No more need for records

Richard A. O'Keefe
Computer Science, University of Otago

November 2003
Fifth draft: May 2012

Abstract

I continue to seek minimal changes to Erlang that will make it better for growing very large systems.

In this report I propose a new compound data type, with a long history in other programming languages, which can be used as a replacement for the `-record` construct. This data type has the same storage requirements as records, but does not require textual coupling between Erlang modules, and is more robust against change, and provides full run-time access to field names for printing and version change.

A revised version of `dict.erl` is attached to show how this works in real code.

1 Goals

The purpose of this proposal is to provide a data type, syntax, and operations that

+ can replace Erlang records
+ can be used to communicate data between separate modules and processes without the use of `.hrl` files
+ can offer enough information for run-time printing and parsing to be fully compatible with source syntax
+ can be reliably distinguished from every other data type
+ can be as efficient as records given type information as inferred by TypEr or the whole-module type inference phase in HiPE

There are some non-goals:

- there is no need to be as efficient as C or Java
- there is no need to support object-oriented style record extensions as this is not something that the current -record syntax supports
- there is no need to support O(1) declarative update — there are papers going back to the 1980s showing up to support O(1) update of immutable arrays in functional languages, but this is not something Erlang programmers expect of records
- there is a need not to support detectable in-place update.

2 State of this document

There will be time to murder and create,
And time for all the works and days of hands
That lift and drop a question on your plate;
Time for you and time for me,
And time yet for a hundred indecisions,
And for a hundred visions and revisions,
Before the taking of a toast and tea.

—T.S.Eliot, *The Love Song of J. Alfred Prufrock*

This document has been edited over nearly a decade; time for a hundred visions and revisions. The scars of its evolution show. Had I but world enough and time to revise it... but it already shows signs of growing vaster than empires. Dear reader, please bear with me, the end is in sight and the mess will be cleared away eventually.

3 What are records in Erlang?

Originally, Erlang had a variety of atomic data types, plus two compound data types. The two compound data types were

list sequences built from [] (empty) and [_|_] (pair), good for element-at-a time operations, but taking $O(n)$ time to reach the nth element;

tuple sequences built from an infinite family of {_,...,_} constructors and by using `list_to_tuple`/1, providing $O(1)$ time to access any element, but requiring $O(n)$ time to build an updated tuple. and not convenient for element-at-a-time operations.

These compound data types are sufficient for all practical purposes, as the OTP libraries demonstrate, but they are not *convenient* for all purposes. Sometimes it is useful to have a compound data structure that does not represent a sequence but an aggregate of possibly dissimilar data, whose components are accessed by name, not position.

Erlang still does not possess such a data structure, but the OTP Erlang compiler provides syntax which simulates it. For example, the following -record declaration can be found in the library file `stdlib/src/dict.erl`:

```
-record(dict, {
    size = 0,                              %# of elements
    n = ?seg_size,                         %# of active slots
    maxn = ?seg_size,                      %Maximum slots
    bso = ?seg_size div 2,                 %Buddy slot offset
    exp_size = ?seg_size * ?expand_load,   %Size to expand at
    con_size = ?seg_size * ?contract_load, %Size to contract at
    empty,                                 %Empty segment
    segs                                   %Segments
}).
```

This describes a record type called `dict` with eight named fields, `size`, `n`, `maxn`, `bso`, `exp_size`, `con_size`, `empty`, and `segs`. The first six of these fields

have default values; when a record of this type is constructed it is not necessary to mention any fields which should have their default values.

Record declarations provide syntactic sugar only. A value of this "type" is a tuple with nine elements; the first is the record type name `dict`, and the remaining eight are the fields in the order they were declared. For example, the expression

```
#dict{empty=Empty,segs={Empty}}
```

is entirely equivalent to

```
{dict, 0, ?seg_size, ?seg_size, ?seg_size div 2,
 ?seg_size*?expand_load, ?seg_size*?contract_load,
 Empty, {Empty}}
```

The run-time representation of a record encodes the record type name and the number of fields, but the field names are not represented at all.

A record value belonging to a record type with n fields requires $n+2$ cells of memory (the arity, because tuples require it, the record type name, to provide limited error checking, and one cell for each field), plus whatever space overheads a tuple would require for garbage collector support. In a functional language with static type checking, such as ML or O'CAML, the space needed would be just n cells plus garbage collector overhead. For the example above, Erlang requires 10 cells where ML would require 8. This is tolerable overhead; a list would require at least 16 cells.

Erlang records have the following advantages:

- much improved clarity in the source code;
- reasonably good space efficiency;
- good time efficiency.

I am concerned with Large Scale Erlang, and records do have some problems which affect large systems.

The major problem is that if two modules need to use the same record type, the declaration is put in a `.hrl` file, which both modules then include. The Erlang run time system knows about modules; it does not know about include files. This makes "version skew" problems possible.

4 Three alternatives

In this section I compare records with abstract patterns, described in earlier work, and with "frames", which are the subject of this report.

Suppose we have a module P which produces records of some type, and another module C which consumes them.

1. Is the preprocessor required?

   records Yes.

   abspats No.

   frames No.

2. In the absence of an always-enforced type system, the only types you can rely on distinguishing are the ones for which there are built-in guard predicates which are guaranteed to be disjoint. Are records disjoint from non-records?

   records No, they are tuples.

   abspats No, they can be anything but new.

   frames Yes. There is a new "frame" data type different from existing types.

3. Are record types disjoint from other record types?

   records No. Nothing stops two modules defining record types with the same name and arity. At run time such records cannot be distinguished.

   abspats No.

   frames No. However, this can be useful. On the other hand, the field names are taken into account when matching in pattern matching, so they are harder to spoof than records.

4. Does it seem likely that the Dialyzer could be made to understand the new feature with reasonable effort?

   records Yes, it already handles them.

   abspats Yes, they are like functions.

   frames Yes, but it's not trivial.

5. Can this way of describing records be used to enforce type constraints on fields at run time?

   records No.

   abspats Yes, for "records" constructed this way. "Forged" records that were constructed some other way will fail to match.

   frames No.

6. Can C add an annotation to a record and send the record back to P?

   records No.

   abspats No.

   frames Yes. If k is a key which is not used in F, then `<{`$k \sim E|F$`}>` produces a frame which can be used in place of F.

7. If P adds one or more fields without changing existing ones, we would expect this to be a **compatible** change. What must be done to C to make it work with the new definition, assuming that C does not create such records, but only uses and updates ones provided by P?

   records Recompile C.

   abspats If P provides backwards-compatible abstract patterns, C should work as is, otherwise C needs revision.

frames C will work as is.

8. If P removes one or more fields without changing the remaining ones, and C does not use the removed fields, we would expect this to be a **compatible** change. What must be done to C to make it work with the new definition?

   records Recompile C.

   abspats If P provides backwards-compatible abstract patterns, C should work as is, otherwise C needs revision.

   frames C will work as is.

9. If P removes one or more fields and C does use the removed fields, we expect that to be an **incompatible** change. When will it be detected?

   records when C is recompiled or when the functions that use those fields are tested.

   abspats when C is loaded (if there is a load-time check for the existence of imported functions, since the requisite abstract patterns will not be there to import) or cross-referenced, or when the functions that use those fields are tested. If a field can be computed from surviving fields, P may be able to provide backwards-compatibility abstract patterns, and then C will work.

   frames only when the functions that use those fields are tested or if an enhanced Dialyzer is used.

10. If P changes the name of a field without changing its meaning, we expect that to be an **incompatible** change. When will it be detected?

    records when C is recompiled.

    abspats when C is loaded (if there is a load-time check for the existence of imported functions, since the requisite abstract patterns will not be there to import) or cross-referenced, or when the functions that use that field are tested.

    frames only when the functions that use that field are tested or if an enhanced Dialyzer is used.

11. If the default value for a field is changed, we expect that to be a **compatible** change. What rebuilding needs to be done?

    records if the dependency between the modules and the `.hrl` file is recorded where a `make`-like tool can see it, C will be recompiled, even though it doesn't need to be.

    abspats if P provides backwards-compatible abstract patterns, no recompilation is needed.

    frames None.

12. If the meaning of a field is changed, we expect that to be an **incompatible** change. When will that be noticed? It's the same answer for all three: when a type checker is used, or if testing is good enough, but possibly never.

13. If P changes the order of the fields, without changing anything else, we expect this to be a **compatible** change. What must be done to make C work with the new version of P?

    records Recompile C.

    abspats Nothing, C will work as is.

    frames Nothing, C will work as is.

14. If we make **compatible** changes to P, such as adding fields without changing their meaning, changing default values (of no interest to a module that only uses and updates records), and/or changing the order of fields, what must be done to C for it to work with *both* versions of P? This is particularly interesting when records made by P may still be present in binaries (thanks to `term_to_binary`/1) or `dets` or Mnesia tables.

    records It can't be done.

    abspats It can be done, with some care.

    frames C doesn't need anything done to it.

15. How much space is required for a record with n fields?

    records $n + 2$ cells plus garbage collector overhead.

    abspats $n + 1$ cells plus garbage collector overhead; you don't have to include the record type name if you don't want to.

    frames $n + 1$ cells plus garbage collector overhead in the usual case. It is possible to construct frames that don't correspond to any declaration in the source code, and in that case the cost can be $2(n+1)$ cells, but that only occurs with `dict`-like use of frames, not with record-like use.

16. How much does it cost to fetch 1 field of n?

    records $O(1)$

    abspats $O(1)$, assuming a tuple mapping.

    frames $O(\lg n)$. Record-like use means small values of n, but the constant factor must be considered as well. Records and abstract patterns are clearly faster. However, see the experimental results below.

17. How much does it cost to fetch cn fields of n, where $0 < c \leq 1$? The answer is $O(n)$ for all of them.

18. How are records printed?

    records as the underlying tuple.

    abspats as the underlying tuple, or whatever the designer mapped it to.

    frames as a frame.

19. Can a type checker be used?

    records Yes; check as concrete type.

abspats Yes; check as concrete type.

frames Yes. The type language must be extended with frame types, and the type checker must be able to deal with subtypes. The Erlang type checker, Dialyzer, can deal with subtypes, so only the easy bit is left.

20. Can a cross referencer find uses of a record type?

    records Yes.

    abspats Yes; abstract patterns are just like functions.

    frames No; there are no record types as such. That is precisely what makes them more robust against compatible changes. However, it *is* usefully possible to locate most uses of field names.

To summarise: abstract patterns offer better static checking and faster code, while frames offer more flexibility and better robustness against compatible changes. Ideally, Erlang should have both.

5 So what are "frames"?

Frames are semantically immutable association lists: finite partial functions from the space of Erlang atoms to the space of Erlang terms. In other words, a frame is a finite set of (key,value) pairs such that values are any Erlang terms, keys are all atoms, and no two keys are equal.

Similar but mutable (or further instantiable for LIFE and IBM PROLOG) data structures have been known by various names in various languages: "tables" in SNOBOL, "Dictionaries" in Smalltalk, "associative arrays" in AWK, "hashes" in Perl, "psi-terms" in LIFE, "items" in IBM PROLOG, "feature structures" in linguistics, and who knows what elsewhere.

The idea of this data type occurred to me a day after I first read about Erlang records, but I did not write it up until September 2003. It turns out that Joe Armstrong had also thought of this data type, and wrote something about it in 2001. I had not then seen what he wrote. He uses the name "proper structs" and slightly different syntax. I do not know how he intended them to be implemented.

The OTP library includes several modules providing related abstract data structures, notably `dict`. The `dict` concrete data structure has been engineered to support medium-large collections (up to several thousands) with arbitrary keys (not just atoms) and fairly frequent changes. An empty `dict` takes 28 words + 3 sets of garbage collector overhead; this is reasonable for large collections but quite unreasonable for a record replacement.

6 Functions for frames

Frames were (re-)invented to replace record syntax, but they have a semantics. The following functions are described for the purpose of talking about the semantics, but they should also be made available for programmers to use.

- `is_frame`(F) = bool().

  If F is a frame, the result is `true`, otherwise `false`. In a guard, succeeds or fails instead of returning `true` or `false`.

  Cost: $O(1)$.

- `frame_to_list`(F) = $[\{K_1, V_1\}, \ldots, \{K_n, V_n\}]$.

  where the keys $\{K_1, \ldots, K_n\}$ are in strictly ascending order. The fact that the order is defined means that it makes sense to match against the result, for example

  ```
  [{day,D},{month,M},{year,Y}] = frame_to_list(Date)
  ```

  Cost: $O(n)$.

- `list_to_frame`($[\{K_1, V_1\}, \ldots, \{K_n, V_n\}]$) = F.

  The argument must be a well formed list; each of its elements must be a pair; and the first element of each pair must be an atom. The atoms need not be distinct. If two pairs have the same key, the first (leftmost) value is used and the later pair is ignored.

  Cost: $O(n \lg n)$. If an adaptive sort like samsort is used, and the keys are in ascending order, $O(n)$.

- `frame_size`(F) = Size.

  The result is the same as `length(frame_to_list(F))` would have been, but this form should be usable in guards.

  Cost: $O(1)$.

- `frame_has`(F, Key) = bool().

  The result is the same as

  ```
  lists:keymember(Key, 1, frame_to_list(F))
  ```

  would have been. In a guard, this succeeds or fails instead of returning `true` or `false`.

  Cost: $O(\lg n)$.

- `frame_value`(F, Key) = Value | error exit.

  The result is the same as

  ```
  ({value,{_,X}} =
       lists:keysearch(frame_to_list(F), 1, Key),
   X)
  ```

  would have been, except of course for not binding X and for a more informative error report. In a guard expression, this makes the guard fail instead of signalling an error.

  Cost: $O(\lg n)$, but see the experimental results below.

- `frame_find`(F, Key) = {Key,Value} | `false`,

  The result is the same as

  ```
  lists:keyfind(Key, 1, frame_to_list(F))
  ```

  would have been[1], except for a more informative error report. As this may create a new tuple, it is not allowed in a guard.

  Cost: $O(\lg n)$.

- `frame_sum`(F1, F2) = F.

  The result is the same as

  ```
  list_to_frame(lists:keymerge(1,
      frame_to_list(F1), frame_to_list(F2)))
  ```

  would have been, except for more informative error reports. As the code above implies, if a key is present in both dictionaries, the value in F1 is used.

  Cost: $O(|F1| + |F2|)$.

- `frame_keys`(F) = $\{K_1, \ldots, K_n\}$.

  The result is the same as

  ```
  list_to_tuple([K || {K,_} <- frame_to_list(F)])
  ```

  would have been, except for more informative error reports. In particular, the tuple of keys is sorted.

  Cost: $O(1)$.

  An earlier draft had this function returning a list of keys, which would in some ways be more convenient. However, with the implementation I have in mind, this is an $O(1)$ operation because the key tuple is already *there*; returning a list would cost $O(n)$. If you want a list, you can convert the tuple.

- `frame_without`(F, Keys) = F1.

  The result is the same as

  ```
  list_to_frame([{K,V} || {K,V} <- frame_to_list(F),
                          false==lists:member(K, Keys)])
  ```

  would have been, except for more informative error reports. The point of filtering keys out in batches instead of one at a time is efficiency; deleting n keys from a frame with m keys one at a time requires $O(nm)$ time.

  Cost: $O(|F| + k \lg k)$ where k = length(Keys).

  For actual use, as opposed to exposition, a better approach would be to subtract (the keys of) one frame from another:

  `frame_difference`(F1, F2) = F

  whose result is the same as

[1] This paper was originally written in 2003, when keyfind did not exist, so it proposed an interface based on keysearch. This is better.

```
frame_without(F1, tuple_to_list(frame_keys(F2)))
```

would have been.

Cost: $O(|F_1| + |F_2|)$.

With the aid of functions like these, we can construct other functions such as

```
frame_map(F, Fun) ->
    list_to_frame([{K, Fun(K,V)} || {K,V} <- frame_to_list(F)]).

frame_filter(F, Pred) ->
    list_to_frame([P || P={K,V} <- frame_to_list(F), Pred(K,V)]).
```

I did not list such functions above, because they are appropriate for a `dict`-like use of frames, not for the intended record-like use. It would, for example, be very difficult to assign any meaningful type to `frame_map`/2.

7 Equality and ordering

The built-in term comparison operators work for records just as well as (and in the same way as) they do for any other tuple. It is important that they should work for frames too.

7.1 Equality

A frame is not equal to anything that isn't a frame. Two frames F1, F2 are equal if and only if `frame_to_list`(F1) is equal to `frame_to_list`(F2).

Thanks to the quirky nature of IEEE floating point comparisons, where +0.0 behaves quite differently from -0.0 yet compares "arithmetically equal" to it, and where there are values x such that x is not "arithmetically equal" to x, Erlang has two equality predicates, `==` and `=:=`. (Sadly, although Erlang got this distinction from Prolog, Erlang has the opposite convention: `==` is "arithmetic equality" (Prolog's `=:=`) and `=:=` is "identity" (Prolog's `==`).) So the actual definition has to be

- F1 =:= F2 iff `frame_to_list`(F1) =:= `frame_to_list`(F2)
- F1 == F2 iff `frame_to_list`(F1) == `frame_to_list`(F2)

7.2 Ordering

It can be useful if ordering is compatible with set inclusion. *Compatible*, not *identical*. Erlang term ordering is a total order; set inclusion is not. What I'm suggesting is that `frame_without`(F, Keys) $\leq$ F, with equality if and only if Keys is [].

There are three ways of viewing a frame as a set:

- it simply *is* a set of pairs;
- we might consider the set of keys (as each key will only occur once);

- we might be interested in the set of values (but that set is not represented, although the *bag* of values is).

The definition below is equally compatible with both the first two views.

1. Frames are ordered just after tuples.
2. To order two frames F1, F2:
   (a) First order them by `frame_size`.
   (b) If they have the same `frame_size`, they have the same relative order that `frame_to_list`(F1) and `frame_to_list`(F2) would have.

In Erlang, `{2} < {1,2}` is `true`, so this ordering of frames is similar to the existing tuple ordering.

8 Constructing frames

While you could use `list_to_frame`/1 to make frames, that function was introduced mainly for expository reasons.
primary ::= '`<{`' [maplet {'`,`' maplet}*['|' expression]$^?$]$^?$ '`}>`'
maplet ::= atom '`~`' expression

The expression `<{}>` is equivalent to `list_to_frame`([]). The cost is $O(1)$.

The expression `<{`$k_1 \sim V_1, \ldots, k_n \sim V_n$`}>` is equivalent to
`list_to_frame`($[\{k_1, V_1\}, \ldots, \{k_n, V_n\}]$),
except, as usual, for error messages, and for not constructing a list or any tuples. The cost is $O(n)$ plus the cost of evaluating the V_i.

One question which is still open is how strictly "is equivalent" should be taken. For efficient implementation, we want to have the values in increasing order of key. One approach would be to have the compiler re-order the code, evaluating the expressions V_i in ascending order of key, so that

```
<{avery ~ f(), tom ~ g(), deacon ~ h(), harry ~ i()}>
```

would be behaviourally equivalent to

```
list_to_frame([
    {avery,  f()},
    {deacon, h()},
    {harry,  i()},
    {tom,    g()}])
```

Another approach would be for the compiler to arrange for the expressions to be evaluated in textual order, and generate additional code to re-order at run time. In this case that would look like

```
(A = f(), B = g(), C = h(), D = i(),
list_to_frame([{avery,A},{deacon,C},{harry,D},{tom,B}]))
```

with the variables being local temporaries. For HiPE-compiled code we'd expect no measurable difference, so for this draft of this report, I have assumed strict textual order.

The keys $k_1, \ldots, k_n$ must be atoms, not general expressions. One reason for this is consistency with pattern syntax, in the next section. Another reason is the compile-time opportunity mentioned above. The major reason is error detection; case shift errors are not uncommon. For the general case, `list_to_frame`/1 may be used.

The expression `<{`$k_1 \sim V_1, \ldots, k_n \sim V_n|F$`}>` is equivalent to
`list_to_frame`$([\{k_1, V_1\}, \ldots, \{k_n, V_n\}|$`frame_to_list`$(F)])$,
except, as usual, for error messages, and for not constructing any list or tuples. The cost is $O(n + m)$ where $m =$ `frame_size`(F).

This syntax is used for constructing a revised frame. Normally the new keys will occur in F, but they need not. That is called updating; updating does not imply in-place mutation. Frames are immutable, just like lists and tuples in Erlang. When one or more of the k_i do not occur as keys in F, we call that "extending" a frame. Extending is slightly more expensive than updating, but it is still linear.

I considered allowing more than one $|F$ part. The idea was that in a frame constructor, $F_1|F_2$ would mean the same as `frame_sum`(F_1, F_2); that is what the function `frame_sum`/2 was introduced for. In particular, `<{`$k_1 \sim V_1, \ldots, k_n \sim V_n|F$`}>` is equivalent to `frame_sum(<{`$k_1 \sim V_1, \ldots, k_n \sim V_n$`}>,` F`)`. I couldn't think of any way that this generalised syntax could be used in pattern matching, and the extra generality doesn't seem to be of any practical use, so I dropped the idea.

We expect that if $\{k_1, \ldots, k_n\} \subseteq$ `frame_keys`(F) then the result will share F's descriptor, otherwise a new descriptor will be allocated.

8.1 Excess generality

The previous subsection allowed both updating and extending a frame. There is no analogue of extension in current Erlang records. Extension comes with two costs:

- if you provide new values for existing fields, the new frame can share the old frame's descriptor. The only way to create new descriptors is through the functional interface. If you want to have an intuitive feel for the space usage of your program, it is well to be aware of when you are creating new descriptors and when not. Creating new descriptors has both direct (memory) and indirect (time) costs.
- if you *intend* to replace the `con_size` slot of a frame D, but by mistake write $\langle\{$`consize` $\sim 1 \mid D\}\rangle$, you would like to be told about it. The previous section would allow this, with the result that you would get a new frame with *both* a new `consize` slot and an old `con_size` slot, and code that fetched the value of `con_size` would quietly get the old (wrong) value.

When you are revising an existing Erlang module to use frames instead of records, you never want extension. You do want safety. Accordingly, in 2012 I decided that only update should be supported, and using a slot name in an update that does not exist in the base frame should be a checked run time error.

This makes it possible to use inline caching to make updates faster.

8.2 What we've lost and why I don't care

An Erlang -record declaration lets you provide default values for fields. For example,

```
-record(date, {year = 2012, month, day = 1})
```

allows us to write any of

```
#date{year = 2012, month = 4, day = 1}
#date{year = 2012, month = 4}
#date{month = 4, day = 1}
#date{month = 4}
```

to express "April Fool's Day, 2012".

There is no analogue of this for frames. The reason is simple: the whole point of frames is to not need and not have a declaration. That means there is no place to put default values.

It doesn't mean that we cannot *have* default values, only that they must be expressed functionally:

```
date(Y, M, D)   -> <{day ~ D, month ~ M, year ~ Y}>.
month(Y, M)     -> <{day ~ 1, month ~ M, year ~ Y}>.
this_year(M, D) -> <{day ~ D, month ~ M, year ~ 2012}>.
```

Default values address a real maintenance problem, that of adding a field to a record without modifying existing code.

The Erlang Reference Manual does not say in section 9.1 what kinds of expressions are legal as default values for record fields. It appears that there is no restriction, except that there is no way for such an expression to mention the values of the fields that were *not* omitted. This is important because the right default value for a field may well depend on the value for other fields.

Record declarations tacitly assume that if each of two fields can be defaulted, both of them together can, as shown in the example above. This is not necessarily the case.

Functional interfaces address the same issue as default values, but more flexibly. The one thing they do not do is to make plain which argument is which, but that's a separate problem applicable to all functions, and there is already a "split procedure names" proposal to address that, where we could write

```
year(Y) month(M) day(D) -> <{day ~ D, month ~ M, year ~ Y}>.
year(Y) month(M)        -> <{day ~ 1, month ~ M, year ~ Y}>.
        month(M) day(D) -> <{day ~ D, month ~ M, year ~ 2012}>.
```

8.3 Joe Armstrong's "proper structs"

Joe Armstrong uses '`~{`', '`=`', and '`}` where I use '`<{`', '`~`', and '`}>`'. This is a minor and superficial difference. I prefer mirror-symmetric brackets, to be consistent with every other kind of punctuation mark based brackets in Erlang. The tilde '`~`' comes from Xerox PARC (see "Pebble", for example.) I believe that there is a greater chance of error if '`=`' is used, because that has other uses in the syntax at this level: `~{foo=bar=ugh()}` might be expected to bind `foo`

and `bar` to the result of `ugh()`; in fact it would find `foo` to `bar` and fail if the result of `ugh()` were not `bar`.

There is a semantic difference, however, and in this respect Joe Armstrong's "proper structs" are closer to IBM PROLOG's "items".

IBM PROLOG "items" are named finite mappings from atoms to terms. IBM PROLOG has no syntactic form for "items", but does have the following built-in predicates:

- `is_item`(Term).
- `item_add`(Item, Key, Value) *changes* Item by adding Key∼Value, unless Item already has Key as one of its keys. The change is undone on backtracking.
- `item_create`(Item, $[K_1, V_1, \ldots, K_n, V_n]$) makes an "item" with maplets $K_1 \sim V_1, \ldots, K_n \sim V_n$ and "group name" [].
- `item_create`(Item, $[K_1, V_1, \ldots, K_n, V_n]$, Name) is like `item_create`/2, but uses Name as the "group name" instead of []. Two "items" unify if and only if they have the same maplets and the same "group name".
- `item_delete`(Item, Key) *changes* Item by removing any maplet with key Key. The change is undone by backtracking. It is not clear what happens if Key is not present to start with.
- `item_list`(Item, List) is like my `frame_to_list`/1.
- `item_put`(Item, Key, Value) *changes* Item by adding Key∼Value, removing any previous maplet for Key. The change is undone by backtracking.
- `item_update`(Item, Key, Value) appears to be the same as `item_put`/3.
- `item_update`(Item, Name) *changes* Item by setting its "group name" to be Name. The change is undone by backtracking. The Name must be an atom.
- `item_value`(Item, Name) unifies Name with the "group name" of Item.
- `item_value`(Item, Key, Value) unifies Value with the value associated with Key in Item. Key must be bound before the call; there is no version of this predicate which enumerates maplets.

From this you can see that IBM PROLOG "items" are, in effect, mutable frames with a special "group name" field. Neither Joe Armstrong nor I wanted to add mutable data structures to Erlang. (If we wanted ML, we know where to find it.)

I decided not to have any special case field; Joe Armstrong decided that it *was* useful[2]. He allows a proper struct to have a tag, playing the rôle of a record type name. For example, instead of

```
#dict{size=0, n=?seg_size, maxn=?seg_size, bso=?seg_size div 2,
     exp_size=?seg_size*?expand_load, empty=Empty,
     con_size=?seg_size*?contract_load, segs={Empty}}
```

[2]Someone said in the Erlang mailing list in 2012 that he now wants to drop this field.

he would write

```
~dict{size=0, n=?seg_size, maxn=?seg_size, bso=?seg_size div 2,
     exp_size=?seg_size*?expand_load, empty=Empty,
     con_size=?seg_size*?contract_load, segs={Empty}}
```

It is not clear to me whether he intends proper struct tags to be atoms or to be arbitrary Erlang terms. Clearly there are good reasons for allowing a tag and for his choice of syntax:

1. Switching from record syntax to proper struct syntax is easy.
2. Because you can ask for the tag of a record, having a tag in a proper struct makes conversion easy. In particular, there is a natural replacement for the record/2 guard test.
3. Two proper structs are equal, or a proper struct matches a proper struct pattern, only if they have the same tag as well as the same field names and field values, again making them close to records.

My experience of rewriting record-using code to frame-using code suggests to me that unless one goes to fairly serious lengths to imitate *all* record syntax in the new syntax, the conversion is *not* that easy. The tags provide very limited protection, and can be simulated by adding a field `''~Tag`. I was also concerned that "records" that are private to a module shouldn't have to pay for tags.

I do not regard the presence or absence of tags as of major importance. In particular, while I think it is better not to have them, I do not think that removing them is a big enough change to deprive Joe Armstrong of credit for the idea, just as the fact that "frame" is easier to say than "proper struct" is not of major importance.

A compromise position would be to say that

> `<`*tag*`{` *maplet*,... `}>`

is identical to

> `<{ ''=`*tag*, *maplet*,... `}>`

where *tag* is an atom or a variable. This would allow the use of tags without requiring them; they would not be special at run time, but only in the syntax.

We could also introduce a type test

- `is_frame`(F, Tag) = bool()

  equivalent to `is_frame`(F) ∧ `frame_has`(F, `''`) ∧
  `frame_value`(F, `''`) =:= Tag.

  Cost: $O(1)$.

8.4 The colour of the paint on the Ark

The Erlang mailing list once again had some discussion of this topic in May 2010, and this report was updated to reflect some of that discussion. I commented, somewhat bitterly, "why do we find it easier to argue about the colour of the paint on the Ark than to climb aboard?" It is of course one of Parkinson's

principles of committology at work; his example was a(n imaginary) committee deciding on a nuclear reactor in minutes but arguing for hours about a bike shed.

Everyone has an opinion about syntax. Worse still, everyone has their own idiosyncratic history of experience with other languages, and regards the nostalgic longings for something that looks like the Good Old Days as a natural intuition of the True Nature of the Good and the Beautiful. The only cure for this is a wide experience with many programming languages, so that you realise there is *no* "natural" way to write *anything* in computing. To a C or PL/I programmer, it is *obvious* that `->` *must* be something to do with pointers to records; to an APL programmer it is *obvious* that a right arrow *must* be a GOTO (what else could it be); to a Pop-2 or S programmer, it *couldn't* be anything other than a left-to-right assignment. And so it goes.

My goals in choosing a syntax were these:

- consistency with the general style of Erlang syntax — this meant that as compound data structures the elements should be separated by commas and should be enclosed in symmetric brackets of some sort;
- unambiguity — this meant that every token used to display the structure of a frame must either have the *same* meaning as its existing uses in Erlang or have *no* existing use in Erlang;
- follow common practice when there is common practice to follow — this means that a frame should be displayed as a sequence of "maplets" with the attribute name (or label) as the left element of each pair and its associated value as the right element, rather than putting labels on the right or segregating labels in one list and values in another; it also meant that when I decided to experiment with allowing an Armstrong-style "record tag", that had to go somewhere at the front;
- use only ASCII or perhaps Latin 1 characters, because while we can now display (some) Unicode characters readily, it is still hard to enter them;
- try not to innovate.

This meant that the basic syntax had to be something like

$$\langle k_1 \mapsto v_1, \ldots, k_n \mapsto v_n \rangle$$

except for the brackets and the $\mapsto$ sign.

Readability matters. If you are reading something linearly, you can tolerate a certain amount of ambiguity. If, however, you are searching and you arrive at a line without having read the previous however-many lines, your tolerance for ambiguity is less.

What should we use for $\mapsto$?

- Erlang already uses colon for module prefix; while the context would help a compiler disambiguate, it would not do much for human beings or syntax colouring editors. Consider the line

```
foo:bar(Ugh),
```

If, following Javascript, you use colon for maplets, your comprehension comes to a juddering halt while you try to figure out if this is a maplet or a remote function call. BAD!

- Erlang already uses the equals sign for pattern matching. Consider the line

```
ok = f(X)
```

  Is this a maplet (as it is in erlson) or is it plain old Erlang testing whether the call `f(X)` returned `ok`? BAD! The S programming language used to use the equal only for keyword arguments, and arrows for assignments. A few years ago it was changed to allow equal for both uses. The compiler has no trouble telling them apart, but I do. There is *no way* I was going to contemplate inflicting that kind of misery on other people.

- Luca Cardelli's "A Polymorphic λ-calculus with Type: Type", DEC SRC RR 10, 198 uses *both* colon and equals in record-like constructions, but with different meanings.

- Erlang already uses the single right arrow to separate the head of a (function, if, case, receive, ...) clause from its body.

- Erlang already uses the single left arrow for generators in list comprehensions. In any case, the direction is wrong.

- Ada uses `=>` for keywords, but I sometimes use a language with both `->` and `=>` and I'm forever using the wrong one. Once again, *no way*.

- The at sign (@) might have been a possibility, but Erlang already allows that in atoms, so we can't use it.

- There's nothing in the top half of Latin 1 that particularly says "maplet".

- That leaves $\sim$ as just about the only ASCII/Latin 1 character with any promise.

And here history comes to our rescue. "Pebble, a Kernel Language for Modules and Abstract Data Types", by Butler Lampson and Rod Burstall, published in Information and Control, volume 76 (1988), pages 278–346, originally written in 1983, has a particular focus on maplets, which they call "bindings". Eric Schmidt's PhD on the System Modeller (using a pure functional language based on Pebble for system configuration, building, and distribution), published in 1982, used a similar notation. And the tilde $\sim$ is precisely the character they used for $\mapsto$.

Section 2.2 "Bindings and Declarations" of the Pebble paper says "An unconventional feature of Pebble is that it treats bindings, such as $x \sim 3$, as values. They may be passed as arguments and results of functions, and they may be components of data structures, just like integers or any other values."

Just what we want.

What about a collection of bindings? Well, it's a set, so the obvious thing is to write a frame inside curly braces. That creates two problems. One is a problem for people: Erlang uses curly braces for *sequences*, not sets. It would be

very confusing for people if $\{a \sim 1, b \sim 2\} = \{b \sim 2, a \sim 1\}$ but $\{\{a, 1\}, \{b, 2\}\} \neq \{\{b, 2\}, \{a, 1\}\}$. It's not a problem for the compiler, which can tell frames from tuples by the presence or absence of the $\sim$ punctuator. The other is a problem for both: *nobody* could tell an empty frame from an empty tuple. We could hack around that by banning empty frames, but it's always a mistake to refuse to believe in zero.

The only bracket characters in ASCII are () [] and {}. Erlang uses all of them, and had to press `<<` and `>>` into service for binaries. This provides a precedent for mirror-symmetric compound bracket tokens. Do we have to follow it? The only additional bracket characters in Latin 1 are the double guillemets, which strictly speaking would be most appropriate for strings, but due to their strong visual resemblance to binary brackets, should never be given any use in Erlang other than as synonyms for binary brackets.

Unicode offers many brackets. Angle brackets are perhaps the most familiar, but they are generally used for tuples, and in any case are easy to confuse with `<>` and are still hard to enter.

Mirror symmetric compound brackets I considered included

- `<| |>` like left and right triangles
- `(| |)` like lunate brackets
- `[| |]` like blackboard bold/"semantic" brackets
- `{| |}` like nothing much
- the same four combinations with `!` in place of `|`
- the same four combinations with the classic Pascal `.` in place of `|`
- `<( )>` looking a bit like lenticular brackets
- `<[ ]>` ditto, but square
- `<{ }>` ditto, but curly
- the same three combinations but inside out, which look horrible.

The combinations using square brackets were ruled out because they are downshifted on a keyboard, while the other characters are upshifted. That's an accident waiting to happen.

Apparently brackets and braces are hard to type on Swedish keyboards: Option-(and Option-) for [and], Option-Shift-(and Option-Shift-) for { and }. This means that the best combination for Erlang programmers using Swedish keyboards would be `<(` and `)>` and the second best would be `<{` and `}>`. Perhaps both could be allowed as synonyms.

I settled on `<{ }>` as conveying (to a mathematician) something of the set-like nature of frames, to an Erlang programmer something of their tuple-like nature, and to a JSON user, something of their dictionary-like nature. However, the last word has not been said. In particular, the token sequences `< (`, `< [`, `< {` are already possible in Erlang, so compound brackets done this way cannot be tokens; they have to be recognised as pairs of tokens. It's still easy to write a parser. A frame occurs where an operand is required; $< (\ldots)$ where an operator is required. It all seems to work OK. It may well be possible to do better.

Anton Lavrik has provided the Erlang community with a number of interesting projects at `https://github.com/alavrik/`. One of them is called erlson, put up in July 2011. "Erlson is a dynamic name-value dictionary data type for Erlang. Erlson dictionaries come with a convenient syntax and can be directly converted to and from JSON." The syntax is basically
erlson ::= '`#{`' [maplet {'`,`' maplet}*'`}`'
maplet ::= (atom|binary) ['`=`' expression]
where an omitted expression is, for the sake of option lists, `true`.

This would have been a workable syntax for frames *before* erlson used it. Now, it is important that the syntax should be *distinct* from erlson syntax, so that frames can be added to Erlang without breaking erlson. There is also an important pragmatic difference between frames and erlson. Frames are new built in data type with a good deal of thought put into making them economical in memory and time. "At runtime, Erlson dictionaries are represented as a list of {Name, Value} tuples ordered by Name. This way, each Erlson dictionary is a valid `proplist` and `orddict` in terms of the correspond[ing] `stdlib` modules." An entry in an Erlson dictionary thus takes 4 words for the {Name,Value} tuple plus 2 words for the list cell pointing to it, or 6 words compared with a frame's 1 or 2. It therefore matters which one you are using: a programmer needs to be able to tell at a glance whether an expression deals with a frame or with an erlson dictionary.

Anton Lavrik has given his permission for erlson syntax to be reused for frames, and if there are no other users of erlson whose code deserves protection all is well for the frame creation syntax. There is, however, one aspect of erlson syntax which we cannot afford to adopt, and that is dot notation. The problem is that erlson uses $D.x$ to access the x slot of dictionary D, and that this usage is recursive. That's not a problem in Java. But Erlang was designed for distributed programming, so node names like `mynode@potter.example.com` are valid Erlang identifiers without any quotes. This means that D.x.y could mean "the x.y slot of D" or "the y slot of (the x slot of D)". Considering

```
D = #{x.y = 1, x = #{y = 2}}
```

we see that both readings could make sense for the same dictionary, even given type information.

That doesn't mean that we can't adopt something with the same *structure* as dot notation for accessing a single slot, only that we can't use either '.' or '@' as the symbol for it. We could, for example, use an infix '#', writing D#x.y or D#x#y without ambiguity.

In May 2012 it became clear that there was going to be even more argument about the colour of paint on the Ark. Dear reader, can we please discuss semantics *first*? Pretty please with knobs on?

Let me demonstrate why the mechanics of typing are not terribly relevant. I have a text editor that was inspired by Emacs, but ran comfortably on PDP-11s. With the advent of VAXen (wow, 4MB of memory!) the command set was enlarged, but nowhere near the level of Emacs (which is, surprisingly, a smaller program than Vim). Now Emacs and Vim and OpenOffice and a whole lot of other editors have an abbreviation facility. It turns out that by turning the comma into a context-sensitive active character we can arrange for a bunch of things to be typed using extremely common characters, such that any keyboard

that doesn't support the characters we need is not going to be usable for Erlang anyway.

```
void comma(void) {
    if (argvalue == implicit) { /* No numeric argument */
        switch (fprev(here)) {  /* Expand a separator? */
            case '=':           /* For frames */
                delete(NORM, here-1);
                strinsert(1, 3, " ~ ");
                return;
            case ',':           /* For lists */
                delete(NORM, here-1);
                insert(1, '|');
                return;
            case ';':           /* For comprehensions */
                delete(NORM, here-1);
                strinsert(1, 4, " || ");
                return;
            case ':':           /* Why is equality =:= ? */
            case '/':           /* Why is inequality =/= ? */
                here--;
                insert(1, '=');
                here++;
                insert(1, '=');
                return;
            case '-':           /* For records */
                delete(NORM, here-1);
                insert(1, '#');
                return;
            case ' ':           /* For tidiness */
                here--;
                insert(1, ',');
                here++;
                return;
        }
        switch (fetch(here)) {  /* Move over a closer? */
            case ')':           /* For expressions */
            case ']':           /* For lists */
                here++;
                return;
            case '}':           /* For tuples and frames */
            case '>':           /* For binaries */
                here += fetch(here+1) == '>' ? 2 : 1;
                return;
        }
        switch (fprev(here)) {  /* Expand an opener? */
            case '<':           /* For binaries */
                strinsert(1, 3, "<<>>");
                here -= 2;
                return;
            case '[':           /* For tuples */
```

```
                delete(NORM, here-1);
                strinsert(1, 2, "{}");
                here -= 1;
                return;
            case '(':               /* For frames */
                delete(NORM, here-1);
                strinsert(1, 4, "<{}>");
                here -= 2;
                return;
        }
    }
    (*dispatch['A'])();
}
```

8.5 Comprehensions

The programming language Haskell has syntax for list comprehensions, and comprehensions may iterate over the elements of lists only. The related programming language Clean also has syntax for array comprehensions, and both kinds of comprehensions may iterate over the elements of arrays as well as the elements of lists. I have long regarded Erlang's Haskell-like limitations in this respect as a defect.

Therefore I considered whether there should be any special syntax for frame comprehension. In my judgement, the answer is *no*; that would be appropriate for `dict`-like uses of frames rather than the intended record-like uses. If this should ever be appropriate, list comprehension can be used in combination with `frame_to_list`/1 and `list_to_frame`/1.

I also considered whether there should be any syntax for iterating over the elements of a frame. Again my judgement was *no*, because that's appropriate for `dict`-like uses. It is always possible to use

```
K <- tuple_to_list(frame_keys(F))
```

or

```
{K,V} <- frame_to_list(F)
```

In this case there is clearly an optimisation to be made: it is possible to traverse a frame directly without building a list which has no enduring use. However, if a compiler is to optimise this, it may as well recognise these forms as any other. No syntax is required for what should be a rare use.

9 Pattern matching

pattern ::= '`<{`' [pmaplet {'`,`' pmaplet}*['|' tvar]$^{?}$]$^{?}$ '`}>`'
pmaplet ::= atom '`~`' pattern
tvar ::= variable | anonymous-variable

In a frame pattern, all of the keys must be atoms, and they must all be different. If there is a $|tvar$ part, the tail variable must be an anonymous variable or a variable which does not occur elsewhere in the same pattern and guard.

`<{` $k_1 \sim P_1, \ldots, k_n \sim P_n$ `}>` $= F$
has the effect of

$$
\begin{array}{l}
F' = F, \\
\texttt{is\_frame}(F'), \\
\texttt{frame\_size}(F') \geq n, \\
P_1 = \texttt{frame\_value}(F',\ k_1), \\
\vdots \\
P_n = \texttt{frame\_value}(F',\ k_n)
\end{array}
$$

where F' is a new variable.

The 2003 draft tested for the size being exactly n. A couple of years of experience writing code using this notation have convinced me that this is practically never what one wants.

`<{` $k_1 \sim P_1, \ldots, k_n \sim P_n | V$ `}>` $= F$
has the effect of

$$
\begin{array}{l}
F' = F, \\
\texttt{is\_frame}(F'), \\
\texttt{frame\_size}(F') \geq n, \\
P_1 = \texttt{frame\_value}(F',\ k_1), \\
\vdots \\
P_n = \texttt{frame\_value}(F',\ k_n), \\
V = \texttt{frame\_without}(F',\ [k_1, \ldots, k_n])
\end{array}
$$

where F' is a new variable. If the variable V is an anonymous variable, the call to `frame_without`/2 is omitted. If the variable V is not an anonymous variable, the call to `frame_without`/2 (which is by now known to be safe) is postponed until *after* the guard.

The cost of matching a frame pattern with n keys against a frame value with m keys is $O(n + m)$ plus the cost of matching the element patterns. In fact we can do better: $O(\min(n + m, n \lg m))$. We expect that `<{timeout~T|_}>` $= F$ will take $O(\lg |F|)$ time, the same as `T` $=$ `frame_value`$(F$,`timeout`$)$ (and in fact a little faster, because the compiler will verify that `timeout` is an atom, while `frame_value`/2 would check at run time).

9.1 A little problem

In 2003, a frame pattern with no tail part required the size to match exactly. This is almost never what you want. The way record syntax is used in Erlang, Haskell, Clean, and O'CAML, record patterns require matches for the specified keys but allow other keys to be present as well.

In 2011, a frame pattern with no tail part allows extra keys. This destroys the symmetry between patterns and expressions, which is somewhat distasteful. However, we *already* have that with -record syntax, and it doesn't seem to be a problem with other languages. It is less distasteful than the very high error rate I found to occur with the exact size convention.

This leaves us with no obvious syntax for "a frame with this set of keys and no others". A guard test is required. Since I have never found myself wanting this, that seems acceptable.

9.2 Excess generality

Do we really need the ability to select "the rest of the frame"? There is nothing like that in records. Including that feature means that pattern matching may allocate memory, something which cannot happen in current Erlang. (Guards may temporarily allocate memory, but not patterns.) Removing this feature simplifies the implementation and means that the distinction between ⟨{...}⟩ and ⟨{...|_}⟩ never arises because the second is not so expressible.

The function `frame_without`/2 would still be available, it just wouldn't be expressible as a pattern match.

Let it be so! From May 2012 on, a frame pattern is just
pattern ::= '`<{`' [pmaplet {'`,`' pmaplet}*]? '`}>`'
with no tail. This aligns frame patterns with record patterns in Haskell, ML, Clean, and above all, Erlang.

9.3 Single-slot selection

The existing -record notation lets you select a single field from a record. We could give ~ a unary syntax for this purpose:
funcname ::= atom | ~ atom | ...

A function like `~size` would act as if there were a definition

```
~size(<{size ~ Size}>) -> Size.
```

That is not part of the present proposal because I have found that pushing all field selection into clause heads seems to result in better code. Compare for example

```
mag1(C) ->
    math:sqrt(~re(C)*~re(C) + ~im(C)*~im(C)).

mag2(<{re ~ R, im ~ I}>) ->
    math:sqrt(R*R + I*I).
```

Doing all the matching in the head means that if the value passed is not a frame, or has no re field, or has no im field, that's caught right away. Using selector functions in the body of a clause delays this checking, possibly until after some side effect that should not have been performed.

9.4 Inline caches and some experimental results

Since accessing a single field is the worst case for frames, it seemed good to get some measurements. Two functions, g1/3 and g2/3, were measured:

```
-record(r, {ack,ick,uck}).

f1() ->
```

```
    g1([#r{ack = I, ick = I, uck = I}
        || J <- lists:seq(1, 1000000), I <- [J rem 256]]).

g1(L) ->
    g1(L, 0, 0).

g1([], Sum, Max) ->
    io:write({Sum,Max}), io:nl();
g1([R|Rs], Sum, Max) ->
    T = R#r.uck,
    g1(Rs, Sum + R#r.ack, if T > Max -> T; true -> Max end).

f2() ->
    g2([<{ick = I, ack = I, uck = I}>
        || J <- lists:seq(1, 1000000), I <- [J rem 256]]).

g2(L) ->
    g2(L, 0, 0).

g2([], Sum, Max) ->
    io:write({Sum,Max}), io:nl();
g2([R|Rs], Sum, Max) ->
    g2(Rs, Sum + ~ack(R), Max max ~uck(R)).
```

Of course, only g1/3 could be measured in a real Erlang system.

Inline caching is an old idea from Smalltalk implementations, see *Efficient Implementation of the Smalltalk-80 System* by L. Peter Deutsch and Allan M. Schiffman in *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, 1983. Most compiled Smalltalk systems use inline caching — my Smalltalk does not.

In Smalltalk, sending the message m to the receiver r is written $r\ m$. For example, extracting the x coördinate of a point might be written

```
aPoint x
```

and compiled to the byte codes

```
pushTemp 0  "where temp0 is aPoint"
send x
```

and thence to native code that does

```
T = temp 0
C = class(T)
if (C == cached_class) {
    F = cached_func
} else {
    cached_func = lookup(method_literal_x, C)
    cached_class = C
}
T = F(T)
```

In a single-cpu implementation, the same idea can obviously be applied to selecting slots from frames. In a multi-cpu implementation, it's not so obvious, but given atomic 64-bit loads and stores together with 32-bit addresses, it can be done without locking. Since this is just a cache, we don't *care* if updates to the cache are not propagated to other processors, so not even memory barriers are needed. So it seemed reasonable to try out simple inline caches to see how much of an improvement might be available.

There were thus three versions, where g2 searched for a slot at run time, while g3 checked an inline cache first. The sources were compiled by hand to an imaginary BEAM-like instruction set. The three arguments are placed in registers R1, R2, R3.

```
g1: switch_on R1, fail, L1, L2, fail
L1: check_atom R1 nil
    print R2
    print R3
    halt
L2: load R4, R1, 0W-2
    load R1, R1, 1W-2
    check_record R4, 'r', 4
    load R5, R4, 2W-3
    add R2, R2, R5
    load R5, R4, 4W-3
    max R3, R3, R5
    goto g1


g2: switch_on R1, fail, L1, L3, fail
L3: load R4, R1, 0W-2
    load R1, R1, 1W-2
    check_frame R6, R4
    load_slot R5, R4, R6, 'ack'
    add R2, R2, R5
    load_slot R5, R4, R6, 'uck'
    max R3, R3, R5
    goto g2

g3: switch_on R1, fail, L1, L4, fail
L4: load R4, R1, 0W-2
    load R1, R1, 1W-2
    check_frame R6, R4
    cached_load_slot R5, R4, R6, 'ack', 0, 0
    add R2, R2, R5
    cached_load_slot R5, R4, R6, 'uck', 0, 0
    max R3, R3, R5
    goto g3
```

These were executed by a fairly simple-minded thread-code emulator that could execute no instructions outside these examples and held all registers in an array.

They were also compiled by hand into C code mechanically following a simple macro-expanding approach, except that the registers were held as local variables in the functions.

The interpreter and C functions were compiled using i686-apple-darwin10-llvm-gcc-4.2 (GCC) 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.9) on a MacBook Pro with a 2GHz Intel Core i7. Only a single core was used. All times are in nanoseconds for a single iteration, thus reflecting the time to fetch two fields and do some arithmetic.

"BEAM"	"HiPE"	function
38.8	7.2	g1, real Erlang R14A
21.4	3.8	g1 (records)
22.2	6.4	g2 (frames, linear search)
20.7	5.0	g3 (frames, inline cache)

Analysis starts with a disclaimer. The BEAM-like emulator I wrote does not handle floats or bignums and ignores integer overflow. It has a max instruction which likewise handles only 30-bit integers. The real Erlang system suffers from not having a `max` instruction, so max has to synthesised from 5 instructions. It also has to be ready for an allocation and garbage collection. If the BEAM-like emulator were extended to handle all the needs of a real system, it would end up *being* BEAM, or very like it. One simple difference is the fact that the experimental emulator, by virtue of emulating only a fragmentary instruction set, is far smaller than the real thing, so gets into cache and stays there.

It is astonishing that frames (with in-line cache) should be faster than records, but the result was repeatable. I suspect some sort of resonance effect. What is clear at a glance is that the cost of using frames is real but small compared with the cost of emulation. HiPE is the one to worry about.

According to Apple's Instruments utility, the "HiPE" version of g3 spent 45% of its time checking number tags, 15% of its time checking frame tags, 11% of its time looking in the inline caches, and 29% of its time doing useful work. The "HiPE" version of g1 spent 28% of its time checking number tags, 31% of its time checking record tags, sizes, and labels, and 41consistent with the useful work taking about 1.5 nsec in both g1 and g3.

As we shall see below, static type information could eliminate all the overheads for records or frames, but since the integer() type includes both fixnums and bignums, it could not eliminate all of the number checking costs.

How realistic was it to use a record with 3 fields? I collected 2311 -record declarations from many thousands of Erlang source files, and got these figures:

- the commonest number of fields is 2,
- the median number of fields is 3,
- the mean number of fields is 5,
- three quarters of the records have 6 or fewer fields,
- 97% have 16 or fewer fields, and
- there is one monster (ssh) with 49 fields.

This is a static count of record declarations, not a dynamic count of record uses, but it offers some guidance.

10 Why not use the `dict` module?

Chris Pressey has forcibly argued in the Erlang mailing list that the `dict` module should be used. He has made no objection to providing intention-revealing syntax for frame construction and pattern matching. His view is that the compiler could support such syntax on top of whatever concrete implementation `dict` uses; and if that implementation is not satisfactory, then `dict` could have a new implementation, that being the point of using abstract data types.

The basic answer is that `dict` and frames are designed to do different jobs. They have different implementation tradeoffs. You can't make `dict` good for the record-like uses that frames are meant for without making it bad for its existing uses.

1. When we know that something is intended to be used in a record-like way, we can use a more space-efficient representation. For example, the record type used by the current implementation of `dict` would require a minimum of 63 cells (assuming zero garbage collector overhead) if stored as a `dict`; if stored as a frame only 9 cells are required.

2. When we know that something is intended to be used in a record-like way, we can use a more time-efficient representation. (Copying 9 cells is faster than copying 63, and no call to `erlang:phash/1` is required.)

3. The implementation of `dict` must use whatever data structures Erlang can offer it. It cannot be better than those data structures. In particular, since existing Erlang compound data types do not change their representations at run-time, neither can `dict`.

4. If the implementation of `dict` is changed to be suitable for record-like uses (lots of small objects, lots of objects with the same set of keys, lots of batch updates that change a significant fraction of the fields), it will be less suitable for its existing uses.

Chris Pressey also suggested using the ETS implementation as the basis for frames. The same objection applies again: ETS tables are engineered for certain uses, making them as bad as they could possibly be for a record replacement. As the reference manual says, "These provide the ability to store very large amounts of data". Frames are supposed to be small, cheap, fast, immutable. You are supposed to be able to have millions of them. Quoting the ETS manual again, "The number of tables stored at one Erlang node is limited. The current default limit is approximately 1400 tables." And, quite disastrously for the intended use of frames, "Note that there is no automatic garbage collection for tables."

ETS tables are important and useful, but they are about as far from what we want here as they could possibly be.

Using atoms as keys also turns out to be useful for efficient implementation. Not absolutely necessary, just useful.

There is another reason, which I have not discussed before. One of Erlang's strengths is interoperability with other languages. Erlang numbers, atoms, and tuples correspond well to Perl numbers, strings, and lists, for example, and equally well to Smalltalk Numbers, Symbols, and Arrays. A common data

structure in "scripting" languages is a dictionary mapping strings to values. This is one of the types supported in the YAML protocol for exchanging data values, for example. Perl has hashes, Smalltalk has IdentityDictionary, which admits Symbols as keys. Allowing keys other than atoms would generalise frames a little too far for interoperability. If frames are added to Erlang, they should be added to UBF as well, making UBF more useful for languages other than Erlang.

Frames are perfectly suited to representing attribute-value associations for SGML parse trees. Thanks to XML namespaces, they are less well suited to XML attributes, but then, XML namespaces make practically everything hard.

11 Frames as mini-modules

We often need to pass functions to other functions. Sometimes we need to pass a bundle of related functions. Erlang behaviours, for example, have handled this by requiring you to pass the name of a callback *module*. It would be useful if we could pass a bundle of functions as a frame, and use it for callbacks as if it were a module.

So far, we have not specified the effect of $E_0 : F(E_1, \ldots, E_n)$ when E_0 evaluates to a frame. Let us define it to be

> `case` E_0
> `of` $\langle\{F \sim \Phi|\alpha\}\rangle \rightarrow \Phi(E_1, \ldots, E_n)$
> `end`

Now we can pass bundles of functions like

```
M = <{ start  ~ fun () -> ... end
     , stop   ~ fun () -> ... end
     , handle ~ fun (Event) -> ... end
    }>,
```

and use them as if the bundle were a module:

```
M:start(),
M:handle(Event),
M:stop()
```

The one way in which this fails to replace modules is that these functions are not recursive. In many cases that will not matter. In some cases, it may.

One simple device would be to change the definition above to

> `case` E_0
> `of` $V_0 = \langle\{F \sim \Phi|\alpha\}\rangle \rightarrow \Phi(V_0, E_1, \ldots, E_n)$
> `end`

after which we could write

```
M = <{ even ~ fun (M, X) -> ... M:odd(Y) ... end
     , odd  ~ fun (M, Y) -> ... M:even(X) ... end
    }>,
U = M:even(X)
```

so no new machinery is needed. (This is a bit like the 'self' parameter in Oberon.) We could even make the choice of passing V_0 or not depend on the arity of Φ: if $n+1$, pass V_0; if n, don't; if anything else, report an error.

12 Frames and JSON

Douglas Crockford "found [JSON], named it, [and] described how it was useful", 2001. It was about that time that I started thinking about frames. Even when the first draft of this document was written in 2003, I was ignorant of JSON and knew very little of JavaScript. IBM Prolog, LIFE's ψ terms, linguist's feature structures, SGML element attributes, SNOBOL and Icon tables, frames, the definition of MESA semantics, Pebble, ... Frames congealed from a vast stew of programming languages. But there was no trace of JavaScript in that stew, and JSON compatibility has never been a goal of the frames design. The goal was to adapt existing ideas from programming languages to find something that could be used to replace records in Erlang.

By a lucky accident, it turns out that frames naturally plug a gap in Erlang representations of JSON data. It must be emphasised that this *is* just a lucky accident, and the imperative language JavaScript must not be allowed to dictate Erlang syntax.

These days JSON is a popular way to send data around the Web and hold it in key-value stores. JSON values are

- null: use `null`
- false: use `false`
- true: use `true`
- numbers: use Erlang numbers, remembering that JSON doesn't actually have integers
- strings: use Erlang binaries
- keys: use atoms for dictionary keys
- arrays: use lists or tuples
- dictionaries: frames are the perfect fit.

The keys in a JSON dictionary must be strings, which can be converted to atoms. JSON dictionaries are usually used for small to medium scale records, so they are the right size to be handled as frames.

The major problem with mapping a JSON dictionary like

```
{"id": 4135, "title": "title2"}
```

to an Erlang frame like

```
<{id ~ 4135, title ~ <<"title2">>}>
```

is the bounded size of Erlang's atom table. However, that is already the subject of an EEP. Until that's fixed, the use of atoms from untrusted sources is a serious vulnerability.

The minor problem is that JSON data often contains many dictionaries with the same keys, and while correct, it is inefficient to ignore this duplication. A JSON decoder ought to maintain a cache of frame descriptors for re-use.

13 Implementation

There are many ways to implement frames.

An obvious way is to use a representation identical to tuples except for the tag on the pointer. Keys and values would be stored with the keys in strictly ascending order. In effect,

$$\langle n, K_1, V_1, \ldots, K_n, V_n \rangle.$$

This requires $2n + 1$ cells plus the usual overheads, and admits both binary search (useful when n is moderately large) and linear search (useful when n is small). Updating such a term requires copying the whole thing.

Another approach would be a binary search tree, which would admit binary search but not linear search. A single update to such a term requires $O(\lg n)$ new space; however for record-like use it is common to update several fields at once, and the space costs could work out higher. Comparing binary trees is trickier than comparing linear sequences.

Another approach would be to use some kind of hash table, so that lookup could be $O(1)$. Comparing hash tables is much trickier than comparing linear sequences. Again, a tuple-like representation with a different tag could be used, and since these objects are fairly small, linear probing with no overflow chains would do.

As a reference point for hash tables, I examined the Java 1.6 HashMap class. For the sake of argument, I assume a 32-bit system with a single header word per object. An *empty* HashMap takes 24 words and each additional entry costs 5 words, not counting more space for the hidden array. That is far too high for this task.

Importantly, as a data type whose representation is completely hidden from the programmer, *several* different representations could be used, depending perhaps on the number of keys.

While Erlang terms may not be changed, this is true at the language level. At the implementation level, a frame could change its representation depending on how it is used, although that would not be kind to the garbage collector.

There are several "functional array" implementation techniques in the literature which could be used. In fact, there are existing proposals that some such scheme *should* be used for tuples so that `setelement`/3 could be $O(1)$, without ceasing to be functional. For example, frame update using `<{`$\ldots|\ldots$`}>` could be lazy; at first you would get just a description of the change, but when the changed frame was matched or scanned (as opposed to used as the basis for further changes) the change would be actually carried out.

What's needed is a notation and an implementation which are sufficiently good to replace records in everyday use. Record update works by copying the whole record, so we can tolerate that for frames.

The representation I propose exploits the fact that most frames will either be produced by a `<{`$\ldots$`}>` form that constructs a whole new frame or by a `<{`$\ldots|F$`}>` form which updates fields that already exist in F rather than adding new fields.

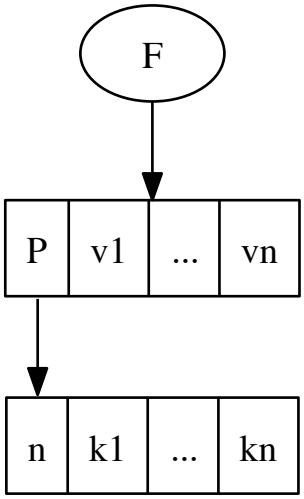
A frame $F = \langle \{k_1 \sim V_1, \ldots, k_n \sim V_n\} \rangle$ is represented by a tagged pointer to

a vector[3]

$$\langle P, V_1, \ldots, V_n \rangle$$

where P points to a tuple

$$\{k_1, \ldots, k_n\}.$$



The expression `<{`$k_1 \sim E_1, \ldots, k_n \sim E_n$`}>` might compile to code like this:

push value of E_1;
⋮
push value of E_n;
permute top n elements as necessary;
push literal $\{k'_1, \ldots, k'_n\}$;
call `make_frame`

where the literal is sorted. The permutation may be necessary because Erlang has strict left-to-right evaluation order, so that the order in which the expressions must be evaluated will not in general be the same as the order in which the values are to be stored.

The key tuple literal is part of the module's constant pool. All the frame-making expressions in a module with the same set of keys could and should share the same key tuple. While making key tuples unique across a whole node might make equality tests fast, frame comparisons are expected to be rare. They have to be correct; they do not need to be optimised. Considering tuple extension, I used to believe that making key tuples unique would cost more than it is worth. Considering the potential benefits of inline caches, I suspect that I was wrong.

[3] *not* a tuple; there is no size field

The expression `<{`$k_1 \sim E_1|F$`}>` might compile to code like this:

push value of E_1;
push value of F;
push literal k_1;
call `update_frame_1`

The expression `<{`$k_1 \sim E_1, k_2 \sim E_2|F$`}>` might compile to code like this:

push value of E_1;
push value of E_2;
maybe swap the two 2 items;
push value of F;
push literal k'_1;
push literal k'_2;
call `update_frame_2`

The swap would be needed if $k_1 > k_2$.

The expression `<{`$k_1 \sim E_1, \ldots, k_n \sim E_n|F|$`}>` might compile to code like this:

push value of E_1;
$\vdots$
push value of E_n;
permute top n elements as necessary;
push value of F;
push literal $\{k'_1, \ldots, k'_n\}$;
call `update_frame`

The `update_frame` operations may do linear search for small frames; for larger frames the fact that the key tuple for update is sorted means that it is possible to use linear-time merge.

If it turns out that $k'_1, \ldots, k'n$ are all already among the keys of F, the result can point to F's key tuple, and this should be the fast path, otherwise a new key tuple must be constructed.

The `is_frame`/1 test is a tag test, or possibly a tuple+frame tag test plus a (first word is atom, not fixnum) test. Both are simple. If `is_frame`/2 is adopted, it needs to check the tag, to check the identity of the first element of the key tuple, and to check the identity of the first element of the payload.

The `frame_size`/1 function returns the tuple size of the keys tuple.

The other functions are obvious combinations of linear scan, linear merge, binary search, and sorting (`list_to_frame`/2, `frame_without`/1).

Pattern matching requires special care, but linear merge will do the job. A naïve approach would turn

$$\texttt{<\{}k_1 \sim P_1, \ldots, k_n \sim P_n\texttt{\}>}$$

into

push term to be matched;
push literal $\{k'_1, \ldots, k'_n\}$;
call `match_frame`;

match P_n against pop();
⋮
match P_1 against pop();

Cleverer approaches are reasonably straightforward. For example, we could use

push term to be matched;
call `check_tuple_size` n;
call `previous_field` k_n;
match P_n against pop();
⋮
call `previous_field` k_2;
match P_2 against pop();
call `earliest_field` k_1;
match P_1 against pop();

where

`check_tuple_size` checks the tuple size and pushes it in the stack, so the top of the stack is *tuple, size.*

`previous_field` requires the top of the stack to be *tuple, #fields left.* It scans backwards for a matching field, leaving the stack as *tuple, #fields left′, value of field* if it finds a match, or fails if it doesn't.

`earliest_field` is like `previous_field`, but removes the tuple and field count from the stack.

13.1 Why atoms, again?

Atoms are, in effect, strings. There are many implementation choices for atoms. Quintus used tagged indices into an extensible table which could grow very large. Older Erlang implementations used tagged pointers to records in a hash table, which could *not* grow very large. The Logix implementation of Flat Concurrent Prolog basically treated atoms as tagged pointers to strings with embedded hash values.

As Erlang is currently implemented, you can test whether two atoms are the same atom or not simply by comparing machine words. This means that for small frames, a linear search for a key can be very efficient.

Joe Armstrong has recently suggested in the Erlang mailing list that the internal representation of atoms should be changed to allow large numbers of atoms to be created cheaply. Having a global atom dictionary means that atoms can be represented uniquely, which makes atom matching very fast. But a global atom dictionary is a shared mutable resource, which requires locking. That's not good for a highly concurrent language. It also requires an atom table garbage collector, which either suspends all other processes, or has to be a fairly complex piece of code running in parallel with the other threads.

If the atom '$c_1 c_2 \ldots c_n$' were represented by a tagged pointer to a block like

$$\langle n, h, c_1 c_2 \ldots c_n \rangle,$$

where n is the number of characters, h is a hash code, and c_1 to c_n are the characters of the atom's name, but these blocks were not unique, then atom equality tests would be slower than they are now, but not prohibitively slow.

I wrote a C program that read 10,000 randomly selected dictionary words, computed size,hash,text representations as shown above, and then did all pairwise comparisons using these two functions:

```
int eqp(struct Word const *p, struct Word const *q) {
    return p == q;
}

int eqh(struct Word const *p, struct Word const *q) {
    return p->hash == q->hash
        && p->size == q->size
        && 0 == memcmp(p->text, q->text, p->size);
}
```

To get some idea of what might happen to atom ordering (or at any rate to an ordering that might be useful for frames), I also measured all comparisons using these functions:

```
int cps(struct Word const *p, struct Word const *q) {
    return strcmp(p->c, q->c);
}

int cph(struct Word const *p, struct Word const *q) {
    return p->h == q->h ? strcmp(p->c, q->c)
         : p->h > q->h ? 1 : -1;
}
```

On a 500MHz SunBlade-100, the measured times were

	cc -xO3	gcc -O6
eqp (inline)	0.4 sec	0.4 sec
eqp (called)	2.5 sec	2.4 sec
eqh (inline)	6.3 sec	7.0 sec
eqh (called)	7.3 sec	7.3 sec
cps (inline)	11.1 sec	11.2 sec
cps (called)	14.8 sec	12.9 sec
cph (inline)	6.6 sec	6.9 sec
cph (called)	9.0 sec	8.0 sec

These times include loop overhead, which is fair, because code scanning a key tuple looking for a key would also include loop overhead. This suggests that using non-unique atoms might make atom equality tests about 7 times more expensive. The good news is that ordering atoms by their hash code and breaking ties by looking at text could make ordering faster, and that this is an ordering that can be computed by an off-line compiler.

The fastest way to compare atoms would be if they were stored uniquely and atom table garbage collection were guaranteed to preserve the relative order of the blocks representing atoms. Then field names could be sorted by address, and compared using single machine instructions. That would require a different approach to compilation, like Kistler's Juice system for Oberon, where the

compiler does semantic checks and outputs a compressed parse tree, and the "loader" generates the actual instructions. That would be a good idea in any case, but it would be a big change to Erlang implementations.

If keys could be arbitrary terms, key comparison would get even more expensive.

14 Binary Encoding

It is necessary to define the representation of frames in the Erlang binary encoding.

Specification of Erlang 4.7, Final Draft (0.6), by Jonas Barklund, describes the Erlang binary encoding in appendix E. The encoding is used in two ways: for sending down a channel to another node, and for converting a term to a binary. Each channel is separately stateful (the atom table), so a term transmitted twice down the same channel may have a different representation each time. Conversion to binary starts in the same state each time, so every time a term is converted to binary the same binary results (although a repeated subterm may be represented differently each time).

Erlang 4.7 binaries begin with the byte 131, to indicate the version. The R9C release of Erlang/OTP still uses that version byte, although some of the details appear to have changed.

The representation of each type of term begins with a distinct byte:

67	reference to atom in atom table
78	atom, to be inserted into the atom table
97	one-byte non-negative integer
98	four-byte signed integer
99	floating-point number as a decimal character string
100	atom, not to be inserted into the atom table
101	reference
102	port
103	process ID
104	tuple
106	empty list
107	string of Latin-1 characters (list of bytes)
108	other list
109	binary
110	signed integer up to 256 bytes
111	even larger signed integer
112	appears to be used for `fun` terms

To this we simply add

113	frame

or, in full,
$TR_{4.7}(\langle\{k_1 \sim v_1, \ldots, k_n \sim v_n\}\rangle) =$
$\langle 113\rangle BE(4,n)TR_{4.7}(k_1)\ldots TR_{4.7}(k_n)TR_{4.7}(v_1)\ldots TR_{4.7}(v_n)$,
where the keys are in sorted order.

That's all we actually need to do. However, the 4.7 binary representation has some problems. It is bulkier than it needs to be, especially for floating-point numbers; it often uses 4 bytes where fewer would do; it is often limited

to 4 bytes where the increasingly common 64-bit machines might have a use for more bytes.

Another issue is that if we send several frames with the same keys, we don't want to keep on sending the same key sequence over and over again, just like we don't want to keep on sending the full text of the same atom over and over again.

14.1 A new binary encoding

Terms are encoded over stateful channels. The state of a channel consists of a term dictionary (an array of remembered terms) and a character encoding state as described in Unicode Technical Report 6, *A Standard Compression Scheme for Unicode*. The term dictionary is initially empty, and the initial character encoding state is as described in UTR-6. As in Erlang 4.7, transmitting a term to another node uses (and updates) the state of the appropriate channel, while `term_to_binary`/1 uses a fresh channel each time.

The notation $x : y$ means $x * 16 + y$. For example, $1 : 3$ means $1 * 16 + 3 = 19$.

The notation $x \perp n$ where $n \geq 0$ means

> if $n < 8$ then $x : n$ else $x : (L + 7), n_{L-1}, \ldots, n_0$ where L is the number of bytes required to hold n, n_{L-1} is the most significant byte of n, and n_0 is the least significant byte of n.

$0 \perp n, c_1, c_2, \ldots, c_n$ The atom '$c_1 c_2 \ldots c_n$'. n is the number of characters, not the number of bytes. The characters are encoded using the method of Unicode Technical Report 6; the state of that encoding persists throughout a binary stream.

$1 \perp n, c_1, c_2, \ldots, c_n$ The string "$c_1 c_2 \ldots c_n$". The characters are encoded like the characters of an atom. This represents a list of integer codes.

$2 \perp n, t_1, t_2, \ldots, t_n$ The list $[X_1, X_2, \ldots, X_n]$, where t_i is the encoding of X_i. In particular, $2 \perp 0$ represents [].

$3 \perp n, t_0, t_1, \ldots, t_n, t_z$ The dotted list $[X_0, X_1, \ldots, X_n | X_z]$, where t_i is the encoding of X_i. X_z should not be a proper list. Note that $3 \perp 0, t_0, t_z$ represents $[X_0 | X_z]$; it does not represent an empty list.

$4 \perp n, t_1, t_2, \ldots, t_n$ The tuple $\{X_1, X_2, \ldots, X_n\}$, where t_i is the encoding of X_i.

$5 \perp n, 0, k_1, \ldots, k_n, v_1, \ldots, v_n$ The frame $\langle \{K_1 \sim V_1, \ldots, K_n \sim V_n\} \rangle$, where k_i (v_i) is the encoding of K_i (V_i) and the K_i are distinct atoms in increasing order.

$5 \perp n, D \perp m, k_1, \ldots, k_n, v_1, \ldots, v_n$ as above, but the descriptor $\langle k_1, \ldots, k_n \rangle$ is also put in the reuse dictionary (for Scheme 1 below, $m = 0$; for Scheme 2 below, m is the slot index).

$5 \perp n, E \perp m, v_1, \ldots, v_n$ as above, but uses the descriptor at slot m in the reuse dictionary.

$6 \perp n, m, f, a, t_1, \ldots, t_n$ The closure
`fun` $(Y_1, \ldots, Y_a)$ `->` $M : F(Y_1, \ldots, Y_a, X_1, \ldots, X_n)$ `end`,
where m, f, a, t_1, ..., t_n are the encodings of module name M, function name F, integer arity A, and closed values $X_1, \ldots, X_n$ respectively. I'm not really pleased with this. In particular, it seems that `fun` $M : F/A$ terms and anonymous `fun`s should have different representations.

$7 \perp 0$

$7 \perp 1$ **IEEE-754 32-bit float in big-endian order**

$7 \perp 2$ **IEEE-754 64-bit float in big-endian order**

$7 \perp 7$ **28-byte decimal format float** Floating point numbers. Here n is the number of 32-bit words in the representation. $7 \perp 0$ represents +0.0. If a number can be converted to 32-bit format without loss of precision, $7 \perp 1$ form is used, otherwise $7 \perp 2$. This doesn't allow for non-IEEE formats or for long double; other variants of $7 \perp k$ could be used. It would also be possible to use $7 \perp 7$ for the FloatString() format of Erlang 4.7, with trailing NULs truncated to fit into 28 bytes (7 words).

$8 \perp n, v_{n-1}, \ldots, v_0$ The non-negative integer $v_{n-1}.256^{n-1} + \ldots + v_0.256^0$, where the v_i are the bytes of the number in big-endian order; n being the smallest possible value. In particular, $8 \perp 0$ is 0.

$9 \perp n, v_{n-1}, \ldots, v_0$ The negative integer whose twos-complement representation is the bitwise complement of $v_{n-1}.256^{n-1} + \ldots + v_0.256^0$. In particular, $9 \perp 0$ is -1.

$A \perp n, b_1, b_2, \ldots, b_n$ The binary `<<`$b_1, b_2, \ldots, b_n$`>>`

$B \perp n, 0, b_1, b_2, \ldots, b_n$,*node*

$B \perp n, 1, b_1, b_2, \ldots, b_n$,*node*

$B \perp n, 2, b_1, b_2, \ldots, b_n$,*node* A reference (0), port (1), or process ID (1) created at a node the encoding of whose name is *node*, with n bytes of additional binary information. According to the Erlang 4.7 specification, n should be something like 5 or 6. This encoding should *not* be fused with the atom coding used for the node name, because by keeping the node name separately encoded we can use back references to keep node names short.

There are two possible schemes for back references.

14.1.1 Scheme 1

Scheme 1 is a 1-pass method for economising on "name-like" constants.

We keep a circular buffer of 256 atoms, references, ports, and PIDs.

$C \perp n$ where $0 \leq n < 256$ refers to the nth preceding entry in that buffer. This is similar to the atom table used in Erlang 4.7, except that doing the encoding requires a costly search. I have implemented this scheme, with $C \perp n$ not changing the circular buffer, and found it to work reasonably well. Using a cleverer data structure could reduce the code of the encoding.

14.1.2 Scheme 2

Scheme 2 is a 2-pass method inspired by Common Lisp and by UBF.

We keep a dictionary of arbitrary terms, of any convenient size (say up to 2^{16} terms. A preliminary pass over a term discovers frequent subterms. The first reference to a frequent subterm also enters it in the dictionary; later references only supply the dictionary index.

$D \perp n, t$ represents term T, where t is the normal encoding of T, and also sets slot n of the dictionary to hold T.

$E \perp n$ pulls out entry n of the dictionary.

This is similar to the use of the atom table in Erlang 4.7, except that it allows the dictionary to have more than 256 entries, and that it allows entries to be other than atoms. For example, they could be PIDs or pairs.

14.2 UBF

Joe Armstrong's UBF seems to work pretty well as a "binary" representation. Adding frames to that requires at least one more reserved character, which for argument's sake I've chosen to be the exclamation mark. The simplest approach would seem to be to start out like a tuple with alternating keys and values, and end with a different character. For example, $\langle\{a \sim 1, b \sim 2, c \sim 3\}\rangle$ might be encoded as `{a 1 b 2 c 3!`.

14.3 Type Inference

Consider the following extract from the `file_io_server` module, converted to use frames.

```
do_start(Spawn, Owner, FileName, ModeList) ->
    ...
        server_loop(<{ handle ~ Handle, owner ~ Owner, mref ~ M,
                       buf ~ <<>>, read_mode ~ ReadMode,
                       unic ~ Unicode_Mode }>)
    ...

server_loop(State = <{mref ~ Mref}>) ->
    receive
        {file_request, From, Reply_As, Request}
          when is_pid(From) ->
            case file_request(Request, State)
              of {reply, Reply, New_State} ->
                 file_reply(From, Reply_As, Reply),
                 server_loop(New_State)
               ; {error, Reply, New_State} ->
                 file_reply(From, Reply_As, Reply),
                 server_loop(New_State)
               ; {stop, Reason, Reply, _New_State} ->
                 file_reply(From, Reply_As, Reply),
```

```
                exit(Reason)
            end
      ; {io_request, From, Reply_As, Request}
          when is_pid(From) ->
            case io_request(Request, State)
              of {reply, Reply, New_State} ->
                 io_reply(From, Reply_As, Reply),
                 server_loop(New_State)
               ; {error, Reply, New_State} ->
                 io_reply(From, Reply_As, Reply),
                 server_loop(New_State)
               ; {stop, Reason, Reply, _New_State} ->
                 io_reply(From, Reply_As, Reply),
                 exit(Reason)
            end
      ; {'DOWN', Mref, _, _, Reason} ->
            exit(Reason)
      ; _ ->
            server_loop(State)
    end.

file_reply(From, ReplyAs, Reply) ->
    From ! {file_reply, ReplyAs, Reply}.

file_request(
    {advise,Offset,Length,Advise},
    State = <{handle ~ Handle}>
) ->
    Reply = ?PRIM_FILE:advise(Handle, Offset, Length, Advise)
    case Reply
      of {error,_} ->
         {stop,normal,Reply,State}
       ; _ ->
         {reply,Reply,State}
    end;
file_request(
    {pread,At,Sz},
    State = <{handle ~ Handle, buf ~ Buf, read_mode ~ ReadMode}>
) ->
    case position(Handle, At, Buf)
      of {ok,_Offs} ->
         case ?PRIM_FILE:read(Handle, Sz)
           of {ok,Bin} when ReadMode =:= list ->
              std_reply({ok,binary_to_list(Bin)}, State)
            ; Reply ->
              std_reply(Reply, State)
         end
       ; Reply ->
         std_reply(Reply, State)
    end;
```

```
file_request(
    {pwrite,At,Data},
    State = <{handle ~ Handle, buf ~ Buf}>
) ->
    case position(Handle, At, Buf)
      of {ok,_Offs} ->
         std_reply(?PRIM_FILE:write(Handle, Data), State)
       ; Reply ->
         std_reply(Reply, State)
    end;
...
file_request(
    close,
    State = <{handle ~ Handle}>
) ->
    {stop,normal,?PRIM_FILE:close(Handle),no_buf(State)};
...
file_request(
    truncate,
    State = <{handle ~ Handle}>
) ->
    case Reply = ?PRIM_FILE:truncate(Handle)
      of {error,_Reason} -> {stop,normal,Reply,no_buf(State)}
       ; _               -> {reply,Reply,State}
    end;
file_request(Unknown, State) ->
    {error,{error,{request,Unknown}},State}.

std_reply(Reply = {error,_}, State) ->
    {error,Reply,no_buf(State)};
std_reply(Reply, State) ->
    {reply,Reply,no_buf(State)}.

no_buf(State) ->
    <{ buf ~ <<>> | State }>.
```

This displays a common pattern. There is a tail recursive loop. One of its arguments is a "state" frame. The loop receives a message and calls a dispatcher function, which may terminate or iterate with the same state or a revised state. Auxiliary functions may be called, and revised states may be returned as elements of tuples.

To make this efficient, we could use static type information. We might, for example, have

```
-type read_mode :: 'list' | 'binary'.
-type state()   :: <{ handle ~ handle(), owner ~ pid(),
                      buf ~ binary(), read_mode ~ read_mode(),
                      mref ~ ref(), unic ~ atom() }>.
-type request() :: {'advise', integer(), integer(), any()}
```

```
                | {'pread', integer(), integer()}
                | ...
-type response():: {'stop','normal',reply(),state()}
                | {'reply',reply(),state()}
                | {'error',any(),state()}.

-spec server_loop(state()) -> void().
-spec file_request(request(), state()) -> response().
-spec std_reply(reply(), state()) -> response().
-spec no_buf(state()) -> state().
```

When frames were first devised, there were experimental type checkers for Erlang. Now the Dialyzer exists and is in common use. Using statically checked types of this kind would let a match like

```
<{ handle ~ Handle, buf ~ Buf }> = State
```

be compiled as

```
Handle := State[-frame tag + 2 words],
Buf    := State[-frame tag + 1 word]
```

The Dialyzer is not yet integrated into the normal compilation process, and type specifications are optional. How far can we get without them, and how?

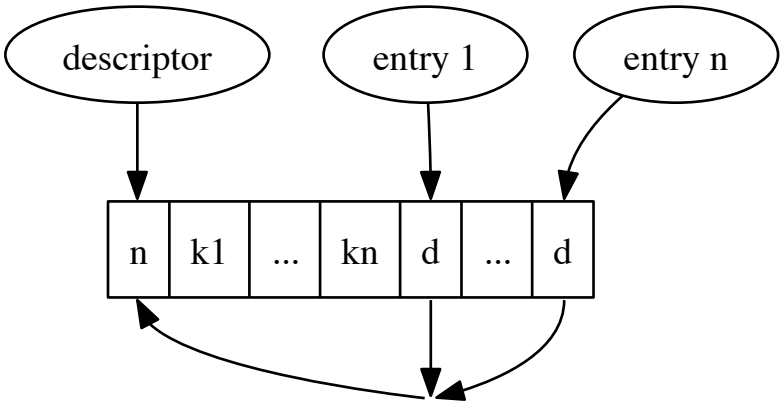
There is apparently a per-module type inference phase in HiPE, which *is* integrated into the Erlang system. I have not yet been able to find any detail about how it works, but the TypEr system comes from the same stable, and that may offer some guidance. In particular, the type lattice used by TypEr could be extended to handle frames very simply, and it would then even handle subtyping. Amongst other things, $\sim f(X)$ can only succeed when X is a frame having an f field.

For this example, we don't need anything that elaborate, because we are only interested in complete information about the keys of a frame and we are not looking for information hidden inside lists or funs, but just being passed around in loops. We can get as far as we need to using a simple first-order data flow analysis which just considers frames, tuples, and "other". It is easy to construct examples where the information we need is hidden by the use of higher order functions, but that's not the way code like this is normally written.

15 Implementation, one more time

In 2012, I think I can now put the pieces together and offer an implementation that works well when used in a record-like way, without type inference but compatiblty with it.

Descriptors are allocated statically whenever possible, using a hash table. Each statically allocated descriptor is augmented with a oneword cache entry that points back to the descriptor. Since the cache entries are adjacent to the descriptor, an entry e corresponds to index $e - contents(e) - contents(contents(e))$. Alternatively, the cache entries could be two words, storing their indices.



15.1 Creation

$\langle\{k_1 \sim V_1, \ldots, k_n \sim V_n\}\rangle \longrightarrow$
    for $i \in [1, n]$
        if V_i is a variable or ground term
            let X_i be V_i
        else
            compute X_i = value of V_i
    `put_frame` t, $n + 1$, &descriptor
    for $i \in [1, n]$
        `put` $X_{\pi(i)}$
    proceed with t

where π is the permutation such that $[k_{\pi(1)}, \ldots, k_{\pi}(n)]$ is sorted.

`put_frame` out, size, descr $\longrightarrow$
    — out is a result register number
    — size is how many words to allocate
    — descr is the address of the descriptor
    allocate size words on the heap;
    set out to be a frame-tagged pointer to that block;
    out[0] := descr;
    leave a pointer to &out[1] ready for more 'put's.

15.2 Selection

$X = \sim k(E) \longrightarrow$
    compute t = value of E
    `check_frame` d, t
    `load_slot` r, d, t, 'k', &dummy

where dummy is a dummy cache entry that points to no descriptor.

`check_frame` descr, frame ⟶
    if frame points to a frame then
        descr := frame[0];
    else
        match failure

We would want three versions of this: one for use in clause heads, where 'failure' means trying the next alternative, one for use in expressions, where 'failure' means raising an exception, and one for type tests returning `true` or `false`. Call those `test_frame`, `check_frame`, and `is_frame` respectively.

`load_slot` out, descr, frame, slot, cache
    — out is a register number
    — descr is a register number
    — frame is a register number
    — slot is an atom
    — cache is a mutable word pointing to a cache entry.
    c := cache;
    if contents(c) ≠ descr then
        i := index of slot in descr;
        if descr is dynamic then
            c := &dummy;
        else
            c := &descr's i'th cache entry;
        cache := c;
    else
        i := index of c;
    out := frame[i];

This uses an inline cache. It is correct even on a multicore system as long as loads and stores of the cache word are atomic, because cache entries are immutable.

15.3 Matching

Matching a frame pattern uses one `check_frame` instruction for the frame as a whole and one `load_slot` instruction for each slot mentioned. It is possible to do better, but this should be good enough to start with.

15.4 Updating

The decision to restrict $\langle\{k \sim V|F\}\rangle$ to updating an existing field means that update is now easy.

$\langle\{k_1 \sim V_1, \ldots, k_n \sim V_n|F\}\rangle$ ⟶
    for $i \in [1, n]$
        if V_i is a variable or ground term
            let X_i be V_i

```
        else
            compute X_i = value of V_i
    compute t as the value of F
    copy_frame r, d, t
    for i ∈ [1, n]
        store_slot r, d, X_i, 'k_i', &dummy
    proceed with a suitably tagged r

copy_frame out, descr, frame ⟶
    if frame points to a frame then
        descr := frame[0];
    else
        match failure
    allocate descr[0]+1 words on the heap
      and let out be the address of that block;
    copy frame[0..descr[0]] to out[0..descr[0]];

store_slot out, descr, value, slot, cache
    — out is a register number
    — descr is a register number
    — value is a register number with the new value
    — slot is an atom
    — cache is a mutable word pointing to a cache entry.
    c := cache;
    if contents(c) ≠ descr then
        i := index of slot in descr;
        if descr is dynamic then
            c := &dummy;
        else
            c := &descr's i'th cache entry;
        cache := c;
    else
        i := index of c;
    out[i] := frame;
```

There are other ways to factor this. For example, `load_slot` and `store_slot` could be split into `find_slot` + `load_indexed` and `find_slot` + `store_indexed`.

Again, I stress that it is possible to do better. With a somewhat different approach to caching and a bit of run-time code generation, we could do one cache lookup for a group of slots, and avoid the (harmless but inefficient) double setting of updated fields. What we have here is enough to get $O(n)$ creation and update and $O(1)$ selection in practice for frames used in a record-like way.

16 References

Not yet written. Mainly to the Erlang manual, and some communications in the Erlang mailing list. IBM PROLOG manual. Haskell report. Clean. O'CAML. YAML. UBF. JSON. Deutsch and Schiffman.

17 Revised version of `dict.erl`

This has no need whatever for the preprocessor.

It is also an example where inline caching would pay off perfectly.

```
%   File    : dict2
%   Reviser: Richard A. O'Keefe
%   Updated: 2003.10.02; 2011.02.22.
%   Purpose: Demonstrate "frame" use.

%   +type and +deftype are as described in Joe Armstrong's thesis;
%   if I've understood it correctly.  I've eliminated all preprocessor
%   use as well as the specific example of records.  This causes trouble
%   in one place.  Most comments are copied from the original.
%   2011: <{...|_}> patterns changed to <{...}>.

-module(dict).
-export([
    append/3,
    append_list/3,
    dict_to_list/1,
    erase/2,
    from_list/1,
    fetch/2,
    fetch_keys/1,
    filter/2,
    find/2,
    fold/3,
    is_key/2,           % has_key(Dict, Key) would be better
    list_to_dict/1,
    map/2,
    merge/3,
    new/0,
    size/1,
    store/3,
    to_list/1,
    update/3,
    update/4,
    update_counter/3
 ]).
-deprecated([
    {dict_to_list,1},   % why doesn't this use Name/Arity format?
    {list_to_dict,1}
 ]).

-inline([seg_size/0, max_seg/0, expand_load/0, contract_load/0]).

seg_size()      -> 16.  % size of each segment
max_seg()       -> 32.  % maximum number of segments
expand_load()   ->  5.  % load factor that triggers expansion
```

```
contract_load() ->  3.  % load factor that triggers contraction

/*
  +deftype alist(K,V) = list({K,V}).
  +deftype pair(K,V) = [K|V].
  +deftype bucket(K,V) = list(pair(K,V)).
  +deftype segment(K,V) = tuple(bucket(K,V)).
  +deftype dict(K,V) = <{
   size     ~ int(),           % number of elements
   n        ~ int(),           % number of active slots
   maxn     ~ int(),           % maximum number of slots
   bso      ~ int(),           % Buddy Slot Offset
   exp_size ~ int(),           % size to expand at
   con_size ~ int(),           % size to contact at
   empty    ~ segment(K,V),    % an always-empty segment
   segs     ~ tuple(segment(K,V))  % segments
   }>.
*/

%+type new() = dict(_,_).

new() ->
    SegSize = seg_size(),
    Empty = list_to_tuple(lists:duplicate(SegSize, [])),
    <{ size ~ 0, n ~ SegSize, maxn ~ SegSize, bso ~ SegSize div 2,
       exp_size ~ SegSize * expand_load(),
       con_size ~ SegSize * contract_load(),
       empty ~ Empty, segs ~ {Empty}
    }>.

%+type is_key(K, dict(K,_)) = bool().

is_key(K, D) ->
    find_key(K, get_bucket(D, K)).

%+type find_key(K, bucket(K,_)) = bool().

find_key(K, [[K|_]|_]) -> true;
find_key(K, [  _  |L]) -> find_key(K, L);
find_key(K, []       ) -> false.

%+type dict_to_list(dict(K,V)) = alist(K,V).

dict_to_list(D) ->
    to_list(D).

%+type to_list(dict(K,V)) = alist(K,V).
% Note: the order in which keys are returned is undefined.
% Sorting would take O(N.lgN) time; this takes O(N).
```

```
to_list(D) ->
    fold(fun (K, V, L) -> [{K,V}|L] end, [], D).

%+type list_to_dict(alist(K,V)) = dict(K,V).

list_to_dict(D) ->
    from_list(D).

%+type from_list(alist(K,V)) = dict(K,V).

from_list(L) ->
    lists:foldl(fun ({K,V}, D) -> store(K, V, D) end, new(), L).

%+type size(dict(_,_)) = int().

size(<{size~Size}>) -> Size.

%+type fetch(K, dict(K,V)) = V.
% Note: run-time error if key not present.

fetch(K, D) -> fetch_val(K, get_bucket(D, K)).

%+type fetch_val(K, bucket(K,V)) = V.

fetch_val(K, [[K|V]|_]) -> V;
fetch_val(K, [  _  |L]) -> fetch_val(K, L).

%+type find(K, dict(K,V)) = {ok,V} | error.

find(Key, D) -> find_val(K, get_bucket(D, K)).

%+ find_val(K, bucket(K,V)) = {ok,V} | error.

find_val(K, [[K|V]|_]) -> {ok,V};
find_val(K, [  _  |L]) -> find_val(K, L);
find_val(_, []       ) -> error.

%+type fetch_keys(dict(K,_)) -> [K].
% Note: "fetch_" is otiose.

fetch_keys(D) -> fold(fun (K, _, L) -> [K|L] end, [], D).

%+type get_bucket(dict(K,V), K()) = bucket(K,V).
%%  This has been fused with get_slot/2.

get_bucket(<{n~N, maxn~MaxN, bso~BSO, segs~Segs}>, K) ->
    H = erlang:phash(Key, MaxN),
    Slot1 = if  H > N -> H - BSO.bso
              ; true  -> H
            end - 1,
```

```
    Size = seg_size(),
    element(Slot1 rem Size + 1, element(Slot1 div Size + 1, Segs)).


%+type erase(K, dict(K,V)) = dict(K,V).
%%  Erase all elements with key K.

erase(K, D0) ->
    {D1,Dc} = on_key_bucket(D0, K, fun (B0) -> erase_key(Key, B0) end,
    maybe_contract(D1, Dc).

%+type erase_key(K, bucket(k,V)) = {bucket(K,V), 0|1}.

erase_key(K, [[K|_]|L]) -> {L,1};
erase_key(K, [  P  |L]) -> {L1,Dc} = erase_key(K, L), {[P|L1],Dc};
erase_key(_, []       ) -> {[],0}.

%+type store(K, V, dict(K,V)) = dict(K,V).

store(K, V, D) ->
    {D1,Ic} = on_key_bucket(D, K, fun (B) -> store_bkt_val(K, V, B) end),
    maybe_expand(D1, Ic).

%+type store_bkt_val(K, V, bucket(K,V)) = {bucket(K,V), 0|1}.

store_bkt_val(K, V, [[K|_]|L]) -> {[[K|V]|L],0};
store_bkt_val(K, V, [P|L1])    -> {L1,Ic} = store_bkt_val(K, V, L), {[P|L],Ic};
store_bkt_val(K, V, [])        -> {[K|V],1}.

%+type append(K, V, dict(K,[V])) = dict(K,[V]).

append(K, V, D) ->
    {D1,Ic} = on_key_bucket(D, K, fun (B) -> append_bkt(K, [V], B) end),
    maybe_expand(D1, Ic).

%+type append_bkt(K, list(V), bucket(K,list(V))) = {bucket(K,List(V)), 0|1}.

append_bkt(K, V, [[K|X]|L]) -> {[[K|X++V]|L], 0};
append_bkt(K, V, [  P  |L]) -> {L1,Ic} = append_bkt(K, V, L), {[P|L1],Ic};
append_bkt(K, V, []       ) -> {[[K|V]],1 1}.

%+type append_list(K, list(V), dict(K,list(V))) = dict(K,list(V)).

append_list(K, Vs, D) ->
    {D1,Ic} = on_key_bucket(D, K,fun (B0) -> append_bkt(Key, L, B0) end),
    maybe_expand(D1, Ic).

%+type update(K, (V->V), dict(K,V)) = dict(K,V).

update(K, F, D) ->
```

```
    {D1,_} = on_key_bucket(D, K, fun (B) -> update_bkt(K, F, B) end),
    D1.

%+type update_bkt(K, (V->V), bucket(K,V)) = {bucket(K,V), 0}.

update_bkt(K, F, [[K|V]|L]) -> U = F(V), {[[K|U]|L], 0};
update_bkt(K, F, [  P  |L]) -> {L1,U} = update_bkt(K, F, L), {[P|L1], U}.

%+type update(K, (V->V), V, dict(K,V)) = dict(K,V).

update(K, F, I, D) ->
    {D1,Ic} = on_key_bucket(D, K, fun (B) -> update_bkt(K, F, I, B) end),
    maybe_expand(D1, Ic).

%+type update_bkt(K, (V->V), V, bucket(K,V)) = {bucket(K,V), 0|1}.

update_bkt(K, F, _, [[K|V]|L]) ->
    {[[K|F(V)]|L], 0};
update_bkt(K, F, I, [  P  |L]) ->
    {L1,Ic} = update_bkt(K, F, I, L), {[P|L1],Ic};
update_bkt(K, _, I, []       ) ->
    {[[K|I]], 1}.

%+type update_counter(K, int(), dict(K,int())) = dict(K,int()).

update_counter(K, I, D) ->
    {D1,Ic} = on_key_bucket(D, K, fun (B) -> counter_bkt(K, I, B) end),
    maybe_expand(D1, Ic).

%+type counter_bkt(K, int(), bucket(K,int())) = {bucket(K,int()), 0|1}.

counter_bkt(K, I, [[K|V]|L]) -> {[[K|V+I]|L], 0};
counter_bkt(K, I, [  P  |L]) -> {L1,Ic} = counter_bkt(K, I, B), {[P|L1],Ic};
counter_bkt(K, I, []       ) -> {[[K|I]], 1}.

%+type fold(((K,V,X) -> X), X, dict(K,V)) = X.

fold(F, X, <{ segs ~ Segs}>) ->
    fold_segs(F, X, Segs, size(Segs)).

%+type fold_segs(((K,V,X) -> X), X, tuple(segment(K,V)), int()) = X.

fold_segs(_, X, _,    0) -> X;
fold_segs(F, X, Segs, I)
    Seg = element(I, Segs),
    fold_segs(F, fold_seg(F, X, Seg, size(Seg)), Segs, I-1).

%+type fold_seg(((K,V,X) -> X), X, segment(K,V), int()) = X.

fold_seg(_, X, _,   0) -> X;
```

```
fold_seg(F, X, Seg, I) ->
    fold_seg(F, fold_bucket(F, X, element(I, Seg)), Seg, I-1).

%+type fold_bucket(((K,V,X) -> X), X, bucket(K,V), int()) = X.

fold_bucket(F, X, [[K|V]|L]) -> fold_bucket(F, F(K, V, X), L);
fold_bucket(_, X, [])        -> X.

%+type map(((K,V)->W), dict(K,V)) = dict(K,W).

map(F, D = <{segs~Segs}>) ->
    <{ segs ~ map_tuple(fun (Seg) -> map_tuple(fun (Bkt) ->
                [ [K|F(K,V)] || [K|V] <- Bkt ], Seg), Segs)
      | D }>.

%+type map_tuple((A->B), {A}) = {B}.
%% This function is not just useful here, it's useful in all sorts of
%% places.  Similarly fold_tuple/3.

map_tuple(F, T) -> list_to_tuple(lists:map(F, tuple_to_list(T))).


%+type filter(((K,V) -> bool()), dict(K,V)) = dict(K,V).

filter(F, D = <{segs~Segs}>) ->
    {Segs1,Dc} = filter_seg_list(F, tuple_to_list(Segs), [], 0),
    maybe_contract(<{segs~Seg1|D}>, Dc).

%+type filter_seg_list(((K,V) -> bool()), list(segment(K,V)),
%                      list(segment(K,V)), int()) = {tuple(segment(K,V)), int()}.

filter_seg_list(F, [Seg|Segs], Fss, Fc0) ->
    {Seg1,Fc1} = filter_bkt_list(F, tuple_to_list(Seg), [], Fc0),
    filter_seg_list(F, Segs, [Seg1|Fss], Fc1);
filter_seg_list(_, [], Fss, Fc) ->
    {list_to_tuple(lists:reverse(Fss, [])), Fc}.

%+type filter_bkt_list(((K,V) -> bool()), list(bucket(K,V)),
%                      list(bucket(K,V)), int()) = {segment(K,V), int()}.

filter_bkt_list(F, [Bkt0|Bkts], Fbs, Fc0) ->
    {Bkt1,Fc1} = filter_bucket(F, Bkt0, [], Fc0),
    filter_bkt_list(F, Bkts, [Bkt1|Fbs], Fc1);
filter_bkt_list(_, [], Fbs, Fc) ->
    {list_to_tuple(lists:reverse(Fbs)), Fc}.

%+type filter_bucket(((K,V) -> bool()), bucket(K,V), bucket(K,V), int())
%      = {bucket(K,V), int()}.

filter_bucket(F, [P=[K|V]|L], R, Fc) ->
```

```
    case F(K, V)
      of true  -> filter_bucket(F, L, [P|R], Fc)
       ; false -> filter_bucket(F, L, R,     Fc+1)
    end;
filter_bucket(_, [], Fb, Fc) ->
    {lists:reverse(Fb), Fc}.

%+type merge(((K,V,V) -> V), dict(K,V), dict(K,V)) = dict(K,V).

merge(F, D1 = <{size~Size1}>, D2 = <{size~Size2}>) ->
    if Size1 < Size2 ->
        fold(fun (K, V1, D) -> update(K, fun (V2) -> F(K, V1, V2) end, V1, D)
             end, D2, D1)
     ; true ->
        fold(fun (K, V2, D) -> update(K, fun (V1) -> F(K, V1, V2) end, V2, D)
             end, D1, D2)
    end.


%+type on_key_bucket(dict(K,V), K, ([[K|V]] -> {[[K|V]],X})) = {dict(K,V), X}.
%%  Apply F to the bucket for key K in dictionary D
%%  and replace the returned bucket.
%%  This has been fused with get_slot/2.

on_key_bucket(D = <{ segs ~ Segs }>, K, F) ->
    H     = erlang:phash(Key, MaxN),
    Slot1 = if  H > N -> H - BSO.bso
              ; true  -> H
            end - 1,
    Size  = seg_size(),
    SegI  = Slot1 div Size + 1,
    BktI  = Slot1 rem Size + 1,
    Seg   = element(SegI, Segs),
    B0    = element(BktI, Seg),
    {B1,Res} = F(B0),
    {<{segs ~ setelement(SegI, Segs, setelement(BktI, Seg, B1)) | D}>, Res}.



%+type get_bucket_s(tuple(segment(K,V)), int()) = bucket(K,V).

get_bucket_s(Segs, Slot) ->
    Size = seg_size(),
    SegI = (Slot-1) div Size + 1,
    BktI = (Slot-1) rem Size + 1,
    element(BktI, element(SegI, Segs)).

%+type put_bucket_s(tuple(segment(K,V)), int(), bucket(K,V)) =
%                   tuple(segment(K,V)).
```

```
put_bucket_s(Segs, Slot, Bkt) ->
    Size = seg_size(),
    SegI = (Slot-1) div Size + 1,
    BktI = (Slot-1) rem Size + 1,
    setelement(SegI, Segs, setelement(BktI, element(SegI, Segs), Bkt)).

%+type maybe_expand(dict(K,V), int()) = dict(K,V).

maybe_expand(D0 = <{size~Size, exp_size~ExpSize}>, Ic)
  when Size + Ic > ExpSize ->
    D1 = case D0
           of <{n~N, maxn~N, bso~BSO, segs~Segs0, empty~Empty}> ->
                 <{maxn~2*N, bso~2*BSO, segs~expand_segs(Segs,Empty)|D0}>
            ; _ -> D0
         end,
    <{n~N1, maxn~MaxN1, bso~BSO1, segs~Segs1}> = D1,
    N2 = N1 + 2,
    Slot1 = N2 - BSO1,
    B = get_bucket_s(Segs1, Slot1),
    Slot2 = N2,
    [B1|B2] = rehash(B, Slot1, Slot2, MaxN),
    Segs2 = put_bucket_s(Segs1, Slot1, B1),
    Segs3 = put_bucket_s(Segs2, Slot2, B2),
    <{size ~ Size + Ic, n ~ N2, exp_size ~ N2*expand_load(),
      con_size ~ N2*contract_load(), segs ~ Segs3 | D1 }>;
maybe_expand(D0 = <{size~Size}>, Ic) ->
    <{size~Size+Ic|D0}>.

%+type maybe_contract(dict(K,V), int()) = dict(K,V).
%+type maybe_contract(dict(K,V), int(), int()) = dict(K,V).

maybe_contract(Dict, Dc) ->
    maybe_contract(Dict, Dc, seg_size()).

% The third argument is passed so that the first clause's guard doesn't
% need macros.  This is the only place where macros were wanted much, &
% with abstract patterns, even this wouldn't have been necessary.

maybe_contract(D0 = <{size~Size, con_size~ConSize, n~N}>, Dc, SegSize)
  when Size - Dc < ConSize, N > SegSize ->
    <{bso~BSO, segs~Segs0}> = D0,
    Slot1 = N - BSO,
    B1 = get_bucket_s(Segs0, Slot1),
    Slot2 = N,
    B2 = get_bucket_s(Segs0, Slot2),
    Segs1 = put_bucket_s(Segs0, Slot1, B1 ++ B2),
    Segs2 = put_bucket_s(Segs1, Slot2, []), % Clear the upper bucket
    N1 = N - 1,
    if  N1 == BSO ->
        <{size ~ Size - Dc, n ~ N1,
```

```
          exp_size ~ N1 * expand_load(), con_size ~ N1 * contract_load(),
          maxn ~ MaxN div 2, bso ~ BSO div 2,
          segs ~ contract_segs(Segs2) | D0}>
      ; true ->
        <{size ~ Size - Dc, n ~ N1,
          exp_size ~ N1 * expand_load(), con_size ~ N1 * contract_load(),
          segs ~ Segs2 | D0}>
    end;
maybe_contract(D0 = <{size~Size}>, Dc, _) ->
    <{size ~ Size - Dc | D0}>.


%+type rehash(bucket(K,V), int(), int(), int()) = [bucket(K,V)|bucket(K,V)].
%%  Yes, we should return a tuple, but this is more fun.

rehash([P=[K|_]|L], Slot1, Slot2, MaxN) ->
    [L1|L2] = rehash(L, Slot1, Slot2, MaxN),
    case erlang:phash(K, MaxN)
      of Slot1 -> [[P|L1]|L2]
       ; Slot2 -> [L1|[P|L2]]
    end;
rehash([], _, _, _) -> [[]|[]].


%+type expand_segs(tuple(segment(K,V)), segment(K,V)) = tuple(segment(K,V)).
%%  Expand the segment tuple by doubling the number of segments.
%%  We special-case the powers of 2 up to 32, this should catch most cases.
%%  N.B. the last element in the segments tuple is
%%  an extra element containing a default empty segment.

expand_segs({B1}, Empty) ->
    {B1,Empty};
expand_segs({B1,B2}, Empty) ->
    {B1,B2,Empty,Empty};
expand_segs({B1,B2,B3,B4}, Empty) ->
    {B1,B2,B3,B4,Empty,Empty,Empty,Empty};
expand_segs({B1,B2,B3,B4,B5,B6,B7,B8}, Empty) ->
    {B1,B2,B3,B4,B5,B6,B7,B8,
     Empty,Empty,Empty,Empty,Empty,Empty,Empty,Empty};
expand_segs({B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12,B13,B14,B15,B16}, Empty) ->
    {B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12,B13,B14,B15,B16,
     Empty,Empty,Empty,Empty,Empty,Empty,Empty,Empty,
     Empty,Empty,Empty,Empty,Empty,Empty,Empty,Empty};
expand_segs(Segs, Empty) ->
    list_to_tuple(tuple_to_list(Segs) ++ lists:duplicate(size(Segs), Empty)).

%+type contract_segs(tuple(segment(K,V))) = tuple(segment(K,V)).
%%  Contract the segment tuple by halving the number of segments.
%%  We special-case the powers of 2 up to 32, this should catch most cases.
%%  N.B. the last element in the segments tuple is
```

```
%%  an extra element containing a default empty segment.

contract_segs({B1,_}) ->
    {B1};
contract_segs({B1,B2,_,_}) ->
    {B1,B2};
contract_segs({B1,B2,B3,B4,_,_,_,_}) ->
    {B1,B2,B3,B4};
contract_segs({B1,B2,B3,B4,B5,B6,B7,B8,_,_,_,_,_,_,_,_}) ->
    {B1,B2,B3,B4,B5,B6,B7,B8};
contract_segs({B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12,B13,B14,B15,B16,
               _,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_}) ->
    {B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12,B13,B14,B15,B16};
contract_segs(Segs) ->
    list_to_tuple(lists:sublist(tuple_to_list(Segs), 1, size(Segs) div 2)).
```

The End