Labeled Tuples

(Informed position)

Chris Casinghino
Jane Street
USA
ccasinghino@janestreet.com

Ryan Tjoa
University of Washington
USA
rtjoa@cs.washington.edu

Abstract

This talk will introduce the *labeled tuples* language feature, which allows programmers to label tuple elements. It is conceptually dual to labeled function arguments, allowing programmers to give a helpful name to constructed values where labeled function arguments permit giving a helpful name to parameters. We present an overview of the design of this feature, arguments for and against its inclusion in the language, and describe a few possible variations and implementation considerations. We have implemented labeled tuples in Jane Street's branch of the OCaml compiler—it has proven extremely popular and quickly found wide use in our code base.

ACM Reference Format:
Chris Casinghino and Ryan Tjoa. 2024. Labeled Tuples: (Informed position). In *Proceedings of Higher-order, Typed, Inferred, Strict: ML Family Workshop 2024 (ML '24).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Language feature overview

The *labeled tuples* extension allows the programmer to label tuple elements. It is conceptually dual to labeled function arguments [2], allowing programmers to give helpful names to constructed values where labeled function arguments permit giving a helpful name to parameters. We have implemented this feature within Jane Street's branch of the OCaml compiler [1].

Here is a motivating example where we want to compute two values from a list and be careful not to mix them up:

```
let sum_and_product ints =
  let init = ~sum:0, ~product:1 in
  List.fold_left ints ~init
    ~f:(fun (~sum, ~product) elem ->
          let sum = elem + sum in
          let product = elem * product in
          ~sum, ~product)
```

This example shows the use of labeled tuples in expressions and patterns. They may be punned like record elements and labeled function arguments. In types, tuple labels are written similarly to function argument labels. For example, the function `f` in the previous example has the type:

```
(sum:int * product:int) -> int
  -> sum:int * product:int
```

Because `sum:int * product:int` is a different type than `product:int * sum:int`, the use of a labeled tuple in this example prevents us from accidentally returning the pair in the wrong order. Labeled tuple types are structural and do not need to be declared before use.

Labeled tuples are useful anytime one wants to use names to explain or disambiguate the elements of a tuple, but declaring a new record feels too heavy. As another example, consider this function from the `Core_unix` library which creates a pipe with descriptors for reading and writing:

```
val pipe :
     ?close_on_exec:bool -> unit
  -> File_descr.t * File_descr.t
```

Which is which? With labeled tuples, we can make it clear just as cheaply as using a labeled function argument:

```
val pipe :
     ?close_on_exec:bool -> unit
  -> read:File_descr.t * write:File_descr.t
```

Tuples may be partially labeled, which can be useful when some elements of the tuple share a type and need disambiguation, but others don't. For example:

```
type min_max_avg = min:int * max:int * float
```

1.1 Reordering and partial patterns

Like records, labeled tuple patterns may be reordered or partial. The compiler only supports reordering / partial matching when it knows the type of the pattern from its context.

So, for example, we can write:

```
# let lt = ~x:0, ~y:42;;
val lt : x:int * y:int = (~x:0, ~y:42)

# let twice_y = let ~y, .. = lt in y * 2;;
val twice_y : int = 84
```

Here, the pattern `~y, ..` is a partial match on the tuple, extracting the first field with label `y` and ignoring the rest. Partial patterns may be used to extract any subset of the fields.

ML '24, September, 2024, Milan, Italy
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

When the type is not known (in the same sense that OCaml requires a type to be known to disambiguate among constructors), the compiler will reject a partial pattern. For example, this program

```
let get_y t =
  let ~y, .. = t in
  y
```

is rejected with this error:

```
File "foo.ml", line 2, characters 8-14:
2 |     let ~y, .. = t in
            ^^^^^^
Error: Could not determine the type of this
partial tuple pattern.
```

This example could be fixed by adding a type annotation to the function's parameter.

Labels may also be repeated in a tuple, and unlabeled elements can be thought of as all sharing the same unique label. When matching on such a tuple, the first occurrence of a label in the pattern is bound to the first corresponding label in the value, and so on. As a result, it's also now possible to partially match on a standard unlabeled tuple to retrieve the first few elements.

2 Is it worth doing?

This language feature sparked considerable debate within Jane Street's OCaml compiler team before we decided to implement it. In this section, we consider some of the arguments for and against, and explain why we decided to add it and what our experience has been in practice.

There is an obvious argument for the feature: tuples are error prone. Because their elements are not labeled, it is easy to mix up two values of the same type. And, for readers, it can be hard to work out the purpose of some member of a large tuple that would be perfectly clear with a label.

One solution is to use a record instead. But this requires a type declaration, which can feel like a heavy-weight solution particularly if the tuple is used in just as an accumulator or the return value from one function. In practice, programmers have adopted several work-arounds. These include:

- Using polymorphic variants as labels in tuples. We see examples in real code that use this approach, as in:
  ```
  let ( `new_price new_price
      , `new_size new_size )
    = ...
  ```
  This approach is lightweight for the programmer, but comes with runtime costs if the compiler can not eliminate the variant blocks.
- Using OCaml's object system. While OCaml's object system does essentially provide a notion of anonymous record that serves similar goals, we do not see programmers use it for this purpose in practice. We believe there are two main reasons: First, OCaml's object-oriented features are relatively rarely used, so programmers do not feel familiar enough to use these in a lightweight way. Second, like the previous approach, this comes with a dynamic cost.
- Simply using a tuple even though its meaning may be unclear. We see this often in practice, as in the example of `Core_unix.pipe` above. We judge the frequency with which this approach is settled for, even when it leads to unclear and error prone code, as strong evidence for the value of labeled tuples.

A final consideration: The feature is popular. At time of writing, it was made available within Jane Street approximately four months ago, and is already used in more than 2500 `.ml` files in our code base.

3 Design considerations, possible extensions, and related systems

3.1 Reordering

Our design intentionally provides only limited facility for reordering and partial matches (these are supported in patterns when the type is known, and expressions are never reordered). An alternative design could be more permissive, providing a notion of record polymorphism that supports reordering of expressions and patterns in all circumstances.

Existing work for more flexible polymorphic records shows they can be implemented with row polymorphism [4] or record kinds [3]. An intermediate option would be to internally canonicalize the order of labeled tuple components (e.g. considering `x:int * y:int` to be the same type as `y:int * x:int`), which could offer more reordering than our design without an explicit notion of record polymophism.

We prefer the current design for several reasons:

- Our goal for the feature is to provide a lightweight dual to labeled function arguments, not a "record" version of polymorphic variants. OCaml's implementation of labeled function arguments makes similar use of type information to type applications.
- Labeled tuples offer better performance than the alternatives described above. Systems with record polymorphism typically require some kind of dynamic field lookup to account for functions that work over different record layouts. Further, performance sensitive code may care about the concrete layout of tuple fields in memory for cache locality reasons—supporting reordering of labeled tuple expressions would make their performance behavior less controllable and predictable.
- The implementation of this feature is considerably simpler than the alternatives.

SML's *flexible records* is the language feature we are aware of that is most closely related to labeled tuples. The primary difference between these features is that flexible records are canonically ordered based on the label names, while

labeled tuples are not, as described above. OCaml's nominally typed records offer convenient reordering and projection operations, taking pressure off of a feature like labeled tuples to do the same—users can use labeled tuples for lightweight local names where that makes sense, and records otherwise.

3.2 Projection

Our current implementation provides no facility for projection from a labeled tuple, though a partial match may be used when the type is known. It would be straightforward to similarly implement projection when the type is known, and the main reason we have not is simply that the most natural syntax is ambiguous with normal record projection. We may add this in the future. Allowing labeled tuple projection even when the type is not known would require row polymorphism, as discussed above.

3.3 Singleton labeled tuples

Since introducing this feature, many users have requested "singleton labeled tuples"—essentially just the ability to label arbitrary values. This feature does not seem desirable for OCaml today in its most general form, as it overlaps in purpose with labeled function arguments—should a function use a labeled argument or a labeled value? Further, it would also be ambiguous to parse due to the syntactic similarities to labeled arguments, though this may be resolvable by requiring parentheses in some cases.

One might consider simply *replacing* labeled arguments with labeled values, but it is not obvious how to achieve some of the conveniences of labeled arguments, like the ability to partially apply a function to any labeled argument. However, a different possible feature, *labeled function returns*, would be adequate to address many of the requests we have seen. Labeled returns would be more straightforward to add to OCaml, and we are considering doing so in the future.

References

[1] 2024. Jane Street OCaml Branch. https://github.com/ocaml-flambda/flambda-backend.

[2] Jacques Garrigue. 2001. Labeled and optional arguments for Objective Caml. In *JSSST Workshop on Programming and Programming Languages, Kameoka, Japan.*

[3] Atsushi Ohori. 1995. A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst.* 17, 6 (nov 1995), 844–895. https://doi.org/10.1145/218570.218572

[4] Mitchell Wand. 1989. Type inference for record concatenation and multiple inheritance. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science.*