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

**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 expressions 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 supports a finer-grained view. As the name suggests, partially-static data allows the components of a value to be classified individually; for example, a list might have a static prefix and a dynamic tail, a tree might have static structure and dynamic labels, or a complex number might have a static imaginary part and a dynamic real part.

Let us look at an example. Here is the standard unstaged code

```
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:

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

Partial evaluation allows the first list to be partially evaluated in the current stage of a program, and the second list to be fully 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.

Moreover, partially-static data supports a finer-grained classification of values into static and dynamic, a single dynamic datum can infect a much larger value. However, the notion of partially-static data supports a finer-grained view. As the name suggests, partially-static data allows the components of a value to be classified individually; for example, a list might have a static prefix and a dynamic tail, a tree might have static structure and dynamic labels, or a complex number might have a static imaginary part and a dynamic real part.

Let us look at an example. Here is the standard unstaged definition of parameterised lists, together with an append function\(^1\):

```
type α list = [] | (: :) of α * α list
(* val (++) : α -> 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:

```
(* val (++) : α list -> α list code -> α list code *)
let rec (++) l r = match l with
  | [] -> r
  | h :: t -> .< h :: .~(t ++ r) >.
```

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

**Keywords** staging, partial evaluation, generic programming

**Categories and Subject Descriptors** D.1.1 [Programming techniques]: Applicative (Functional) Programming

**Figure 1: partially-static data**

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

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

The `ps` interface (Figure 1) relates a type `t` to its partially-static counterpart `ps`. From a general transformation on types (Sheard and Diatchki). From a modular implicity (White et al. 2015) for overloaded functions.

The code type represents quoted expressions, which may be executed at some future stage. The brackets `.<e>`, build a quoted expression of type `t code` from an expression of type `t`. 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 `++`:

```
type α listps =
  [] | (::) of α * α listps | Dyn of α list code
(* let rec dyn : α listps -> α list code *)
let rec dyn = function
  | [] -> .< [] >.
  | h :: t -> .< h :: .~(dyn t) >.
  | Dyn l -> l
(* val (++) : α listps -> α listps -> α listps *)
let rec (++) l r = match l with
  | [] -> r
  | h :: t -> h : : (t ++ r)
  | Dyn l -> .< .~l ++ .~(dyn r) >.
```

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 `t` to the second list `r`. The function `dyn` converts `r` 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 `cd` 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 `listps` from `list` 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 `tps` by replacing each recursive occurrence of `t` in the definition, and adding an additional top-level constructor `Dyn of t code`.

\[\text{D.1.1 [Categories and Subject Descriptors]: Applicative (Functional) Programming} \]

**Keywords** staging, partial evaluation, generic programming

**Figure 1: partially-static data**

```
module Fix(S: sig t end) = struct
  type ps = ps
  type t
  val sta : t -> t code
  val dyn : ps -> t code
  val cd : t code -> ps
end
```

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

The code type represents quoted expressions, which may be executed at some future stage. The brackets `.<e>`, build a quoted expression of type `t code` from an expression of type `t`. 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 `++`:

```
type α listps =
  [] | (::) of α * α listps | Dyn of α list code
(* let rec dyn : α listps -> α list code *)
let rec dyn = function
  | [] -> .< [] >.
  | h :: t -> .< h :: .~(dyn t) >.
  | Dyn l -> l
(* val (++) : α listps -> α listps -> α listps *)
let rec (++) l r = match l with
  | [] -> r
  | h :: t -> h : : (t ++ r)
  | Dyn l -> .< .~l ++ .~(dyn r) >.
```

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 `t` to the second list `r`. The function `dyn` converts `r` 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 `cd` 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.
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`, an open-recursive version of `list` which uses a second parameter `ρ` where α list usually appear in the definition:

```
type (α, ρ) listr = [] | (::) of α * ρ
```

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

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

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

```
module Lps = Fixps(struct type (α, ρ) t = (α, ρ) 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 bi-functor over the open-recursive version of the type. For example, here is a generic `fold` parameterised by an implicit bifunctor S of type `MAP` (Figure 3) for a type `S.t`:

```
let rec fold {S: MAP} (α, β) f g = function
  | `Nil` = f (S.map id (fold f) ρ)
  | `Dyn c` = g c (fold f ρ)
```

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

Here is an instance of `MAP2` for `listr`:

```
implicit module ListF = struct
  type (α, ρ) t = (α, ρ) listr
  let map f g = function
    | [] -> [] | h :: t -> f h :: g t
end
```

and a new definition of `++` built from the generic `fold`:

```
(* val (++) : α L.t → α L.t → α L.t *)
let (++) l r = fold (function 'Will → r | c → 'R c) l r
```

**Generic Folds for Partially-Static Data.** Figure 3 also introduces an extended bifunctor interface, `MAP2 ps`, that adds an multi-stage map operation. Using `MAP2 ps` we can build a generic fold over partially-static data, parameterised by a bifunctor S and two PS instances:

```
let rec foldps {S: MAP2} (α, β) f g = function
  | `Nil` = f (S.map id (fold f) ρ)
  | `Dyn c` = g c (fold f ρ)
```

Given functions `f` and later that build partially-static and dynamic values of types `B.ps` and `B.code` from values of the open partially-static and dynamic values of types `(A.ps,B.ps) S.t` and `(A,B.t) S.t` code, `foldps` builds a partially-static value of type `B.ps` from the closed partially-static type `(A.ps,A.t) Fixps(S).ps`. As the implementation shows, the `now` function is used on static data, and the `later` function is passed to the `fold` function defined above to handle the dynamic case.

The `foldps` function relies on a PS instance for `Fixps(S)`. Figure 4 defines a suitable instance, built from the `MAP2 ps` instance and a PS instance for the parameter type.

Finally, here is an instance of `MAP2 ps` for `listr`:

```
module type MAP2 = sig
  type (α, β) t
  val map : (α → γ) → (β → δ) → ((α, β) t) → (γ, δ) t
end
module type MAP2 ps = sig
  include MAP2
  val mapps : (α → γ code) → (β → δ code) → ((α, β) t) → (γ, δ) t code
end
```

Figure 3: Bifunctors, with and without staging

```
implicit module Fixps {S: MAP2 ps} {P: PS} =
  struct
  type t = P.t Fix(S).t
  type ps = (P.ps,P.t) Fixps(S).ps
  let rec sta ('R x) = 'Sta (S.map P.sta sta x)
  let rec dyn = function
    | 'Sta x -> < 'R . ~(S.mapps P.dyn dyn x) >.
    | 'Dyn c -> c
  end
  let cd c = 'Dyn c
end
```

**Figure 4:** PS instance for fixpoints

```
implicit module ListFps = struct
  include ListF
  let mapps f g = function
    | [] -> < [] >.
    | h :: t -> < -(f h) >..< -(g t) >.
end
```

and a new definition of `++` built from the generic `foldps`:

```
(* val (++)' : {A:PS} → ((A.ps, A.t) Listr, t as β) → β *)
let (++)' l r =
  foldps ( function [] → r | c → 'Sta c)
   (< function [] → . ~(dyn r) | c → 'R c >.) l r
```

**Ongoing Work**

We have seen how to derive the partially-static form of a type, along with generic operations over partially-static data, like `foldps`. The `MAP2 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 `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.

**Acknowledgements.** This work was supported by Microsoft Research through its PhD Scholarship Programme.

**References**


