Partially static data as free extension of algebras

Jeremy Yallop
jeremy.yallop@cl.cam.ac.uk
Department of Computer Science and Technology
University of Cambridge
England

Tamara von Glehn
T.L.Von-Glehn@dpmms.cam.ac.uk
Department of Pure Mathematics and Mathematical Statistics and Newnham College
University of Cambridge
England

Ohad Kammar
ohad.kammar@cs.ox.ac.uk
Department of Computer Science and Balliol College
University of Oxford
England

Abstract
Partially-static data structures are a well-known technique for improving binding times. However, they are often defined in an ad-hoc manner, without a unifying framework that ensures full use of the equations associated with each operation.

We present a foundational view of partially-static data structures as free extensions of algebras for suitable equational theories, i.e. the coproduct of an algebra and a free algebra in the category of algebras and their homomorphisms. By precalculating these free extensions, we construct a high-level library of partially static data representations for common algebraic structures. We demonstrate our library with common use-cases from the literature: string and list manipulation, linear algebra, and numerical simplification.

Keywords multi-stage compilation, metaprogramming, partial evaluation, partially static data, universal algebra

1 Introduction
Partial evaluation, multi-stage programming, and related approaches, can improve the performance of programs by utilising the distinction between static inputs (available now) and dynamic inputs (available later). The classification extends from inputs to expressions: static expressions, depending only on static inputs, can be evaluated in advance, combining the resulting dynamic expressions into a more specialized program with improved performance.

Unfortunately, a straightforward application of this approach often produces disappointing results. Naive binding-time analyses can infect large parts of a program, since any expression that has a dynamic variable as a sub-expression is considered dynamic, resulting in a residual program in which most expressions remain unreduced.

For example, suppose that in the expression \((x + y) + z\) the variables \(x\) and \(z\) are static, and \(y\) is dynamic. Then \(x + y\) is also dynamic, because it has \(y\) as a sub-expression and, in turn, the entire expression is dynamic, with no non-trivial sub-expressions:

\[ ((x^{sta} + y^{dyn})^{dyn} + z^{sta})^{dyn} \]

The situation improves if we use the associative and commutative properties of \(+\) to rewrite the expression, quarantining the dynamic expression \(y\), and exposing a static sub-expression \(x + z\):

\[ ((x^{sta} + z^{sta})^{sta} + y^{dyn})^{dyn} \]

However, it is impractical to rewrite all programs in this way. Variables may have different binding times on different invocations of a function; in such circumstances it is not possible to rewrite expressions to group static sub-expressions together. Furthermore, this kind of rewriting, using well-defined algebraic laws, is better performed by a computer than by a human programmer.

Partially-static data [Mogensen 1988] replaces the all-or-nothing distinction between static and dynamic with hybrid data structures, parts of which are in the present, and others in the future. Partially-static operations [Thiemann 2013] perform some computation statically, despite the presence of unknown data. Further, the implementation of these operations uses laws to reorder the dynamic portions, normalizing and optimizing the residual portions of the program.

While the spirit of this approach is uniform, the concrete data structures in each case are very different. Our central contribution is to conceptualise them with a universal property, in terms of the operations and equations we utilise. Universality translates into a functional specification which we need to implement and validate, and replaces the uncertainty of designing a new data structure with the precise activity of implementing a specification.

Contributions. Following a survey of motivating examples drawn from the literature (§2), we present the following contributions:

- §3 introduces our approach informally, using partially-static monoids as an extended use-case.
- §4 presents the universal properties for law-respecting partially-static structures as free extensions of algebra for the associated equational theories.
We precalculate the free extensions of common algebraic structures: monoids, commutative rings and semirings, and abelian groups. For brevity, we present only monoids in detail.

§5 describes a high-level library of partially-static structures based on §3 and §4. The parameterized modules in the library can be instantiated with concrete implementations of algebraic interfaces — a monoid for strings, a ring for complex numbers, and so on — to provide drop-in modules that perform optimizations using the associated laws.

2 Motivating examples

A principled approach to partially-static data that takes algebraic laws into account can improve the output of a wide variety of staged without altering the non-local structure of the generating program. We present several illustrative examples as MetaOCaml programs [Kiselyov 2014].

Printf. Functions whose arguments arrive at different times lend themselves well to a multi-stage approach. The typed sprintf function studied by Danvy [1998] and Asai [2009] is one such example, since the format string is typically known in advance of the values passed as subsequent arguments. Yallop and White [2015] use staging to turn sprintf from a function into a code generator; however, a naive approach results in code that contains too many catenations. For example, the following call to sprintf generates a function that prints two integer arguments with "ab" interposed:

```ocaml
fun (int -> lit "a") ++ (lit "b" ++ int)
```

When sprintf is staged using a straightforward binding time analysis the result contains four catenations:

```ocaml
.< (int ++ lit "a") ++ (lit "b" ++ int)>
```

With our library the expression above is further reduced to

```ocaml
.< (int ++ lit "a") ++ (lit "b" ++ int)>
```

Power. Consider the staged power function, fun x —> x^n, with n statically known. Once again, a naive approach generates suboptimal code. The staged power function of Taha [2003] builds a computation with too many multiplications, including an unnecessary multiplication by 1:

```ocaml
power 5 .<x>. \(\sim\) .< 1 * (x * (x * (x * (x * x)))) >.
```

Using the fact that integers with multiplication form a commutative monoid, our commutative semiring implementation reduces the 5 multiplications to 3:

```ocaml
power 5 .<x>. \(\sim\) .< let y = x * x in let z = y * y in x * z >.
```

Linear algebra. Linear algebra offers many opportunities for optimisation via multi-stage specialization and numerical simplification such as: the Fast Fourier Transform [Kiselyov et al. 2004], Gaussian elimination [Carette and Kiselyov 2011], and matrix-vector multiplication [Aktemur et al. 2013]. The inner product illustrates the principle: given a statically-known vector \(s = [1; \theta; z]\) and a dynamic vector \(d = [x; y; z]\), a naively-staged inner product function might generate the following code:

```ocaml
.< (1 * x) + (0 * y) + (2 * z) >.
```

Using the fact that integers form a commutative semiring, our library generates the following simpler code:

```ocaml
.< x + (2 * z) >.
```

all and any. The examples so far all involve constructing and then residualizing partially-static values. It is also sometimes useful to compute with partially-static values before residualization.

The all function takes a predicate \(p\) and a list \(l\), and returns true iff every element of \(l\) satisfies \(p\). Our approach supports defining a variant of all that operates on partially-static lists, with interleaved static and dynamic portions, and that produces partially-static booleans. Since a single element that does not satisfy \(p\) is enough to determine the result of all, the result may be static even where the input is partially unknown:

```ocaml
all even ([2; 4] ++ .<x>. ++ [3])
```

```ocaml
\(\sim\) even 2 \&\& even 4 \&\& .< any even x >. \&\& even 3
```

With our library the expression above is further reduced to the static value false, using the fact that booleans form a commutative semiring. The dual function any can be defined similarly.

In the examples above, partially static operations (+, *, etc.) are explicit. Our approach also covers partially-static datatypes without operations or equations:

Possibly-static data. When instantiating the universal property for the empty theory, i.e. data with no operations and no laws, the free extension degenerates into ordinary sum types, yielding a partially-static structure known as possibly-static, whose values are either entirely static or entirely dynamic. Possibly-static values can be used to write programs that can accept a particular input as static or dynamic:

```ocaml
is_digit (Dyn c) \(\sim\) Dyn .< is_digit .~c >.
```

```ocaml
is_digit (Sta '3') \(\sim\) Sta true
```

\(^1\)The full code involves injections from static and dynamic values into a common type (§3).
Partially static data as free extension of algebras

Partially-static algebraic datatypes. More generally, inductive algebraic datatypes can be seen as initial algebras for a multi-sorted signature, i.e. free algebras of operations without laws. These datatypes are useful in programs that perform staged computation. Lists with possibly-dynamic tails are a common example of a more general family of partially-static datatypes [Inoue 2014; Kaloper-Meršinjak and Yallop 2016; Sheard and Diatchki 2002].

For example, a variant of map that takes functions for both static and dynamic values can traverse the initial portion of a partially-static list, leaving traversal of the dynamic tail for later. Supposing \( \text{lst} = 0 :: 1 :: \ldots \).

```haskell
module type MONOID = sig
  type t
  val \text{\texttt{}} : t
  val \text{\texttt{\|}} : t \rightarrow t \rightarrow t
end
```

**Figure 1.** Monoids and their laws

Partially-static monoid: interface. What do we need from a partially-static monoid \( \text{PS} \)?

First, if \( \text{PS} \) is to stand in for other monoids in multi-stage programs, it must implement the \text{MONOID} interface in a way that satisfies the familiar laws (Figure 1). Ideally, \( \text{PS} \) should be canonical: expressions that are statically equivalent under the monoid laws should have the same representation in \( \text{PS} \).

Second, it should be possible to use partially-static values in place of fully-static or fully-dynamic values, and so \( \text{PS} \) should support injections from static and dynamic data.

Third, it should be possible to residualize computations in \( \text{PS} \) – i.e. to turn partially-static monoid values into code. Generalizing a little, it should be possible to inspect partially-static data – or at least, to inspect its static structure – both in order to residualize and to perform transformations such as \( \text{all} \) (§2). Residualization, and destruction in general, should also preserve the monoid laws, so that programs that are equivalent under the monoid laws should residualize to programs that are also equivalent under the laws.

Finally, since the aim is to improve generated code, performing as much computation as possible in advance, \( \text{PS} \) should never unnecessarily convert static values to dynamic values.

These requirements suggest the following interface, which includes the \text{MONOID} interface (line 2), and supports injections from static and dynamic values (lines 4 and 5) along with a mapping into another monoid \( C \) based on mappings for static and dynamic values (lines 6–8):

```haskell
module type \text{PS} = sig
  include \text{MONOID}
  type \text{sta}
  val \text{\texttt{\texttt{\texttt{\|}}}} : \text{sta} \rightarrow t
  val \text{\texttt{\texttt{\texttt{\|}}}} : \text{sta var} \rightarrow t
  module \text{Eva}(C: \text{MONOID}) : sig
    val \text{\texttt{\texttt{\texttt{\|}}}} : (\text{sta var} \rightarrow C.t) \rightarrow (\text{sta var} \rightarrow C.t) \rightarrow t \rightarrow C.t
  end
end
```

The \text{var} type, discussed further in §5, is a retract of code that represents only dynamic variables.

Partially-static monoid: implementation. How do we implement \( \text{PS} \) while satisfying the monoid laws, use pre-computed static values, and generate optimal code?

Starting from the four \( \text{PS} \) operations: \( \text{\texttt{\texttt{\texttt{\|}}}}; \text{\texttt{\texttt{\texttt{\|}}}}; \text{\texttt{\texttt{\texttt{\|}}}}; \text{\texttt{\texttt{\texttt{\|}}}} \); and \text{dyn}, we naively define the following tree type, with one constructor for each:

```haskell
type \text{t} = \text{Unit} | \text{Mul of t x t}
      | \text{Sta of string} | \text{Dyn of string var}
```

However, this implementation ignores the monoid laws, allowing many different representations for values (such as \text{Mul (Unit,Unit) and Unit}) that ought to be considered equal. Applying the laws eliminates the redundancy, flattening the nesting so that the association is all in one direction:

```haskell
type \text{t} = \text{Nil} | \text{Cons of atom x t}
and \text{atom} = \text{Sta of string} | \text{Dyn of string var}
```

Now \( \text{\texttt{\texttt{\texttt{\|}}}} \) can be defined by the familiar append function which, of course, respects the monoid laws.

We might take things one step further. It is clearly desirable for \text{sta} to be a homomorphism with respect to \( \text{\texttt{\texttt{\texttt{\|}}}} \), i.e.

```haskell
\text{sta x \text{\texttt{\texttt{\texttt{\|}}}} sta y = sta (x \text{\texttt{\texttt{\texttt{\|}}}} y)}
```

This suggests that adjacent static values in the list should be coalesced (Figure 5). With a little care it is possible to enforce the constraint in the type, using a GADT index [Garrigue and Normand 2011] instantiated by \( s \) or \( d \) to track whether a list starts with a static or dynamic element:

```haskell
\text{type _ alt = Empty : _ alt}
  | ConsS : string x d alt \rightarrow s alt
  | ConsD : string var x _ alt \rightarrow d alt
\text{type alt ex = T : _ alt \rightarrow alt ex}
```

The existential type \text{alt ex} hides the index to build a parameterless type that can be used to implement \text{t} in \( \text{PS} \).
This representation is still not quite canonical, since ConsS can store empty strings, a shortcoming that can be overcome with further type trickery [Kiselyov and Shan 2007]. It is (comparatively) straightforward to define an append_alt function that catenates two alt values, combining adjacent static strings using the standard \(^{\star}\) operator.

Finally, eva interprets a value of type \(t\) in some other monoid \(C\), mapping constituent static and dynamic values individually, and mapping monoid operations to the operations of \(C\). In other words, the following expression
\[
eva f g (s \otimes (d \otimes (s \otimes \ldots \otimes 1)))
\]
becomes
\[
f s \otimes c (g d \otimes c (f s \otimes c \ldots \otimes c 1c))
\]
As an optimization, the unit may be omitted where the value is non-empty, so that eva \(f g (s \otimes 1)\) becomes \(f s\) rather than \(f s \otimes c 1c\).

A common use of eva is residualization, which turns a partially-static value into a fully-dynamic value. Residualization is implemented by instantiating \(C\) to the monoid that maps \(x \otimes y\) to \(. \cdot -x \cdot \cdot -y\cdot\) and \(1\) to \(. \cdot "\cdot "\cdot\), and supplying the function that residualizes a single string value and the identity function as the two arguments of eva. Then
\[
eva lift_string id (s_1 \otimes (d \otimes (s_2 \otimes 1))) \sim . \cdot -((\text{lift_string } s_1) \cdot \cdot d \cdot \cdot -((\text{lift_string } s_2) >).
\]
Here is the implementation of Ps_\_string. (App. A gives implementations for append_alt and Eva_alt.)

```ocaml
module type Ps_\_string = struct
  type t = alt_en and sta = string
  let sta s = T (ConsS (s, Empty))
  let dyn d = T (ConsD (d, Empty))
  let 1 = empty
  let (\&) = append_alt
  module Eva = Eva_alt
end
```

**Improving printf.** §2 showed the effects of the partially-static monoid on the code generated by a staged sprintf function. We now show how to transform the implementation of sprintf to achieve those effects.

Figures 2 and 3 give minimal interfaces for unstaged and staged formatted printing. The type \(t\) represents format specifications; its two parameters respectively represent the result and the input type of a sprintf instantiation. The following three operations construct format strings: \(lit\) \(s\) is a format string that accepts no arguments and prints \(s\); \(x ++ y\) catenates \(x\) and \(y\); \(int\) is a format string that accepts and prints an integer argument. Finally, sprintf combines a format string with corresponding arguments to construct formatted output. Asai [2009] gives further details.

Here is an implementation of Figure 2 in continuation-passing style (CPS), using an accumulator:

```ocaml
let type (\_\_\_\_) t
val lit : string -> (a, a) t
val (++) : (\(\beta, a\) t -> (y, \(\beta\) t -> (y, a) t
val int : (a, int code -> a) t
val sprintf : (string, a) t -> a

Figure 2. printf signature

let type (\(\alpha,\rho\) t = (string -> a) -> string -> \(\rho\)
val lit x k s = k (s ^ x)
val (++) f g k = f (g k)
val int k s x = k (s ^ string_of_int .~x).
let sprintf p = p id ""

With this implementation, a format string is a function accepting a continuation argument \(k\) and an accumulator \(s\). Both list and int call \(k\) directly, passing an extended string; 
++ is simply function composition. The function sprintf passes the identity function as a top-level continuation along with an empty accumulator.

Staging sprintf is straightforward. We treat format strings statically; arguments and, consequently, the accumulator, are dynamic. The ++ function is left unchanged, and the remainder of the implementation acquires brackets and escapes to match the assignment of static and dynamic classifications:

```

The generated code (§2) is suboptimal precisely because the staging is straightforward: every catenation is delayed, even where both operands are available in advance.

Staging using our partially-static monoid is also straightforward. The steps are as follows, starting from the unstaged implementation: replace \(\text{string with Ps_\_string.t, replace } ^{\star}\) and "" with \(\otimes\) and \(1\), insert sta and dyn to inject static and dynamic expressions, and replace the top-level continuation with the residualization function described above:

```ocaml
let type (\(\alpha,\rho\) t = (Ps_\_string.t ->\(\alpha\)) -> Ps_\_string.t ->\(\rho\)
val lit x k s = k (s \otimes sta x)
val int k s x = k (s \otimes dyn .<string_of_int ."x">)
let sprintf p = p cd 1
```

Jeremy Yallop, Tamara von Glehn, and Ohad Kammar
These equations are called axioms. The example signature and axioms above form the description of an algebraic structure. To describe the universal property for partially static data, the symbols in $\Sigma$ are in $\text{Ax}$, for example, the three monoid axioms $P_1$, $P_2$, and $P_3$.

Put together, the description of an algebraic structure is called a presentation $\mathcal{P}$, given by a pair $\langle \Sigma, \text{Ax}_\mathcal{P} \rangle$ consisting of a signature $\Sigma$ and a set $\text{Ax}_\mathcal{P}$ of axioms over this signature. The example signature and axioms above form mon — the presentation of monoids (cf. Fig. 1).

An algebra for a presentation is a mathematical implementation of such specifications. Formally, given a presentation $\mathcal{P}$, a $\mathcal{P}$-algebra $A$ is a pair $\langle A, \rho \rangle$ consisting of a set $|A|$, called the carrier of the algebra, and, for each operation symbol $f : n \in \Sigma$, an $n$-ary function $f_A : |A|^n \rightarrow |A|$, such that all the axioms in $\text{Ax}_\mathcal{P}$ hold. For example, noting that a nullary function is a constant, a monoid-algebra is a monoid.

Finally, given two $\mathcal{P}$-algebras $A, B$, a $\mathcal{P}$-homomorphism $h : A \rightarrow B$ is a function between the carriers $h : |A| \rightarrow |B|$ that respects the operations: for each operation symbol $f : n \in \Sigma$, and for every $n$-tuple $\bar{a} = (a_1, \ldots, a_n)$ of $|A|$-elements, we have $h(f_A(a_1, \ldots, a_n)) = f_B(h(a_1), \ldots, h(a_n))$. For example, a monoid-homomorphism $h : A \rightarrow B$ is a function that satisfies $h(1_A) = 1_B$ and $h(x \otimes y) = h(x) \otimes h(y)$, i.e. the familiar notion of a monoid homomorphism.

For each presentation $\mathcal{P}$, the collection of $\mathcal{P}$-algebras and $\mathcal{P}$-homomorphisms between them forms a category $\mathcal{P}$-$\text{Alg}$, with the identities and composition given by the identity functions and the usual composition of functions. We have an evident functor $|-| : \mathcal{P}$-$\text{Alg} \rightarrow \text{Set}$ that forgets the algebra structure on objects and the homomorphism requirement on morphisms.

The forgetful functor $|-|$ always has a left adjoint $F_\mathcal{P} : \text{Set} \rightarrow \mathcal{P}$-$\text{Alg}$. Concretely, its object map on a set $X$ yields the term algebra over $X$: the set of $\Sigma$-terms with variables in $X$, quotiented by the deductive closure of $\text{Ax}_\Sigma$ under the derivations of equational logic. For example, the free monoid over $X$ is the set of finite sequences with $X$-elements, as every pair of $\Sigma_{\text{mon}}$-terms are equivalent to the sequence formed by their tree-fringe, with the unit elements omitted, represented by a spine. The unit of the adjunction, $\eta^\mathcal{P} : X \rightarrow F_\mathcal{P}X$, maps an element $x \in X$ to its equivalence class as a term. For mon, $\eta_{\text{mon}}^\mathcal{P}(x)$ is the one-element sequence $[x]$. The adjunction itself assigns to every function $f : X \rightarrow |A|$ its homomorphic extension $\Rightarrow^\mathcal{P} f : F_\mathcal{P}X \rightarrow A$, which evaluates (the equivalence class of) a term in the algebra $A$, with $X$-variables substituted according to $f$. For example, taking $A$ to be the natural numbers with multiplication:

$\eta^\mathcal{P} : X \rightarrow F_\mathcal{P}X$ maps an element $x \in X$ to its equivalence class as a term. For mon, $\eta_{\text{mon}}^\mathcal{P}(x)$ is the one-element sequence $[x]$. The adjunction itself assigns to every function $f : X \rightarrow |A|$ its homomorphic extension $\Rightarrow^\mathcal{P} f : F_\mathcal{P}X \rightarrow A$, which evaluates (the equivalence class of) a term in the algebra $A$, with $X$-variables substituted according to $f$. For example, taking $A$ to be the natural numbers with multiplication:

$[x; y; z] \Rightarrow^\mathcal{P} \eta_{\text{mon}}^\mathcal{P}(x) \mapsto (x \mapsto 2, y \mapsto 3, z \mapsto 4) = 2 \cdot 3 \cdot 4 = 24$

The categories $\mathcal{P}$-$\text{Alg}$ have coproducts $A \oplus B$, and their concrete structure is given as follows. The carrier $|A \oplus B|$ is the $\Sigma$-$\text{Alg}$-term algebra over the disjoint union $|A| \amalg |B|$ quotiented by the deductive closure of the axioms in $\mathcal{P}$, together with the equations of the form:

$$f(t_1a_1, \ldots, t_nh_{a_n}) \equiv f(t_1f_A(a_1, \ldots, a_n))$$

for every $f : n \in \Sigma$, and analogous equations for $B$. The coproduct injection $i_B^\mathcal{P} : A \rightarrow A \oplus B$ maps $a$ to the equivalence class of $i_1a$, and similarly for $B$. For every pair of homomorphisms $h_1 : A \rightarrow C$, $h_2 : B \rightarrow C$, the unique coCartoupling homomorphism $[h_1, h_2] : A \oplus B \rightarrow C$ interprets a term over $|A| + |B|$ as the corresponding $|C|$-element, once each variable $i_1x$ is substituted by $h_1(x)$. A free extension of an algebra $A$ by a set $X$ is the coproduct of the algebra $A$ with the free algebra over $X$, namely $\text{ps}(A, X) \equiv A \oplus F_\mathcal{P}X$. Combining the universal properties of coproducts and adjunctions, it is characterised by an algebra $\text{ps}(A, X)$ together with a homomorphism $\iota_X : X \rightarrow \text{ps}(A, X)$, and a function $\iota_A : A \rightarrow \text{ps}(A, X)$, such that for every other pair of a homomorphism $h : A \rightarrow C$ and a function $e : X \rightarrow |C|$, there exists a unique homomorphism $\text{eva}(h, e) : \text{ps}(A, X) \rightarrow C$ satisfying $\text{eva}(h, e) \circ \iota_A = h$ and $\text{eva}(h, e) \circ \iota_X = e$. Partly static data as free extension of algebras

![Figure 4. Partially-static monoid: dropping 1](image)

![Figure 5. Coalescing adjacent static values](image)
4.2 Conceptual justification

We have two different arguments for using free extensions of algebras as the appropriate functional specification for partially static data. In both, the algebra $A$ stands for the static datatype, and the set $X$ stands for a collection of dynamically-known values. The free extension $\text{ps}(A, X)$ then supports the first two operations for partially-static data:

\[
st \coloneqq \iota_A : A \rightarrow \text{ps}(A, X) \quad \text{dyn} \coloneqq \iota_X : X \rightarrow \text{ps}(A, X)
\]

In the first argument, the universal property requires a conceptual leap: we have no direct justification to the existence of the map $\text{eva}(h, e)$ for every other algebra $C$. However, if we strengthen the requirements of partially static data to allow any homomorphic post-processing, and not just late-binding, we indeed obtain the existence of the desired homomorphism $\text{eva}(h, e)$ and the associated two equations. The uniqueness requirement represents minimality of the datatype.

For the second argument, we observe the following fact:

**Proposition 4.1.** Let $P$ be a presentation, and $X$ a set. Assume a choice of a set $\text{ps}(A, X)$ for every algebra $A$, a homomorphism $\iota_A : A \rightarrow \text{ps}(A, X)$, and a function $\iota_X : X \rightarrow \text{ps}(A, X)$ such that:

- For every function $e : X \rightarrow |A|$ there is a unique homomorphism $\text{eva}(id, e) : \text{ps}(A, X) \rightarrow A$ satisfying:
  \[
  \text{eva}(id, e) \circ \iota_A = \text{id} \quad \left| \text{eva}(h, e) \right| \circ \iota_X = e
  \]
- For every homomorphism $h : A \rightarrow B$, there is a unique homomorphism $\text{ps}(h, X) : \text{ps}(A, X) \rightarrow \text{ps}(B, X)$ satisfying:
  \[
  \text{ps}(h, X) \circ \iota_A = \iota_B \circ h \quad \text{ps}(h, X) \circ \iota_X = \iota_X
  \]

Then $\text{ps}(A, X)$, together with $\iota_A, \iota_X$ and $\text{eva}(h, e) \coloneqq \text{eva}(id, e) \circ \text{ps}(h, X)$ form the free extension of $A$ with $X$.

While more technical, this justification adds a uniformity requirement. First, partially static datatypes should exist for every algebra. Second, as the datatype stores representations of hybrid terms consisting only of $A$ elements and $X$ elements, the functor $\text{ps}(\cdot, X)$ represents a uniformity assumption about the way $A$ elements are stored. The uniqueness requirements require this representation to be minimal.

4.3 Algebraic structure

As an example, the free extension of a monoid $A$ with a set $X$ has as carrier the set:

\[
|\text{ps}(A, X)| \coloneqq A \times \bigoplus_{n \in \mathbb{N}} (X \times A)^n
\]

As a more complicated example, recall that a commutative ring $(A, 0, \oplus, 1, \otimes)$ where $(A, 0, \oplus)$ forms an abelian group, $(A, 1, \otimes)$ forms a commutative monoid, together with a distributivity law $x \otimes (y \oplus z) \equiv (x \otimes y) \oplus (x \otimes z)$. The free extension of a commutative ring $A$ with a set $X$ is the commutative ring $A[X]$ of multinomials with coefficients in $A$ and variables in $X$.

5 A library for partially-static data

We designed a general-purpose library for generating simplified algebraic code using the principles of §3 and §4.

**Interface.** The $\text{PS}_{\_\text{string}}$ module of §3 generalizes to support arbitrary monoids by parameterizing by the $\text{MONOID}$ instance (Figure 6). Practical reasons lead us to double-parameterize our $\text{PS}_{\_\text{string}}$ with static and dynamic variants of the same monoid. In principle, the dynamic monoid can be built automatically from the static variant:

```ocaml
module Delay\_string(M:MONOID):MONOID with type t = M.t code = struct type t = M.t code
let l = .< M.l >.
let (\@) x y = .< M.(.\@ X .\@ y) >. end
```

but in practice it is better to avoid cross-stage persistence that captures locally-bound values ($M.l$ and $M.(\@)$), and so we instead define each dynamic monoid separately.

Instantiating $\text{PS}_{\_\text{string}}$ with the string monoid and its dynamic variant as parameters recovers the $\text{PS}_{\_\text{string}}$ of §3.

Figure 7 shows the type of a second module in our library, $\text{PS}_{\_\text{string}}$, which defines partially-static commutative rings, and is parameterized by concrete static and dynamic implementations of the $\text{CRING}$ interface. The $\text{PS}_{\_\text{string}}$ and $\text{PS}_{\_\text{string}}$ interfaces are identical except for the algebraic signatures $\text{MONOID}$ and $\text{CRING}$. Unfortunately, we do not know how to define both interfaces as instances of a more general signature, since OCamll’s abstraction over module types supports only fully-known or entirely abstract module types. The generalization here seems to need an intermediate form of abstraction that specifies some components of a signature (such as the type $t$ used in the definition of $\text{Eva}$) while leaving others (such as the particular operations in the algebraic signature) unspecified.

**var and code.** The standard type of dynamic values in Meta-OCaml is code: a value of type $\text{code}$ represents a dynamic expression of type $t$. Instead, our library uses a type $\text{var}$, which represents only a subset of code values, namely those dynamic expressions representing variables.

Using $\text{var}$ in place of $\text{code}$ serves two purposes. First, it allows us to freely duplicate or discard dynamic values, which is unsafe for general quoted expressions, since they may perform effects. Second, it allows us to add ordering information, making $\text{var}$ values suitable for use as keys in associative data structures. A $\text{var}$ value is a pair of a code value and an unique integer identifier that is used to support ordering:

**type $\alpha$ var = $\alpha$ code * int**
Conversion from var to code simply projects the first element of the pair. Conversion from a general dynamic expression of type code to var inserts a let binding for the expression using the freshly-bound variable as the first var component and a freshly-generated integer as the second.

This use of let-insertion is a standard technique in multi-stage programming and partial evaluation, particularly when specializing in direct style [Bondorf 1992]. Typical implementations of let-insertion involve delimited control [Kiselyov 2014] or algebraic effects [Yallop 2017]; the most recent release of BER MetaOCaml supports let-insertion natively.

6 Conclusion and further work

We have used free extensions of algebras as a functional specification of partially-static data, and described a high-level library for using them to produce efficient staged code.

In the future, we would like to explore this approach to Kaloper-Meršinjak and Yallop’s [2016] generic treatment of partially-static algebraic datatypes [Jones et al. 1993; Sheard and Diatchki 2002] (cf. §2). We would also like to use free extensions of free theories to partially evaluate code using effect handlers [Bauer and Pretnar 2015].

Acknowledgements. Supported by the European Research Council grant ‘events causality and symmetry — the next-generation semantics’, the Engineering and Physical Sciences Research Council grant EP/N007387/1 ‘Quantum computation as a programming language’, and a Balliol College Oxford Career Development Fellowship. We would like to thank Jacques Carette, Chung-chieh Shan, Sam Staton, and Gordon Plotkin for fruitful discussions and suggestions.

References


module Ps(A:MONOID)(B:MONOID with type t = A.t code): sig
  include MONOID
  type sta = A.t
  val sta : sta → t
  val dyn : sta var → t
module Eva(C: MONOID) : sig
  val eva : (sta → C.t) → (sta var → C.t) → t → C.t
end

Figure 6. Interface to partially-static monoids

module Ps(A:CRING)(B:CRING with type t = A.t code): sig
  include CRING
  type sta = A.t
  val sta : sta → t
  val dyn : sta var → t
module Eva(C: CRING) : sig
  val eva : (sta → C.t) → (sta var → C.t) → t → C.t
end

Figure 7. Interface to partially-static commutative rings
A Partially-static monoids

The \texttt{var} type

\begin{verbatim}
type \alpha \texttt{var} = \alpha \text{ code} \times \text{int}
let \texttt{var_counter} = \textbf{ref} 0

let \texttt{var} : \alpha \texttt{var} = \textbf{fun} \texttt{x} -> \texttt{incr \ var_counter}; (\texttt{x}, !\texttt{var_counter})

let \texttt{cd_of_var} : \alpha \texttt{var} -> \alpha \text{ code} = \texttt{fst}
\end{verbatim}

\subsection*{Partially-static monoids}

\textbf{The \texttt{var} type}

\begin{verbatim}
type \alpha \texttt{var} = \alpha \text{ code} \times \text{int}

let \texttt{var_counter} = \textbf{ref} 0

let \texttt{var} : \alpha \texttt{var} = \textbf{fun} \texttt{x} -> \texttt{incr \ var_counter}; (\texttt{x}, !\texttt{var_counter})

let \texttt{cd_of_var} : \alpha \texttt{var} -> \alpha \text{ code} = \texttt{fst}
\end{verbatim}

\textbf{B Efficient residualization for partially-static strings}

A residualizing function that builds a single n-ary catenation from a partially-static string monoid value may be defined as follows:

\begin{verbatim}
module PsΩ_string = PsΩ(String_monoid)(String_code_monoid)
\end{verbatim}
let residualize_string_list : string code list → string list code = fun l → List.fold_right (fun h t → .<h :: .~t>.) l .<[]>.

let nary_cd : Psₘ.string.t → string code = fun s →
let module E = Psₘ.string.Eva(String_code_list) in
  .< String.concat ""
  .~(residualize_string_list
    (E.eva
      (fun x → [.<x>.])
      (fun x → [x])
      s)) >.

Here is an example of nary_cd in action:

nary_cd (sta "a" ⊗ dyn (var .< x >.) ⊗ sta "c")
  ~> .<String.concat "" ["a"; x; "c"]>.