Generic Partially-Static Data (Extended Abstract)

David Kaloper-Meršinjak Jeremy Yallop

University of Cambridge, UK

dk505@cl.cam.ac.uk jeremy.yallop@cl.cam.ac.uk

Abstract

We describe a generic approach to defining partially-static data and corresponding operations.

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

Keywords staging, partial evaluation, generic programming

1. 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 a 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:

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

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

TyDe'16, September 18, 2016, Nara, Japan
ACM. 978-1-4503-4435-7/16/09...
http://dx.doi.org/10.1145/2976022.2976028

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

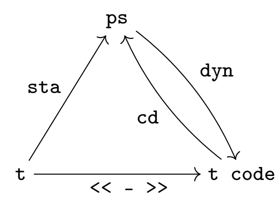

Figure 1: partially-static data

```
module Fix(S: sig type (_,_) t end) = struct
  type α t = [‘R of (α,α t) S.t]
end
module Fix_ps(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
```

Figure 2: Fixpoints and partially-static fixpoints

```
(* 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 `.< 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 list_ps =
  [] | (::) of α * α list_ps | Dyn of α list code
(* let rec dyn : α list_ps → α list code *)
let rec dyn = function
    []     → .< [] >.
  | h :: t → .< h :: .~(dyn t) >.
  | Dyn l  → l
(* val (++) : α list_ps → α list_ps → α list_ps *)
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 `l` is encountered, at which point it constructs a piece of code that prepends `l` 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 `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.

2. Partially-Static Data, Generically

The construction of $\texttt{list}_{ps}$ 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 $\texttt{t}_{ps}$ 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`, an open-recursive version of `list` which uses a second parameter ρ where α `list` would 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`[2]:

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

Similarly, an application of a second fixpoint operator, $\texttt{Fix}_{ps}$ (also Figure 2), gives us a partially-static version of lists:

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

3. 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 $\texttt{MAP}_2$ (Figure 3) for a type `S.t`.

```
(*val fold: {S: MAP2 } → ((α,β) S.t → β) → α Fix(S).t → β*)
let rec fold {S: MAP2 } f (‘R x) = f (S.map id (fold f) x)
```

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

Here is an instance of $\texttt{MAP}_2$ 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 ‘Nil → r | c → ‘R c) l
```

Generic Folds for Partially-Static Data. Figure 3 also introduces an extended bifunctor interface, $\texttt{MAP}_{2\,ps}$, that adds an multi-stage $\texttt{map}_{ps}$ operation. Using $\texttt{MAP}_{2\,ps}$ we can build a generic fold over partially-static data, parameterised by a bifunctor `S` and two `PS` instances:

```
(* val foldps : {S: MAP2 } → {A: PS} → {B:PS} →
 ((A.ps,B.ps) S.t → B.ps) → ((A.t,B.t) S.t → B.t) code
 → (A.ps,A.t) Fixps(S).ps → B.ps *)
let rec foldps {S: MAP2 } {A:PS} {B:PS} now later =
  function ‘Sta v → now (S.map id (foldps now later) v)
         | ‘Dyn c → B.cd .< fold .~later .~c >.
```

Given functions `now` and `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) S.t` and `(A.t,B.t) S.t code`), $\texttt{fold}_{ps}$ builds a partially-static value of type `B.ps` from the closed partially-static type `(A.ps,A.t)` $\texttt{Fix}_{ps}$`(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.

[2] Some technical notes: we define `Fix` as a functor to make use of higher-kinded polymorphism, and we use structural variants (distinguished by a backtick on constructors) to sidestep problems with type generativity.

```
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
  let cd c = ‘Dyn c
end
```

Figure 4: `PS` instance for fixpoints

The $\texttt{fold}_{ps}$ function relies on a `PS` instance for $\texttt{Fix}_{ps}$`(S)`. Figure 4 defines a suitable instance, built from the $\texttt{MAP}_{2\,ps}$ instance and a `PS` instance for the parameter type.

Finally, here is an instance of $\texttt{MAP}_{2\,ps}$ for `listr`:

```
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 $\texttt{fold}_{ps}$:

```
(*val (++): {A:PS} → ((A.ps,A.t) Lps.t as β) → β → β *)
let (++) {A:PS} l r =
  foldps (   function [] →  r        | c → ‘Sta c)
         (.< function [] → .~(dyn r) | c → ‘R c >.) l
```

4. Ongoing Work

We have seen how to derive the partially-static form of a type, along with generic operations over partially-static data, like $\texttt{fold}_{ps}$. The $\texttt{MAP}_{2\,ps}$ 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 `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

J. Gibbons. *Datatype-Generic Programming*, pages 1–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

J. Gibbons and B. c. d. s. Oliveira. The essence of the iterator pattern. *J. Funct. Program.*, 19(3-4):377–402, July 2009.

O. Kiselyov. The design and implementation of BER MetaOCaml. In *Functional and Logic Programming*, LNCS. Springer, 2014.

T. Sheard and I. S. Diatchki. Staging algebraic datatypes. Unpublished manuscript. `http://web.cecs.pdx.edu/~sheard/papers/stagedData.ps`.

L. White, F. Bour, and J. Yallop. Modular implicits. ACM Workshop on ML 2014 post-proceedings, September 2015.