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 would ensure 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

Program optimizers, such as partial evaluators, optimizing compilers, multi-stage programming languages and super-compilers, improve the performance of programs by distinguishing static inputs (available now) from dynamic inputs (available later). The classification extends from inputs to expressions: static expressions, depending only on static inputs, can be reduced in advance, leaving specialized programs built from dynamic expressions.

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

For example, suppose that in the expression

\[(x + y) + z\]

\(x\) and \(z\) are static, and \(y\) is dynamic. Then \(x + y\) is also dynamic, because it has \(y\) as a sub-expression; in turn, the entire expression is dynamic, with only trivial static sub-expressions:

\[((x + y)^D) + z^D\]

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

\[((x + y)^S)^D + y^D\]

However, it is impractical to manually rewrite all programs in this way. Moreover, 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 [16] 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 [24] 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. For concreteness we present our approach as a Haskell library, frex (with a second implementation in BER MetaOCaml [14]); however, the formulation transfers straightforwardly to other settings.

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 algebras for the associated equational theories.
We precalculate the free extensions of common algebraic structures: monoids, commutative rings, distributive lattices, and abelian groups, and show that the formulation also extends to algebraic datatypes.

- §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 commutative ring for complex numbers, and so on — to provide drop-in modules that perform optimizations using the associated laws.

- Appendix B shows that the algebraic simplifications performed by partially-static structures lead to improved performance on representative benchmarks.

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 functions without altering the non-local structure of the generating program. After briefly recapping Haskell’s compile-time metaprogramming facilities we present several illustrative examples as Haskell programs.

2.1 Template Haskell

Template Haskell [21] extends Haskell with constructs for compile-time metaprogramming that support running user-defined computations during compilation to generate code. The frex library, and the examples in this paper, use Typed Template Haskell [17], a recent enhancement in which the types of the generated code are reflected in the generating program.

The type constructor Code (an alias for Template Haskell’s Q (Term a)) represents code values: a value of type Code e is an unevaluated piece of code that will produce a value of type e. Type judgements $\Gamma \vdash e : \tau$ assert that the expression e has type $\tau$ in context $\Gamma$ and deferred to stage n, where n is a natural number. The following two constructs transport expressions between stages:

- **BRACKET**
  \[
  \Gamma \vdash \tau^{n+1} e : \tau \quad \text{ESCape} \quad \Gamma \vdash \tau^n e : \text{Code } \tau \quad \Gamma \vdash \tau^{n+1} \$e : \tau
  \]

(The full typing rules additionally distinguish between escapes that occur within brackets and top-level escapes [21].)

Finally, the function lift in the type class Lift provides an interface to cross-stage persistence. When a variable bound at stage n appears in an expression at stage $n + 1$, Template Haskell automatically inserts a call to lift to convert the value to a piece of code, which is inserted via an escape:

\[
\text{let } x = 3 \text{ in } [ x ] \leadsto \text{let } x = 3 \text{ in } [ \$ (\text{lift } x) ]
\]

2.2 Examples

We now turn to examples of partially-static data in action:

**Printf.** Functions whose arguments arrive at different times lend themselves well to a multi-stage approach. The typed sprintf function studied by Danvy [9] and Asai [2] is one such example, since the format string is typically known in advance of the values passed as subsequent arguments. Yallop and White [26] 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:

\[
\text{sprintf } ((\text{int } ++ \text{ lit } "a") ++ (\text{lit } "b" ++ \text{int}))
\]

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

\[
[\lambda x \ y \rightarrow (("\text{show } x \) ++ "a") ++ "b") ++ \text{show } y ]
\]

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

\[
[\lambda x \ y \rightarrow \text{show } x \ ++ \ ("ab" ++ \text{show } y ) ]
\]

(Frex can also generate the more efficient code that makes a single call to an n-ary catenation function (§5.4).

**Power.** Consider the staged power function, $\lambda x \rightarrow x^n$, with n statically known. Once again, a naive approach generates sub-optimal code. The staged power function of Taha [22] builds a computation with too many multiplications, including an unnecessary multiplication by 1:

\[
\text{power } 5 [x] \leadsto [1 \times (x \times (x \times (x \times (x \times x))))]
\]

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

\[
\text{power } 5 [x] \leadsto [\text{let } y = x \times x \text{ in let } z = y \times y \text{ in } x \times z ]
\]

**Linear algebra.** Linear algebra offers many opportunities for optimization via multi-stage specialization and numerical simplification such as: the Fast Fourier Transform [15], Gaussian elimination [5], and matrix-vector multiplication [1]. 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:

\[
[ (1 \times x) + (0 \times y) + (2 \times z) ]
\]

Using the fact that integers form a commutative ring, frex generates the following simpler code:

\[
[ x + (2 \times z) ]
\]

**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 if every element of l satisfies p. Frex supports defining a variant of all that operates on partially-static
Partially static data as free extension of algebras

```haskell
class Monoid t where
  ℧ : t
  (⊛) : t → t → t
         ℧ ⊛ (y ⊛ z) ≡ (x ⊛ y) ⊛ z

Figure 1. Monoids and their laws
```

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 \( \text{all} \), the result may be static even where the input is partially unknown:

```haskell
all even (sta [2, 4] ++ var [x] ++ sta [3])
      ~ even 2 ⊛ even 4 ⊛ [ all even x ] ⊛ even 3
```

With `frex` the expression above is further reduced to the static value `false`, using the fact that booleans form a distributive lattice. The dual function `any` can be defined similarly.

In the examples above, partially static operations (+, ++, etc.) are explicit. `frex` also supports 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:

```haskell
isDigit (Dyn c) ~ Dyn [isDigit $c ]
isDigit (Sta '3') ~ Sta True
```

**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 [11, 13, 20].

For example, `frex` can be used to define a variant of `sum` that operates on partially-static lists and can traverse the initial portion of a list, leaving traversal of the dynamic tail for later:

```haskell
sumps (1 ∶ x 2 ∶ x 3 ∶ x dyn [t]) ~ [6 + sum 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. §5 generalizes the concrete example to arbitrary monoids and to other structures.

**Partially-static monoid: interface.** What operations should a partially-static monoid `PS_m` support?

First, if `PS_m` 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).

```haskell
instance Monoid m ⇒ Monoid (PS_m m) where ...
```

That is, if `m` satisfies the `Monoid` interface, then `PS_m m` ought to satisfy it, too.

Ideally, `PS_m` should be `canonical`: expressions that are statically equivalent under the monoid laws should have the same representation in `PS_m`.

Second, it should be possible to use partially-static values in place of fully-static or fully-dynamic values, and so `PS_m` should support injections from static and dynamic data:

```haskell
sta_m :: Monoid m ⇒ m → PS_m m
dyn_m :: Monoid m ⇒ Code m → PS_m m
```

Third, it should be possible to residualize computations in `PS_m` — i.e. to turn partially-static monoid values into code:

```haskell
cd_m :: (Lift m, Monoid (Code m)) ⇒ PS_m m → Code m
```

More generally, it should be possible to inspect partially-static data — or, at least, its static structure — in order to residualize and to perform transformations such as `all` (§2).

```haskell
eva_m :: (Monoid m, Monoid n) ⇒ (m → n) → ([Code m] → n) → PS_m m → n
```

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, `PS_m` should never unnecessarily convert static values to dynamic values.

**Partially-static monoid: implementation.** How do we implement `PS_m` while satisfying the monoid laws, pre-computing static values, and generating optimal code?

Starting from the four `PS_m` operations: \( \text{∅, ⊕, sta}_m; \) and `dyn_m`, we naively define the following tree type, with one constructor for each:

```haskell
type PS_m m = Unit | Mul (PS_m m) (PS_m m)
           | Sta m | Dyn (Code m)
```

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:

```haskell
type PS_m m = Nil | Cons (Atom m) (PS_m m)
```

Now `⊕` can be defined by the familiar append function which, of course, respects the monoid laws.

```haskell
"""
We might take things one step further. It is clearly desirable for \(\text{sta}_\Box\) to be a homomorphism with respect to \(\circ\), i.e.

\[
\text{sta}_\Box\ x \otimes \text{sta}_\Box\ y \equiv \text{sta}_\Box\ (x \otimes y)
\]

This suggests that adjacent static values in the list should be coalesced (Figure 4). With a little care it is possible to enforce the constraint in the type, using a GADT index \([19, 27]\) instantiated by either \(S\) or \(D\) to track whether a list starts with a static or dynamic element:

\[
\begin{align*}
\text{data} & \quad \text{Phase} = S \mid D \\
\text{data} & \quad \text{Alt} :: \text{Phase} \rightarrow * \rightarrow * \rightarrow * \mid \text{where} \\
& \quad \phantom{\text{data Alt :: Phase \rightarrow * \rightarrow * \rightarrow * \mid \text{where}}}
\operatorname{Empty} :: \text{Alt} \, \text{any} \, s \, d \\
& \quad \phantom{\text{data Alt :: Phase \rightarrow * \rightarrow * \rightarrow * \mid \text{where}}}
\operatorname{ConsS} :: s \rightarrow \text{Alt} \, D \, s \, d \rightarrow \text{Alt} \, S \, s \, d \\
& \quad \phantom{\text{data Alt :: Phase \rightarrow * \rightarrow * \rightarrow * \mid \text{where}}}
\operatorname{ConsD} :: d \rightarrow \text{Alt} \, S \, s \, d \rightarrow \text{Alt} \, D \, s \, d
\end{align*}
\]

An existential type can hide the phase index to build a type that can be used to implement \(\text{PS}_\Box\):

\[
\begin{align*}
\text{data} & \quad \text{PS}_\Box \ m = P :: \text{Alt} \, _\Box \ m \ (\text{Code} \ m) \rightarrow \text{PS}_\Box \ m \\
\text{data} & \quad \text{PS}_\Box \ m = P :: \text{Alt} \, _\Box \ m \ (\text{Code} \ m) \rightarrow \text{PS}_\Box \ m
\end{align*}
\]

This representation is still not quite canonical, since \(\text{ConsS}\) can store empty monoid elements; this is unavoidable in the general case, since it is not always possible to determine whether a monoid element should be considered empty. (For example, the monoid of endofunctions does not support equality.)

It is straightforward to define a function that catenates two \(\text{Alt}\) values, combining adjacent static strings using the standard \(\oplus\) operator. §5.1 gives a slightly generalized version of the cationation function for \(\text{Alt}\).

Finally, \(\operatorname{eva}_\Box\) interprets a value of type \(\text{PS}_\Box\) in some other monoid \(\mathcal{M}\), mapping constituent static and dynamic values individually, and mapping monoid operations to the operations of \(\mathcal{M}\). In other words, the following expression

\[
\operatorname{eva}_\Box(f \circ g) (s \circ (d \circ (s \circ \ldots \circ \ldots \oplus \ldots)))
\]

becomes

\[
f \circ s \circ (d \circ (f \circ s \circ \ldots \circ \ldots \circ \circ \ldots))
\]

As an optimization, the unit may be omitted where the value is non-empty, so that \(\operatorname{eva} f g (s \circ \ldots)\) becomes \(f s\) rather than \(f s \circ \ldots \circ \circ \ldots\).

A common use of \(\operatorname{eva}_\Box\) is residualization, which turns a partially-static value into a fully-dynamic value. Residualization is implemented by instantiating \(n\) to the monoid that maps \(x \circ y\) to \([s x \oplus s y]\) and \(\bot\) to \(["""]\), and supplying the function \(\text{tlift}\) that residualizes a single value and the identity function as the two arguments of \(\operatorname{eva}\). Then

\[
\begin{align*}
\operatorname{eva}_\Box\text{ tlift}\, \text{id} (s_1 \circ (d \circ (s_2 \oplus \ldots))) \rightarrow \\
& [ s_1 \leftrightarrow s d \leftrightarrow s_2 ]
\end{align*}
\]

Here is the implementation of \(\text{PS}_\Box\_\text{string}:

```kotlin
class Format f where
  type Acc f a where
    lit :: String \rightarrow f a a
    cat :: f b a \rightarrow f c b \rightarrow f c a
    int :: f a (Acc f Int \rightarrow a)
    sprintf :: f (Acc f String) a \rightarrow a

Figure 2. Format signature
```

Improving \(\text{printf}\). §2 showed the effects of the partially-static monoid on the code generated by a staged \(\text{printf}\) function. We now show how to transform the implementation of \(\text{printf}\) to achieve those effects.

Figure 2 gives a minimal interface for formatted printing. The type constructor \(f\) represents format specifications; its two parameters respectively represent the result and the input type of a \(\text{printf}\) instantiation. The following three operations construct format strings: \(\text{lit}\) is a format string that accepts no arguments and prints \(s\); \(\text{cat}\) is a format string that accepts \(x\) and \(y\), and \(\text{int}\) is a format string that accepts and prints an integer argument. Finally, \(\text{sprintf}\) combines a format string with corresponding arguments to construct formatted output. Asai [2] gives further details.

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

```kotlin
newtype Fmt r a = 
  Fmt { fmt :: (String \rightarrow r) \rightarrow String \rightarrow a }

instance Format Fmt where
  type Acc Fmt a = a
  lit = Fmt $ \k \rightarrow k (s \rightarrow x)
  int = Fmt $ \k \rightarrow k (s \rightarrow show x)
  cat = Fmt f (fmt f . fmt g)
  str = Fmt $ \k \rightarrow k (s \rightarrow x)
  sprintf p = fmt p id ""

With this implementation, a format string \(\text{fmt}\) is a function accepting a continuation argument of type \(\text{String} \rightarrow r\) and an accumulator of type \(\text{String}\). Both \(\text{lit}\) and \(\text{int}\) call \(\text{fmt}\) directly, passing an extended string; \(\text{cat}\) is simply function composition. The function \(\text{sprintf}\) passes the identity function as a top-level continuation along with an empty accumulator.

Staging \(\text{sprintf}\) is straightforward. We treat format strings statically; arguments and, consequently, the accumulator, are dynamic. The cat function is left unchanged, and
Partially static data as free extension of algebras

\[ S_1 \oplus S_2 \oplus S_3 \oplus S_4 \sim S_1 \oplus S_2 \oplus S_4 \]

**Figure 3.** Partially-static monoid: dropping \( \bot \)

\[ S_1 \oplus S_2 \oplus S_3 \oplus S_4 \sim S_1 \oplus (S_2 \oplus S_4) \]

**Figure 4.** Coalescing adjacent static values

the remainder of the implementation acquires brackets and escapes to match the assignment of static and dynamic classifications:

\[
\text{newtype } \text{FmtS} \ r \ a = \\
\text{FmtS} \ (\text{FmtS} :: (\text{Code } \text{String} \rightarrow r) \rightarrow (\text{Code } \text{String} \rightarrow a))
\]

\[
\text{instance } \text{Format } \text{FmtS} \text{ where} \\
\text{type } \text{Acc } \text{FmtS} = \text{Code} \\
\text{lit } x = \text{FmtS } \backslash k \ s \rightarrow k \ [ \ s ++ \ x ] \\
f \ \text{cat} \ g = \text{FmtS} \ (\text{FmtS } f \ . \ \text{fmtS } g) \\
\text{int } = \text{FmtS } \backslash k \ s \ x \rightarrow k \ [ \ s ++ \ \text{show } s x ] \\
\text{str } = \text{FmtS } \backslash k \ s \ x \rightarrow k \ [ \ s ++ \ s x ] \\
\text{sprintf } p = \text{fmtS } p \ \text{id} \ [[ \ \text{"} ]
\]

The generated code (§2) is sub-optimal 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 \( P \oplus \) String, replace cat and "" with @ and \( \bot \), insert sta@ and dyn@ to inject static and dynamic expressions, and replace the top-level continuation with the residualization function described above:

\[
\text{newtype } \text{FmtPS} \ r \ a = \\
\text{FmtPS} \ (\text{FmtPS} :: (P \oplus \text{String} \rightarrow r) \rightarrow (P \oplus \text{String} \rightarrow a))
\]

\[
\text{instance } \text{Format } \text{FmtPS} \text{ where} \\
\text{type } \text{Acc } \text{FmtPS} = \text{Code} \\
\text{lit } x = \text{FmtPS } \backslash k \ s \rightarrow k \ (s \oplus \text{sta}@ \ x) \\
f \ \text{cat} \ g = \text{FmtPS} \ (\text{FmtPS } f \ . \ \text{fmtPS } g) \\
\text{int } = \text{FmtPS } \backslash k \ s \ x \rightarrow k \ (s \oplus \text{dyn}@ \ x) \\
\text{str } = \text{FmtPS } \backslash k \ s \ x \rightarrow k \ (s \oplus \text{dyn}@ \ x) \\
\text{sprintf } p = \text{FmtPS } p \ \text{id} \ [[ \ \text{"} ]
\]

This implementation statically constructs a canonical representation before residualizing, eliminating nesting and redundant catenations with \( \bot \).

§5.4 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 \( \Sigma \) a pair \( (O_\Sigma, \text{arity}_\Sigma) \) consisting of a set \( O_\Sigma \) whose elements we call operation symbols, and a function \( \text{arity}_\Sigma : O_\Sigma \rightarrow \mathbb{N} \) assigning to each operation symbol a natural number called its arity. For example, monoids use the signature given by

\[
O_{\text{mon}} := \{ \bot, \oplus \}, \quad \text{arity}_{\text{mon}}(\bot) := 0, \quad \text{arity}_{\text{mon}}(\oplus) := 2
\]

Given a signature \( \Sigma \), the functional specification is given by a set of equations between terms built from the operation symbols in \( \Sigma \) and according to their corresponding arities. These equations are called axioms (over the signature \( \Sigma \)). For example, the monoid axioms \( A \times_{\text{mon}} \) are given in Figure 1.

Put together, the description of an algebraic structure is called a presentation \( P \), given by a signature \( \Sigma_P \) and a set \( A \times_P \) of axioms over this signature. The example signature and axioms above form \( \text{mon} \) — the presentation of monoids.

An algebra for a presentation is a mathematical implementation of such specifications. Formally, given a presentation \( P \), a \( P \)-algebra \( A \) is a pair \( (A, \delta) \) consisting of a set \( |A| \), called the carrier of the algebra, and, for each operation symbol \( f : n \in \Sigma_P \), an \( n \)-ary function \( f_A : |A|_n \rightarrow |A| \), such that all the axioms in \( A \times_P \) hold. For example, noting that a nullary function is a constant, a \( \text{mon} \)-algebra is a monoid.

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

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

The forgetful functor \( |-| \) always has a left adjoint \( F_P : \text{Set} \rightarrow P \text{-Alg} \). Concretely, its object map on a set \( X \) yields the term algebra over \( X \): the set of \( \Sigma_P \)-terms with variables in \( X \), quotiented by the deductive closure of \( A_X \) 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 is equivalent to the sequence formed...
by their tree-fringe, with the unit elements omitted, repre-
represented by a spine. The unit of the adjunction, \( \eta^P : X \to |P| \times X \) maps an element \( x \in X \) to its equivalence class as a term. For \( \text{mon} \), \( \eta^{\text{mon}}(x) \) is the one-element sequence \([x]\). The
adjunction itself assigns to every function \( f : X \to |A| \) its homomorphic extension \( \Rightarrow f : P \times X \to A \), 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:

\[
[x; y; z] \Rightarrow_{\text{mon}} \{ x \mapsto 2, y \mapsto 3, z \mapsto 4 \} = 2 \cdot 3 \cdot 4 = 24
\]

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

\[
f\left( i_1 a_1, \ldots, i_n a_n \right) \equiv i_1 f_A(a_1, \ldots, a_n)
\]

for every \( f : n \in \Sigma_P, a_1, \ldots, a_n \in |A| \), and analogous equations for \( B \). The coproduct injection \( i_0^A : A \to A \oplus B \) maps \( a \) to the equivalence class of \( i_0 a \), and similarly for \( B \). For every pair of homomorphisms \( h_1 : A \to C, h_2 : B \to C \), the unique coproduct homomorphism \( [h_1, h_2] : A \oplus B \to C \) interprets a term over \( |A| + |B| \) as the corresponding \( |C| \)-element, once each variable \( i_0 x \) 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 \( ps(A, X) \equiv A \oplus F_P X \).

Combining the universal properties of coproducts and adjunctions, it is characterised by an algebra \( ps(A, X) \) together with a homomorphism \( \iota_A : A \to ps(A, X) \), and a function \( \iota_X : X \rightarrow ps(A, X) \), 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 \( eva(h, e) : ps(A, X) \to C \) satisfying \( eva(h, e) \circ \iota_A = h \) and \( eva(h, e) \circ \iota_X = e \).

4.2 Conceptual justification

We have two different arguments for using free extensions of algebras as the appropriate fundamental 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 \( ps(A, X) \) then supports the first two operations for partially-static data:

\[ \text{sta} := \iota_A : A \to ps(A, X) \quad \text{dyn} := \iota_X : X \to ps(A, X) \]

In the first argument, the universal property requires a conceptual leap: we have no direct justification to the existence of the map \( eva(h, e) \) 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 \( eva(h, e) \) 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 \( ps(A, X) \) for every algebra \( A \), a homomorphism \( \iota_A : A \to ps(A, X) \), and a function \( \iota_X : X \to ps(A, X) \) such that:

- For every function \( e : X \rightarrow |A| \) there is a unique homomorphism \( eva(id, e) : ps(A, X) \rightarrow A \) satisfying:

\[
eva(id, e) \circ \iota_A = id \quad eva(h, e) \circ \iota_X = e
\]

- For every homomorphism \( h : A \rightarrow B \), there is a unique homomorphism \( ps(h, X) : ps(A, X) \rightarrow ps(B, X) \) satisfying:

\[
ps(h, X) \circ \iota_A = \iota_B \circ h \quad ps(h, X) \circ \iota_X = \iota_X
\]

Then \( ps(A, X) \), together with \( \iota_A, \iota_X \) and \( eva(h, e) := eva(id, e) \circ ps(h, X) \) 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, the datatype stores representations of hybrid terms consisting only of \( A \) elements and \( X \) elements, the functor \( ps(-, X) \) 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:

\[
|ps(A, X)| := A \times \sum_{n \in \mathbb{N}} (X \times A)^n
\]

As a more complicated example, recall that a commutative ring is \( (A, 0, \ast, \odot, 1, \otimes) \) where \( (A, 0, \ast, \otimes) \) forms an abelian group, \( (A, 1, \otimes) \) forms a commutative monoid, together with a distributivity law \( x \otimes (y \otimes z) \equiv (x \otimes y) \otimes (x \otimes z) \). 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

The principles introduced for the partially-static monoid in §3 and formalised in §4 extend straightforwardly to other settings. This section presents a general interface to partially-static data via free extensions as a Haskell library, \texttt{frex}, along with additional examples for particular algebraic structures. Appendix A presents a second implementation in MetaOCaml.

The \texttt{frex} library is defined in three parts. The first part is an interface to coproducts indexed by an algebraic structure (§5.1), along with instances for particular structures (monoids, commutative rings, abelian groups, and so on). The second part is a monadic interface to free algebraic structures (§5.2) that, combined with the coproduct interface, gives a definition of free extensions (§5.3). Finally, it is often possible to improve
class (alg a, alg b, alg (Coprod alg a b)) =>
    Coprod alg a b where
    data family Coprod alg a b :: *
    inl :: a → Coprod alg a b
    inr :: b → Coprod alg a b
    eva :: alg c =>
        (a → c) → (b → c) → Coprod alg a b → c

Figure 5. The Coproduct interface

on the generic residualization function for particular structures; §5.4 gives some representative examples of more effective residualization.

5.1 Coproducts

Figure 5 defines the coproduct interface as a Haskell type class, Coprod, with three parameters and four components.

The three parameters alg, a, and b, respectively represent a type class for a particular algebraic structure and the two types that comprise the coproduct. For example, Coprod Monoid Int String represents the coproduct of Int and String in the category of monoids. The first two class constraints alg a and alg b constrain the instantiation of the parameters to types that have instances of alg. For example, the instantiation Coprod Monoid Int String is only allowed if there exist type class instances of Monoid for Int and String.

The first component, Coprod, is the type of values of the coproduct of a and b in the category alg. Coprod is an associated data type [8], whose definition varies with each instance of the Coproduct class. For example, as shown below, Coprod Monoid Int String, a coproduct in the category of monoids, is built on the alternating sequence of §3, whereas Coprod Set Int String, a coproduct in the category of sets, is the familiar binary sum type. The final class constraint alg (Coprod alg a b) ensures that each instantiation of Coprod is an instance of the algebraic structure alg. The second and third components, inl and inr, inject values of a and b into Coprod. The final component eva is a kind of fold that produces a value of type c from a value of type Coprod and functions a and b to c. The constraint alg c ensures that c is also an instance of the algebraic structure associated with the instance—the example, there must be a Monoid instance for the type Coprod Monoid a b.

Coproduct of monoids The partially-static structure for monoids in §3 is an instance of the more general coproduct of monoids.

Figure 6 gives the Coproduct instance for the Monoid class constraint, built from Monoid instances a and b. The Coprod type is defined as a sequence of alternating a and b elements using Alt (§3), with the first type parameter hidden by an existential, to allow either a-prefixed or b-prefixed sequences. (To reflect the more general setting, the s and d elements from §3 are renamed a and b, and the constructors are renamed similarly.) Both inl and inr construct singleton sequences from elements, while eva maps the alternating elements into the target monoid, whose ⊗ operation is used to combine the results. As a small optimization, eva does not map the Empty constructor to ⊗ except in the case where the input sequence has no elements.

The constraints in the Coproduct class specify that each data instance Coprod alg a b is an instance of alg. For example the type Coprod Monoid a b should be an instance of Monoid.

Figure 6 defines the Monoid instance for Coprod Monoid a b, where a and b also have instances of Monoid.

The ⊗ and ⊙ operations respectively construct an empty sequence and concatenate two sequences. Prepending an a element to an a-prefixed sequence combines the element
class Monoid m ⇒ CMonoid m  
⋯ ⊗ x ⋯ x

class CMonoid c ⇒ CGroup c  
⋯ (y ⊗ z) ⋯ (x ⊗ y) ⊗ z

where cinv :: c → c  
⋯ x ⊗ y ⋯ y ⊗ x
⋯ cinv x ⊗ x ⋯ 1

Figure 8. Commutative monoids and abelian groups

instance (CMonoid a, CMonoid b) ⇒  
Coproduct CMonoid a b where  
data Coprod CMonoid a b = Inl a | Inr b  
in1 a = C a ⊙ 1 ; inr b = C 1 b  
eva f g (C a b) = f a ⊗ g b

Figure 9. Commutative monoids coproduct

class Set a  
instance Set a

instance Coproduct Set a b where  
data Coprod Set a b = Inl a | Inr b  
in1 = Inl 1 ; inr = Inr  
eva f g (Inl x) = f x ; eva f g (Inr y) = g y

Figure 10. Coproduct of structures with no equations

with the head of the sequence using the ⊗ operation of the a monoid, and similarly for b, mutatis mutandis.

Coproducts of commutative monoids and abelian groups

Figure 8 shows the interface to commutative monoids (CMonoid) and abelian groups (CGroup). The CMonoid inherits the methods from the Monoid interface and adds the commutativity law. The only difference between Monoid and CMonoid is the set of laws tacitly associated with the class.

However, the addition of the commutativity law to the monoid interface leads to quite a different coproduct structure (Figure 9). Applying commutativity to the alternating sequence structure of the monoid coproduct allows the elements of each constituent monoid to be brought together and multiplied. The alternating sequence consequently collapses into a two-element sequence — that is, a Cartesian product of sets. (The coproduct of abelian groups is isomorphic to the commutative monoid coproduct, and not shown.)

Coproduct of sets

Figure 10 shows the instance for Coproduct in the category of sets. The first Coproduct parameter is instantiated to Set, a type class without constraints or methods, to which every Haskell type belongs. Then the associated Coprod type is simply the familiar type of binary sums (called Either in the Haskell standard library), with in1 and inr as its two constructors, and eva corresponding to the familiar either function.

5.2 Free extensions

§5.1 provides a general interface to coproducts. However, computing with partially-static data requires a particular form of coproduct: a free extension — i.e. the coproduct of an algebra and a free algebra. This section defines a general interface to free algebras that can be combined with Coproduct to give free extensions.

Figure 11 defines a type class Free indexed by a constraint, alg, and a type x. An instance Free alg x represents the free algebra for alg with variables in x; for example, Free Monoid (Code Int) represents the free monoid with variables in Code Int.

There are three class members. First, FreeA alg x is the type of values in the free algebra; as with Coprod, the definition of the type varies with each instance. For instance, the free algebra for Monoid is a list of variables, while the free algebra for CMonoid is a multiset. Second, pvar injects a variable into FreeA. Finally, the monadic pbind maps a value of FreeA into another algebra c via a function that injects variables into c. The class constraint alg (FreeA alg x) stipulates that there must be an alg instance for FreeA so that, for example, the type Free CMonoid m must also support the monoid operations 1 and ⊗, besides pvar.

Free algebras: monoids, commutative monoids, sets, abelian groups

Figure 12 shows implementations of free algebras for the structures discussed in §5.1: monoids, sets, commutative monoids and abelian groups. The free monoid with variables in x is simply a list of x values. The pbind function maps a free monoid value into any other monoid:

(pvar x₁ ⊗ pvar x₂ ⊗ ⋯ ⊗ pvar xₙ) `pbind` f

∼ (f x₁ ⊗ f x₂ ⊗ ⋯ ⊗ f xₙ)

A value in the free algebra for sets is simply a single variable, and pbind is application.

The CMonoid and CGroup free algebras are multisets (bags) and finite maps (dictionaries) with integer values, respectively. The implementations are straightforward, and omitted.

pvar x ⊗ cinv (pvar y) ⊗ pvar x ⊗ cinv (pvar y)

∼ ( x ⇒ 2 , y ⇒ 2 )

1https://hackage.haskell.org/package/base-4.10.0.0/docs/Data-Either.html
Partially static data as free extension of algebras

```haskell
class Ring a where
  (\@) :: a -> a -> a
  rneg :: a -> a ; 0, 1 :: a
  a @ b @ c = a @ (b @ c)
  a @ b = b @ a
  a @ 0 = a
  a @ rneg a = 0
  (a @ b) @ c = a @ (b @ c)
  a @ 1 = a
  a @ b @ b = b @ a
  a @ (b @ c) = (a @ b) @ (a @ c)
```

Figure 15. Commutative rings

```haskell
data Multinomial x a = MN (Map (MultiSet x) a)

instance (Ring a, Ord x) =>
  Coprod Ring a (FreeA Ring x) where
data Coprod Ring a (FreeA Ring x) =
  CR (Multinomial x a)

inl a = CR (MN (singleton empty a))
inr (RingA (MN x)) = CR (MN (map initMN x))
eva f g (a + bx^2y) = eva f g (g x) @ (g x) @ (g y)
```

Figure 16. Coproduct of a commutative ring and a free commutative ring

```
the free extension, as is the case with commutative rings. The free commutative ring on a set X consists of
multinomials, i.e. finite sums of products of variables in X with integer coefficients. As mentioned in §4.3, the coproduct of
a commutative ring A with a free commutative ring then consists of multinomials with coefficients in A (Figure 16).
Figure 14 illustrates the behaviour of a computation in a
partially-static commutative ring: the call to dot constructs
multinomials that residualize to multiplications and additions.

inl maps an element b of A to the constant term b, while
inr maps each term with coefficient n to the same term with
coefficient (1 @ n) (n times, or using @ for negative n).
eva evaluates the multinomial using the ring operations for
addition and multiplication:
eva f g (a + bx^2y) = eva f g (g x) @ (g x) @ (g y)
```

A free commutative semiring (an algebra for the operations
and axioms of a ring except those involving \$\oplus\$) is the
same but with natural number coefficients instead of inte-
ger, and the free extension is defined analogously.

**Coproduct of distributive lattices** A distributive lattice is a commutative semiring \((A, 0, \oplus, 1, \otimes)\) which additionally satisfies the absorption rules

\[ a \otimes (a \oplus b) = a, \quad a \oplus (a \otimes b) = a. \]
instance Functor f ⇒

Coproduct (Alg f) (FreeA (Alg f) a) (FreeA (Alg f) b) where

newtype Coprod (Alg f) (FreeA (Alg f) a) (FreeA (Alg f) b) =
   L (FreeA (Alg f) (Coproduct Set a b))

| stint | x = L (fmap Inl x) |
| int | y = L (fmap Inr y) |
| eva g h (L e) = e `pbind` eva (g . pvar) (h . pvar) |

**Figure 17.** Coproduct of two free F-algebras

For example, booleans form a distributive lattice with && and || as ⊗ and || as ⊕.

As in the case of commutative rings, the coproduct of a distributive lattice A and a free distributive lattice on a set X consists of multisetes over X with coefficients in A. However, the fact that multiplication is idempotent (a ⊗ a ≅ a) means that duplicates of variables within a term can be ignored, so the MultiSet of Figure 16 is replaced with Set. In addition, the second absorption rule means that any term of the sum which is a multiple of another term with the same coefficient is redundant and can be dropped.

**Coproduct of initial F-algebras** In addition to the familiar structures discussed above, the algebraic approach naturally subsumes earlier work on staged algebraic data types [12, 13, 20] that is discussed further in §6.

An algebraic data type is the initial algebra for a presentation consisting of a functor F and no axioms. In other words, it is constructed as the free F-algebra over the empty set.

For any algebraic structure, the coproduct of two free algebras is easy to calculate: it is given by the free algebra on the coproduct of their underlying sets. Figure 17 shows this in the case of F-algebras.

The free extension of an algebraic data type T := FreeA (Alg f) Empty is thus of this form, where the type a is the empty type and b is Code(T).

For example, the signature functor

IList X := ⊥ + Int × X

has initial algebra the type IntList of integer lists. The free extension of IntList is isomorphic to the free IList-algebra over Code(IntList).

### 5.3 Partially-static data

Figure 18 defines a general interface to partially-static data.

The first two lines define a free extension as the coproduct of an algebra and a free algebra: FreeExt C for a constraint indexed by an algebra and a set and FreeExt for the corresponding type component. The sta function generalizes the sta@ function for monoids (§3). If a is an instance of the algebraic structure alg, then sta injects a value of type a into the corresponding free extension. The dyn function similarly generalizes dyn@, injecting variables from a free algebra into the associated free extension. Finally, cd generalizes cd@, instantiating the evaluation function eva to build a value in Code from a value in the free extension. The auxiliary tlift function turns a value in a into a typed code value, for any type a that is an instance of Lift, the class of residualizable types.

### 5.4 Improving residualization

The generic partially-static representations in this section have the pleasant property that they do not prematurely residualize values: the static structure part of each structure is available for further computation right up to the point when cd is called.

However, since the generic operations of Figure 18 have no knowledge about the particular structures involved, the generated code sometimes shows opportunities for improvement, as illustrated below. The interface to partially-static data in some earlier work (e.g. [13]) provides cd as the only way to inspect partially-static data. The coproduct view improves on this approach, providing two additional ways of inspecting partially-static values: the eva function, and (in the Haskell implementation) the Coprod type. With eva and Coprod it becomes possible to perform further optimizations at the point of code generation.

**Residualization for monoids** The Monoid interface (Figure 1) exposes nullary and binary constructors ⊞ and ⊕. However, for some monoids it is most efficient to combine more than two elements in a single operation. For example, the partially-static monoid structure generates the following code for the printf example in §3:

\[
\text{[ s1 ++ $d ++ s2 ]}
\]
Partially static data as free extension of algebras

However, depending on the representation of strings, it may be more efficient to generate a single call to an n-ary cat-
nation function:

\[
[\text{concat } [s_1, s_2, s_3]]
\]

It is straightforward to write an alternative to \text{cd} specialized to the monoid of strings that generates this more efficient code\(^2\).

**Residualization for commutative monoids** Similarly, for the \text{power} function (§2.2), the \text{cd} function instantiated with \text{CMonoid} generates code that is optimal if the constraint is to use only operations in the \text{CMonoid} interface (where \(\circ\) is instantiated to \(*\)): generates code with 7 multiplications by default:

\[
\text{power } [x] 7 \rightsquigarrow x * x * x * x * x * x * x * x
\]

However, the \text{CMonoid} interface is a rather limited target for code generation. If the target language is expanded to use \text{let}-binding it is possible to save intermediate results, and so improve the performance of generated code\(^3\):

\[
\text{power } [x] 7 \\
\rightsquigarrow \text{let } a = x * x \text{ in let } b = a * a \text{ in x * a * b}
\]

As in the previous example, these improvements depend only on a specialized \text{cd}; the implementation of \text{power} and the partially-static representations are untouched.

### 5.5 Duplication of code

Since the aim of partially-static data structures is to avoid unnecessary computation in generated code, it is important to avoid duplicating or discarding expressions. In languages where the evaluation of an expression may have side effects, avoiding duplication and discarding is even more crucial.

However, in some of our examples, such as \text{power} and \text{dot}, quoted expressions injected with \text{dyn} may appear either several times or not at all in the output of \text{cd}.

Fortunately, there are standard techniques available to address this issue. It is common in partial evaluators and multi-stage programming languages to convert programs into a form where every non-trivial expression is \text{let}-bound [6, 14, 25] using a function (commonly named \text{genlet}) that accepts a dynamic expression e, inserts a \text{let}-binding for e at some higher point in the code, and returns the bound variable:

\[
g (\text{genlet } [\text{f x }]) \rightsquigarrow [\text{let } y = f x \text{ in ... } $(g [y])]
\]

Automatic conversion to ANF form in the LMS multi-stage programming framework [18] serves a similar purpose

\text{let}-insertion combines straightforwardly with partially-static data; however, we have omitted it from the exposition for simplicity. The MetaOCaml implementation (Appendix A) uses \text{let}-insertion to avoid duplication.

\(^2\)See the \texttt{Printf.cdStrings} function in the supplementary material

\(^3\)See \texttt{Power.cdPower} in the supplementary material

### 6 Related work

We consider two main classes of related work — previous structured approaches to partially-static data and ad-hoc implementations of particular partially-static structures — before touching briefly on the use of partially-static data in supercompilers, optimizing compilers, and other tools.

**Structured approaches to partially-static data** The existence of general schemes for partially-static structures without laws — i.e. datatypes — is well-known. The standard partial evaluation textbook [12] informally describes a representation of partially-static lists that generalizes to arbitrary recursive types (p527). Sheard and Diatchki [20] describe a similar, but more concrete, scheme for deriving the staged versions of particular datatypes in the multi-stage programming language MetaML [23]. Kaloper-Meršinjak and Yallop [13] incorporate Sheard’s scheme into a generic programming framework based on the initial algebra view of datatypes.

The present work builds on these foundations, showing how partially-static datatypes arise as a particular instance of the general view of partially-static algebraic structures as free extensions of algebras.

There have been fewer previous attempts to construct a general view of partially-static structures with laws. Thiemann [24] considers partially static operations, which incorporate algebraic laws, in a partial evaluation context. Thiemann’s [24] vision of specialization that considers algebraic structure is an inspiration for this work. However, the implementation is quite different: Thiemann’s [24] design operates via repeated rewriting, whereas the structures described in the present work are reduced using the evaluation mechanism of the host language. It consequently appears unlikely that the rewriting approach is suitable for use in multi-stage programming.

**Partially-static data in partial evaluation** Partially-static data has been used in partial evaluation from early times. Mogensen [16] introduced the concept in a study of partially-static lists, and several authors followed suit (e.g. [10, 12]). Sheard and Diatchki notes [20] that the term acquired the more specific meaning of static containers with dynamic elements.

**Partially-static data in multi-stage programming** Partially-static data is frequently employed in multi-stage programming, where eliminating unnecessary operations is an essential aspect of generating optimal code.

The seminal finally tagless work by Carette et al. [7] makes use of a static-dynamic type — i.e. a record that holds a dynamic representation and, optionally, an additional static value of the same value — to improve generated code in a staged embedded lambda calculus. The implementation additionally uses partially-static representations that implement ring simplification rules for zero addition and unit multiplication.
We have used free extensions of algebras as a functional code. Our approach combines the following attributes: extensible and modular: level library, specification of partially-static data, and described a high-static structure is a list with a possibly-dynamic tail. Combining the techniques in this paper with Carette and Kiselyov’s modular approach by instantiating numeric signatures with free extensions is a promising avenue for future exploration.

Yallop [25] makes use of several partially-static structures in the staging of an implementation of the Scrap Your Boilerplate generic programming library, including a partially-static structure for monoids that reassociates subexpressions in similar manner to the free extension presented here.

Drawing lessons from supercompilation, Inoue [11] makes use of partially-static data that is updated to reflect equalities between values during the static exploration of the dynamic branches of a staged program. The primary partially-static structure is a list with a possibly-dynamic tail.

7 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, frex, that uses them to produce efficient staged code. Our approach combines the following attributes:

Extensible and modular: The partially-static interface (sta, dyn, cd) operates uniformly over algebraic structures. Adding Coproduct and Free instances for an algebraic structure is sufficient to make the structure available for use in optimizations. Similarly, adding an instance of an algebraic class interface is sufficient to make the type available for use in optimizations. For example, since the standard library provides a Monoid instance for the Maybe type of optional values, frex will use the monoid laws to optimize programs involving Maybe even though frex itself makes no mention of Maybe.

Unifying: Partially-static data is a well-known technique for binding-time improvement, and ad-hoc implementations of structures that implement some algebraic simplifications are found throughout the literature (§6). The observation that partially-static data can be viewed as free extensions of algebras exposes and clarifies the structure underlying these ad-hoc implementations.

Reusable: This paper explores a universal view of partially-static data using a concrete library (frex) in a particular language (Haskell). However, the underlying ideas can be reused in many contexts: free extensions can be used to structure optimizers in other multistage languages, optimizing compilers, partial evaluators, supercompilers, program generators, and so on.

Practical: The effectiveness of algebraic optimization using with free extensions for partially-static data is evident both from the simplifications visible in generated code, and from benchmarks (§B).

In the future we would like to use free extensions of free theories to partially evaluate code using effect handlers [3].

Acknowledgments

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


Jeremy Yallop, Tamara von Glehn, and Ohad Kammar
A MetaOCaml implementation

Several features of Haskell — type classes, abundant polymorphism, typed quotation, succinct syntax — make it a convenient language for expounding the ideas in this paper. However, the view of partially-static data structures as free extensions of algebras is independent of any language, and transfers straightforwardly to a variety of settings. This appendix sketches a second implementation in the multi-stage programming language BER MetaOCaml [14]. The implementation is included as supplementary material.

Haskell is a lazy, pure language with support for compile-time metaprogramming, in which programs are structured using implicitly-resolved instances of type classes. In contrast, BER MetaOCaml is an eager, impure language with support for run-time metaprogramming, in which programs are structured using explicitly combined modules. Despite these differences, both languages provide the basic ingredients for implementing partially-static data structures as free extensions: the ability to define datatypes and a distinction between static and dynamic values.

A.1 Parameterization

The various generic components — definitions of coproducts, free extensions and so on, are defined within a signature \( \text{Sig} \) containing a parameterized module \( \text{ops} \) that builds an interface with a set of operations for a type:

```haskell
module type Sig = sig
module Ops (X: TYPE) : sig module type OP end ...
```

The partially-static representation for each algebraic structure instantiates \( \text{ops} \) appropriately. For example, here is the definition of the monoid interface:

```haskell
module Monoid_ops (T: TYPE) = struct
module type OP = sig
  type t = T.t
  val \!\! : t
  val (\*) : t \rightarrow t \rightarrow t
end
end
```

A.2 Algebras

Each additional element in \( \text{Sig} \) depends on \( \text{ops} \). An algebra is a pair of a type and a set of operations for the type:

```haskell
module type Algebra = sig
module T : TYPE
module Op : Ops(T).OP end
```
A.3 Coproducts

A coproduct is an aggregation of three instances T, A, B, of an algebra, together with injections from A and B into T and an evaluation function that maps values from T into any other algebra C:

```ocaml
definition COPRODUCT = sig
  module T : Algebra
  module A : Algebra
  module B : Algebra

  val inl : A.T.t -> T.T.t
  val inr : B.T.t -> T.T.t

end
```

The various coproduct implementations are expressed as parameterized modules that build instances of COPRODUCT, constraining A and B to match the parameters. Here is the signature of the parameterized module that builds a coproduct of monoids:

```ocaml
definition Coproduct_monoid (A : S.Algebra) (B : S.Algebra) : sig
  module T : Algebra
    val a : A.T.t
    val b : B.T.t

  end
```

A.4 Presentations and free algebras

A presentation of an algebra is a definition of the free algebra, with supports injection of variables, and a bind function that performs substitution:

```ocaml
definition PRES = sig
  module Free(X : Setoid) : sig
    module Alg : Algebra
      val x : X.t -> Alg.T.t
  end

  module Bind(X : Setoid) (C : Algebra) : sig
    val (>>=) : Free(X).Alg.T.t -> (X.t -> C.T.t) -> C.T.t
  end
```

The partially-static structure for an algebra A aggregates A, a second implementation of the algebra, the var and sta injections, and the cd and eva destructors. Since MetaOCaml is an impure language, it is crucial to avoid duplicating or discarding expressions, and so dynamic injections are from a distinct type var rather than arbitrary code values (Cf. §5.5).

```ocaml
definition PS = sig
  module A : Algebra
  include Algebra
  val sta : A.T.t -> T.t
  val var : A.T.t var -> T.t
```

A.5 Programming with partially-static data

Although the set of abstraction mechanisms available in the language leads to a structure that is quite different from the Haskell implementation, the programming experience is rather similar. For example, the `printf` example is expressed identically:

```ocaml
printf ((int ++ lit "a") ++ (lit "b" ++ int))
```

B Performance evaluation

The central contribution of this paper is a unification of various existing optimizations based around partially-static data. While some of the examples, such as the optimized `printf` and `all` functions, are new, the fact that partially-static data can improve the performance of code generated by multi-stage programs (§6) is well-known. Consequently, the primary question of interest is whether it is possible to generate particular code — for instance, to generate code in which no opportunities for algebraic optimizations with static data remain. The performance of particular generated programs is a secondary concern in this work.

Nevertheless, it is reassuring to see that the code generated by the frex library for several examples evinces improved performance over both the original unstaged program and naively staged versions that make no use of partially-static data.

The measurements in this section were taken on a Debian Linux system running the 4.9.0 kernel on an AMD FX(tm)-8320 eight-core processor with 16GB memory. Haskell code was compiled with GHC 8.0.2, and OCaml code with BER MetaOCaml n104.

We consider two representative examples: matrix multiplication, in Haskell, and the staged `printf` function, in OCaml.

B.1 Matrix multiplication

Figure 19 shows the performance of four implementations of 10x10 matrix multiplication in Haskell. The naive implementation is a one-line implementation based on representations of matrices as lists of lists:

```hsaskell
[[dot a b | b <- transpose n] | a <- m]
```

The figures for linear represent the performance of a popular Haskell library of the same name, based on a vector representation. There are two staged implementations, both of which are instantiations of the one-line implementation above with appropriate instances. The naive staging simply unrolls the loop, turning the list traversal into an arithmetic
expression. The partially-static version additionally uses frex to perform algebraic simplification.

As the sparsity of the matrix increases, the algebraic simplifications performed by the partially static version significantly increase its advantage over naive staging.

### B.2 Printf

Figure 20 shows the performance of three implementations of the printf function — unstaged, staged, and partially static (with improved residualization (§5.4))), written using the OCaml version of frex (Appendix A). Once again, the partially-static version gains an edge as the opportunities for algebraic simplification increase — in this case, as the number of subexpressions to reassociate grows.