Generic partially-static data (extended abstract)

David Kaloper-Meršinjak  Jeremy Yallop
University of Cambridge
dk505@cam.ac.uk  jeremy.yallop@cl.cam.ac.uk

Static vs dynamic

A central feature of multi-stage programming is the distinction between static and dynamic expressions, i.e. between those expressions which can be evaluated in the current stage of a program, and those that can be evaluated only in a future stage. This distinction underlies the performance improvements that are the primary goal of multi-stage programming: by performing as much work as possible in the current stage, the residual code that is executed in future stages can be made more efficient.

Whether a particular expression is static or dynamic depends on its free variables: an expression depending only on static data is static, while an expression with dynamic dependencies must be treated as dynamic. Effective multi-stage programming often involves restructuring programs (for example, by CPS conversion), to increase the number of classes that can be classified as static.

An alternative, less invasive approach to moving computation into the static phase is to focus on data rather than on expressions. Once more, with a naive classification of values into static and dynamic, a single dynamic datum can infect a much larger value. However, the notion of partially-static data applies to a wide variety of data types. The partial classification, written \(~e\), splices a dynamic value. Partially-static also encompasses dynamic; the operation analyses the prefix of the first list until a dynamic tail is encountered, at which point it constructs a piece of code that prepends \(~\) to the second list.

Let us look at an example. Here is the standard unstaged definition of parameterised lists, together with an append function \(+\):  

```ocaml
(* val (+ +) : α list α list → α list *)

let rec (+ +) l r = match l with
| [] -> r
| h :: t -> h :: (t ++ r)
```

And here is a variant of \(+\) that treats the second list as dynamic:

```ocaml
(* val (++): α list → α list code → α list code *)

let rec (++: l r = match l with
| [] -> r
| h :: t -> \(~h\.~(t ++ r))
```

The code type represents quoted expressions, which may be executed at some future stage. The brackets \(~\) build a quoted expression of type \α code\ from an expression of type \α. Antiquotation, written \~e, splices a code value \e into a quoted expression.

Finally, here is a definition of partially-static lists, with possibly-dynamic tails, with a corresponding definition of \++:

```ocaml
type α listα = [] | (:::) of α * α listα | Dyn of α list code
```

1 We use the multi-stage language BER MetaOCaml (Kiselyov 2014), extended with modular implicits (White et al. 2015) for overloaded functions.

### Figure 1: partially-static data

```
module type PS = sig
  type t
  val dyn : ps → t code
  val sta : t → ps
  val cd : t code → ps
end
```

### Figure 2: Fixpoints and partially-static fixpoints

```
module Fix(S: sig type (_,_) t end) = struct
  type α t = [\'R of (α,α t) S.t]
end

module Fixps(S: sig type (_,_) t end) = struct
  type (α,β) ps = [\'Sta of (α,α β) ps S.t]
  | \'Dyn of β Fix(S).t code]
end
```

This last \++ operation analyses the prefix of the first list until a dynamic tail is encountered, at which point it constructs a piece of code that prepends \(~\) to the second list. The function \~\) converts \(~\) to a dynamic value that can be spliced into the generated code.

The notion of partially-static data applies to a wide variety of data types. The PS interface (Figure 1) relates a type \t to its partially-static counterpart \ps by means of several operations. The interface supports moving values forward in time, with an operation \sta that builds a partially-static value from a static value, and an operation \dyn that converts a partially-static value into a fully dynamic value. Partially-static also encompasses dynamic; the operation \cd builds a partially-static value from a dynamic value.

### Partially-static data, generically

The construction of \ps\ from \listt\ is an instance of a more general transformation on types (Sheard and Diatchki). From a definition for a type \t, we can obtain a partially-static counterpart \tpst\ by replacing each recursive occurrence of \t in the definition, and adding an additional top-level constructor \Dyn of \t code.
In fact, we can express this transformation as a fixpoint operation on type functions — or rather, on type definitions written in an open-recursive style. For example, here is a definition of listr which uses a second parameter \( \rho \) where \( \alpha \) listr would usually appear in the definition:

```plaintext
let rec fold {S: MAP} {A:PS} {A,B:PS} l r = fold (function | h : : t | c | ('R c) l r |
```

Applying a fixpoint operator, Fix (Figure 2) to listr builds a closed recursive definition, isomorphic to list2:

```plaintext
module L = Fix(struct type (α,β) t = (α,β) listr end)
```

Similarly, an application of a second fixpoint operator, Fixps (also Figure 2), gives us a partially-static version of lists:

```plaintext
module Lps = Fixps(struct type (α,β) t = (α,β) listr end)
```

### Generic operations on partially-static data

Besides abstractions for constructing partially-static types it is useful to construct generic operations over data of those types.

#### Generic folds

Gibbons (2007) shows how to obtain a variety of generic operations over a data type — maps, folds, unfolds, and more — from a fixpoint over the open-recursive version of the type. For example, here is a generic fold parameterised by an implicit bifunctor \( S \) of type \( \text{MAP}_2 \) (Figure 3) for a type \( S \cdot t \).

```plaintext
(*val fold: {S: MAP} \to (α,β) S.t \to β\)*)
```

Given a function \( f \) that builds a \( β \) from a value of the open-recursive type \( S \cdot t \), fold builds a \( β \) from the closed type \( \text{Fix}(S) \cdot t \).

Here is an instance of \( \text{MAP}_2 \) for lists:

```plaintext
implicit module ListF = struct
  type (α,τ) t = (α,τ) listr
  let map f g = function | h :: t | f h :: g t end
end
```

and a new definition of \( \leftrightarrow \) built from the generic fold:

```plaintext
(* val (\leftrightarrow): α L.t \to α L.t \to α L.t *)
```

```plaintext
let (\leftrightarrow) l r = fold (function 'Nil \to r | c \to 'R c) l
```

#### Generic folds for partially-static data

Figure 3 also introduces an extended bifunctor interface, \( \text{MAP}_2 ps \), that adds an multi-stage \( \text{map}_{ps} \) operation. Using \( \text{MAP}_2 ps \) we can build a generic fold over partially-static data, parameterised by a bifunctor \( S \) and two PS instances:

```plaintext
(*val foldps: {S: MAP} \to (A,PS) S.t \to β\)*)
```

Given functions \( \text{now} \) and \( \text{later} \) that build partially-static and dynamic values (of types \( B.ps \) and \( B.ps \) code) from values of the open partially-static and dynamic values (of types \( A.ps,B.ps \) and \( A.t,B.t \) code), \( \text{fold}_{ps} \) builds a partially-static value of type \( B.ps \) from the closed partially-static type \( (A.ps,A.t) \text{Fix}_{ps}(S) \cdot ps \).

As the implementation shows, the \( \text{now} \) function is used on static data, and the \( \text{later} \) function is passed to the fold function defined above to handle the dynamic case.

The \( \text{fold}_{ps} \) function relies on a PS instance for \( \text{Fix}_{ps}(S) \). Figure 4 defines a suitable instance, built from the \( \text{MAP}_2 ps \) instance and a PS instance for the parameter type.

```plaintext
finally, here is an instance of \( \text{MAP}_2 ps \) for \( \text{listr} \):
```

```plaintext
module type MAP2 = sig
  type (α,β) t
  val map : (α \to γ) \to (β \to δ) \to (α,β) t \to (γ,δ) t
end
```

```plaintext
module type MAP2 ps = sig
  include MAP2
  val map_{ps}: (α \to γ code) \to (β \to δ code) \to (α,β) t \to (γ,δ) t code end
```

### Ongoing work

We have seen how to derive the partially-static form of a type, along with generic operations over partially-static data, like \( \text{fold}_{ps} \).

The \( \text{MAP}_2 ps \) instances used by these generic operations are not arduous to define, but we are investigating ways to generalize them to avoid the need to explicitly support staging. Requiring that \( \text{MAP} \) be a traversable functor (Gibbons and Oliveira 2009) seems promising, but a naive approach requires cross-stage persistence and introduces administrative terms into generated code.

We are also interested in defining partially-static versions of types without requiring open recursion.

Finally, we plan to complete the generic programming toolbox with support for other operations, apply it to larger examples, and release our MetaOCaml code as a reusable library.

### References


