SML# with Natural Join

Tomohiro Sasaki Katsuhiro Ueno Atsushi Ohori
Tohoku University
{tsasaki,katsu,ohori}@riec.tohoku.ac.jp

Abstract

This paper reports on the extension of SML# with *natural join* operator commonly used in database query. The extension is based on the database typing of Ohori and Buneman and an HM(X)-style constraint polymorphic typing. Based on this typing and type inference algorithm, the seamless SQL integration of SML# is extended with natural join. The extended SML# is available as a version 3.2.0.

1. Introduction

Natural join is a basic operation in database programming to combine two partial descriptions. In the relational data model, this is introduced as an operation ($\bowtie$) to combine two tables sharing some column names, as seen in the following example:

id	name
1	Amy
7	Dr. C
9	Togetoge

$\bowtie$

id	salary
5	50
7	100
9	300

$=$

id	name	salary
7	Dr.C	100
9	Togetoge	300

where the result relation is more descriptive. As observed in [1], this operation can be generalized to complex database objects constructed from atomic data, labeled records, and set (list) by interpreting it as an operation to compute the least upper bound $x \sqcup y$ of two partial descriptions x, y ordered by information content.

The general motivation of this research is to integrate this operation in ML. This integration should open up a new possibility of ML in data-intensive application as investigated in [2], including recently emerging web and cloud applications. By combining the techniques of seamless SQL integration [8] in ML, this extension also enables us to support natural joins in the integrated SQL. In this paper, we develop an ML-style type system and type inference algorithm for natural join, implement them in SML#, and extend the SML# SQL integration to support natural join.

Our development of a type system and a type inference algorithm is based on a type inference system for record and database operations presented in [7]. In this system, natural join is given with the following typing:

$$\{t_3 = t_1 \sqcup t_2\}, \Gamma \rhd \lambda x.\lambda y.\texttt{\_join}(x, y) \ : \ t_1 \to t_2 \to t_3$$

where $t_3 = t_1 \sqcup t_2$ is a syntactic constraint that restricts possible instances τ_1, τ_2, τ_3 of t_1, t_2, t_3 to satisfy the predicate $\tau_3 = \tau_1 \sqcup \tau_2$ denoting the fact that the type-level join of τ_1 and τ_2 is equal to τ_3. The type-level join $\tau_1 \sqcup \tau_2$ is defined as the least upper bound operation of the following ordering on the set of database types:

$$\begin{aligned} b &\sqsubseteq b \text{ for any atomic type } b \\ \{l_1 : \tau_1, \ldots, l_n : \tau_n\} &\sqsubseteq \{l_1 : \tau'_1, \ldots, l_n : \tau'_n, \ldots\} \text{ if } \tau_i \sqsubseteq \tau'_i \\ \tau \ set &\sqsubseteq \tau' \ set \text{ if } \tau \sqsubseteq \tau' \end{aligned}$$

Here we restrict type-level join to "flat record types" ("relations in the first normal form") by omitting the third case and restricting τ'_i to be equal to τ_i in the second case. To introduce this form of constraint typing in an ML-style polymorphic type system, we follow the HM(X) approach of type inference with constrained types [6] and introduce polymorphic type of the form $\forall \bar{t}.C \Rightarrow \tau$ so that the join term above has the following polymorphic type:

$$\forall t_1, t_2, t_3.\{t_3 = t_1 \sqcup t_2\} \Rightarrow t_1 \to t_2 \to t_3$$

Different from the HM(X) approach, however, we do not require an abstracted constraint set C to be satisfiable. As show in [7], satisfiability checking of a set of join constraints involving free type variables is an NP-hard problem. The solution proposed in [7] is to delay the satisfiability checking until type variables are instantiated to ground types. We adopt this strategy, and allow possibly unsatisfiable C in $\forall \bar{t}.C \Rightarrow \tau$. By this extension, the following polymorphic type is inferred:

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

which shows an actual interactive session in SML#. With this extension, the SQL integration of SML# presented in [8] can easily be extended adding the case for natural join in the toy-term generation algorithm. In the extended system, the following SQL expression can be directly written, type-checked, and evaluated:

```
# _sql db =>
 select #t.name, #t.salary
 from (#db.Employees natural join #db.Salary) as t;
val it = fn
: ['a#{Employees: 'b, Salary: 'c},
   'b#{}, 'c#{}, 'd, 'e, 'f, 'g,
   'h#{name: 'd, salary: 'f}.
     ('h = 'b join 'c) => 'a db -> ('d * 'f) query]
# SQL.fetchAll (_sqleval it connection);
val it = [("Dr.C", 100), ("Togetoge", 300)]
  : (string * int) list
```

Before presenting the technical development, we briefly mention some closely related works. Our type system for join is based on [7] for record and database operations. After this, a number of related type system for records have been presented. In particular, [10] presented an encoding method of record concatenation, and [9] presents a constraint-based type inference system that can represent various features of row variables. One of our aim is to achieve a full and complete integration of SQL in ML-style polymorphic language by integrating the typing technique of database join [7]. In a general perspective of integrating database query facility in a general purpose programming language, the approaches of LINQ [5], Ferry [4] and Links [3] all share motivations similar to ours. In the complete article version, we shall provide detailed comparisons and discussions to those and other related works.

In the rest of this abstract, we outline the type system and the type inference algorithm with soundness theorem, and describe our implementation.

2. Type system and type inference

We consider the following expressions (ranged over by e), the sets of monotypes (ranged over by τ) and ML-style polytypes (σ):

$$\begin{array}{rcl} e & ::= & c^b \mid x \mid \lambda x.e \mid e\,e \mid \{l = e, \ldots, l = e\} \\ & & \mid \texttt{\_join}(e, e) \mid \texttt{let}\ x = e\ \texttt{in}\ e \\ \tau & ::= & b \mid t \mid \tau \to \tau \mid \{l : \tau, \ldots, l : \tau\} \\ \sigma & ::= & \forall \overline{t}.C \Rightarrow \tau \end{array}$$

$\texttt{\_join}(e_1, e_2)$ denotes the natural join of two records e_1 and e_2. Let C be a finite set of constraints of the form $\tau_3 = \tau_1 \sqcup \tau_2$ mentioned in Section 1. This set constrains the possible set of type substitutions.

Let Γ be a finite map from variables to polytypes. The type system is given by the set of rules deriving a judgment of the form

$$C, \Gamma \vdash e : \tau$$

. We only show a few rules that deal with constraints as follows:

$$\frac{C, \Gamma \vdash e_1 : \tau_1 \quad C, \Gamma \vdash e_2 : \tau_2 \quad (\tau = \tau_1 \sqcup \tau_2) \in C}{C, \Gamma \vdash \texttt{\_join}(e_1, e_2) : \tau}$$

$$\frac{C_1 \cup C_2, \Gamma \vdash e_1 : \tau_1 \quad C_1, \Gamma\{x : \forall \overline{t}.C_2 \Rightarrow \tau\} \vdash e_2 : \tau \quad (FTV(\Gamma) \cup FTV(C_1)) \cap \overline{t} = \emptyset, FTV(C_2) \subseteq \overline{t}}{C_1, \Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}$$

$$\frac{[\overline{\tau}/\overline{t}]C' \in C \quad \tau' = [\overline{\tau}/\overline{t}]\tau \quad \text{for some } \overline{\tau}}{C, \Gamma\{x : \forall \overline{t}.C' \Rightarrow \tau\} \vdash x : \tau'}$$

Constraints are introduced with the first rule for natural join. The second and the third rule are for constrained type-abstraction and type-instantiation. As mentioned in Section 1, we allow C to be unsatisfiable and delay satisfiability checking until type variables are instantiated to ground types. As shown in [7], this strategy yields sound typing.

The `let` rule is a special case of HM(X) rule for let-abstraction where C_2 does not contain any type variables that are free in C_1 and Γ. Owing to this restriction, our type system smoothly interacts with our strategy of delaying the constraint satisfiability checking. However, note that due to this restriction, it may restrict programs to less polymorphic; for example, f of the following example

$$\lambda x.\texttt{let}\ f = \lambda y.\texttt{\_join}(x, y)\ \texttt{in}\ e$$

does not have a polytype because the type of y connects to that of x through the constraint of $\texttt{\_join}(x, y)$, whereas if `_join` could be replaced with a record constructor, at least y would be generalized.

For this calculus, we define a type inference algorithm $\mathcal{WC}$ as an extension to the one defined in [7]. We define $\mathrm{Cls}(C, \Gamma, \tau)$ to be the function that returns a pair $(C_1, \forall \overline{t}.C_2 \Rightarrow \tau)$ such that $\overline{t}$ is a largest set satisfying the following: $(FTV(\Gamma) \cup FTV(C_1)) \cap \overline{t} = \emptyset$, $C = C_1 \cup C_2$ (C_1, C_2 disjoint), and C_2 is a set of constraints each of which involves some $\overline{t}$ and does not contain any $FTV(\Gamma) \cup FTV(C_1)$. We omit its detailed definition. We also write $S(C)$ for the operation to applying S to C with the following additional satisfiability check: it fails if the resulting constraint set contains a unsatisfiable ground constraint. The cases of $\mathcal{WC}$ for `let` and variables are given below. The other cases including the one for $\texttt{\_join}(e_1, e_2)$ are essentially the same as those of [7].

$$\begin{array}{l} \mathcal{WC}(C, \Gamma, x) = \text{let } (\forall \overline{t}.C' \Rightarrow \tau) = \Gamma(x) \\ \qquad\qquad S = [\overline{t'}/\overline{t}] \qquad (\overline{t'} \text{ fresh}) \\ \qquad\qquad \text{in } (C \cup S(C'), \emptyset, S(\tau)) \end{array}$$

$$\begin{array}{l} \mathcal{WC}(C, \Gamma, \texttt{let}\ x = e_1\ \texttt{in}\ e_2) = \\ \quad \text{let } (C_1, S_1, \tau_1) = \mathcal{WC}(C, \Gamma, e_1) \\ \qquad (C_1', \sigma) = \mathrm{Cls}(C_1, S_1(\Gamma), \tau_1) \\ \qquad (C_2, S_2, \tau_2) = \mathcal{WC}(C_1', S_1(\Gamma)\{x : \sigma\}, e_2) \\ \quad \text{in } (C_2, S_2 \circ S_1, \tau_2) \end{array}$$

$\mathcal{WC}$ returns *Failure* if either $\Gamma(x)$ does not exist, a wrong or unsatisfiable ground constraint is added to C, or a recursive call of $\mathcal{WC}$ returns *Failure*.

THEOREM 1 (Soundness of $\mathcal{WC}$). *If $\mathcal{WC}(C, \Gamma, e) = (C', S', \tau)$, then $S(C) \subseteq C'$ and $C', S(\Gamma) \vdash e : \tau$.*

The proof is standard using the following property: If $C, \Gamma \vdash e : \tau$ then $S(C \cup C'), S(\Gamma) \vdash e : S(\tau)$ for any S and C'.

3. Implementation

We have implemented the extension reported in this paper in the SML# compiler [11] version 3.2.0, which we plan to release on September 16th.

As seen in Section 1, we add $\texttt{\_join(}e_1\texttt{, }e_2\texttt{)}$ expression and `natural join` SQL syntax. For the former, we implement its type-check only. `_join` always raises an exception if it evaluates. There is a subtle technical issue to evaluate polymorphic natural join; it requires the record label sets and layout information at runtime, whereas they are static attributes. Establishing an evaluation model of polymorphic join is left for future work. In contrast, the latter works fine with relational database engines. Following the strategy presented in [8], we translate `natural join` into a toy-term that are never evaluated. The `_join` appears in the toy-term to achieve the polymorphic typing of `natural join`. Actual evaluation of `natural join` is done on a SQL database server. Examples of interactive executions is shown in Section 1.

References

[1] P. Buneman, A. Jung, and A. Ohori. Using powerdomains to generalize relational databases. *Theor. Comput. Sci.*, 91(1):23–56, 1991.

[2] P. Buneman and A. Ohori. Polymorphism and type inference in database programming. *ACM T. Database Syst.*, 21(1):30–74, 1996.

[3] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: web programming without tiers. In *Proc. International Conference on Formal Methods for Components and Objects*, pages 266–296, 2007.

[4] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. FERRY: database-supported program execution. In *Proc. ACM SIGMOD Conference*, pages 1063–1066, 2009.

[5] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *Proc. ACM SIGMOD Conference*, pages 706–706, 2006.

[6] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1), 1999.

[7] 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.

[8] A Ohori and K. Ueno. Making Stanard ML a practical database programming language. In *Proc. ACM International Conference on Functional Programming*, pages 307–319, 2011.

[9] F. Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4), 312–347, 2000.

[10] D. Remy. Typing record concatenation for free. In *Proc. ACM Symposium on Principles of Programming Languages*, 1992.

[11] SML# Project. `http://www.pllab.riec.tohoku.ac.jp/smlsharp/`