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 (⋈) to combine two tables sharing some column names, as seen in the following example:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Amy</td>
<td>50</td>
</tr>
<tr>
<td>7</td>
<td>Dr. C</td>
<td>100</td>
</tr>
<tr>
<td>9</td>
<td>Togetoge</td>
<td>300</td>
</tr>
</tbody>
</table>

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 \( \sqcup \) 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:

\[
\{ t_3 = t_1 \sqcup t_2 \}, \Gamma \vdash \lambda x.\lambda y.\_\text{join}(x, y) : t_1 \rightarrow t_2 \rightarrow t_3
\]

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

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

Here we restrict type-level join to “flat record types” ("relations in the first normal form") by omitting the third case and restricting \( \tau' \) to be equal to \( \tau \) 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 \( \forall \tau. C \Rightarrow \tau \) so that the join term above has the following polymorphic type:

\[
\forall t_1, t_2, t_3. \{ t_3 = t_1 \sqcup t_2 \} \Rightarrow t_1 \rightarrow t_2 \rightarrow t_3
\]

Different from the HM(X) approach, however, we do not require an abstracted constraint set \( C \) to be satisfiable. As shown 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 \( \forall \tau. C \Rightarrow \tau \). By this extension, the following polymorphic type is inferred:

\[
\# \text{fn} (x, y) \Rightarrow \_\text{join}(x, y);
\text{val it = fn} : \{ \text{\'a\#{}{}, \'b\#{}{}, \'c\#{}} \}\,
\text{('c = 'a join 'b) => 'a * 'b -> 'c}
\]

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:

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

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.
2. Type system and type inference

We consider the following expressions (ranked over by $e$), the sets of monotypes (ranked over by $\tau$) and ML-style polytypes ($\sigma$):

$e ::= e^b \mid x \mid \lambda x.e \mid e.e \mid \{ \ldots \}$

$\mathsf{let} (e, e) = e$ in $e$ in $e$

$\tau ::= b \mid t \mid \tau \Rightarrow \tau \mid \{ \ldots \}$

$\sigma ::= \forall \tau. C \Rightarrow \tau$

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

Let $\Gamma$ 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:

$\frac{C, \Gamma \vdash e_1 : \tau_1; C, \Gamma \vdash e_2 : \tau_2}{C, \Gamma \vdash \mathsf{join}(e_1, e_2) : \tau}$

$\frac{C_1 \cup C_2, \Gamma \vdash e_1 : \tau_1; C_1, \Gamma \vdash \{ x : \forall \tau. C_2 \Rightarrow \tau \} \vdash e_2 : \tau}{(\mathsf{FTV}(\Gamma) \cup \mathsf{FTV}(C_1)) \cap \tau = \emptyset, \mathsf{FTV}(C_2) \subseteq \tau} \quad \frac{C_1, \Gamma \vdash \mathsf{let} \ x = e_1 \ in \ e_2 : \tau}{\{ \tau/\tau \}' C \subset C, \ {\tau/\tau}' \Rightarrow \tau \vdash e : \tau'}$

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 $\mathsf{let}$ 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 $\Gamma$. 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

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

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

For this calculus, we define a type inference algorithm $\mathsf{WC}$ as an extension to the one defined in [7]. We define $\mathsf{Cls}(C, \Gamma, \tau)$ to be the function that returns a pair $(C_1, \forall \tau. C_2 \Rightarrow \tau)$ such that $\tau$ is a largest set satisfying the following: $(\mathsf{FTV}(\Gamma) \cup \mathsf{FTV}(C_1)) \cap \tau = \emptyset, C = C_1 \cup C_2 (C_1, C_2$ disjoint), and $C_2$ is a set of constraints each of which involves some $\tau$ and does not contain any $\mathsf{FTV}(\Gamma) \cup \mathsf{FTV}(C_1)$. 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 an unsatisfiable ground constraint. The cases of $\mathsf{WC}$ for $\mathsf{let}$ and variables are given in Table 1. The other cases including the one for $\mathsf{join}(e_1, e_2)$ are essentially the same as those of [7].

$\mathsf{WC}(C, \Gamma, x) = \{ \forall \tau, C' \Rightarrow \tau \} = \Gamma(x)

S = \{ \tau/\tau \}\quad (\tau \text{ fresh})$

in $(C \cup S(C'), \emptyset, S(\tau))$

$\mathsf{WC}(C, \Gamma, \mathsf{let} \ x = e_1 \ in \ e_2) = \{ (C_1, S_1, \tau_1) = \mathsf{WC}(C, \Gamma, e_1) \}

(C_2, \sigma) = \mathsf{Cls}(C_1, S_1, \tau_1),

(C_2, S_2, \tau_2) = \mathsf{WC}(C_2, S_1(\{ x : \sigma, e_2 \}), in (C_2, S_2 \circ S_1, \tau_2))$

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

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

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

3. Implementation

We have implemented the extension reported in this paper in the SML# compiler [1] version 3.2.0, which we plan to release on September 16th.

As seen in Section 1, we add $\mathsf{join}(e_1, e_2)$ expression and natural join to SQL syntax. For the former, we implement its type-check only. $\mathsf{join}$ 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 $\mathsf{join}$ 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


http://www.pllab.riec.tohoku.ac.jp/smlsharp/