Partially static data as free extension of algebras

Short paper

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 expresssions 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:

$$((\mathtt{x}^{sta} + \mathtt{y}^{dyn})^{dyn} + \mathtt{z}^{sta})^{dyn}$$

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`:

$$((\mathtt{x}^{sta} + \mathtt{z}^{sta})^{sta} + \mathtt{y}^{dyn})^{dyn}$$

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.

Conference'17, July 2017, Washington, DC, USA
2017 ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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:

```
sprintf ((int ++ lit "a") ++ (lit "b" ++ int))
```

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

```
.<fun x y →
 ((""^ string_of_int x) ^"a") ^"b") ^string_of_int y>.
```

But strings form a monoid under catenation, and so this code is equivalent to the following more efficient code, which is generated by our library:

```
.<fun x y → string_of_int x ^ ("ab"^ string_of_int y)>.
```

(Our library can also generate the more efficient code that makes a single call to an *n*-ary catenation function.)

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`:

```
power 5 .<x>.  ⇝ .< 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:

```
power 5 .<x>. ⇝
  .< 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; 0; 2]` and a dynamic vector `d = [x; y; z]`, a naively-staged inner product function might generate the following code:

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

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

```
.< 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 `t` 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[1]:

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

```
is_digit (Dyn c) ⇝ Dyn .< is_digit .~c >.
is_digit (Sta '3') ⇝ Sta true
```

[1] The full code involves injections from static and dynamic values into a common type (§3).

```
module type MONOID = sig
  type t
  val 𝟙 : t
  val (⊗) : t → t → t
end
```

$$\mathbb{1} \otimes x \equiv x \equiv x \otimes \mathbb{1}$$
$$x \otimes (y \otimes z) \equiv (x \otimes y) \otimes z$$

Figure 1. Monoids and their laws

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 `lst = 0 :: 1 :: .<t>.`, we have:

$\texttt{map}_{\texttt{ps}}$ `(succ, .<succ>.) lst` $\rightsquigarrow$ `.<1 :: 2 :: map succ t>.`

3 Monoids

At the heart of each example in §2 is a partially-static algebraic structure. This section introduces a concrete structure for the partially-static monoids of strings, beginning from design considerations and concluding with a concrete implementation. The implementation generalizes straightforwardly to arbitrary monoids (§5).

Partially-static monoid: interface. What do we need from a partially-static monoid $\texttt{PS}_\otimes$?

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

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

Third, it should be possible to residualize computations in $\texttt{PS}_\otimes$ – 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 `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, $\texttt{PS}_\otimes$ should never unnecessarily convert static values to dynamic values.

These requirements suggest the following interface, which includes the `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):

```
1  module type PS⊗ = sig
2   include MONOID
3   type sta
4   val sta : sta → t
5   val dyn : sta var → t
6   module Eva(C: MONOID) : sig
7    val eva : (sta → C.t) →
8              (sta var → C.t) → t → C.t
9   end
10 end
```

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

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

Starting from the four $\texttt{PS}_\otimes$ operations: $\mathbb{1}$; $\otimes$; `sta`; and `dyn`, we naively define the following tree type, with one constructor for each:

```
type t = Unit | Mul of t * t
       | Sta of string | Dyn of string var
```

However, this implementation ignores the monoid laws, allowing many different representations for values (such as `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:

```
type t = Nil | Cons of atom * t
 and atom = Sta of string | Dyn of string var
```

Now $\otimes$ 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 `sta` to be a homomorphism with respect to $\otimes$, i.e.

`sta x` $\otimes$ `sta y` $\equiv$ `sta (x` $\otimes$ `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:

```
type _ alt = Empty : _ alt
           | ConsS : string * d alt → s alt
           | ConsD : string var * _ alt → d alt
type alt_ex = T : _ alt → alt_ex
```

The existential type `alt_ex` hides the index to build a parameterless type that can be used to implement `t` in $\texttt{PS}_\otimes$.

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 `^` 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 ⊗ (d ⊗ (s ⊗ ... ⊗ 𝟙)))
```

becomes

```
f s ⊗C (g d ⊗C (f s ⊗C ... ⊗C 𝟙C))
```

As an optimization, the unit may be omitted where the value is non-empty, so that `eva f g (s ⊗ 𝟙)` becomes `f s` rather than `f s ⊗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 ⊗ y` to `.<.~x ^ .~y>.` and `𝟙` to `.<"">.`, 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 (s1 ⊗(d ⊗ (s2 ⊗𝟙))) ⇝
.< .~(lift_string s1) ^ .~d ^ .~(lift_string s2) >.
```

Here is the implementation of $\texttt{Ps}_{\otimes}\texttt{\_string}$. (App. A gives implementations for `append_alt` and `Eva_alt`.)

```
module type Ps⊗_string = struct
 type t = alt_ex 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:

```
type (_,_) t
val lit : string → (α, α) t
val (++) : (β, α) t → (γ, β) t → (γ, α) t
val int : (α, int → α) t
val sprintf : (string, α) t → α
```

Figure 2. `printf` signature

```
type (_,_) t
val lit : string → (α, α) t
val (++) : (β, α) t → (γ, β) t → (γ, α) t
val int : (α, int code → α) t
val sprintf : (string code, α) t → α
```

Figure 3. Staged `printf` signature

```
type (α,ρ) t = (string → α) → string → ρ
let lit x k s = k (s ^ x)
let (++) f g k = f (g k)
let 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 `lit` 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:

```
type (α,ρ) t = (string code → α) → string code → ρ
let lit x k s = k .<.~s ^ x>.
let int k s x = k .<.~s ^ string_of_int .~x >.
let sprintf p = p id .<"">.
```

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 `string` with $\texttt{Ps}_{\otimes}\texttt{\_string.t}$, replace `^` and `""` with `⊗` and `𝟙`, insert `sta` and `dyn` to inject static and dynamic expressions, and replace the top-level continuation with the residualization function described above:

```
type (α,ρ) t = (Ps⊗_string.t →α) → Ps⊗_string.t →ρ
let lit x k s = k (s ⊗ sta x)
let int k s x = k (s ⊗ dyn .<string_of_int .~x>.)
let sprintf p = p cd 𝟙
```

Figure 4. Partially-static monoid: dropping $\mathbb{1}$

Figure 5. Coalescing adjacent static values

This implementation statically constructs a canonical representation before residualizing, eliminating nesting and redundant catenations with $\mathbb{1}$.

App. B gives a second residualization function for partially-static string monoids that generates a single call to n-ary concat rather than a sequence of binary catenations.

4 Universality: free extension of algebras

To describe the universal property for partially static data, we first recall some basic universal algebra, which allows us to discuss classes of algebraic structures uniformly.

4.1 Rudimentary universal algebra

Like datatypes, descriptions of algebraic structures consist of an interface and a functional specification for this interface. The interface is given by an algebraic *signature* Σ: a pair $(O_\Sigma, \mathrm{arity}_\Sigma)$ consisting of a set O_Σ whose elements we call *operation symbols*, and a function $\mathrm{arity}_\Sigma : O_\Sigma \to \mathbb{N}$ assigning to each operation symbol a natural number called its *arity*. For example, monoids use the signature given by

$$O_{\mathbf{mon}} \coloneqq \{\mathbb{1}, \otimes\}, \quad \mathrm{arity}_{\mathbf{mon}}(\mathbb{1}) \coloneqq 0, \quad \mathrm{arity}_{\mathbf{mon}}(\otimes) \coloneqq 2$$

We later use the more compact set-like notation $\{\mathbb{1} : 0, \otimes : 2\}$. Given a signature Σ, the functional specification is given by a set of equations between terms built from the operation symbols in Σ and according to their corresponding arities. These equations are called *axioms* (over the signature Σ). For example, the three monoid axioms $\mathrm{Ax}_{\mathbf{mon}}$ are:

$$1 \otimes x \equiv x \qquad x \otimes 1 \equiv x \qquad (x \otimes y) \otimes z \equiv x \otimes (y \otimes z)$$

Put together, the description of an algebraic structure is called a *presentation* $\mathcal{P}$, given by a pair $(\Sigma_\mathcal{P}, \mathrm{Ax}_\mathcal{P})$ consisting of a signature $\Sigma_\mathcal{P}$ and a set $\mathrm{Ax}_\mathcal{P}$ 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 $(|A|, -_A)$ consisting of a set $|A|$, called the *carrier* of the algebra, and, for each operation symbol $f : n$ in $\Sigma_\mathcal{P}$, an n-ary function $f_A : |A|^n \to |A|$, such that all the axioms in $\mathrm{Ax}_\mathcal{P}$ hold. For example, noting that a nullary function is a constant, a **mon**-algebra is a monoid.

Finally, given two $\mathcal{P}$-algebras A, B, a $\mathcal{P}$-homomorphism $h : A \to B$ is a function between the carriers $h : |A| \to |B|$ that respects the operations: for each operation symbol $f : n$ in $\Sigma_\mathcal{P}$, and for every n-tuple $\vec{a} = (a_1, \ldots, a_n)$ 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 **mon**-homomorphism $h : A \to B$ is a function that satisfies $h(1_A) = 1_B$ and $h(x \otimes_A y) = h(x) \otimes_B h(y)$, i.e. the familiar notion of a monoid homomorphism.

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

The forgetful functor $|-|$ always has a left adjoint $F_\mathcal{P} : \mathbf{Set} \to \mathcal{P}$-Alg. Concretely, its object map on a set X yields the *term algebra over* X: the set of $\Sigma_\mathcal{P}$-terms with variables in X, quotiented by the deductive closure of Ax_Σ 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 $\Sigma_{\mathbf{mon}}$-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, $\eta^\mathcal{P} : X \to |F_\mathcal{P} X|$ maps an element $x \in X$ to its equivalence class as a term. For **mon**, $\eta^{\mathbf{mon}}(x)$ is the one-element sequence $[x]$. The adjunction itself assigns to every function $f : X \to |A|$ its homomorphic extension $\gg\!=^\mathcal{P} f : F_\mathcal{P} X \to A$, 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:

$$[x; y; z] \gg\!=^{\mathbf{mon}} \{x \mapsto 2, y \mapsto 3, z \mapsto 4\} = 2 \cdot 3 \cdot 4 = 24$$

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

$$f(\iota_1 a_1, \ldots, \iota_n a_n) \equiv \iota_1 f_A(a_1, \ldots, a_n)$$

for every $f : n$ in $\Sigma_\mathcal{P}$, $a_1, \ldots, a_n$ in $|A|$, and analogous equations for B. The coproduct injection $\iota_1^\oplus : A \to A \oplus B$ maps a to the equivalence class of $\iota_1 a$, and similarly for B. For every pair of homomorphisms $h_1 : A \to C$, $h_2 : B \to C$, the unique cotupling homomorphism $[h_1, h_2] : A \oplus B \to C$ interprets a term over $|A| + |B|$ as the corresponding $|C|$-element, once each variable $\iota_i x$ is substituted by $h_i(x)$. 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 $\mathrm{ps}(A, X) \coloneqq A \oplus F_\mathcal{P} X$. Combining the universal properties of coproducts and adjunctions, it is characterised by an algebra $\mathrm{ps}(A, X)$ together with a homormophism $\iota_A : A \to \mathrm{ps}(A, X)$, and a function $\iota_X : X \to \mathrm{ps}(A, X)$, such that for every other pair of a homomorphism $h : A \to C$ and a function $e : X \to |C|$, there exists a unique homomorphism $\mathrm{eva}(h, e) : \mathrm{ps}(A, X) \to C$ satisfying $\mathrm{eva}(h, e) \circ \iota_A = h$ and $\big|\mathrm{eva}(h, e)\big| \circ \iota_X = e$.

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 $\mathrm{ps}(A, X)$ then supports the first two operations for partially-static data:

$$\mathrm{sta} \coloneqq \iota_A : A \to \mathrm{ps}(A, X) \qquad \mathrm{dyn} \coloneqq \iota_X : X \to \mathrm{ps}(A, X)$$

In the first argument, the universal property requires a conceptual leap: we have no direct justification to the existence of the map $\mathrm{eva}(h, e)$ 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 $\mathrm{eva}(h, e)$ 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 $\mathcal{P}$ be a presentation, and X a set. Assume a choice of a set $\mathrm{ps}(A, X)$ for every algebra A, a homomorphism $\iota_A : A \to \mathrm{ps}(A, X)$, and a function $\iota_X : X \to \mathrm{ps}(A, X)$ such that:*

- *For every function $e : X \to |A|$ there is a unique homomorphism $\mathrm{eva}(\mathrm{id}, e) : \mathrm{ps}(A, X) \to A$ satisfying:*

$$\mathrm{eva}(\mathrm{id}, e) \circ \iota_A = \mathrm{id} \qquad \left|\mathrm{eva}(h, e)\right| \circ \iota_X = e$$

- *For every homomorphism $h : A \to B$, there is a unique homomorphism $\mathrm{ps}(h, X) : \mathrm{ps}(A, X) \to \mathrm{ps}(B, X)$ satisfying:*

$$\mathrm{ps}(h, X) \circ \iota_A = \iota_B \circ h \qquad \mathrm{ps}(h, X) \circ \iota_X = \iota_X$$

Then $\mathrm{ps}(A, X)$, together with ι_A, ι_X and $\mathrm{eva}(h, e) \coloneqq \mathrm{eva}(\mathrm{id}, e) \circ \mathrm{ps}(h, X)$ 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 $\mathrm{ps}(-, X)$ 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:

$$\left|\mathrm{ps}(A, X)\right| \coloneqq A \times \sum_{n \in \mathbb{N}} (X \times A)^n$$

As a more complicated example, recall that a commutative ring $(A, \mathbb{0}, \oplus, \ominus, \mathbb{1}, \otimes)$ where $(A, \mathbb{0}, \oplus, \ominus)$ forms an abelian group, $(A, \mathbb{1}, \otimes)$ 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 `PS⊗_string` module of §3 generalizes to support arbitrary monoids by parameterizing by the `MONOID` instance (Figure 6). Practical reasons lead us to doubly-parameterize our $\mathrm{PS}_\otimes$ with static and dynamic variants of the same monoid. In principle, the dynamic monoid can be built automatically from the static variant:

```
module Delay⊗(M:MONOID):MONOID with type t = M.t code
= struct type t = M.t code
         let 𝟙 = .< M.𝟙 >.
         let (⊗) x y = .< M.(.~x ⊗ .~y) >. end
```

but in practice it is better to avoid cross-stage persistence that captures locally-bound values (`M.𝟙` and `M.(⊗)`), and so we instead define each dynamic monoid separately.

Instantiating $\mathrm{PS}_\otimes$ with the string monoid and its dynamic variant as parameters recovers the `PS⊗_string` of §3.

Figure 7 shows the type of a second module in our library, $\mathrm{Ps}_{\otimes\oplus}$, which defines partially-static commutative rings, and is parameterized by concrete static and dynamic implementations of the `CRING` interface. The $\mathrm{Ps}_\otimes$ and $\mathrm{Ps}_{\otimes\oplus}$ interfaces are identical except for the algebraic signatures `MONOID` and `CRING`. Unfortunately, we do not know how to define both interfaces as instances of a more general signature, since OCaml'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 `Eva`) while leaving others (such as the particular operations in the algebraic signature) unspecified.

`var` *and* `code`. The standard type of dynamic values in MetaOCaml is `code`: a value of type `t code` represents a dynamic expression of type `t`. Instead, our library uses a type `var`, which represents only a subset of `code` values, namely those dynamic expressions representing variables.

Using `var` in place of `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 `var` values suitable for use as keys in associative data structures. A `var` value is a pair of a `code` value and an unique integer identifier that is used to support ordering:

```
type α var = α code * int
```

```
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
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
end
```

Figure 7. Interface to partially-static commutative rings

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

Baris Aktemur, Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2013. Shonan Challenge for Generative Programming: Short Position Paper. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation (PEPM '13)*. ACM, New York, NY, USA, 147–154. https://doi.org/10.1145/2426890.2426917

Kenichi Asai. 2009. On typing delimited continuations: three new solutions to the printf problem. *Higher-Order and Symbolic Computation* 22, 3 (01 Sep 2009), 275–291. https://doi.org/10.1007/s10990-009-9049-5

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.* 84, 1 (2015), 108–123.

Anders Bondorf. 1992. Improving Binding Times Without Explicit CPS-conversion. *SIGPLAN Lisp Pointers* V, 1 (Jan. 1992), 1–10. https://doi.org/10.1145/141478.141483

Jacques Carette and Oleg Kiselyov. 2011. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Sci. Comput. Program.* 76, 5 (2011), 349–375. https://doi.org/10.1016/j.scico.2008.09.008

Olivier Danvy. 1998. Functional Unparsing. *J. Funct. Program.* 8, 6 (Nov. 1998), 621–625. https://doi.org/10.1017/S0956796898003104

Jacques Garrigue and Jacques Le Normand. 2011. Adding GADTs to OCaml: the direct approach. (September 2011). OCaml Users and Developers Workshop 2011.

Jun Inoue. 2014. Supercompilation via staging. In *Fourth International Valentin Turchin Workshop on Metacomputation.*

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

David Kaloper-Meršinjak and Jeremy Yallop. 2016. Generic Partially-static Data (Extended Abstract). In *Proceedings of the 1st International Workshop on Type-Driven Development (TyDe 2016).* ACM, New York, NY, USA, 39–40. https://doi.org/10.1145/2976022.2976028

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Lecture Notes in Computer Science, Vol. 8475. Springer International Publishing, 86–102.

Oleg Kiselyov and Chung-chieh Shan. 2007. Lightweight Static Capabilities. *Electron. Notes Theor. Comput. Sci.* 174, 7 (June 2007), 79–104.

Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. 2004. A Methodology for Generating Verified Combinatorial Circuits. In *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT '04).* ACM, New York, NY, USA, 249–258. https://doi.org/10.1145/1017753.1017794

Torben Ægidius Mogensen. 1988. Partially Static Structures in a Self-Applicable Partial Evaluator. In *Partial Evaluation and Mixed Computation*, D. Bjørner, A.P. Ershov, and N.D. Jones (Eds.).

Tim Sheard and Iavor S. Diatchki. 2002. Staging Algebraic Datatypes. Unpublished manuscript. (2002). http://web.cecs.pdx.edu/~sheard/papers/stagedData.ps.

Walid Taha. 2003. A Gentle Introduction to Multi-stage Programming.. In *Domain-Specific Program Generation (Lecture Notes in Computer Science)*, Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky (Eds.), Vol. 3016. Springer, 30–50. https://doi.org/10.1007/978-3-540-25935-0_3

Peter Thiemann. 2013. Partially Static Operations *(PEPM '13).* ACM, New York, NY, USA, 75–76. https://doi.org/10.1145/2426890.2426906

Jeremy Yallop. 2017. Staged Generic Programming. *Proc. ACM Program. Lang.* 1, ICFP, Article 29 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110273

Jeremy Yallop and Leo White. 2015. Modular Macros. (September 2015). OCaml Users and Developers Workshop 2015.

A Partially-static monoids

The var type

```
type α var = α code * int

let var_counter = ref 0

let var : α. α code → α var =
  fun x → incr var_counter; (x, !var_counter)
let cd_of_var : α. α var → α code = fst

type d = D and s = S

type (α, 'start) alt =
    Empty : (α, _) alt
  | ConsS : α * (α, d) alt → (α, s) alt
  | ConsD : α code * (α, _) alt → (α, d) alt

type α alt_ex = T : (α, _) alt → α alt_ex

module Eva_alt (C : MONOID) =
struct
 let eva f g (T c) =
  let rec eva' : type start.
    (A.t, start) alt → C.t = function
    | Empty → C.𝟙
    | ConsS (a, Empty) → f a
    | ConsD (b, Empty) → g b
    | ConsS (a, m) → C.(f a ⊗ eva' m)
    | ConsD (b, m) → C.(g b ⊗ eva' m)
  in eva' c
end
```

Partially-static monoids

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

 let consS : type start.
    A.t → (A.t, start) alt → (A.t, s) alt =
   fun a → function
     | Empty → ConsS (a, Empty)
     | ConsS (a', m) → ConsS (A.(a ⊗ a'), m)
     | ConsD _ as r → ConsS (a, r)

 let consD : type start.
   B.t → (A.t, start) alt → (A.t, d) alt =
   fun b → function
     | Empty → ConsD (b, Empty)
     | ConsS _ as r → ConsD (b, r)
     | ConsD (b', m) → ConsD (B.(b ⊗ b'), m)

 let rec append_alt : type start start'.
  (A.t, start) alt → (A.t, start') alt → t =
  fun l r → match l, r with
   | l, Empty → T l
   | Empty, r → T r
   | ConsS (a, m), r →
     let T m' = append_alt m r in T (consS a m')
   | ConsD (b, m), r →
     let T m' = append_alt m r in T (consD b m')
 let (⊗) (T l) (T r) = append_alt l r
 let 𝟙 = T Empty
 let sta a = T (ConsS (a, Empty))
 let dyn b = T (ConsD (cd_of_var b, Empty))
 module Eva = Eva_alt
end

module String_monoid =
struct
  type t = string
  let (⊗) = (^)
  let 𝟙 = ""
end

module String_code_monoid =
struct
  type t = string code
  let (⊗) x y = .< .~x ^ .~y >.
  let 𝟙 = .< "" >.
end

module Ps⊕_string =
  Ps⊕(String_monoid)(String_code_monoid)
```

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:

```
module String_code_list = struct
  type t = string code list
  let 𝟙 = []
  let (⊗) = (@)
```

```
end

let residualize_string_list
   : string code list → string list code =
  fun l →
    List.fold_right (fun h t → .<.~h :: .~t>.) l .<[]>.

let nary_cd :  Ps⊕_string.t →string code =
 fun s →
  let module E = Ps⊕_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"]>.
```