op = :: \end{cases} \\
 & \mathcal{D}_{xs}\left(\dfrac{p\,;\Delta \vdash xs : \mathbf{list}\ \tau'' \quad p\,;\Delta[xs \to \mathbf{list}\ \tau''] \vdash f_1(xs) : \mathbf{list}\ \tau' \quad p\,;\Delta[ys \to \mathbf{list}\ \tau'] \vdash f_2(ys) : \tau}{p\,;\Delta \vdash (f_2 \circ f_1)(xs) : \tau}\right) \\
= & \ \mathbf{let} \\
 & \quad (e_2', \oplus_2) = \mathcal{D}_{ys}(p\,;\Delta[ys \to \mathbf{list}\ \tau'] \vdash f_2(ys) : \tau) \\
 & \quad (e_1', @) = \mathcal{D}_{xs}(p\,;\Delta[xs \to \mathbf{list}\ \tau''] \vdash f_1(xs) : \mathbf{list}\ \tau') \\
 & \ \mathbf{in} \\
 & \quad (\mathbf{integral}_{\oplus_2}\ (ys \Rightarrow e_2'[ys @ f_1(\mathbf{tail}(xs))/ys])\ 0_{\oplus_2}\ e_1', \oplus_2)
\end{array}
$$

Figure 5.4: Symbolic differencing of FunDIF functions

**Example 5.5.3 (Differences)** *The programs from Example 5.2.1 are used to illustrate how the difference works. We have chosen not to clutter the example with too many details, which is why the entire type derivation trees are not included. However, it should be clear from the context that the programs and subexpressions in fact can be typed in this way.*

We difference the map function applied to a list of naturals.

$$\begin{aligned} & \mathcal{D}_{xs}(p\,;[xs \to \textbf{list nat}] \vdash map_f(xs) : \textbf{list nat}) \\ = \;& \mathcal{D}_{xs}(p\,;[xs \to \textbf{list nat}] \vdash \textbf{integral}_{::}\ (ys \Rightarrow f(\textbf{hd}\ ys))\ ([]\ \textbf{as list nat})\ xs : \textbf{list nat}) \\ = \;& ([f(\textbf{hd}\ xs)]\ \textbf{as list nat}, @) \end{aligned}$$

We difference the filter function applied to a list of naturals.

$$\begin{aligned} & \mathcal{D}_{xs}(p\,;[xs \to \textbf{list nat}] \vdash filter_p(xs) : \textbf{listnat}) \\ = \;& (\textbf{let}\ \ x = \textbf{hd}\ xs\ \textbf{in if}\ p(x)\ \textbf{then}\ [p(x)]\ \textbf{as list nat else}\ []\ \textbf{as list nat}, @) \end{aligned}$$

We difference the suffixsum function applied to a list of naturals.

$$\begin{aligned} & \mathcal{D}_{xs}(p, xs \to \textbf{list nat} \vdash suffixsum(xs) : \textbf{list nat}) \\ = \;& \mathcal{D}_{xs}(p, xs \Rightarrow \textbf{list nat} \vdash \textbf{integral}_{::}\ (ys \Rightarrow sum(ys))\ ([]\ \textbf{as list nat})\ xs : \textbf{list nat}) \\ = \;& ([sum(xs)]\ \textbf{as list nat}, @) \end{aligned}$$

We difference the composition of the map and filter function applied to a list of naturals. Furthermore, we apply a couple of simple semantical

equivalences to end up with simpler expressions.

$$
\begin{aligned}
& \mathcal{D}_{xs}(p, xs \to \textbf{list nat} \vdash (map_f \circ filter_p)(xs) : \textbf{list nat}) && (5.1)\\
= \;& \textbf{let} && (5.2)\\
& \quad (e_2', \oplus_2) = ([f(\textbf{hd}\ ys)]\ \textbf{as list nat}, @) && (5.3)\\
& \quad (e_1', @) = (\textbf{let}\ \ x = \textbf{hd}\ xs\ \textbf{in} && (5.4)\\
& \qquad\qquad \textbf{if}\ p(x)\ \textbf{then}\ ([x]\ \textbf{as list nat}) && (5.5)\\
& \qquad\qquad \textbf{else}\ ([]\ \textbf{as list nat}), @) && (5.6)\\
& \quad \textbf{in} && (5.7)\\
& \quad (\textbf{integral}_{\oplus_2}\ (ys \Rightarrow e_2'[ys\ @\ filter_p(\textbf{tail}(xs))/ys])\ 0_{\oplus_2}\ e_1', \oplus_2) && (5.8)\\
= \;& (\textbf{integral}_{@}\ (ys \Rightarrow [f(\textbf{hd}\ (ys\ @\ filter_p(\textbf{tail}(xs))))])) && (5.9)\\
& \quad ([]\ \textbf{as list nat}) && (5.10)\\
& \quad (\textbf{let}\ \ x = \textbf{hd}\ xs\ \textbf{in} && (5.11)\\
& \quad\ \textbf{if}\ p(x)\ \textbf{then}\ ([x]\ \textbf{as list nat}) && (5.12)\\
& \quad\ \textbf{else}\ ([]\ \textbf{as list nat})), @) && (5.13)\\
\equiv \;& (\textbf{integral}_{@}\ (ys \Rightarrow ([f(\textbf{hd}\ (ys))]\ \textbf{as list nat}))\ ([]\ \textbf{list nat}) && (5.14)\\
& \quad (\textbf{let}\ \ x = \textbf{hd}\ xs\ \textbf{in} && (5.15)\\
& \quad\ \textbf{if}\ p(x)\ \textbf{then}\ ([x]\ \textbf{as list nat}) && (5.16)\\
& \quad\ \textbf{else}\ ([]\ \textbf{as list nat})), @) && (5.17)\\
\equiv \;& (\textbf{let}\ \ x = \textbf{hd}\ xs\ \textbf{in if}\ p(x) && (5.18)\\
& \quad \textbf{then integral}_{@}\ (ys \Rightarrow ([f(\textbf{hd}\ (ys))]\ \textbf{as list nat})) && (5.19)\\
& \qquad\quad ([]\ \textbf{as list nat})\ ([x]\ \textbf{as list nat}) && (5.20)\\
& \quad \textbf{else integral}_{@}\ (ys \Rightarrow ([f(\textbf{hd}\ (ys))])\ \textbf{as list nat})) && (5.21)\\
& \qquad\quad ([]\ \textbf{as list nat})\ ([]\ \textbf{as list nat}), @) && (5.22)\\
\equiv \;& (\textbf{let}\ \ x = \textbf{hd}\ xs\ \textbf{in if}\ p(x) && (5.23)\\
& \quad \textbf{then}\ ([f(x)]\ \textbf{as list nat}) && (5.24)\\
& \quad \textbf{else}\ ([]\ \textbf{as list nat}), @) && (5.25)
\end{aligned}
$$

(5.14): We use that the list $ys$ is never empty when the anonymous function of the integral is applied and thus $\textbf{hd}\ (ys @ filter_P(\textbf{tail}\ xs)) = \textbf{hd}\ ys$.
(5.18): We lift out the let binding and the conditional.
(5.23): The integral expression is applied to a constant expression in each branch and can thus be evaluated.
We difference the composition of the suffixsum and the map function applied to a list of naturals. Again we use some semantic equivalences to rewrite

the resulting expression.

$$
\begin{array}{lll}
 & \mathcal{D}_{xs}(p, [xs \to \textbf{list nat}] \vdash (suffixsum \circ map_f)(xs) : \textbf{list nat}) & (5.26)\\
= & \textbf{let} & (5.27)\\
 & \quad (e_2', \oplus_2) = ([sum(ys)]\ \textbf{as list nat}, @) & (5.28)\\
 & \quad (e_1', @) = ([f(\textbf{hd}\ xs)]\ \textbf{as list nat}, @) & (5.29)\\
 & \textbf{in} & (5.30)\\
 & \quad (\textbf{integral}_{\oplus_2}\ (ys \Rightarrow e_2'[ys\ @\ map_f(\textbf{tail}(xs))/ys])\ 0_{\oplus_2}\ e_1', \oplus_2) & (5.31)\\
= & (\textbf{integral}_{@}\ (ys \Rightarrow ([sum(ys\ @\ map_f(\textbf{tail}(xs)))]))\ \textbf{as list nat}) & (5.32)\\
 & \quad ([]\ \textbf{as list nat})\ ([f(\textbf{hd}\ xs)]\ \textbf{as list nat})\ , @) & (5.33)\\
\equiv & ([sum(([f(\textbf{hd}\ xs)]\ \textbf{as list nat})\ @\ map_f(\textbf{tail}(xs)))] & (5.34)\\
 & \ \textbf{as list nat}, @) & (5.35)\\
\equiv & ([sum(map_f(xs))]\ \textbf{as list nat}, @) & (5.36)
\end{array}
$$

(5.35): The integral expression is applied to a list of constant length and thus we can unfold the definition of integral.
(5.36): We use the equivalence $[x]\ @\ xs = x :: xs$ and the declaration of the $map_f$ function. ■

The following should be noted on $\mathcal{D}$:

- Not all operators returned by $\mathcal{D}$ are FunDIF operators, because when differencing singleton lists we return an "operator string", which are used in the case of singleton list ipo's (remember the definition $[n_1]$ @+ $[n_2] = [n_1 + n_2]$). However, we think that they can be simulated in FunDIF (for example is $x$ @+ $y \equiv [hd(x) + hd(y)]$).

- In all applications of $\mathcal{D}$, the typing judgments used in subsequent applications of $\mathcal{D}$ are sub derivations of the argument derivation, except in the case of let expressions. However, the let typing derivation has the sub derivations $p\ ; \Delta \vdash e_1 : \tau'$ and $p\ ; \Delta[x \to \tau'] \vdash e_2 : \tau$ and we conjecture that from these, we can create a derivation of $p\ ; \Delta \vdash e_2[e_1/x] : \tau$, which is used on the right hand side.

- $\mathcal{D}$ could easily be generalized to derivations of the form $p\ ; \Delta \vdash [e_1, \ldots, e_n] : \textbf{list}\ \tau$.

- There is no difference rule for conditionals, because a conditional is in general a non-monotonic construct.

- In the chain rule the resulting expression is parameterized over the resulting operator of the difference of $f_2$ and $0_\oplus$ should be understood as the neutral element for the operator $\oplus$.
- The expression returned by $\mathcal{D}_{xs}$, may be undefined when applied to the neutral element of the source ipo.

## 5.6 Incremental Computation Using Differences

Continuing the examples from the preceding sections we show, how functions can be computed incrementally using differences. First it should be noted that the incremental functions cannot be expressed using FunDIF, because that requires that we can declare functions with more than one argument and return more than one value.

However, the incremental version can be implemented using SML. At the time of writing, we have not tried to implement these functions using SML.

## 5.7 Correctness Of The Difference Map

The purpose of this section is to show that the difference actually produces the correct difference function. First we need to conjecture that the typing derivation on the right hand side of the difference mapping of a let expression, can be constructively produced by the typing derivations of the left hand side. That is:

**Conjecture 5.7.1 (Substitution)** *Assume $\Gamma$ and $\Delta$ are function- and variable typing environments respectively, and assume that $e_1$ and $e_2$ are expressions. Then*

$$\Gamma;\Delta \vdash e_1 : \tau' \wedge \Gamma;\Delta[x \to \tau'] \vdash e_2 : \tau \quad \Rightarrow \quad \Gamma;\Delta \vdash e_2[e_1/x] : \tau \qquad (5.37)$$

■

At the time of writing we have not come further than this.

## 5.8 Related Work

In this section we discuss some of the theory related to the Finite Differencing presented in Chapter 4 and the Concrete Difference Calculus of Chapter 5. We first discuss, how our ideas are related to *deforestation* and *dynamic programming*. Then we discuss, what the similarities and differences are in relation to FunSETL by Brixen [2005]. Finally we recall the theory of Finite Differencing by Paige [1981], Selective Memoization by Acar et al. [2003] and Magic Sets Transformation by Bancilhon et al. [1986] and discuss, how the theory of our version of Finite Differencing and FunDIF is related to this.

### 5.8.1 Deforestation and Dynamic Programming

Deforestation is a program transformation technique for eliminating intermediate datastructures between function calls as introduced by Wadler [1990]. In Chapter 5 we introduced a chain rule for differencing the composition of functions. The result of differencing the composition of functions is an integral expression, where we iterate over the difference of the first function. However, integral expressions create intermediate results, but as we saw in Example 5.5.3 using some semantic equivalences we can often eliminate integral expressions in difference functions, because the integral expressions is often applied to a list of fixed length (see composition of map and filter). Thus, when we construct an incremental version of a function from the difference function, where we do not produce an intermediate result between map and filter.

Dynamic programming is a programming method for solving optimization problems (it is described by Cormen et al. [2001]). The idea of dynamic programming is to identify the structure of a solution, recursively define the values of the solution, compute the values of a solution in a bottom up fashion and then create a solution. The discovery of an incremental version of a function using difference functions is in some sense related to the step in dynamic programming where we go from top down (recursion) to bottom up computation. Example 4.2.31 shows, how the Fibonacci function can be computed bottom up using difference functions, where the difference function is found from a top down computation of the Fibonacci function. Thus, one could imagine that if we extend FunDIF with a kind of recursion, then it might be possible to automatically create bottom up computations from top down. However, this is very speculative, because it is always "dangerous" to add recursion to a non-recursive language.

### 5.8.2 FunSETL

We do not describe FunSETL here, because it was presented in Chapter 3. FunSETL and FunDIF are similar, because they both have restrictive expressive power, where recursion is not allowed and where a basic construct is included for a restricted form of iteration. FunSETL has the *fold* expression (multiset version) and FunDIF has the integral expression (on lists) as iterative construct. The integral expression from FunDIF can be expressed by the foldright construct from SML, if we use the accumulating parameter to keep track of the list we are iterating over. The declaration is shown here:

```
1 fun integral operator f  c xs =
2   \#2(foldr (fn (x,(acclist,accval))
3             => (x :: acclist, operator(f(x :: acclist),accval)))
4       ([],c) xs)
```

It only makes sense to apply the incrementalization algorithm for FunSETL (described by Brixen [2005][p.76]) once to incrementalize a FunSETL function. If we inspect the incrementalization algorithm we can see that function calls under a fold expression that depend on the bound variables by the anonymous functions are not incrementalized. However, the theory of Finite Differencing from Chapter 4 does not have this restriction. In Example 4.2.26 we saw that $suffixsum$ could be differenced twice using the Finite Differencing theory and Example 5.5.3 shows how $suffixsum$ is differenced once in FunDIF, but from the result it should be clear that the difference algorithm of FunDIF applies again. Thus, we can construct an incremental version of suffixsum that runs in constant time. However, if we for one moment assume that the FunSETL fold expression were applied to lists and that we formulated the $suffixsum$ computation using *fold* instead of integral, then only the outer fold expression would be eliminated in the incremental version (the *sum* function would remain), that is, FunSETL only removes a linear factor in the running time.

It should be noted that the incrementalization technique using differences only applies to monotonic functions. If we declare a FunSETL function that computes the sum of a multiset of integers then this function is not monotonic, but the FunSETL incrementalization transforms this program correctly. However, the current version of the finite differencing theory cannot handle this.

### 5.8.3 Finite Differencing

Paige [1981] and Paige and Koenig [1982] provide a description of *Finite*

*Differencing.* Applicative expressions are expressions with free variables and are thereby considered functions of these free variables. Finite differencing is a program transformation technique for maintaining the results of *applicative expressions* within code-blocks.

The transformations described by Paige [1981] and Paige and Koenig [1982] are developed for SETL programs. SETL is a set-based imperative programming language. It has beyond the basic types, sets, finite maps and tuples. However, the finite difference techniques should also be applicable to other programming languages.

SETL is introduced in this section by example and we show, how Paige's finite differencing can be applied to to these examples. We are not concerned with the input and output constructions of SETL and thus we only introduce *blocks* (sequences of statements) of SETL code to illustrate the finite differencing techniques and to explain, how finite differencing can be used to optimize programs.

First let us see an example of a SETL program.

**Example 5.8.1** *SETL program that computes the square of the ten first prime numbers and outputs the squares in each iteration (based on Eratosthenes Sieve).*

```
1:  A := {2} ;
2:  Z := 2;
3:  while(#A < 10)
4:     Z := Z + 1;
5:     B := {y ∈ A | Z  mod y ≠ 0}
6:     if #A = #B then A := A with Z;
7:      end if ;
8:     C := {x^2 | x ∈ A} ;
9:     ⟨output  C⟩
10:  end while;
```

The loop checks if a number $Z$ is a prime, by checking if any of the primes found so far is a divisor for $Z$ and if this is not the case then $Z$ must be a prime. When the while loop terminates, the set $A$ contains the first ten prime numbers and $C$ contains their squares. During execution the squares are stored in $C$ and in each iteration $C$ is printed on the screen (the specific code for this is not included). ■

The idea of finite differencing is to *differentiate* an applicative expression $E = f(x_1, \ldots, x_n)$ with respect to a block of code $B$. This means that, if we are given an applicative expression $E = f(x_1, \ldots, x_n)$ and a block of code

$B$ then the result of $E$ is maintained during the execution of $B$, when $B$ changes one of the variables $x_i$. That is, each time there is a change $dx_i$ to variable $x_i$ in the block $B$, then $dx_i$ is replaced by the code $\delta E\langle dx_i\rangle = \delta^- E\langle dx_i\rangle\, dx_i \delta^+ E\langle dx_i\rangle$, where $\delta^- E\langle dx_i\rangle$ and $\delta^+ E\langle dx_i\rangle$ are blocks of code containing *pre- and post-derivative* code with respect to the change $dx_i$. The pre- and post-derivative ensures that the result of $f(x_1,\ldots,dx_i,\ldots,x_n)$ is available in the variable $E$ after executing $\delta^- E\langle dx_i\rangle\, dx_i \delta^+ E\langle dx_i\rangle$. Furthermore each computation in $B$ of $f(x_1,\ldots,x_n)$ is replaced by $E$, because finite differencing ensures the invariant $E = f(x_1,\ldots,x_n)$ at all lines of the original code (except the statements that change the value of a variable $x_i$). In many cases this results in a program speedup of $B$ and in some cases even an asymptotic speedup. The derivative of an entire program, together with some initialization code to declare $E$, can be used to generate a program that is semantically equivalent to the original program.

The goal of finite differencing is to speedup the execution of a program by differentiating one or more applicative expression with respect to the entire program, where the pre- and post-derivative code is "cheap" and where we can benefit from using $E$ instead of recomputing it again and again. Paige and Koenig [1982] show that the derivative operator is linear ($\delta E\langle B_1 B_2\rangle = \delta E\langle B_1\rangle\, \delta E\langle B_2\rangle$) and that there exists a chain rule for differentiating with respect to applicative expressions that depend on each other.

Let us see, how the applicative expression $E = \{x^2 \mid x \in A\}$ can be differentiated with respect to the program from Example 5.8.1.

**Example 5.8.2** *Let us name the code block from Example 5.8.1 for $B$ and let us differentiate $E = \{x^2 \mid x \in A\}$ with respect to $B$.*

Using that the difference operator is linear and because the expression $E$ is a function of $A$, we get that $\delta E\langle B\rangle$ is

1: $\delta^- E\langle A := \{2\}\rangle$;
2: $A := \{2\}$;
3: $\delta^+ E\langle A := \{2\}\rangle$;
4: $Z := 2$;
5: **while**$(\#A < 10)$
6: $\quad Z := Z + 1$;
7: $\quad B := \{y \in A \mid Z \mod y \neq 0\}$
8: $\quad$ **if** $\#A = \#B$ **then**
9: $\quad\quad \delta^- E\langle A := A$ **with** $Z\rangle$;
10: $\quad\quad A := A$ **with** $Z$;
11: $\quad\quad \delta^+ E\langle A := A$ **with** $Z\rangle$;

12: **end if** ;
13: $C := E$
14: $\langle output\ \ C \rangle$ ;
15: **end while**;

The above code is equivalent to

1: $E := \{2^2\}$ ;
2: $A := \{2\}$ ;
3: $Z := 2$;
4: **while**$(\#A < 10)$
5: $Z := Z + 1$;
6: $B := \{\forall y \in A \mid Z \mod y \neq 0\}$
7: **if** $\#A = \#B$ **then**
8: $E := E$ **with** $Z^2$;
9: $A := A$ **with** $Z$;
10: **end if** ;
11: $C := E$;
12: $\langle output\ \ C \rangle$
13: **end while**;

where we have replaced $\{x^2 \mid x \in A\}$ by $E$, and where the post-derivatives are empty, because only pre-derivatives are needed to maintain $E$. ■

Let us see another example, where the chain rule is applied.

**Example 5.8.3** *The following example contains two applicative SETL expressions $E_1$ and $E_2$, where $E_1$ takes all the numbers greater than 5 from the set $A$ and $E_2$ adds one to each such element. Differentiate $E_2, E_1$ with respect to $A := A$ **with** $x$.*

$E_1 = \{x \in A \mid x \geq 5\}$ ;
$E_2 = \{x + 1 \mid x \in E_1\}$ ;

By the chain rule given by Paige and Koenig [1982] we get that

$$
\begin{aligned}
& \delta E_2, E_1 \langle A := A \textbf{ with } x \rangle \\
= \ & \delta E_2 \langle \delta E_1 \langle A := A \textbf{ with } x \rangle \rangle \\
= \ & \delta E_2 \left\langle \delta^- E_1 \langle A := A \textbf{ with } x \rangle ; A := A \textbf{ with } x; \delta^+ E_1 \langle A := A \textbf{ with } x \rangle \right\rangle \\
= \ & \delta E_2 \langle \textbf{if } x \geq 5 \textbf{ then } E_1 := E_1 \textbf{ with } x \ \textbf{ endif } ; A := A \textbf{ with } x \rangle \\
= \ & \delta E_2 \langle \textbf{if } x \geq 5 \textbf{ then } E_1 := E_1 \textbf{ with } x \ \textbf{ endif } \rangle \, \delta E_2 \langle A := A \textbf{ with } x \rangle \\
= \ & \textbf{if } x \geq 5 \textbf{ then } \delta^- E_2 \langle E_1 := E_1 \textbf{ with } x \rangle \, E_1 := E_1 \textbf{ with } x \textbf{ endif } ; A := A \textbf{ with } x \\
= \ & \textbf{if } x \geq 5 \textbf{ then } E_2 := E_2 \textbf{ with } x^2; E_1 := E_1 \textbf{ with } x \textbf{ endif } ; A := A \textbf{ with } x
\end{aligned}
$$

Thus, if we have a program, where the statement $A := A$ **with** $x$ is executed somewhere in the body and where the invariants $E_1 = \{x \in A \mid x \geq 5\}$ and $E_2 = \{x + 1 \mid x \in E_1\}$ hold before the execution of $A := A$ **with** $x$ then they also hold after executing $A := A$ **with** $x$ by replacing it by the derivative code above. ■

The applicative expressions $E_1$ and $E_2$ from Example 5.8.3 can be considered a function composition of a function filtering all elements greater than 5 and a function adding one to all these elements. That is, $E_1$ and $E_2$ is a composition of specialized *filter* and *map* functions and the general case of *filter* and *map* would not be more complicated than this.

The derivatives are not unique, because we could choose all derivatives to have the empty pre-derivative and as post-derivative just use $E := f(x_1, \ldots, x_n)$, that is, recompute the applicative expression every time there is a change to one of its arguments. However, this is not a very good derivative, because it often introduces more computations than it removes. Paige [1981] and Paige and Koenig [1982] describe recommended derivatives for many SETL constructs.

In the examples the applicative expressions have been *continuous*, that is, there exist "cheap" derivative code to maintain the results of the applicative expressions, when there is a change to one of the arguments. However, this is not always the case. The SETL expression $E = \{x \in A \mid x > q\}$ is not continuous in $q$ (but still in $A$) and thus there exists no good derivative for the statement $q := q + 1$.

Some of the inspiration of the Finite Differencing from Chapter 4 has come from the Finite Differencing work from this section. Let us compare the work of this section with our work of finite differencing.

The work by Paige and Koenig [1982] is highly related to incremental computation, because the applicative expressions corresponds to functions, where the results need to be maintained cheaply, when there is a change on the arguments. That is, the derivative $\delta E \langle dx_i \rangle$ (where $E = f(x_1, \ldots, x_n)$) contains a "cheap" computation of $f(x_1, \ldots, dx_i, \ldots, x_n)$ possibly using the result $E = f(x_1, \ldots, x_n)$. Thus the derivative can be considered an incremental computation of $f(x_1, \ldots, dx_i, \ldots, x_n)$ from $f(x_1, \ldots, x_n)$. However, we are only interested in constructing incremental versions of functions and therefore the entire function should be considered the applicative expression. Furthermore, the differentiation algorithm only caches the result of the applicative expression and thus if we want to use intermediate results, when we compute a function incrementally (like in FunSETL) we need to make this explicit when differentiating the function. That is, we need to split up the applicative expressions representing the function into dependent applicative

expressions. Thus, the detection of intermediate results does not come for free using the Finite Differencing explained in this section.

### 5.8.4 Memoization

Acar et al. [2003] presents a functional programming language MFL with built in *selective memoization*, which means that during execution, return values from function calls are *memoized* and if the same function call (or an equivalent function call) is met again, then the memoized result is used instead of recomputing from scratch. Control flow of the program is used to determine, which memoized results should be stored and retrieved.

The important issues of memoization are

1 *equality testing*: Since the results of all function calls are stored in a *memoization table*, we need a way to determine, when a memoized value can be used instead of recomputing a term. This means that we need to do an equality test on the arguments to a function call, which can be computationally expensive, if recursive data structures are used. This can change the time complexity of the program for the worse.

2 *dependencies*: We need a way to figure out how the input to a function is related to the output, because often many inputs to a function give the same output. This means that we need a way to identify *equivalence classes* of input that are mapped to the same output value.

3 *memory usage*: Because all the results of function calls are stored, this can eat up a lot of memory, especially if recursive data structures are used. Thus, there is a need for a policy to decide when memoized results should be discarded.

Acar et al. [2003] introduces the core constructs of MFL. It is called selective, because the framework allows to some extent the programmer to have control over the equality testing, identification of dependencies and memory usage. The semantics of MFL uses the branching information (control flow) of a program to determine, which results should be memoized for the different branches of the computation. The branching information is used as a *key* in the memoization table (a way of testing for data dependencies).

Below we see an example of a function expressed in the same language as the examples given by Acar et al. [2003].

**Example 5.8.4** *Write a MFL function that computes the sum of the elements of a list of integers.*

The following function solves the problem.

```
1: mfun sum(xs' : !(int list)) =
2: let !xs = xs' in return(
3:   mcase (unbox xs) of
4:     nil  ⇒ 0
5:   | cons(h, t) ⇒ h + sum(!t)
6: ) end
```

Parameters to functions are called *resources* and to get their value we need to use a **let** construct. When a type is marked with a ! it signals the framework that there is a *dependency* between the value of this type and the result of a computation (! can also be thought of as a constructor). The memoization table requires (because hashing is used) that elements of $!\tau$ types can be indexed, that is, a map from a value to a unique index. For lists this means that they need to be *boxed*. A box means that a unique tag is created an attached to the value (identical values get the same tag). Therefore the programmer needs to manually *box* and *unbox* list values. The other constructs used in the program have the expected semantics. However, if the *sum* function is only called once we can not benefit from the memoized results (identical recursive calls does not occur). ■

Memoization is related to incremental computation: If we have computed $f(x)$ and we also want to compute $f(y \oplus x)$ then if $f(y \oplus x)$ encounters some of the same function calls as $f(x)$ they can be retrieved from memory. As an example, assume that we have computed $sum(xs)$ for some list $xs$ using the *sum* function from Example 5.8.4 and assume that we want to compute $sum(\mathbf{cons}(x, xs))$ for some $x$. The computation of $sum(\mathbf{cons}(x, xs))$ can then use the memoized result $sum(xs)$, which essentially means that the *sum* function is computed incrementally. Acar et al. [2003] also gives an example showing, how *quicksort* can be computed incrementally using memoization and that the running time of *quicksort* applied to $\mathbf{cons}(x, xs)$, where $quicksort(xs)$ and its sub calls are memoized, is $\mathcal{O}(n)$, where $n$ is the length of the list $x :: xs$. Thus, illustrating that memoization is related to incremental computation.

Memoization differs from finite differencing, because memoization is dynamic (results are memoized at run time) where as finite differencing generates incremental programs statically from the original source code. Furthermore memoization does not "intelligently" select the intermediate results that could be interesting for future computations. However, in selective memoization, where the programmer marks the variables for which the output depends, makes memoization more "intelligent", but unfortunately it is

still the programmers responsibility to mark the dependency variables.

### 5.8.5 Magic Sets Transformation

The *Magic Sets Transformation* is a transformation used to optimize logic programs. The Magic sets transformation is introduced by Bancilhon et al. [1986]. In this section we present a brief overview of the ideas of the Magic Sets Transformation.

Let us first see an example of a logic program.

**Example 5.8.5** *The reflexive transitive closure of a relation.*

Assume that the universe of elements is $\{a, b, c, d, e\}$ and assume that we have a relation $rel = \{(b, c), (c, d)\}$. The following logic program computes the reflexive transitive closure of the elements that $b$ are related to.

```
closure(X,X).
closure(X,Y) :- closure(X,Z) , rel(Z,Y).
```

where the query is $closure(b, X)$.

$closure(X, X)$ means that all elements in the universe are related to them selves (reflexivity). $closure(X, Y) : -closure(X, Z), rel(Z, Y)$ means that $(X, Y)$ is in the closure relation, if $X$ and $Z$ are related in the closure relation and $Z$ and $Y$ are related in the $rel$ relation for some Z. Iterated usage of the last rule generates the transitive closure. The query $closure(b, X)$ means that we wish to know all $X$ such that $closure(b, X)$, which is a selection and projection of the *closure* relation. ■

Bottom up evaluation means that the entire *closure* relation from Example 5.8.5 is computed in the following way: First the rule $closure(X, X)$ is applied to yield that $\{(a, a), (b, b), (c, c), (d, d), (e, e)\} \subseteq closure$. Then the rule $closure(X, Y) : -closure(X, Z), rel(Z, Y)$ is used from right to left: We know all the elements of the relation $rel$ and a subset of the *closure* relation, which can be used to discover new elements of *closure*. This process is iterated until the entire *closure* relation is computed. Then we select only those elements that has $b$ as the first component and project away the $b$.

The Magic Sets Transformation optimizes logic programs, where queries are computed in this manner. The idea of the Magic Sets Transformation is that when we issue a query we often know that we are only interested in the part of the result. In the closure relation we are only interested in the elements that has $b$ as the first component. Analyzing the rules for the *closure* relation we can see that during computation we only need to

maintain the elements of *closure* that have $b$ as the first component. Thus we can write the following equivalent logic program.

**Example 5.8.6** *A "magic" computation of the query.*

Declare:

```
magic(b).
closure(X,X) :- magic(X)
closure(X,Y) :- closure(X,Z),rel(Z,Y),magic(X)
```

where the query is $closure(b, X)$. This is another logic program, but where the query gives the same result as the query from Example 5.8.5. However, during a bottom up computation the *magic* relation ensures that we do not compute the entire *closure* relation in the query from Example 5.8.5. ■

The process of constructing the program in Example 5.8.6 from the program in Example 5.8.5 is referred to as *Magic Sets Transformation* by Bancilhon et al. [1986] and the process can be automated for a certain class of logic programs. However, the new program can be much larger than the original program, because there may be a need to construct many magic relations and many new versions of each of the relations in the original program.

This optimization strategy is not really related to Finite Differencing, because the query is not optimized with respect to a small change in the input. However, the optimization seems highly related to *partial evaluation* (explained by Jones [1997]). The "magic" relation ensures that the relations are specialized with respect to a certain kind of query, where a number of the arguments are static. In Example 5.8.6 we saw, how a "magic" relation made the first argument to the *closure* relation static.

### 5.8.6 Incremental View Maintenance

As explained in Chapter 2 view maintenance is essential in database theory for the ERP system area. Gupta and Mumick [1999] introduces *incremental view maintenance*. Incremental view maintenance means that if we have declared a materialized view on some relations, how can we *incrementally* update the materialized view when records are inserted, modified or deleted. Furthermore we would also like to know, whether or not we need access to the relations the view is declared on when we need to update the view.

Gupta and Mumick [1999] give an overview of some of the algorithms that can be applied for incremental view maintenance depending on the expression power of view definition language. For example is Paige's Finite Differencing theory is applied in this area.

# Chapter 6

# Summary and Future Work

In Chapter 2 we saw that the reporting technologies used today are not suited for real-time computation of report functions, which confirms that there is room for improvement in the ERP system area for real-time computation of report functions.

Chapter 3 is a preliminary argument for Hypothesis 1. It shows that it is possible to achieve real-time computation of report functions using declarative specifications and automatic incrementalization. However, only a small number of functions have been implemented using FunSETL and thus it only shows that the idea looks promising.

Chapter 4 and Chapter 5 contains the first iteration of developing a calculus, where report functions can be expressed declaratively and automatically incrementalized, and which should be more powerful than FunSETL. However, the calculus is still under development and there is still a long road ahead of us. In the current version it is only possible to express very simple functions of one argument and it can maybe be hard to see, how it can scale to report functions.

Thus, we need to

- Continue the development of the finite differencing calculus and realize the calculus in a programming language (possibly FunDIF).
- Show that the finite differencing calculus can be applied to report functions.
- Maybe look in to incrementalization of curried functions.

The first two points are obvious extensions of the work in progress. However, we have not investigated any attention to the last point yet (it seems more

complex than the first two). Incrementalizing curried functions could be very interesting, because this could probably be used to simulate OLAP cubes giving fast replies to OLAP queries.

# Bibliography

U. A. Acar, G. E. Blelloch, and R. Harper. Selective memoization. *SIGPLAN Not.*, 38(1):14–25, 2003. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/640128.604133.

A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantics foundations and query execution. *The VLDB Journal - The International Journal on Very Large Data Bases*, 15(2):121–142, 2006. ISSN 1066-8888. doi: http://dx.doi.org/10.1007/s00778-004-0147-z.

F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, New York, NY, USA, 1986. ACM. ISBN 0-89791-179-2. doi: http://doi.acm.org/10.1145/6012.15399.

D. Brixen. Inkrementelle metoder til REA-baseret rapportering. Master's thesis, Department of Computer Science, University of Copenhagen, 2005. URL `http://www.charme.dk/detkanendatalog/Specialer/danielbrixenspeciale.pdf`.

H. chih Yang, R.-L. H. Ali Dasdan, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001. ISBN 0262531968.

J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communication of the ACM*, 51(1):107–113, 2008. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1327452.1327492.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, volume 28(6), pages 237–247, New York, 1993. ACM Press. URL `citeseer.ist.psu.edu/flanagan93essence.html`.

H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems, The Complete Book*. Prentice Hall, 2002. ISBN 0-13-031995-3.

L. Golap and M. T. Özsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, 2003. ISSN 0163-5808. doi: http://doi.acm.org/10.1145/776985.776986.

R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics, Second Edition*. Addison-Wesley, 2004. ISBN 0-201-55802-5.

A. Greef, M. Fruergaard Pontoppidan, and L. Dragheim Olsen. *Inside Microsoft Dynamics AX 4.0*. Microsoft Press, 2006. ISBN 0-7356-2257-4.

A. Gupta and I. S. Mumick. Maintenance of materialized views: problems, techniques, and applications. pages 145–157, 1999.

M. R. Hansen and H. Rischel. *Introduction to Programming using SML*. Addison-Wesley, 1999.

T. Hvitved. Architectual analysis of microsoft dynamics nav. 2008.

N. D. Jones. *Computability and complexity: from a programming perspective*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0-262-10064-9.

H. Levy and F. Lessman. *Finite Difference Equations*. Dover Publications, Inc., New York, NY, USA, 1992. ISBN 0-486-67260-3.

P. M. Lewis, A. Bernstein, and M. Kifer. *Databases and Transaction Processing*. Addison Wesley, 2002. ISBN 0-201-70872-8.

Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst.*, 20(3):546–585, 1998. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/291889.291895.

M. N. Nissen and K. F. Larsen. FunSETL—functional reporting for ERP systems. In O. Chitil, editor, *Draft Pre-Proceesings for IFL 2007*, number 12-07 in Technical Report series of the Computing Laboratory, University of Kent, UK, pages 268–289, September 2007.

R. Paige. *Formal Differentiation: A Program Synthesis Technique*, volume 6 of *Computer Science and Artificial Intelligence*. UMI Research Press, 1981. Revision of Ph.D. dissertation, New York University, 1979.

R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, 1982. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/357172.357177.

B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002. ISBN 0-262-16209-1.

A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts, 4 th. Edition*. McGraw-Hill, 2002. ISBN 0-07-112268-0.

D. Studebaker. *Programming Microsoft Dynamics NAV*. Packt Publishing Ltd., 2007. ISBN 978-1-904811-74-9.

D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.

M. Torgersen. Querying in C#: How language integrated query (LINQ) works. In *OOPSLA Companion*, pages 852–853, 2007.

P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:344–358, 1990.