An Introduction to Remy's Fast Polymorphic Record Projection

Limsoon Wong

Real World Computing Partnership Novel Function
Institute of Systems Science Laboratory
Mui Keng Terrace, Singapore 0511.
Email: limsoon@iss.nus.sg

Abstract

Traditionally, a record projection is compiled when all fields of the record are known in advance. The need to know all fields in advance leads to very clumsy programs, especially for querying external data sources. In a paper that had not been widely circulated in the database community, Remy presented in programming language context a constant-time implementation of the record projection operation that does not have such a requirement. This paper introduces his technique and suggests an improvement to his technique in the context of database queries.

1 Introduction

In a prototype we have recently constructed for querying over heterogenous genome data sources [9], there is a need to compile queries in which many information used by traditional systems are missing. Consider the innocent-looking query below for example.

```
\DB1 => \DB2 =>
{(#name:n, #age:a, #sex:s) |
  (#name:\n, #age:\a, ...) <- DB1,
  (#name: n, #sex:\s, ...) <- DB2}
```

This query, in comprehension syntax [3], is a function that joins DB1 and DB2 on the #name field. However, there are several crucial differences between it and what is encountered in a relational system:

- The schema for DB1 is not given. We know that it is a set of records and that each record has a #name and a #age field. We do not know what other fields are in the record and we do not know what are the type of values that can be found in these two fields.
- The schema for DB2 is not given. We know that it is a set of records and that each record has a #name and a #sex field. We do not know what other fields are in the record and we do not know what are the type of values that can be found in these two fields.
- This function may be applied to different pairs of relations, not all having the same schema. That is, it can be applied first to some DB_1' and DB_2' and then be applied a second time to some DB_1'' and DB_2'' that do not have the same schemas as DB_1' and DB_2'.

Such an example arise from our prototype because the prototype has to work on many genetic 'databases.' These sources are highly heterogenous. They range from traditional relational databases to non-traditional complex structured files to data generated by specialized software packages. While some of these sources have schemas that are accessible, many lack them. Even for those which provide their schemas, these schemas are not described using a single unified format. In addition, these data sources are regularly evolving and hence their schemas are changing regularly too. Lastly, practically all these data sources are remote.

Due to the above characteristics, it is not possible to compile our queries using traditional techniques, which requires precise knowledge of types to calculate field offsets at compile time. Let us first consider some simple techniques that do work. We can store a record as a list of field-name-field-value pairs. Then field selection or record projection can be done by simply traversing the list to look for the pair having the right field name. This technique obvious requires a linear number of operations and is thus too inefficient. It can be improved by storing a record as an array of field-name-field-value pairs sorted on field name. Then record projection can be done by binary-searching the array. This has logarithmic cost but is still inefficient.

What is desired is a technique that (1) has constant time efficiency and (2) has low space overhead. In this paper, we introduce in Section 2 a technique due to Remy that posseses these two properties. In his original paper [16], which is not widely circulated in the database community, he presented his technique and concentrates on showing that it has low space overhead. In this paper, we compare in Section 3 its time performance with other techniques and we suggest in Section 4 an improvement in the context of querying homogenous collections.

2 Remy's Technique

The technique of Remy for implementing fast poylmorphic record projection can be found in the paper [16]. As this paper had not been widely circulated, we describe his technique in detail below.

- We take advantage of the fact that when a record object is created, we know how many fields it has and what are the labels naming these fields.
- First assign to each label that appears in a program a distinct number. Let $\mathrm{A}(l)$ be the number assigned to label l.
- We do not have to know what is the type of each field of a record. Hence, we use the notation $[l_1, ..., l_n]$ to stand for the *abstract* type of a record which has exactly the fields named by l_1, ..., l_n.
- For each abstract type $[l_1, ..., l_n]$, we create a directory $\mathrm{D}_{[l_1,...,l_n]}$ as described below.

  The directory is an $m + 1$ cell array of numbers. Here m is a 'magic' number such that $\mathrm{A}(l_1) \bmod m$, ..., $\mathrm{A}(l_n) \bmod m$ are distinct integers.

  Let cell 0 contains the number m. Let cell $1 + \mathrm{A}(l_i) \bmod m$ contains the number i, for $1 \leq i \leq n$. The contents of the remaining cells are not important. That is $\mathrm{D}_{[l_1,...,l_n]}[1 + \mathrm{A}(l_i) \bmod m] = i$, if $1 \leq i \leq n$.
- Each record object of abstract type $[l_1, ..., l_n]$ is stored as an array of $n+1$ pointers. Let O be a typical record object whose abstract type is $[l_1, ..., l_n]$. We describe its contents below.

  In cell O[0], we store the pointer to the directory $\mathrm{D}_{[l_1,...,l_n]}$. In cells O[1], ...O[$n$], we store pointers to values of fields l_1, ..., l_n of the record object respectively. (Actually, we might be able to store the values there directly.)
- To compile record selection, say $r.l$, the following steps can be used. Let O be the record object that r has been compiled into. Then *deref*(O[*deref*(O[0])[1 + A(l) mod *deref*(O[0])[0]]]) fetches out the value of field l of r.

This method is quite efficient in terms of both space and speed. In terms of speed, a record projection involves 4 array fetches, 2 pointer dereferencing, and 1 mod computation. This is the *constant* cost for access to any field of any record. In terms of space, there is 1 directory for each *abstract* type and one extra array slot for each record. A few cells get wasted if the corresponding m is not chosen wisely. But the wastage is minimal (unlikely to be more than the number of labels in the abstract type). This wastage is mitigated by the fact that different real records share the same directory.

The method can be made more efficient by picking m to be $2^n - 1$. Then the mod computation can be replaced by faster bitwise intersection. Such a choice of m increases directory wastage; however, this wastage is mitigated by the fact that directories are shared. It is also interesting to observe that, with this modification, the Remy directory scheme becomes specialized to a kind of extendible hashing scheme used in accessing external data [7].

The basic implementation of the technique above needs to be adjusted slightly when querying external databases. Under this situation, it is often not possible to know in advance the labels or field names that will be found in incoming data. Hence label numbers, abstract types, and directories have to be created on-the-fly as external data streams are parsed. However, it is often possible to make special arrangements so that efficiency is improved.

For example, instead of having the an external heterogenous data stream being trasmitted as

```
{ (#name:"john",#age:10),
  (#name:"bill",#age:20),
  (#name:"mary",#age:15,#note:"naughty"),
...}
```

we could arrange to have it transmitted as

```
$d1 (#name,#age)
$d2 (#name,#age,#note)
{ ($d1, "john", 10),
  ($d1, "bill", 20),
  ($d2, "mary", 15, "naughty"),
...}
```

So that when `$d1 (#name,#age)` is parsed, we create the Remy directory $D_{[name,age]}$ dynamically, if it does not already exist, and associate it with `$d1`. Then when we come to parsing `($d1, "john", 10)`, we can create the record directly using the previously created directory.

3 Comparisons

In this section we compare three alternative implementations of a simple query requiring 750000 projection operations. The first being the implementation of Remy described in the previous section. The second is an implementation based on binary search, which illustrates the efficiency of Remy's technique over typical implementations of polymorphic record projection. The third is an implementation using ordinary record projection, which is monomorphic and illustrates that Remy's technique is only eight times slower. In the next section, we suggest an improvement to Remy's technique that roughly triples its efficiency.

The query to be implemented is:

```
{ if x.#1 = y.#1 then x.#0 else y.#0 |
      \x <- DB1, \y <- DB2 }
```

For each `x` in `DB1` and each `y` in `DB2`, if the `#1` field of `x` is equal to the `#1` field of `y`, this query includes the `#0` field of `x` in its output; it includes the `#0` field of `y` otherwise. Note that #*l* is the name of a field; it is not the position of a field.

In every run, the cardinality of `DB1` and `DB2` is fixed at 500 records each. we use a nested loop in each of our implementation and thus giving 750000 projection operations in each run. We repeat the runs by varying the number of fields from 2 to 40. The Remy and the binary-search implementations are reused without change when the number of fields is changed; this illustrates the flexibility of polymorphic record projection. The monomorphic implementation has to be changed each time we change the number of fields; this highlights the clumsiness of monomorphic projection.

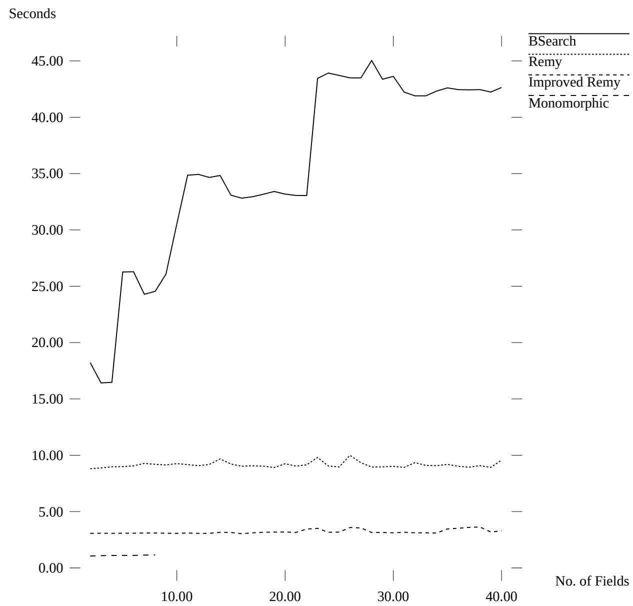

Figure 1: Performance Comparisons of Record Projection Techniques

The experiments are performed on a Sparc 20 machine. The results are shown in Figure 1. The topmost curve is the binary-search version, the next curve is the Remy implementation, the bottom curve is the monomorphic version. The third curve is the improved Remy implementation to be described in the next section.

Both the monomorphic implementation and the Remy implementation have constant-time performance with respect to the number of fields. On other hand, the binary-search implementation becomes progressively worse, in a logarithmic manner, as the number of fields increases. This shows that the technique of Remy is a clear improvement over traditional polymorphic projection implementations.

Remy's implementation takes about 9 seconds to complete each run. The monomorphic implementation takes about 1.1 seconds. So Remy's method is eight times slower. However, we used the *same* program in Remy's case and 7 different programs in the monomorphic case. We would have to use 39 different programs if we did not stop the experiment for the monomorphic implementation at eight fields. Therefore, Remy's technique offers a great deal of flexibility at a rather attractive cost.

In terms of space usage, Remy's technique is very good as well. Its records are four bytes (the size of one integer, the magic number) bigger than the monomorphic implementation. In addition, it has an overhead of one directory entry (shared by 1000 records) in each run. Thus Remy's technique has very small space overhead. In contrast, the binary-search implementation has a space overhead for each record that is linearly proportional to the number of fields in the record because a record is represented as an array of field-name-field-value pairs that is sorted on field name. Thus the binary-search method (and other typical polymorphic projection implementations) is extremely space inefficient.

4 An Improvement

In this section, we discuss a technique for further improving the performance of Remy's technique in the context of programming with homogenous collection types.

If the set of records is homogenous, the offset of any field in one record must be the same as all other records in the same set. The Remy projection operation can be divided into an offset computation, which is the same for all these records, and the actual retrieval of field value based on the offset. So the basic idea here is to move the offset computation outside the loop, whenever we are iterating over homogenous collections.

We illustrate this idea with the test query from the previous section. The Remy implementation of the test query is as follow:

```
fun TestRemy DB1 DB2 =
  map (fn X => map (fn Y =>
    if Record.Proj 1 X = Record.Proj 1 Y
    then Record.Proj 0 X
    else Record.Proj 0 y) DB2) DB1
```

It is in the syntax of ML [12], where sets are simulated using lists and field names are represented using integers. The operation `Record.Proj` l E is the projection of field l of record E, where E is a representation of a record using Remy's method. The syntax `fn` X `=>` E is ML's way of defining a function that takes input X and produces output E. The syntax `map` F E is ML's way of applying a function F to each element in the list E.

`TestRemy` works correctly regardless of what `DB1` and `DB2` are, so long as both of them are lists of records having a field whose name is 0 and a field whose name is 1. In particular, it works even when `DB1` and `DB2` are not homogenous. For example, it works even when some records in `DB1` have 3 fields and some have 4 fields. This makes Remy's technique very general because it can be used to implement polymorphic record projection for systems based on parametric polymorphism [13, 17, 2, 15, 16, etc.] and for systems based on subtype polymorphism [4, 5, 6, etc.].

However, not every system needs this kind of generality in record projection. In particular, relational databases have homogenous sets. It is possible to take advantage of homogeneity to speed up record projection using Remy's technique.

To do so, we recall that Remy record projection consists of two steps. The first step is the computation of an offset based on field name and the magic number associated with a Remy directory. The second step uses the offset to indexed into a Remy record to retrieve the value of the required field. That is the `Record.Proj` l E operation can be broken into an offset computation `Record.Offset` l E and an indexing operation `Record.Jump` n E, where n is the offset returned by the `Record.Offset` operation.

If the set we are mapping over is homogenous, then all its records share the same Remy directory. Therefore, we can apply the idea of code motion [1]. We compute the offset only for the first record. This offset can be reused for the remaining records.

To implement this idea, let us first implement a memoization function [11].

```
fun Memoize () =
  let
    val Cell = ref (Record.Proj)
    val _ = Cell :=(fn L => fn R =>
     let
       val Offset = Record.Offset L R
       val V = Record.Jump Offset R
     in Cell:=(fn L =>Record.Jump Offset);
       V
     end)
  in Cell
  end
```

Calling `Memoize()` gives us a pointer to a function f L E. The first time f is called with field name l and record r, it executes `Record.Jump (Record.Offset` l r`)` r. The next time f is called with field name l' and record r', it executes `Record.Jump` n r', where n is the value of `Record.Offset` l r computed during the first call. Note that l' is ignored. Hence the first time the full cost of record projection is incurred, but all subsequent calls incur less than half that cost.

Then the improved implementation is the program below, where $!E$ is ML's syntax for dereferencing the pointer E.

```
fun TestRemy' DB1 DB2 =
  let
    val (cx0,cx1) =(Memoize(),Memoize())
    val (cy0,cy1) =(Memoize(),Memoize())
  in map (fn X => map (fn Y =>
       if (!cx1 1 X) = (!cy1 1 Y)
       then (!cx0 0 X)
       else (!cy0 0 Y)) DB2) DB1
  end
```

The performance of the improved program is also given in Figure 1. It runs at an average of 3 seconds. This is a three-fold improvement over the original implementation.

Note that it is also possible to turn the binary-search implementation into a 'amortized constant-time' operation by moving offset computation outside the loop. However, its large space overhead remains. So Remy's technique is still better.

We must be very careful in the use of the improvement technique described above. For example, it is not sound to replaced `TestRemy'` by the program `Foo` below.

```
val (cx0,cx1) =(Memoize(),Memoize())
val (cy0,cy1) =(Memoize(),Memoize())

fun Foo DB1 DB2 =
  map (fn X => map (fn Y =>
    if (!cx1 1 X) = (!cy1 1 Y)
    then (!cx0 0 X)
    else (!cy0 0 Y)) DB2) DB1
```

This program creates the cells `cx0`, `cx1`, `cy0`, and `cy1` statically and uses them in all invocations of `Foo`. Then the first call to `Foo DB1 DB2` will work correctly. However, if we try to call `Foo DB1' DB2'` later, where `DB1'` and `DB2'` are different in types from `DB1` and `DB2`, this call will not be computed correctly. The reason is that records in `DB1'` and `DB2'` should use Remy directories different from those used by `DB1` and `DB2`. However, the memo cells are write-once only and so the second call will use the incorrect Remy directories. (Our original program `TestRemy'` will work all the time because it creates new memo cells each time it is called.)

The technique of Remy has been used in our prototype system [9] for querying heterogenous data sources. A newer version of our system is also able to automatically perform the optimization suggested above.

5 Conclusion

We have given a detailed description of Remy's implementation of polymorphic record projection. We have shown that it is attractive in terms of time as well as space. We have suggested an improvement that produces a three-fold speed-up in queries on homogenous bulk data types.

We summarize some of the advantages of Remy's technique below.

- It is efficient both in space and in time.
- It is very flexible and can be used in languages based on parametric polymorphism or subtyping.
- It works even when both homogenous and heterogenous collections are present.
- It offers compositionality. We can design the implementation of set independently from the implementation of records. This orthogonality allows the same programs to work on set of integers, set of records, and more usefully, set of variants. A variant [8] is a data type whose values can be one of several distinct types/structures; it is sometimes called a tagged union in data modeling literature [10].
- It is very straightforward to implement.
- Although we have not discussed hash tables, the idea behind Remy's technique can be used in adaptive hash tables. In fact, it is closely related to the extendible hashing technique [7].

To close this paper, we want to mention another technique, due to Ohori [14], for implementing fast polymorphic record projection. It can achieve a slightly smaller contant-time performance than Remy's. However, its implementation is more complex and is closely dependent on the underlying type system. In particular, it does not generalize to subtyping.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[2] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Germany, October, 1992*, pages 140–154. Springer-Verlag, October 1992.

[3] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[4] L. Cardelli. Amber. In *LNCS 242: Combinators and Functional Programming*, pages 21–47. Springer-Verlag, 1986.

[5] L. Cardelli. A semantics for multiple inheritance. *Information and Computation*, 76(2):138–164, 1988.

[6] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The Modula-3 type system. In *Proceedings 16th Annual ACM Symposium on Principles of Programming Languages*, pages 202–212, Austin, Texas, January 1989.

[7] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

[8] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, 1992.

[9] K. Hart, L. Wong, C. Overton, and P. Buneman. Using a query language to integrate biological data, August 1994. Talk given at the Meeting on the Interconnection of Molecular Biology Databases, Stanford.

[10] R. Hull and C. K. Yap. The Format model: A theory of database organisation. *Journal of the ACM*, 31(3):518–537, July 1984.

[11] D. Michie. Memo functions and machine learning. *Nature*, 268:19-22, 1968.

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

[13] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli, a polymorphic language with static type inference. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 46–57, Portland, Oregon, June 1989.

[14] A. Ohori. A compilation method for ML-style polymorphic record calculi. In A. W. Appel, editor, *Proceedings of 19th ACM Symposium on Principles of Programming Languages*, pages 154–165, 1992.

[15] D. Remy. Typechecking records and variants in a natural extension of ML. In *Proceedings of 16th Symposium on Principles of Programming Languages*, pages 77–88, 1989.

[16] D. Remy. Efficient representation of extensible records. In P. Lee, editor, *Proceedings of ACM SIGPLAN Workshop on ML and its Applications*, pages 12–16, 1992.

[17] M. Wand. Complete type inference for simple objects. In *Proceedings of 2nd IEEE Symposium on Logic in Computer Science*, pages 37–44, Ithaca, New York, June 1987.