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 guarantees 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 \(\oplus\):

\[
\begin{align*}
type \ \alpha \ \text{list} &= [] \mid (:\) \ \alpha \ \text{*} \ \alpha \ \text{list} \\
(* \ \text{val} \ (\oplus)) : \ \alpha \ \alpha \ \text{list} &\to \alpha \ \text{list} \to \alpha \ \text{list} \\
\text{let rec} \ (\oplus) \ 1 \ r &= \text{match} \ 1 \ \text{with} \\
&[] \to r \\
&h :: t \to h :: (t \oplus r)
\end{align*}
\]

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

\[
\begin{align*}
type \ \alpha \ \text{list}_\text{ps} &= [] \mid (:\) \ \alpha \ \text{*} \ \alpha \ \text{list}_\text{ps} \\
(* \ \text{val} \ (\oplus)) : \ \alpha \ \text{list}_\text{ps} &\to \alpha \ \text{list}_\text{ps} \to \alpha \ \text{list}_\text{ps} \\
\text{let rec} \ (\oplus) \ 1 \ r &= \text{match} \ 1 \ \text{with} \\
&[] \to r \\
&h :: t \to h :: (t \oplus r)
\end{align*}
\]

1. Static vs Dynamic

\[
\begin{align*}
type \ \alpha \ \text{list}_\text{ps} &= [] \mid (:\) \ \alpha \ \text{*} \ \alpha \ \text{list}_\text{ps} \\
(* \ \text{val} \ (\oplus)) : \ \alpha \ \text{list}_\text{ps} &\to \alpha \ \text{list}_\text{ps} \to \alpha \ \text{list}_\text{ps} \\
\text{let rec} \ (\oplus) \ 1 \ r &= \text{match} \ 1 \ \text{with} \\
&[] \to r \\
&h :: t \to h :: (t \oplus r)
\end{align*}
\]

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

\[
\begin{align*}
\text{module type} \ \text{PS} &= \text{sig} \\
\text{type} \ t \\
\text{type} \ ps \\
\text{val} \ \text{dyn} : ps \to t \ \text{code} \\
\text{val} \ \text{sta} : t \to ps \\
\text{val} \ \text{cd} : t \ \text{code} \to ps
\end{align*}
\]

\[
\begin{align*}
\text{let rec} \ (\oplus) \ l \ r &= \text{match} \ l \ \text{with} \\
&[] \to r \\
&h :: t \to h :: (t \oplus r)
\end{align*}
\]

\[
\begin{align*}
\text{module Fix}(S: \text{sig} \ t \ \text{code} \ \beta) = \text{struct} \\
\text{type} \ t = [\text{R of } (\alpha, \alpha \ t) \ S.t] \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{module Fixps}(S: \text{sig} \ t \ \text{code} \ \beta) = \text{struct} \\
\text{type} \ (\alpha, \beta) \ ps = [\text{Sta of } (\alpha, \alpha, \beta) \ ps \ S.t] \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{let rec} \ (\oplus) \ l \ r &= \text{match} \ l \ \text{with} \\
&[] \to r \\
&h :: t \to h :: (t \oplus r)
\end{align*}
\]

\[
\begin{align*}
\text{module} \ \text{Fix}(S: \text{sig} \ t \ \text{code} \ \beta) = \text{struct} \\
\text{type} \ t = [\text{R of } (\alpha, \alpha \ t) \ S.t] \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{module Fixps}(S: \text{sig} \ t \ \text{code} \ \beta) = \text{struct} \\
\text{type} \ (\alpha, \beta) \ ps = [\text{Sta of } (\alpha, \alpha, \beta) \ ps \ S.t] \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{let rec} \ (\oplus) \ l \ r &= \text{match} \ l \ \text{with} \\
&[] \to r \\
&h :: t \to h :: (t \oplus r)
\end{align*}
\]
2. Partially-Static Data, Generically

The construction of listp from list is an instance of a more general transformation on types (Sheard and Diatchki). From a definition for a type \(\tau\), we can obtain a partially-static counterpart \(\tau_{ps}\) by replacing each recursive occurrence of \(\tau\) in the definition, and adding an additional top-level constructor dyn of \(\tau\) 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 \(\rho\) where \(\alpha\) list would usually appear in the definition:

\[
\text{let rec sta ('R x) = 'Sta (S.map P.sta sta x)}
\]

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

\[
\text{module L}_{ps} = \text{Fix}_{ps}(\text{struct type } \alpha \rightarrow \beta \rightarrow \alpha \text{ listr end})
\]

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 generically 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 a fold parameterised by an implicit bifunctor \(\alpha\) of type \(\text{fold}_{ps}\) (Figure 3) for a type \(\tau\).

\[
\text{module L = Fix(struct type } \alpha \rightarrow \beta \rightarrow \alpha \text{ listr end)}
\]

Figure 3 also introduces an extended bifunctor interface, \(\tau_{ps}\), that adds multi-stage administrative terms into generated code. Using \(\text{fold}_{ps}\), we can build a generic fold over partially-static data, parameterised by a bifunctor \(\alpha\) and two \(\text{PS}\) instances:

\[
\text{map} = \text{function } [] \rightarrow [] | h :: t \rightarrow h :: \text{ps} \text{ g t end}
\]

Generic Folds for Partially-Static Data. Figure 3 also introduces an extended bifunctor interface, \(\tau_{ps}\), that adds multi-stage administrative terms into generated code. Using \(\text{fold}_{ps}\), we can build a generic fold over partially-static data, parameterised by a bifunctor \(\alpha\) and two \(\text{PS}\) instances:

\[
\text{map} = \text{function } [] \rightarrow [] | h :: t \rightarrow h :: \text{ps} \text{ g t end}
\]

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

module type MAP = sig
  type (\(\alpha,\beta\)) t
  val map : (\(\alpha \rightarrow \gamma\) code) \(\rightarrow \beta \rightarrow \alpha \text{ listr}\) \(\rightarrow \gamma,\delta\) code\(\rightarrow \alpha \text{ listr}\) t
end

module type MAP = sig
  type (\(\alpha,\beta\)) t
  val map : (\(\alpha \rightarrow \gamma\) code) \(\rightarrow \beta \rightarrow \alpha \text{ listr}\) \(\rightarrow \gamma,\delta\) code\(\rightarrow \alpha \text{ listr}\) t
end

module type MAP = sig
  type (\(\alpha,\beta\)) t
  val map : (\(\alpha \rightarrow \gamma\) code) \(\rightarrow \beta \rightarrow \alpha \text{ listr}\) \(\rightarrow \gamma,\delta\) code\(\rightarrow \alpha \text{ listr}\) t
end

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 \(\text{fold}_{ps}\). The \(\text{map}_{ps}\) 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.

Acknowledgements

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

References


