# Efficient representation of extensible records

Didier Rémy
INRIA-Rocquencourt*

June 12, 2001

## Abstract

We describe a way of representing polymorphic extensible records in statically typed programming languages that optimizes memory allocation, access and creation, rather than polymorphic extension.

## Introduction

Type systems for records have been studied extensively in recent years. New operations on records have been proposed such as polymorphic extension that builds a record from an older one without knowing its fields. Such operations are very powerful, and were not always provided as primitive constructs in untyped languages.

In comparison to the numerous results on the type theory of records, there has been less interest in their compilation. Many languages still have monomorphic operations on records, e.g. most implementations of ML [HMT90, Wei89, Ler90]. Others, that have more powerful records, use association list techniques, eventually improved by caching.

Safe untyped languages require that the presence of fields is checked before access. The use of association lists interleaves dynamic checking with access to fields. In strongly typed languages, the presence of fields is statically checked. Thus the representation of records by association lists performs superfluous runtime checks, and it seems that cheaper solutions could be found.

We propose a representation of records based on a simple perfect-hash coding of fields that allows access in constant time with only a few machine instructions which can be dropped to a single instruction whenever the set of fields of the record is statically known. Creation can be performed in time proportional to the size of the record, and allocates a vector of size the number of fields plus one. Only the polymorphic creation of records has to pay more in time and memory.

In section 1, the specificity of records as data product structures serves as an introduction to the conditions for which our representation will work in practice. The method is described in detail in section 2 as the encoding of partial functions from labels to values with finite domains. In section 3 we extend the method to records with defaults. These are total functions from labels to values that are constant almost everywhere. As an application we get safe standard records in an untyped language. In section 4, we discuss how to handle pathological cases in order to prevent bad behavior.

## 1 Records and their specificity

Records are product data structures. Each piece of information is stored with a key, more commonly called a label, that is used to retrieve the information. There is at most one value associated to a label. By *field* a pair noted $a \mapsto v$ of a label $a$ and a value $v$. Such data structures are of common use in computer science. However our definition of records is too vague for choosing a good representation of records. It is necessary to know the average and the maximum size of these structures, to know whether they are created incrementally, the frequency of the different operations and which ones are privileged. It is obvious that different programs will give different answers. These questions can only be answered in general, or the answers that we give below can also be taken as assumptions for which our representation of records will work in practice.

Records are provided with three operations. *Creation* builds a record with a finite number of labels together with values associated to these labels. *Extension* takes a record and builds a new one that has all fields of the first one plus a finite number of new fields. *Access* takes a record and a label and returns the value associated to that label. It fails if the label is not in the domain of the record.

Records have a relatively small number of labels. At most a couple of hundred, on average less than ten. Non incremental creation and access are both very frequent and are privileged. There are usually many records with the same domain. Space and time are equally

*Author's address: INRIA, B.P. 105, F-78153 Le Chesnay Cedex. Email: Didier.Remy@inria.fr

important.

The simplest representation of records by association lists is good for very small records but it makes access to large records too slow. Balance trees would have better performances for large records but the overhead has also to be paid for small records; they also require too much memory. General hashed tables will also have an overhead that is not acceptable for small records.

## 2 Extensible records with polymorphic access

In this section we consider records as partial functions from labels to values, with finite domains. The problem is to find a representation for such functions, such that under the assumptions of the above section, the operations on records can be performed efficiently. By performing, we mean evaluating in general. In particular, this applies to compilation where some of the evaluation can be done statically.

Standard monomorphic operations on non extensible records should not be penalized by the introduction of more powerful records. Although it is always possible to keep two kinds of records coexisting, the new records should replace the older, weaker ones. It should be left to the compiler to recognize that some record operations are monomorphic and thus can better be compiled. Indeed, this will not be possible for all compilation schema.

A record $r$ is a partial function from labels to values with a finite domain $(a_i \mapsto v_i)^{i \in 1..n}$. The simplest decomposition of this function is

$$\begin{array}{ccccc} \text{Labels} & \xrightarrow{h} & [1,n] & \xrightarrow{v} & \text{Values} \\ a_i & \longmapsto & i & \longmapsto & v_i \end{array}$$

The total function $v$ stores the components of the record, while the the partial function $h$, called the header, describes how labels are mapped to indices. This decomposition suggests the representation of $r$ in a vector $R$:

$$\begin{cases} 0 \mapsto H \\ i \mapsto v_i & i \in [1,n] \end{cases}$$

where $H$ represents the function $h$.

The partial function $h$ needs to be defined at least on the domain of $r$ and it should better be injective on the domain of $r$ too. Any such function would work, since $v$ is then defined by

$$(h \upharpoonright dom\,(r))^{-1} \circ r$$

up to permutation of indexes with identical values.

If all records are coded such that their headers (the representations may differ provided they implement the same function) only depend on their domains, then compilation of operations on records whose set of fields is statically known can be optimized by partially evaluating their header. The access become a single indirect read to fetch the value of the record on that field. The creation is always in that case, since it builds a record with $n$ fields from nothing. The header can be computed statically and shared between all records built by the same function. The cost is reduced to allocating and filling $n+1$ fields of a vector. More generally, headers can be shared between all records that have the same domain by keeping all existing headers in a table.

Polymorphic access and polymorphic extension must use the headers. For sake of simplicity, we consider that labels are integers. The parser and the printer would deal with the isomorphism between integers and names in a real language.

Finding a good representation of $h$ seems as difficult as finding a good representation of $r$. There are two differences, though. Since the header is shared, we are allowed a little more flexibility on the size of $H$. Also, $h$ is a function on integers, thus we are allowed to use arithmetic and logic operation on integers. There is no hope of finding a direct representation of $h$ by arithmetic operations, since its domain is completely arbitrary. At least some mapping between integers has to be an arbitrary map represented by a vector of integers for instance. A mixed decomposition of the header $h$ is:

$$\mathbb{N} \xrightarrow{(\_\bmod p)} [0,p-1] \xrightarrow{\eta} [1,n]$$

where $(\_\bmod p)$ is injective on the domain of $r$. Such a decomposition is always possible, to the price of a higher $p$. In practice the smallest $p$ that works is on average twice the size of $n$ for a few labels and three to four times for larger sets of labels. Since the header is shared, this is very acceptable.

The interest of this decomposition of $h$ is that it can be compiled efficiently and coded in a vector $H$:

$$\begin{cases} 0 \mapsto p \\ j \mapsto \bar{\eta}(j-1) & j \in [1,p] \end{cases}$$

The partial function $\eta$ must be extended into a total function on $[1,p]$. We write it $\bar{\eta}$ the unconstrained extension of $\eta$.

In too cases below, We will also be interested in two particular extensions of $\eta$ below that we write $\hat{\eta}$ and $\check{\eta}$. The former $\hat{\eta}$ is an extension of $\eta$ with values of $[1,n]$, thus it makes $r$ a total function. The later $\check{\eta}$ extends $\eta$ outside of $[1,n]$, for instance 0, which provides a membership test to the domain of $\eta$ by testing $\bar{\eta}$ for equality to zero.

The domain of $r$ must also be coded in $H$ for polymorphic extension. All its labels can be listed at the end of $H$.

$$\begin{cases} 0 \mapsto p \\ 1 \mapsto n \\ 2+j \mapsto \bar{\eta}(j) & j \in [0,p-1] \\ 1+p+i \mapsto a_i & i \in [1,n] \end{cases}$$

The access can be optimized whenever the domain of the record, and consequently the header, are statically known. Such information is expected to be found by the typechecker. This cannot always be the case, however.

In order to rely on the types to know the domains of records, the attendances of fields, given by the types of records, must correspond exactly to their domains. This implies that the restriction of a record on a field modifies its header, since its changes the attendance. This is one possible semantics for restriction. Another one is to take the restriction of fields as a retyping function, that is, a function that evaluates as the identity. The choice is between an expensive active restriction that allows access optimizations or a cheap retyping restrictions that forbids them.

# 3 Records with defaults

In this section we consider records as partial functions from labels to values, constant almost everywhere. The problem is now to recognize whenever the field does not belong to the domain of the record (we mean the explicitly defined values, here), in which case the default value is returned. The membership test might be expensive in time or in space.

There is a cheap solution based on the same technique as above. In fact, we coded records by total function on labels, and described the domain separately in order to implement the extension of fields. Thus we could apply them outside of their domain (but get a value of unpredictable type).

Let $r$ be a record. Consider the record $r'$ equal to $id \upharpoonright dom(r)$. An arbitrary label $a$ is in the domain of $r$ if and only if it is equal to $r'.a$. The auxiliary record $r'$ only depends on the domain of $r$ is already be coded in the header $H$ as the domain of $r$. Remember that $i$ is the index in $R$ where the value of label $a_i$ is stored. Thus it is the index where $a_i$ is in $R'$, which is also in $H$ at position $2+p+i$, provided $i = \eta(a_i \bmod p)$.

We simply shift the indices in $R$ to place the default value at position 1:

$$\begin{cases} 0 \mapsto H \\ 1 \mapsto d \\ 1+i \mapsto v_i \quad i \in [1,n] \end{cases}$$

We compute the application of $r$ to the label $a$ as follows. First compute the index $i$ associated to $a$, that is $H.(a \bmod (H.0))$. If $H.(2+p+i)$ is equal to $a$, then the label belongs to the domain of $r$ and the result is $R.(1+i)$ otherwise it is the default $R.(1)$.

One must be careful to use the $\hat{\eta}$ extension of $\eta$. The $\check{\eta}$ extension is still possible, but the above membership test must be preceded by a membership test to the domain of $\eta$, and in case of failure the default value should also be used.

In fact, the encoding of records with defaults can be easily adapted to implement safe classical records in an untyped language: a dynamic type error is raised when the membership test fails instead of returning the default.

| $n$ | 3 | 5 | 8 | 11 | 23 | 30 | 40 | 60 | 100 | 200 |
|---|---|---|---|---|---|---|---|---|---|---|
| $p_{ave}$ | 4 | 9 | 18 | 30 | 86 | 132 | 207 | 400 | 902 | 2565 |
| $p_{max}$ | 11 | 18 | 29 | 48 | 148 | 206 | 298 | 576 | 1195 | 3053 |

Figure 1: Average header size

# 4 About efficiency

This section does not consider the case of records with defaults for sake of simplicity, but it can be adapted very easily to them.

The previous representation of records is very attractive since it implements linear time access, uses very little memory, keeps the same performance on monomorphic records as if all records were monomorphic. However we must check the following points. First, that the size of the header does not get too large. Secondly, that the polymorphic extension does not have too bad performance, even though it is not privileged. Last, that pathological cases can be handled.

The computation of the integer $p$ is at the heart of every question. The problem is given a set of integers $D$, find a small integer $p$ such that $(\_ \bmod n)$ is injective on $D$. The integer $p$ does not need to be minimal, even if computed at compile time, since a larger header might make polymorphic extension more efficient. However, the minimal $p$ gives a lower bound on the size of the header. For instance if $D$ is randomly chosen, and the set of labels is large enough in comparison to the size $n$ of $D$, the probability that $p$ disambiguates $n$ integers is

$$\frac{p!}{(p-b)!\,p^n}$$

The formula that gives the average smallest $p$ in function of $n$ is simple, but figure 1 gives an experimental result on the average $p_{ave}$ and the largest $p_{max}$ for on a hundred runs per column.

For small records, $10^4$ runs did not give very different maximal $p$. The dispersion of $p$ is shown by figure 2

The figures show that under 30 labels, the size of $D$ does not exceeds, in average, four times the number of fields, and exceeds very rarely twice the average. For large records (above 50 fields) the header becomes very large. It is clear that another solution must be applied for large records. Even if one wished to push this limit to a hundred labels, there is always a rank that can be reached in practice (even if it is pathological) for another representation should be used.

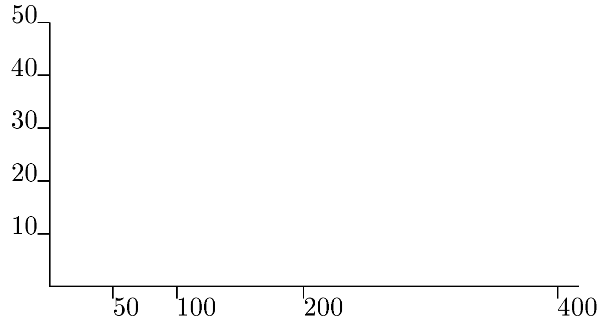


Figure 2: Dispersion of the header size

Since, we cannot avoid a mixed representation, if we want to handle large size records in order to represent them with reasonable size headers, we use tags to distinguish between the two representations. For instance, negative integers can be used as tags for an alternate representations. There are many possibilities for the alternate representations, and since these are pathological cases, in the sense that they do not meet the requirement that we set in section 1, we do not care much about the efficiency of the alternate representation. We propose, two possible solutions that fit well with the regular representation.

The first solution is whenever $a$ and $b$ are equal to $j$ modulo $p$ to assign $\eta(j)$ with an integer $-q$ such that $q$ is in $[2,p]$ and modulo $q$ distinguishes $a$ and $b$ and with values that are not in the image of $\eta$.

The second solution replaces perfect hashing by hashing with linear probing ([Sed88], Chapter 16). The header is $H$ is

$$\begin{cases} 0 \mapsto p & \\ 1 \mapsto n & \\ 2+j \mapsto \bar{\eta}(j) & j \in [0, p-1] \\ 1+p+i \mapsto a_i & i \in [1, n] \\ 2+n+p+j \mapsto a_{\eta(j)} & j \in [0, p-1] \end{cases}$$

for labels that do not conflict. When $2+n+p+(a_i \mathrm{mod} p)$ is already occupied, the label $a_i$ is placed at the smallest free position after, say $j$, and $\eta(j)$ is filled with $i$.

The first method still gives access in linear time, but headers are more difficult to compute. The may be much faster on average, but require larger headers. In both cases there is a lot of freedom on how to chose $p$, according to how many conflicts are accepted. Letting $p$ be about 3 times the size of $r$ may avoid searching for optima while limiting the number of clashes. The first method is more flexible, since a conflict for one label does not need to double the size of the header or recompute another header: it simply uses the holes of the actual header. Then, it can also be used even for average size records in some cases in order to compile more efficient extension.

The efficiency of polymorphic extension has not been considered yet, since it was not a privileged operation. It has to dynamically compute a new header, which may be very expensive — and at least proportional to the size of the record it extends. The time for computing the header now becomes important. There are different cases (we exclude large size records, that should be represented otherwise):

- There is no need to compute a new $p$,
- The average case for computing a new $p$,
- The worse cases for computing a new $p$.

In the average case, computing a new header means that 3 different $p$'s must be probed per extension, since headers are in average 3 times the size of $n$. The optimistic unit cost $U$ is the one of a loop that contains at least one vector write and one modulo instruction. The cost for a probe is $pU$, since the failure is probable to happen at the end. Thus the average cost for creating a new header is $3pU$ plus two other $pU$ for filling the header.

However, a record with a very compact header may be extended with a label that will make the header get closer to its average size. Then about $nU$ probes may be needed, making the cost for the new header increase to $pnU$.

On the other hand, it is very probable that the extended header already exists or is trivial. If the label $a$ of the extended field is taken at random, there is a probability of $(p-n)/p$ that the $n+1$ fields will still be disambiguated by $p$. In that case, the cost for creating a new header is the same as copying the old one plus modifying a few fields $pU$.

The creation of a new header may be avoided most of the time since it is very probable that it has already be created. This requires that all existing headers are stored in a dictionary. This will save both space and time. The keys in the dictionary are the domains of headers that can be kept ordered in $H$, so that equality tests are not too expensive, and the whole cost of searching would be lower than the minimal cost of extension.

Active restriction of fields can be implementing along the same ideas. The header is looked up in the table. If it does not exist, then the new header need not be the smallest one, provided that it is of reasonable size. This avoids the expensive cost of finding the smallest integer modulo which all elements of the domain as distinguished.

# 5 Other compilation schemas

There are three different ideas in the above representation of records

1. The value of fields and the position of fields are represented separately. The header that describes the later is shared between all records having the

same set of fields, two records with the same domain always have the same fields at the same position.

2. The header can be represented by a modulo followed by a projection.

3. Different representation of the headers can live together.

The first point is crucial in our representation. Any representation of records that does not originally respect this point can still be used to implement headers, then values can be stored in vectors as above. Sharing of headers will save the large amount of space required by association lists or balanced trees.

The representation of the header itself is not important. We described one possibility that is very convenient for small and medium size records. But many other representations are possible. We choose to represent the header as a structure that is interpreted both by extension and access. It could also be a closure for the access part, together with a description of the domain that is needed for the extension. This is the tag vs closure duality.

If the extension and the restriction of fields are themselves closed with the header, records could really be viewed as objects with two methods for access and extension.

## Conclusion

We have presented a way of representing records with or without defaults that allows efficient access and creation. Only the more powerful features such as polymorphic extension, or true restriction of fields have to pay a higher price.

Our representation of record with defaults can be used to implement safe access in an untyped language.

An orthogonal application could be the representation of feature terms that are very related to records.

## Thanks

These ideas originate in discussions with Xavier Leroy. They have been mentioned for the first time in [Ler90] and tested in the untyped version of Zinc, the ancestor of Caml-Light.

## References

[HMT90] Robert Harper, Robin Milner, and Mads Tofte. *The definition of Standard ML*. The MIT Press, 1990.

[Ler90] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 1990.

[Sed88] Robert Sedgewick. *Algorithms*. Computer Science. Addison-Wesley, second edition edition, 1988.

[Wei89] Pierre Weis. *The CAML Reference Manual*. INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 1989.