SML# in Industry: A Practical ERP System Development [*]

Atsushi Ohori [†], Katsuhiro Ueno[*] [‡]

Tohoku University
{ohori, katsu}@riec.tohoku.ac.jp

Kazunori Hoshi, Shinji Nozaki, Takashi Sato,
Tasuku Makabe, Yuki Ito

NEC Solution Innovetors, Ltd. [§]
{k-hoshi, s-nozaki}@wh.jp.nec.com, t-sato@yk.jp.nec.com
{t-makabe, yu-ito}@wr.jp.nec.com

Abstract

This paper reports on our industry-academia project of using a functional language in business software production. The general motivation behind the project is our ultimate goal of adopting an ML-style higher-order typed functional language in a wide range of ordinary software development in industry. To probe the feasibility and identify various practical problems and needs, we have conducted a 15 month pilot project for developing an enterprise resource planning (ERP) system in SML#. The project has successfully completed as we have planned, demonstrating the feasibility of SML#. In particular, seamless integration of SQL and direct C language interface are shown to be useful in reliable and efficient development of a data intensive business application. During the program development, we have found several useful functional programming patterns and a number of possible extensions of an ML-style language with records. This paper reports on the project details and the lessons learned from the project.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (Functional) Programming

Keywords Standard ML; Business Application; Database Programming; Record Polymorphism

[*] **This is the author's version of the work (as of 2014-06-16). It is posted here for your personal use. Not for redistribution. The definitive Version of Record will be published in ACM ICFP '14, September 1–6, 2014, Copyrigh2014 ACM 978-1-4503-2873-9 /14/09 http://dx.doi.org/10.1145/2628136.2628164**

[†] Atsushi Ohori and Katsuhiro Ueno have been partially supported by Grant-in-aid for scientific research (B), grant no:25280019.

[‡] Katsuhiro Ueno has also been partially supported by Grant-in-aid for scientific research (young investigator B), grant no:24700021.

[§] On April 1st, 2014, NEC Software Tohoku, Ltd. became a part of NEC Solution Innovators, Ltd.

1. Introduction

Advantages of ML-style polymorphic languages have been widely recognized in academia. Higher-order functions allow programmers to write elaborate code in a concise and declarative manner. Polymorphic typing and a sophisticated module system enhance the reliability of the system through static type checking and data abstraction. Despite these advantages, ML-style languages have not yet been widely used in industry. Among major software companies in Japan, for example, very little is known about ML and there is virtually no experience in using ML in software production. As a developer of an ML-style language SML# [13], we find this situation unfortunate. As a software company, we also find an ML-style language as a significant potential in enhancing the productivity and reliability in our ordinary software production. Declarative programming, static type inference, signature checking, safe and scalable module system should all have significant positive impact not only in symbolic computation such as theorem provers and compilers but also in ordinary software development such as inventory management, management accounting, and resource planning.

Based on these general observations, NEC Solution Innovetors, Ltd. and Tohoku University have started joint research aiming at establishing a programming environment of SML# for general software development. SML# is a new ML-style language being developed at Tohoku University [13]. In addition to its conformance with the Definition of Standard ML [8], it supports several practically useful features including: seamless integration SQL (the standard database query language), direct C language interface, concurrent GC and native multithreading support on multicore CPU. Among them, seamless SQL integration and direct C language interface should be particularly useful in data intensive business software development.

As the first step in this endeavor, we have conducted a 15 month pilot project to develop an enterprise resource planning (ERP) system in SML#, which will be used in NEC Solution Innovetors, Ltd. to manage the company's projects and related resources. The goal of this project is to probe the feasibility of SML# and to identify various practical problems and needs in using SML# for ordinary software development in industry. As we shall explain in the next section, an ERP system is chosen by considering this goal. The system development has successfully completed, and the ERP system is currently under evaluation with the actual databases containing the company's project, personnel and accounting information. Although we do not have statistical measurement of the productivity of using SML# at this moment, the development has been completed as we planned with the expected productivity. The SML# features of the seamless database integration and direct C library interface have indeed been shown to be useful.

In addition to the completed ERP system, we have obtained project management skills in designing and developing business applications in ML. As a software development project using a new language in a company, we had to manage a number of problems including programming education, software quality control, and design method. After sorting out these problems, the program development has been smoothly done with a satisfactory result. During the development, we have found several useful polymorphic programming idioms that can serve as *functional design patterns* for data intensive business software development in ML. Another important assets we have acquired through this project are the insights toward improving ML-style languages. Experience with database programming revealed the needs and interesting research issues concerning flexible manipulation of records in a polymorphic language. These lessons learned from our project should also be useful for anyone who plans to adopt an ML-style language in industry.

The rest of the paper is organized as follows. Section 2 describes the overview of the system we have developed and miscellaneous problems we have tackled in carrying out the project. Section 3 describes the system architecture and the details of the development. Section 4 discusses evaluation of the project management and new findings through the project. Section 5 discusses the lessons learned through the project and suggests future improvement of ML-style languages. Section 6 reports the current status and initial evaluation of the developed system. Section 7 concludes the paper with a brief description of our future plan.

2. The Project Overview

This section outlines the system we have developed, and discusses miscellaneous problems we have overcome out in carrying out the project.

2.1 The target ERP system

We have set our goal to develop an ERP system that has the following functionality.

1. It reports the current status of each software project in the company based on the company database which are updated frequently.
2. It simulates the future costs and expected profits of existing and planned projects under a given set of parameters on resources entered by the operator.

There have been growing needs among managers and executives in the company for accurate estimate of each project status at any given time, which is not easily available in the currently used system. So an ERP system having these functionalities is itself a highly desired system in the company. In addition, this target system is appropriate for our purpose of probing the feasibility and identifies various practical problems in using SML#. Firstly, it represents a typical business application involving extensive database accesses, various business logic programming, and report generation. Secondly, since we do not have much previous experience in developing a similar system, the development must contain an entire software development process: requirement identification, system architecture selection, system design, program development, testing and evaluation. Finally, through the use of the developed system in NEC Solution Innovetors, Ltd., we can evaluate the system's maintainability and extensibility as well as its performance and overall quality.

2.2 Miscellaneous issues in carrying out the project

To carry out the first software development project using ML in the company, we had to sort out a number of problems before starting the program development. We list some of them below.

- Programming education.

  Since none of the software engineers in NEC Solution Innovetors, Ltd. had previous experience in ML, we have set up an education course on ML programming and the related techniques. We have designed and given a 6 day intensive course, covering ML programming basics, SML# features, system programming, and an example system development.
- Acquiring domain knowledge.

  As stated earlier, we do not have much previous experience in an ERP system development. So we have studied the basics of management accounting, and have analyzed the company's accounting principles and resources. Sharing these domain knowledge has been crucial for our combined development team to understand, analyze and review the program code.
- Software quality control.

  The software development must conforms to the ordinary quality standard set by the company. This requires to make documents for: requirement definition, primary design, functional design, program design, and test design. Since there is no standard method available for ML, we had adopted the existing standards for most of the cases and have adjusted during the project . For example, we have adapted class diagram with ML-style signatures for program design.
- Design method.

  We had to develop and share a design method for the system, i.e. how to analyze the problem and how to represent it as a programming system. For this purpose, again, we have found very little resources for ML-based system development. As a compromising starting point, we have adopted the model-view-controller approach [6], and have tried to refine their components for ML based system. One positive outcome of this effort is *abstract model construction* we shall present in Section 4.1.

Once we had sorted out these problems, program development has been largely smooth.

3. The ERP System

This section describes the architecture design and the details of the system development.

3.1 The system architecture

To design the overall architecture of this system, we regard the ERP system as a web application and decompose the system into its major components by adopting the model-view-controller architecture. We then design the system by elaborating each component. The following are the final structure of the system.

- **The Model.** This part consists of SML# modules realizing the services the ERP system offers, including current status reporting and a several forms of simulations. The set of modules are organized into the following two layers.
  1. *Persistent data (raw models).* Most of the data are stored in a large relational database system serving through SQL Server. Figure 1 shows a simplified ER diagram of the database schema. In the actual database, table names and column names are represented in Japanese and they are directly handled by SML# programs. It contains the current status of company's various resources, including projects, employees, contracts and orders and is updated frequently. This database is directly accessed through SML# SQL integration [12]. In addition, the system reads data files in CVS and TVS formats, containing semi-stable data such as the

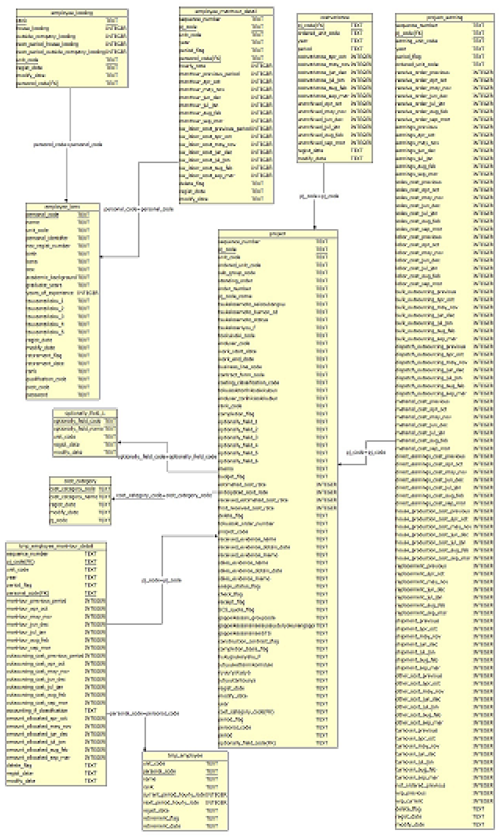


Figure 1. A Simplified ER Diagram of the Database

current labor costs for each rank of employees, and the budgets of the current half year term.

These data are represented as a set of ML modules and are regarded as *raw models* used by the *abstract models* described below.

2. *Abstract models.* They are modules that realize the required services such as status reporting and several form of simulations. They are implemented by ML programs using raw models.

This two-layered organization is our new design patterns that refines conventional MVC pattern, which we describe in details in Section 4.1,

- **The Controller.** A set of functions that are called from the dispatcher function to perform appropriate action according to the user request. The dispatcher function is called from `main` function of the web application framework described below, and performs the following loop.
  - get a user request,
  - initialize the time,
  - analyze the user request and call the corresponding controller function,
  - generate a view and return a response to the user.
- **The View.** This part consists of a set of ClearSilver template files for various report formats. These files are manipulated by the SML# ClearSilver binding described below.

```
_require "basis.smi"
_require "./form.smi"
_require "./Log.smi"
_require "./Exception.smi"
_require "./Fcgi.smi"
structure Server =
struct
  datatype method = GET | POST
  type request = ...
  type response = ...
  val main : (request -> response) -> unit
  val uriOf : string list -> string
end
```

Figure 2. Interface File Example : Web Server Framework

- **Libraries**
  - Web Server Framework. A simple framework written in SML# that provides datatypes for requests, responses and forms, and the sever function

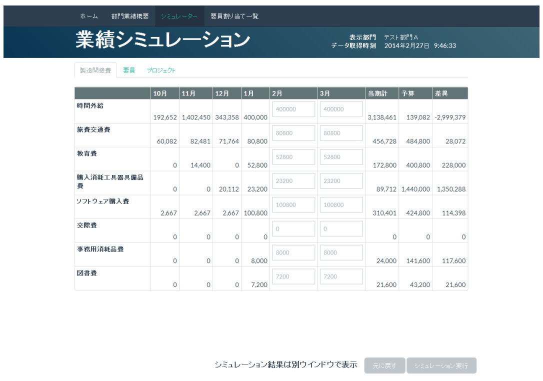
    `main : (request -> response) -> unit`
  - Common Libraries including those for CVS and TVS file parsers, time and date format conversions.
  - Global functions for system configuration and logging.
  - External (third party) Libraries
    - FastCGI for web server framework. C API (`FCGI_getchar` etc) are directly imported through SML# C FFI.
    - JSON parser. We have used Standard ML of New Jersey's implementation.
    - ClearSilver HTML Template System. We write a simple C stub functions and import them through SML# C FFI.
    - JavaScript Libraries: Bootstrap, Prototype, jQuery, ccchart.

3.2 The development details

The development has been done in two stages: the prototype development and the development of the system version 1. About 30% of prototype codes are re-used, and the rest are re-written in the version 1 system. Most coding has been done by NEC Solution Innovetors, Ltd. Tohoku University has developed the web server framework and provided the necessary language extensions and supports. To achieve industry strength ML coding, we have held weekly meeting for code review and re-factoring. This has been quite effective in enhancing the quality of the system and boosting ML programming skills of the project members.

ML components have been developed using the separate compilation feature of SML#. For each major module, we first decide the initial interface by writing an interface file of SML#. As an example, Figure 2 shows the interface file of the web server framework. Using these interface files, we have developed each component separately. For each SML# source file, SML# separate compiler produces an object file in the standard ELF format. SML# object files, C object files and libraries are then linked together by SML#, which is invoked through the Unix `make` system. This model works effectively among our team.

The developed system consists of about 25 k lines of code excluding third-party libraries, among which 12 k lines are written in SML#. The code size details are given in the following table.



業績シミュレーション

表示部門 テスト部門A
データ取得時刻 2014年2月27日 9:46:33

	10月	11月	12月	1月	2月	3月	当期計	予算	差異
時間外給	192,652	1,402,450	343,358	400,000	400000	400000	3,138,461	139,082	-2,999,379
旅費交通費	60,082	82,481	71,764	80,800	80800	80800	456,728	484,800	28,072
教育費	0	14,400	0	52,800	52800	52800	172,800	400,800	228,000
購入消耗工具器具備品費	0	0	20,112	23,200	23200	23200	89,712	1,440,000	1,350,288
ソフトウェア購入費	2,667	2,667	2,667	100,800	100800	100800	310,401	424,800	114,398
交際費	0	0	0	0	0	0	0	0	0
事務用消耗品費	0	0	0	8,000	8000	8000	24,000	141,600	117,600
図書費	0	0	0	7,200	7200	7200	21,600	43,200	21,600

シミュレーション結果は別ウインドウで表示

Figure 3. Information Window of the System

type	files	total size (k lines)
SML# interface file	61	2.1
SML# source file	61	9.7
ClearSilver template	17	3.5
c source file	1	0.09
html files	10	5.4
css files	11	1.2
js files	6	2.4

The following table shows the summary of the production costs.

type	activities	man-hour
prototype	education & analysis	320
	design	320
	coding and test	1120
version 1	education & analysis	640
	design	320
	coding and test	1280

A new member has joined at the start of version 1 system. The relatively high education cost is inevitable for this project that adopts a new language and new method. Considering this and the fact that we have developed an entirely new system from scratch with problem analysis and prototype developing, we regard the development of the project efficient, and the overall development quite satisfactory.

As we shall report in Section 6, the developed system is currently under evaluation. Overall, the system shows expected behavior. Figures 3 shows a simulation window.

4. Evaluation of the Project and New Findings

As we shall discuss in detail in the next section, we have found several needs of improvements of an ML-style functional language, but the overall development has been quite satisfactory. We have confirmed that functional programming and SML# features have contributed to the productivity of the project, as we have expected. We have also found and developed some useful programming patterns for data intensive business applications. In this section, we discuss some of them below.

4.1 Abstract model construction that refines MVC pattern

We started the system design by adopting the popular MVC pattern, where the notion of models represent data with a set of procedures. In a simple web application such as a system to display a picture associated to some key, a model naturally corresponds to persistent data stored in a database system. The required procedures are retrieving, updating, and deleting the designated data element. In this simple situation, a database can be properly abstracted through object-relational mapping. However, in a typical business application such as the one we have developed, large and complex data are stored in a relational database as a highly normalized (in the sense of relational database theory [7]) set of flat relations. For example, the company database we had to deal with in this project (whose outline is shown in Figure 1) has 15 tables, each has 4 to 103 columns. The total number of columns is 395. To retrieve a data for each user request, we have to make complicated database joins among a subset of 15 tables, taking account of the database schema design, and to project the results on some particular columns. User responses are then computed from the retrieved column values. We do not think that a database consisting of 15 tables with 395 columns is exceptionally large. For such a large relational database with highly normalized schema, it is very difficult to make an object relational mapping that can deal with various user requests.

```
fun findEmployeeNameByCode employeeCode =
  let
    infix andAlso ==
    val r = SQL.fetchOne
      (_sqleval
        (let
           open SQL
         in
           _sql db =>
             select #e.NAME as employeeName
             from #db.M_PDBS as e
             where (#e.PCD == (toSQL employeeCode))
         end)
        (DBConnector.conn ()))
  in
    #employeeName r
  end
```

Figure 4. SQL integration used in abstract model construction

The seamless integration of SQL in an ML-style language achieved by SML# provides an ideal solution to this problem. In SML#, database queries (SQL commands) are polymorphically typed ordinary ML expressions denoting a list of records. It is then a routine matter to use higher-order functions and to construct a desired data structure on top of whatever complex record structures returned by a query expression. We conceptually regard this layer of computation, *abstract model construction*, which refines the usual MVC pattern, i.e. the conventional *model* M in MVC pattern is refined to two layers of *raw models* of databases and *abstract models* constructed by higher-order functions on top of raw models.

Figure 4 shows a fragment of a simple function that is used in abstract model construction, where we replace the actual Japanese column names with simple labels. We note that `_sql db => select ...` is an expression having a polymorphic type, and not an ad-hoc embedding. See [12] for the details. This property guarantees complete static typechecking through ML's polymorphic type inference. This feature completely relieves problematic runtime query failure due to schema mismatch. Considering the size of databases and complexity of SQL queries used, this property should otherwise be difficult to obtain. Seamless SQL integration in SML# enables the programmer to develop required abstract models through easy, declarative, and type safe programming. This is the most contributing factor in productivity of our development.

4.2 Data intensive programming with record polymorphism

Record polymorphism of SML# provides a powerful mechanism for modular programming in data intensive application development. As we mentioned, database query results contain labeled records with a large number of fields. For example, a frequently appearing typical type in our system contains 35 fields. Each module processes only some subset of the fields. To organize a system consisting of a number of such modules in a modular and type safe way, record polymorphism plays a central role. For example, a utility function that filters out low-ranked software projects has the following interface type

```
val excludeE : ['a#{rank:rank}. 'a list -> 'a list]
```

which represents the precise polymorphism and type constraint this utility function has.

In addition to writing various utility functions through record polymorphism, we have also found several *functional design patterns* that have general usefulness. Here we show a typical one below. In our system, we generate result HTML files through ClearSilver templates. This is done by filling each hole in a template with specific values. In order to organize this process in a modular and type safe way, we find the following pattern using polymorphic record update, which we present below. This pattern has not yet fully exploited in the current system; we plan to re-factor the system with this pattern in near future. Suppose we have two attributes `A` and `B` and that they are independently computed by two functions `fillA` and `fillB`. This is a simplified model of decomposing the template filling task into a set of independent modules. In an actual situation, `{A, B}` becomes a set of sets of attributes. This situation is cleanly represented by the following design pattern. We represent each field of type τ as τ `option`, where `NONE` indicates a non-filled hole and `SOME` v indicates the hole filled with v. The function `fillA` is then implemented as a function of the form

```
fun fillA (x as {A = NONE, ...}) =
    let val v = computing a value to be filled in A
    in x # {A = v}
    end
  | fillA (x as {A = SOME _, ...}) =
    raise AlreadyFilled
```

where `x # {A = v}` is polymorphic record update operation which creates a new record from `x` by modifying the field `A`. This function has the following polymorphic type.

```
fillA : ['a#{A: τ option}. 'a -> 'a]
```

This type precisely represents the fact that `fillA` operates only on field `A` and that it can compose with any function that operates on any other fields. `fillB` has a similar structure. A template package is defined as a module containing the following

```
val init = {A=NONE, B=NONE}
fun check (x as {A=SOME _, B= SOME _}) = x
  | check _ = raise ThereAreMissingField
fun fill F = check (F init)
```

where `init` represents the empty template, and `fill` invokes hole filling function `F` and dynamically ensures that all the fields are filled through function `check`. The controller to dispatch `fillA` and `fillB` can then be implemented as

```
fun dispatch AB () = fill (fillB o fillA)
```

by composing `fillA` and `fillB` functions.

This record pattern scales to set of subsets of fields and is generally helpful in compositional development of a component having multiple attributes/aspects.

```
structure FCGI =
struct
  exception IO of string
  val accept = _import "FCGI_Accept" : () -> int
  val finish = _import "FCGI_Finish" : () -> ()
  val setExitStatus =
     _import "FCGI_SetExitStatus" : int -> ()
  val FCGI_getchar =
     _import "FCGI_getchar" : () -> int
  fun getchar () =
      SOME (chr (FCGI_getchar ()))
      handle Chr => NONE
   ...
end
```

Figure 5. FCGI binding through SML# C FFI

4.3 Development with best-of-breed components

In a large system development in industry, selecting best-of-breed components from available choices is essential. The direct C interface of SML# has greatly contributed to this practice in the project. As we show in Section 3, we have used several external libraries such as FastCGI and ClearSilver. Figure 5 shows a fragment of FastCGI module written and used in the project. In this code, `FCGI_`$\cdots$ is a C function provided by the FCGI package. With this feature, we can directly import a library package almost instantly as far as the package provides a C API. SML#'s separate compilation system links the referenced packages. Through this mechanism, the programmer can select best suited packages from a rich collection of library packages with C API.

5. Insights Towards Better Record Programming

Manipulation of record structures is one of the central aspects in our ERP system. This should be true for many data-intensive business applications. In textbook examples, records are rather small and are typically used as parameters to a function or a datatype constructor. In business applications using databases, however, we frequently deal with records with a large number of fields. Experience with record programming in our project have given us various invaluable insights towards extending and improving record programming primitives in SML# and ML-style functional languages. In this section, we discuss some important ones.

5.1 Uniform operations on a set of fields.

A record often contains a set of related fields such as labor cost fields of all the months. We have often encountered cases of writing similar or same code for each such field. Such a program would become much more concise and maintainable if the language supports uniform operations such as map and fold for a set of record fields. Using first-class polymorphic field selectors, this can already be achieved in SML# to some extent. For example, summing up a given list F of fields in a record X can be coded as follows.

```
foldr (fn (f,S) =>(f X + S)) 0 F
```

For example, in SML#, we can write the following function

```
fn X => foldr (fn (f,S) =>(f X + S)) 0 [#jan, #feb]
  : ['a#{feb: int, jan: int}. 'a -> int]
```

that adds up `jan` and `feb` fields in a given record. This polymorphic idiom is itself useful for cases where fields are accumulated to a single value. However, we often encounter the cases where each field of one record is accumulated to the same or corresponding field in another record. To deal with this situation, it is desirable for the language to contain a mechanism to treat labels as first-class entities so that we can write, for example, the code of the form:

```
fn (X,Y) =>
  foldr (fn (L,Y) => Y # {L = #L X + #L Y})
        Y [<jan>, <feb>]
```

This pseudo code would increment each of `jan` and `feb` fields in `Y` record with the corresponding field in `X` record. `<jan>` and `<feb>` are hypothetical "label literals" which are bound to `L` and used as a label. This mechanism would significantly improve database programming. At this moment, we do not know whether or not such extension is possible. We regard this an interesting future work on ML-style languages with records.

It has been suggested to us that "`Fieldslib`" of Camlp4 [14] provides first class labels through type-directed code generation using the language module system. We note however that the functionality we want is the ability to abstract a polymorphic pattern that access multiple records and creates new records according to a given set of labels. In a language where record types must be declared and field access are monomorphic, even addressing the problem of flexible and modular manipulation of record structure is difficult. We also note that, as shown above, field access functions in SML# are already first-class polymorphic functions, on which we can map and fold.

5.2 Type-safe dynamic views on records.

Another insight we have gained through our database programming is the usefulness of customized "views" on a large flat record. In a relational database schema, each table contains a normalized flat record. For those relational schema, we often encountered cases where some fields are properties of an entity determined by a key stored in another field. In such case, we have to write codes that transform a flat record into a nested record. Although there is no difficulty, writing each such code is tedious and makes the entire program lengthy. Since the code is determined by the source record type and the target record type, language support to provide a view that automatically generates those code from types is highly desirable. We believe that previous results on functional type isomorphism [1, 5] can be used to integrate such a mechanism in SML# and other ML-like languages with records.

Another observation we have during database programming in this project is that such a view should also support uniform treatment of implicitly enforced *non-null* constraint. In a relational schema, each field can contain the so called "null value" (unless otherwise explicitly specified as *non-null*). In our SQL integration, a field with null value is represented as an option type. However, this modeling is over strict; in may cases non-null constraint is omitted and context information such as some value in some field determines that theses fields are restricted to be non-null values. In such case, a view mechanism that dynamically checks and converts τ `option` to τ would be quite useful. The programmer then invokes a view only once when he/she knows that some set of fields are non-null. This mechanism would free the programmer from tedious burden of checking the case of `NONE` and writing error handing code. Again we believe this mechanism can be incorporated based on existing research on type isomorphisms mentioned above.

5.3 Natural join in a programming language

The pattern for template filling we have shown in Section 4.2 works reasonably well but is not ideal. It uses `NONE` of an option type to represent non-filled hole. A better approach would be to join locally computed set of attributes together to obtain a complete set of attributes to be filled in a template. Let τ to be the set of attributes required for a template. Since τ usually consists of a large set of attribute fields, we would like to decompose it into a set $\{\tau_1, \ldots, \tau_n\}$, each of which is independently computed. In relational databases, this problem corresponds to decomposing a large relation τ into a set of relations $\{\tau_1, \ldots, \tau_n\}$. The desired relation is obtained by joining the set of component relations. In the relational model, this natural join operation is restricted to flat relations, but can be generalized to a term algebra containing records and sets [2, 9]. Using the notions developed in those works, we can regard this decomposition as an equation

$$\tau_1 \sqcup \cdots \sqcup \tau_n = \tau$$

where $A \sqcup B$ is the least upper bound of two structures A and B. In our setting, we can think of this as an operation that recursively joins nested records. For example.

$$\begin{aligned} & \{\texttt{id:int,proj:}\{\texttt{name:string}\}\} \\ \sqcup\ & \{\texttt{id:int,proj:}\{\texttt{rank:int}\}\} \\ =\ & \{\texttt{id:int, proj:}\{\texttt{name:string,rank:int}\}\} \end{aligned}$$

Using this notion, the template filling example can be better represented as follows.

We independently write functions to compute partial values

```
val fill₁ : unit -> τ₁
...
val fillₙ : unit -> τₙ
```

each of which generates a piece of information as a nested record τ_i. The template module can be

```
fun fill (F:τ) = ...
```

which simply takes all the necessary information and fills the template completely. Its type safety is straightforwardly specified by the type constraint of nested record type τ. Dispatching the set $\texttt{fill}_1$, ..., $\texttt{fill}_n$ of hole filling methods can be done

```
fill (Join (fill₁ (), ..., fillₙ ()))
```

where `Join` is a nested application of a polymorphic nested record joining function `_join` having the constrained type of the form:

```
_join : 'a * 'b -> 'c where 'c = 'a ⊔ 'b
```

This solution is much more robust and modular since the completeness and consistence of the information needed to fill all the template holes are represented and checked by the type system.

Although the current SML# does not support this operation, the necessary type theoretical machinery has been well established in [3, 10] and a language having this operation has been proposed [11]. A preliminary extension to the SML# type system has been done. In the extension, the following code can be written

```
val fn (x,y) => _join(x,y);
  : ['a#{}, 'b#{}, 'c#{}('a,'b). 'a * 'b -> 'c]
```

where `'c#{}('a,'b)` represents the least upper bound constraint. Currently, this extension is only for typing natural join of SQL commands, and does not support runtime execution. We hope that with further development of the implementation method, we can support user-level natural joins on nested records.

Another possible approach to this problem would be to provide those elaborate record operations through language independent query [4]. It remains to be investigated how language independent query can be uniformly integrated in a host language polymorphic type system.

6. Initial Evaluation of the Developed System

Our industry-academia project of developing the ERP system has been completed at the end of March 2014. The current status of the system and our plan of its deployment are the following.

1. After the completion, the system has been put into a trial use in a department of NEC Solution Innovetors, Ltd., where the development team belongs. The department consists of about 60 engineers.
2. The trial use have continued in the department for a few months. During this period, we have improved and polished the system.
3. After the improvement, the company plans to start using the system in the division consisting of 5 departments for thorough evaluation of the system's functionality and usability. This evaluation will be completed before September 2014.
4. After September 2014, the company plans to deploy the system in the entire Tohoku branch of NEC Solution Innovetors, Ltd.

This current plan is almost exact scenario we had drawn up at the start of the project in December 2013. At that time, the Tohoku branch of NEC Solution Innovetors, Ltd. was an independent company located in Tohoku, called NEC Software Tohoku, Ltd., which was merged into NEC Solution Innovetors, Ltd. in April, 2014.

The steps 1 and 2 of the above deployment process have progressed as we expected, and we have a positive perspective of completing the entire deployment process as planned. In particular, improvement of the system has been smoothly done. We have so far found several problems and shortcomings of the system. Some major ones include the following.

- Long system start-up time. The system start-up time was much longer than we expected. Our analysis showed that this delay was due to redundant database connection calls and redundant calls to a date computation function.
- Insufficient treatment of incomplete database states. One main goal of the system is to obtain real-time status of each project. To achieve this, the system uses the data stored in the company database, which is continuously updated. It is then inevitable that some data such as workloads of some project is partially filled, with some missing values. Such incomplete database states showed anomalous results.
- Insufficient problem analysis and design. At the time of designing the system, we analyzed the company's business model and constructed our internal data model by extending the existing one for real-time status reporting. During the evaluation, we found that certain financial data in the company are only meaning full at the end of each month, and due to this, the system behaved anomalously in its few functions.

We have successfully improved all the problems so far found. All the improvements have been done mostly by the company team. For long initialization time, we have reused an established database connection and have factored out date computation, resulting in significant reduction of the start-up time. For the incomplete database states, we have added codes to check the anomalous value, to warn the operator and to solicit the operator for missing values. We have also improved our internal model so that the system estimates those values that are only available at the end of each month. These improvements have been done quite smoothly and efficiently without much help of the academic team. This experience confirms our confidence in maintaining and extending functional codes written in SML#.

7. Conclusions and Future Plan

We have reported on our project of developing an enterprise resource planning (ERP) System using a functional programming language SML#. This project is our initial attempt to establish a programming environment for an ML-style higher-order typed functional language in wide range of ordinary software development in industry. The development project has successfully completed as we have planned, demonstrating the feasibility of SML# in software production in industry. Evaluation of the project has confirmed the usefulness of the seamless SQL integration and direct C interface of SML#. By defining an abstract model layers on top of database access through higher-order functions, the ERP system has been easily structured in a natural and type-safe manner. The project has also given invaluable insights on possible future extensions to an ML-style language for data intensive business application developments, including uniform operations on record fields, type-safe dynamic views on records, and introduction of natural join on nested records.

This is our initial attempt toward establishing a functional programming environment for our general software development in industry, and a number of challenging issues remain to be investigated before a functional language such as SML# can become a mainstream production language. Here we only mention one of them which is, we believe, the most important one. Throughout the project, which has been mainly carried out in NEC Solution Innovetors, Ltd., a major software company in Japan, we have frequently encountered a difficulty of analyzing a problem and making a basic program design for each programming task. This has been mainly due to the lack of common abstraction concepts in breaking down a problem into programming component in a functional language. To improve this situation, we have identified and shared several patterns during the project both in coerce and fine grain. The notion of abstract model construction on database access we reported in Section 4.1 and the polymorphic record update pattern in Section 4.2 are typical examples. Collecting such useful functional programming patterns and sharing them with industry will be of great help in disseminating functional languages in industry. As one of important future work of our academia-industry collaboration, we would like to collect such patterns, and ultimately to establish functional programming methodology that is readily usable in industry.

Acknowledgments

We thank the anonymous reviewers for helpful comments and suggestions for improving the presentation of the paper.

References

[1] G. Barthe. A computational view of implicit coercions in type theory. *Mathematical. Structures in Comp. Sci.*, 15(5):839–874, 2005.

[2] P. Buneman, A. Jung, and A. Ohori. Using powerdomains to generalize relational databases. *Theoretical Computer Science*, 91(1):23–56, 1991.

[3] P. Buneman and A. Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–74, 1996.

[4] J. Cheney. S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *Proc. ACM International Conference on Functional Programming*, 403–416, 2013.

[5] R. Di Cosmo. A short survey of isomorphisms of types. *Mathematical. Structures in Comp. Sci.*, 15(5):825–838, 2005.

[6] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[7] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[8] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[9] A. Ohori. Semantics of types for database objects. *Theoretical Computer Science*, 76:53–91, 1990.

[10] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proc. ACM Conference on LISP and Functional Programming*, pages 174–183, 1988.

[11] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proc. the ACM SIGMOD conference*, pages 46–57, 1989.

[12] A. Ohori and K. Ueno. Making Standard ML a practical database programming language. In *Proc. ACM ICFP*, pages 307–319, 2011.

[13] SML#. http://www.riec.tohoku.ac.jp/smlsharp/, 2006 – 2014.

[14] Fieldslib. https://github.com/janestreet/fieldslib