Partially-Static Data as Free Extension of Algebras

JEREMY YALLOP, University of Cambridge, United Kingdom
TAMARA VON GLEHN, University of Cambridge, United Kingdom
OHAD KAMMAR, University of Oxford, United Kingdom

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 to 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.

1 INTRODUCTION

The defining feature of multi-stage programming is putting fine-grained control over beta reduction in the hands of the programmer. For example, the multi-stage programming language Typed Template Haskell [Peyton Jones 2016] extends Haskell with two constructs. The first construct, quotation (written ⟦ e ⟧), prevents beta reduction, turning an arbitrary expression into a value:

```
⟦ f x ⟧      -- f x should not be reduced
```

The second construct, antiquotation (written $e), re-enables beta reduction inside a quotation:

```
⟦ f $(g x) ⟧     -- g x should be reduced
```

A notable property of this style of quotation is support for reduction *under lambda*, with support for quoting *open terms*:

```
⟦ \x → $(f ⟦x⟧) ⟧     -- f ⟦x⟧ should be reduced
```

Authors' addresses: Jeremy Yallop, jeremy.yallop@cl.cam.ac.uk, Department of Computer Science and Technology, University of Cambridge, United Kingdom; Tamara von Glehn, T.L.Von-Glehn@dpmms.cam.ac.uk, Department of Pure Mathematics and Mathematical Statistics, and Newnham College, University of Cambridge, United Kingdom; Ohad Kammar, ohad. kammar@cs.ox.ac.uk, University of Oxford, Department of Computer Science, and Balliol College, Wolfson Building, Parks Road, Oxford, OX1 3QD, United Kingdom.

This fine-grained control over evaluation provides a basis for optimisation. In the execution of a multi-stage program, each stage executes code constructed from quoted expressions in the previous stage. Careful insertion of quotations and antiquotations allows terms depending only on known values to be reduced during generation, leaving only terms depending on unknown values to be reduced in the following stage.

The standard introductory example of multi-stage programming [Taha 2003] is the power function, implemented in terms of multiplication:

```
power :: Int → Int → Int
power x 0 = 1
power x n = x * power x (n - 1)
```

Under the assumption that the exponent n will be available to the generating stage, the programmer adds staging annotations and changes the types accordingly. (Here Code, an alias for Typed Template Haskell's Q (TExp a), is the type of quoted expressions):

```
power :: Code Int → Int → Code Int
power x 0 = ⟦1⟧
power x n = ⟦$x * $(power ⟦x⟧ (n - 1))⟧
```

Now power is no longer a function on integers, but a code generator: the application of power to a variable x and an integer n evaluates to an expression specialized to the exponent, in which the overhead of the recursive call and the branch have been eliminated:

$$\text{power } ⟦x⟧ \; 6 \quad \leadsto \quad ⟦ \; x * (x * (x * (x * (x * (x * 1))))) \; ⟧$$

However, while the generated code is likely to outperform the unstaged power, there is evidently room for further improvement. First, there is a needless multiplication by 1. More significantly, the number of multiplications can be further reduced by **let**-binding intermediate results:

$$⟦ \; \textbf{let } y = x * x \textbf{ in let } z = y * y \textbf{ in } z * y \; ⟧$$

Evidently, beta reduction alone will not perform these simplifications, no matter how we coax it by inserting quotation and antiquotation annotations. No series of beta reductions will perform these simplifications, which are justified by the algebraic properties of multiplication — in this case, that integers with multiplication form a monoid.

Similar difficulties occur in a wide variety of staged programs, as we illustrate in §4. In programs that are staged by quoting expressions the structure of the generated code is determined by the evaluation structure of the generator, and algebraic laws play no part in code generation.

How might we deal with this difficulty? First, we might restructure the source program (e.g. defining power via an auxiliary function square). However, relying on the programmer to take account of algebraic laws does not scale well. Second, we might post-process generated code to perform algebraic simplifications. However, this risks giving up the most desirable properties of multi-stage programming: namely that code transformations can be implemented by the programmer within the language itself. Finally, we might introduce specialized numeric representations that simplify generated code using laws of the underlying algebraic structure. This is, in our view, the most promising approach, and is widely used in the multi-stage programming literature (§7).

This paper builds on this third approach, re-examining the foundations of these *partially-static* structures, which are typically approached in *ad-hoc* fashion, and conceptualising them with a single *universal property*, in terms of the operations and equations involved. 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*; however, the formulation transfers straightforwardly to other settings. (In the extended version of this paper we describe a second implementation of *frex* in the multi-stage language BER MetaOCaml [Kiselyov 2014].)

The *frex* library has a number of appealing features. First, *frex* applies algebraic simplification during the normal evaluation of a generating stage of a multi-stage program — i.e. *it turns algebraic simplification into beta reduction*, improving the performance of generated programs. Second, *frex* provides drop-in replacements for many common algebraic structures that make it possible to repurpose existing polymorphic code for staging. Third, by unifying a wide variety of partially-static structures under the single concept of a *free extension*, *frex* exposes a tiny user-facing interface consisting of just three simple functions together with existing algebraic classes.

The **contributions** of this paper are as follows:

- §2 provides a general introduction to the principles underlying partially-static structures with algebraic laws, starting with binary operations with no equations, and showing the effect on the representation of adding associativity and commutativity laws.
- §3 shows how the considerations in §2 identify each partially-static structure with a *free extension* — i.e. the coproduct of an algebra and a free object in the appropriate category.
  The unifying view provided by *free extensions* transforms programming with partially-static data, as monads have transformed programming with effects: it guides the definition of instances and programs, clarifies semantic properties, and provides a high-level framework in which many common patterns can be uniformly abstracted.
- §4 uses free extensions to define a variety of partially-static instances for algebraic structures: monoids (§4.1), commutative monoids and abelian groups (§4.5), sets (§4.8), commutative rings and semirings (§4.11), distributive lattices (§4.13) and algebraic data types (§4.15).
  In each case we illustrate the structures with examples drawn from the literature, showing how *frex* can be applied to programming problems such as arithmetic, pretty-printing, linear algebra and list manipulation, to improve generated code by algebraic simplifications.
  As the representative benchmarks in §6 demonstrate, these simplifications translate directly into faster running times.
- The major part of this paper is intended to be accessible, since we hope that *frex* and its techniques will be widely adopted by functional and multi-stage programmers, as well as in other settings such as compiler optimisation and partial evaluation. However, readers with some familiarity with universal algebra will find a more formal justification for using free extensions to represent partially-static structures in §5.
- §7 contextualizes our contributions among the work on partially-static data.

2 DEFINING PARTIALLY-STATIC STRUCTURES

How might we build implementations of algebraic operations that support computation with partially-static data? This section sketches a general approach to defining these *partially-static algebras* that can be used as drop-in replacements for standard instances in type-class polymorphic code. §4 describes the implementation of this approach as a high-level, modular and extensible library, which applies directly to several examples drawn from the literature.

We start with the simplest non-trivial algebraic structure. A *magma* consists of a set a along with a binary operation. We define a Haskell class Magma and introduce a picture form, representing the binary constructor • as a binary branch in a tree:

```
class Magma a where (•) :: a → a → a
```

There are many instances of Magma, since any binary operation for a type forms a magma. The most general instance simply represents the tree structure directly:

```
data Tree a where                    instance Magma (Tree a) where
    Leaf :: a → Tree a                   (•) = Branch
    Branch :: Tree a → Tree a → Tree a
```

Here is a second Magma instance, for multiplication for integers, using a type isomorphism $Int_\times$ to distinguish the instance from magmas for other integer operations such as addition or subtraction:

```
newtype Int× = Int× Int
instance Magma Int× where (Int× x) • (Int× y) = Int× (x × y)
```

Then we can define trees using the Magma operation together with integer values, and interpret those trees using the Magma instance for $Int_\times$:

$$\bullet \xrightarrow{\text{instantiate}} \times \xrightarrow{\text{reduce}} 6$$
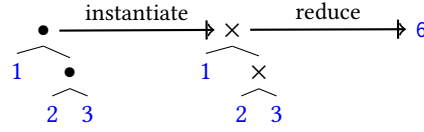
So the reduced form of a magma term instantiated to $Int_\times$ is (isomorphic to) a single integer.

We would also like to compute with open terms containing free variables x_1, x_2, such as these:
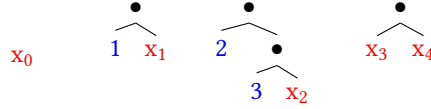
The type Tree (**Either Int** (Code **Int**)) gives a simple representation for these mixed trees. However, this representation keeps all trees, even those without free variables, in unreduced form.

Free Variables and Binding-Time Analysis. In our setting free variables correspond to what are called *dynamic* variables in the partial evaluation literature, and integer values correspond to *static* values. The division into static and dynamic also extends to terms: dynamic terms are terms that mention some dynamic variables; other terms are static.

A classification of terms into static and dynamic is called a *binding-time analysis*. Fig. 1 shows a binding-time analysis for a magma tree. The left child contains a dynamic variable, and so it is classified as dynamic. The right child contains only static values, and so it is classified as static. The root tree has a dynamic left child, and so it is classified as dynamic.
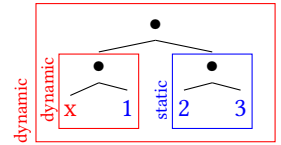
The aim of the binding-time analysis is to identify static sub-terms, since it is only those terms that can be reduced. Here is an illustration: the right sub-term is static, and consequently reduced, but no further reduction can take place, since all remaining terms are dynamic:

Fig. 1. Binding-time analysis

$$\bullet \xrightarrow{\text{instantiate}} \times \xrightarrow{\text{reduce}} \times$$
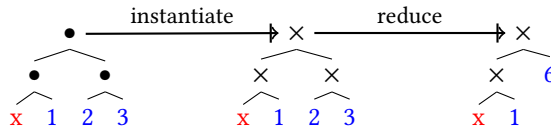
Even before instantiation it was evident that the dynamic terms would block reduction.

We can define a representation for mixed trees that takes binding-times into account, allowing reduction of static sub-trees. We start with a definition of binding-times, BindingTime, and an

indexed type BT that reflects BindingTime at the type level, so that we can enforce constraints about binding-times in representations of data:

```
data BindingTime =                    data BT :: BindingTime → * where
       Sta                              BTSta :: BT Sta
     | Dyn                              BTDyn :: BT Dyn
```

Using BindingTime we define an indexed type SD to stand in for **Either** a (Code a), and a function btSD that computes the binding time for an SD value:

```
data SD :: BindingTime → * → * where       btSD :: SD bt a → BT bt
  S :: a → SD Sta a                        btSD (S _) = BTSta
  D :: Code a → SD Dyn a                   btSD (D _) = BTDyn
```

Finally, the Mag type represents magma terms that do not contain unreduced static subtrees. Leaves may be static or dynamic (LeafM); if the left branch of a tree is static then the right must be dynamic (Br1); if the left branch is dynamic then the right may be static or dynamic (Br2):

```
data Mag :: BindingTime → * → * where         btMag :: Mag bt a → BT bt
  LeafM :: SD bt a → Mag bt a                 btMag (LeafM m) = btSD m
  Br1 :: Mag Sta a → Mag Dyn a → Mag Dyn a    btMag (Br1 _ _) = BTDyn
  Br2 :: Mag Dyn a → Mag r   a → Mag Dyn a    btMag (Br2 _ _) = BTDyn
```

Here is the magma instance itself. The definition of • uses binding times to reduce the number of cases to the three circumstances of interest: the case where both operands are static (in which case the elements must be coalesced), and the two cases that correspond to Br1 and Br2:

```
instance Magma a ⇒ Magma (Exists Mag a) where
  E a • E b = case (btMag a, btMag b, a, b) of
     (BTSta, BTSta, LeafM (S a), LeafM (S b)) → E (LeafM (S (a • b)))
     (BTSta, BTDyn, l,              r)         → E (Br1 l r)
     (BTDyn, _   ,  l,              r)         → E (Br2 l r)
```

The Mag instance uses an existential type to hide the BindingTime index in order to conform to the Magma interface. Existential types will come in useful on several occasions, so we define a general Exists type that hides the first parameter b of a binary type constructor f, using kind polymorphism to support arbitrary index kinds:

```
data Exists (f :: k₁ →k₂ →*) a where E :: f b a → Exists f a
```

Equality and Associativity. Trees are considered equal if they have the same shape and if corresponding leaves are equal. Furthermore, trees are considered equal if they reduce to equal trees.

Adding laws to the algebraic structure groups trees into larger equivalence classes. For example, an associativity law equates trees with different branching structures:

Here *a*, *b*, *c* stand for arbitrary terms.

A magma with an associativity law is called a *semigroup*. We introduce a corresponding Haskell class, Semigroup, with the same members as Magma, and an additional obligation for instances, stated in a comment: the implementation of • must be associative:

```
class Magma a ⇒ Semigroup a    -- a • (b • c) ≡ (a • b) • c
```

For example, the instance for integers with multiplication can be made into a valid `Semigroup` instance, since multiplication is associative:

$$\textbf{instance } \text{Semigroup } \textbf{Int}_\times$$

Magmas for non-associative operations, such as subtraction, do not give rise to semigroups.

Standard Haskell does not provide a way of ensuring that instances obey the laws of their class. However, some extensions, such as Liquid Haskell, provide various means of checking that instances are law-abiding.

With the associativity law, trees are equal if they have the same sequence of leaves; branching structure is no longer relevant.
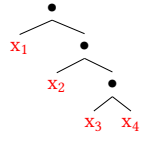

2.0.1 Equivalence and Normal Forms. It is convenient to pick a canonical representative of each equivalence class, i.e. a *normal form*. Without the associativity law the canonical representative of each class is the tree that has been reduced as much as possible.

For trees without any free variables the most-reduced tree is a single static element, such as an element of $\textbf{Int}_\times$. For trees with *only* free variables, there is no reduction, and a normal form is a fully right-associated tree — equivalent to a non-empty value of type [`Code a`]. (This is called the *free semigroup*.)

What is the normal form for a tree with both integers and free variables? For magmas the normal form is represented by `Mag`, which ensures that static subtrees are reduced. For semigroups it is additionally possible to reassociate the tree to make adjacent nodes (i.e. nodes that are adjacent in the left-to-right leaf order) into siblings (i.e. nodes with the same parent):
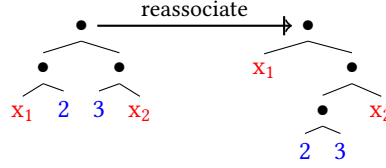
Making adjacent static nodes into siblings makes it possible to reduce the parent node. So a normal form for trees with both static and dynamic elements is a fully right-associated tree with no adjacent static nodes:
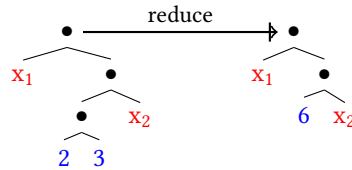
In the partial evaluation literature, a change in term structure that increases the number of static terms is called a *binding-time improvement*.

As with `Magma`, we can define a datatype for mixed semigroup trees in normal form. Besides ensuring that there should be no unreduced static subtrees the `Semi` type adds a new constraint that all trees are kept in right-associated form, by making the first arguments of the two binary-branching constructors `ConsS` and `ConsD` atoms rather than trees:

```
data Semi :: BindingTime → * → * where
  LeafS :: SD bt a → Semi bt a
  ConsS ::        a → Semi Dyn a → Semi Dyn a
  ConsD :: Code a → Semi r   a → Semi Dyn a
```

It is convenient to define auxiliary functions that add static and dynamic elements to the left of the tree. The consS function adds a static element to a Semi tree; if there is a static element in leftmost position already, consS combines the two elements using •:

```
consS :: Magma a ⇒ a → Exists Semi a → Exists Semi a
consS h (E (LeafS (S s)))   = E (LeafS (S (h • s)))
consS h (E t@(LeafS (D _))) = E (ConsS h t)
consS h (E (ConsS s t))     = E (ConsS (h • s) t)
consS h (E t@(ConsD _ _))   = E (ConsS h t)
```

The consD function is simpler, because dynamic elements may be added to the left of any tree:

```
consD :: Code a → Exists Semi a → Exists Semi a
consD h (E t) = E (ConsD h t)
```

Finally, here are Magma and Semigroup instances for Semi:

```
instance Semigroup a ⇒ Magma (Exists Semi a)          instance Semigroup a ⇒
 where E (LeafS (S s)) • l = consS s l                        Semigroup (Exists Semi a)
       E (LeafS (D d)) • l = consD d l
       E (ConsS h t) • l = consS h (E t • l)
       E (ConsD h t) • l = consD h (E t • l)
```

The final two cases of • may traverse the entire left operand to handle the case where the final static element on the left should be coalesced with the initial static element on the right:
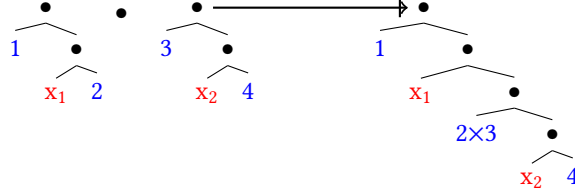
Commutativity. Adding an associativity law made trees equivalent that were previously distinct and made trees reducible that were previously irreducible. Adding a second law for commutativity coalesces more equivalence classes and adds further opportunities for reduction:

The CSemigroup class adds a law to make • commutative:

```
class Semigroup a ⇒ CSemigroup a    -- a • b ≡ b • a
```

Since many types of element have no notion of ordering, we must represent ordering directly in the tree structure. We introduce a new tree constructor, with an unordered bag of *n* children:
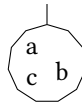
The ability to reorder subtrees significantly simplifies the normal form. Since all the static elements can be moved to one end, and subsequently reduced, the normal form for a mixed static-dynamic commutative semigroup is simply a pair of an optional single static element and an unordered bag of dynamic variables:
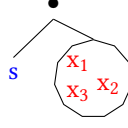
Here is the normal form in Haskell, using the standard multiset to represent bags:

```
data CSemi a = CSemi (Maybe a) (MultiSet (Code a))
```

Then l • r is a pair whose components are built from the corresponding parts of l and r using • for the underlying magma lifted to `Maybe` values for the static component and multiset union for the bag of dynamic variables:

```
instance CSemigroup a ⇒ Magma (CSemi a) where
  CSemi s1 d1 • CSemi s2 d2 = CSemi (s1 •? s2) (union d1 d2)
    where Nothing •? m = m
            m •? Nothing = m
            Just m •? Just n = Just (m • n)
```

3 PARTIALLY-STATIC DATA: A GENERAL INTERFACE

§2 introduced the ideas behind computing with partially-static algebras. We now look at how to turn these ideas from a design pattern into a general abstraction that can be instantiated with particular algebras to support partially-static computation that turns algebraic equations into beta reduction. The general abstraction is the core of our Haskell library, *frex*, which provides an extensible and modular interface to partially-static data. The ideas behind *frex* are not tied to any particular language, and in the extended version of this paper we also describe a second implementation of *frex* in MetaOCaml.

3.1 Partially-Static Data: Requirements

With the aim of constructing an interface PS to partially static data, we start with a list of requirements, distilled from the discussion up to this point.

First, the PS type is intended to work with a variety of algebraic structures, and so we parameterise it by an algebraic structure and the type of the data it represents:

$$\text{PS :: } (* \to \text{Constraint}) \to * \to *$$

The first parameter is a *constraint* that stands for a type class such as Magma or CSemigroup.

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

```
sta :: algebra a ⇒ a → PS algebra a
dyn :: Code a → PS algebra a
```

(We will extend these type signatures with further constraints in §3.3.) In §2, sta and dyn took various forms: LeafM (S -) and LeafM (D -) for magmas, and LeafS (S -) and LeafS (D -) for semigroups. For commutative semigroups the injections can be written as follows

$$\text{sta}_{CS} = \lambda\text{s} \to \text{CSemi (\textbf{Just} s) empty}$$
$$\text{dyn}_{CS} = \lambda\text{d} \to \text{CSemi \textbf{Nothing} (singleton d)}$$

Furthermore, PS algebra itself should be an instance of algebra, since it is intended to stand in for contexts where an algebra instance is expected. For example, the partially-static structure for semigroups should be a semigroup instance.

```
class (algebra a, algebra b, algebra (Coprod algebra a b)) ⇒ Coproduct algebra a b
 where data family Coprod algebra a b :: *
       inl :: a → Coprod algebra a b
       inr :: b → Coprod algebra a b
       eva :: algebra c ⇒ (a → c) → (b → c) → Coprod algebra a b → c
```

Fig. 2. The Coproduct interface

3.2 Coproducts

These considerations suggest designing *frex* around *coproducts* — that is, not the familiar binary sums that Haskell calls `Either`, but the more general notion of coproducts in a category, whose representation varies according to the category.

Fig. 2 defines the coproduct interface as a Haskell type class, Coproduct, with three parameters and four components.

The three parameters algebra, a, and b, respectively represent a type class for a particular algebraic structure and the two types that comprise the coproduct. For example, Coproduct Monoid `Int String` represents the coproduct of `Int` and `String` in the category of monoids. The first two class constraints algebra a and algebra b constrain the instantiation of the parameters to types that have instances of algebra. For example, the instantiation Coproduct 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 algebra. Coprod is an *associated data type* [Chakravarty et al. 2005], whose definition varies with each instance of the Coproduct class. For example a coproduct in the category of *monoids*, Coprod Monoid `Int String`, is an alternating sequence similar to the Semi type of §2 (and defined more precisely in §4.1), whereas the type Coprod Set `Int String`, a coproduct in the category of sets, is the familiar binary sum type (§4.8).

The final class constraint algebra (Coprod algebra a b) ensures that each instantiation of Coprod is an instance of the algebraic structure algebra — for example, there must be a Monoid instance for the type Coprod Monoid a b. 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 generalizes the standard **either** function, producing a value of type c from a value of type Coprod and functions from a and b to c. For example, if algebra has an operation • then eva behaves as follows:

$$\text{eva f g (inl } s_1 \bullet \text{ inr } d_1 \bullet \text{ inl } s_2 \bullet \dots) \quad \rightsquigarrow \quad \text{f } s_1 \bullet \text{ g } d_1 \bullet \text{ f } s_2 \bullet \dots$$

where the • operations on the left are from Coprod algebra a b and the • operations on the right are from c. (In particular, eva inl inr is the identity.) The constraint algebra c in the type of eva ensures that c is also an instance of the algebraic structure associated with the instance: for example, a coproduct of monoids can only be eliminated into a monoid.

3.3 Free Objects and Free Extensions

§3.2 provides a general interface to coproducts. However, computing with partially-static data requires a particular form of coproduct, where the left type is some type a and the right type is generated by quoted terms Code a. Furthermore, while there are no constraints on the left algebraic structure, the right structure is always *free*, since Code values do not have computational behaviour or additional equations.

In other words, we are interested in what are called *free extensions*: coproducts of algebras and free objects. This section defines a general interface to free algebras and shows how they combine with Coproduct to give free extensions.

```
class algebra (FreeA algebra x) ⇒ Free algebra x where
   data family FreeA algebra x :: *
   pvar :: x → FreeA algebra x
   pbind :: algebra c ⇒ FreeA algebra x → (x → c) → c
```

Fig. 3. Free algebraic structures, Free

```
FreeExt_C :: (* → Constraint) → * → Constraint
type FreeExt_C algebra a = Coproduct algebra a (FreeA algebra (Code a))

FreeExt :: (* → Constraint) → * → *
type FreeExt algebra a = Coprod algebra a (FreeA algebra (Code a))
```

Fig. 4. Free extensions: constraint alias FreeExt$_C$ and type alias FreeExt

Fig. 3 defines a type class Free indexed by a constraint, algebra, and a type x. An instance Free algebra x represents the free algebra for algebra with variables in x; for example, Free Semigroup (Code **Int**) represents the free semigroup with variables in Code **Int**.

There are three class members. First, FreeA 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 Semigroup is a non-empty list of variables, while the free algebra for CSemigroup is a non-empty 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 algebra (FreeA algebra x) stipulates that there must be an algebra instance for FreeA so that, for example, the type FreeA Semigroup m must also support the operation ● in addition to pvar.

Fig. 4 shows how the Coproduct and Free classes combine to give the definition of a free extension as a coproduct of an algebra and a corresponding free algebra. There are two definitions: FreeExt$_C$ defines a constraint that may appear to the left of a fat arrow (⇒), while FreeExt names the type associated with the FreeExt$_C$ instance. For example, FreeExt Semigroup **Int** defines the type of the free extension of semigroups for integers as an alternating sequence (the coproduct) of integers and lists of integer code values (the free algebra).

The definitions of Fig. 4 form a crucial part of the general partially-static data interface (§3.5).

It is worth emphasizing that the free extension structure may look quite different from both the static and the dynamic instance. For example, in the semigroup of integers with multiplication each expression reduces to a single integer; in the *free* semigroup each expression reduces to a non-empty sequence of names; in the *semigroup free extension* each expression reduces to a sequence of integers and names, with the additional constraint that there are no adjacent integer elements.

These considerations lead to new definitions for sta and dyn. We drop the provisional name PS in favour of using FreeExt directly, and write:

```
sta :: (algebra a, FreeExt_C algebra a) ⇒ a → FreeExt algebra a
sta = inl
dyn :: (Free algebra (Code a), FreeExt_C algebra a) ⇒ Code a → FreeExt algebra a
dyn = inr . pvar
```

The first definition defines sta, the function that builds partially-static representations from static values, as the left injection into the free extension of an algebra. The second definition defines dyn as the composition of the injection into the free object and the injection into the free extension. The constraints ensure that there are algebra and Free algebra instances available for the static and dynamic components and FreeExt$_C$ algebra instances available for the results.

Viewing partially-static algebraic structures as free extensions makes explicit some additional requirements on the implementation of instances. For example, sta should be a homomorphism with respect to each operation •, i.e.:

$$\text{sta } x \bullet \text{sta } y \quad \equiv \quad \text{sta } (x \bullet y)$$

Furthermore, eva should preserve the laws of the algebra, so that terms that are equivalent under the laws remain equivalent when eva is applied.

3.4 From Partially-Static to Fully Dynamic

The eva function is a general-purpose destructor for coproducts. The most common use of eva with partially-static data is *residualization*: turning partially-static values into fully-dynamic values.

A residualization function can be obtained from eva as follows. First, specialize eva to FreeExt, where the second type parameter to Coproduct is instantiated to FreeA algebra (Code a):

```
eva_FE :: (FreeExtCon algebra a, algebra c) ⇒
          (a → c) → (FreeA algebra (Code a) → c) → FreeExt algebra a → c
eva_FE = eva
```

Next, instantiate the return type of eva_FE to Code a:

```
eva_Code :: (FreeExtCon algebra a, algebra (Code a)) ⇒
   (a → Code a) →(FreeA algebra (Code a) → Code a) →FreeExt algebra a →(Code a)
eva_Code = eva_FE
```

Finally, supply suitable arguments for the first two parameters of eva_Code. The first argument to eva converts static values to code; this is the purpose of tlift, a typed variant of Template Haskell's lift function that provides an interface to cross-stage persistence:

```
tlift :: Lift a ⇒ a → Code a
tlift = liftM TExp . lift
```

The second argument to eva builds Code values from values of a free object; this can be accomplished with pbind (§3.3).

This series of specializations produces the following residualization function, which is the final component of *frex*'s general interface to partially-static data.

```
cd :: (Lift a, Free algebra (Code a), algebra (Code a), FreeExt_C algebra a) ⇒
      FreeExt algebra a → Code a
cd = eva tlift (`pbind` id)
```

Here is cd in action:

$$\text{cd } ((\text{dyn } [\![x]\!] \bullet \text{sta } 2) \bullet (\text{sta } 3 \bullet \text{dyn } [\![y]\!])) \quad \leadsto \quad [\![x \times 6 \times y]\!]$$

And, of course, cd preserves the laws of the algebra, since eva does, so that equivalent partially-static computations are residualized to equivalent code.

3.5 Using *frex*

Fig. 5 summarises *frex*'s interface to partially-static data.

How does one use *frex* to write programs? In order to use *frex* to program with partially-static representations for an algebraic structure such as Monoid or Ring, two things are needed: a free extension instance for the structure, and instances of the structure for particular types such as **Int** or **String**. In many cases, these requirements will be met by the combination of *frex* and existing Haskell libraries. Here are some common scenarios:

```
FreeExt_C :: (* → Constraint) → * → Constraint
type FreeExt_C algebra a = Coproduct algebra a (FreeA algebra (Code a))

FreeExt :: (* → Constraint) → * → *
type FreeExt algebra a = Coprod algebra a (FreeA algebra (Code a))

sta :: (algebra a, FreeExt_C algebra a) ⇒ a → FreeExt algebra a
sta = inl

dyn :: (Free algebra (Code a), FreeExt_C algebra a) ⇒ Code a → FreeExt algebra a
dyn = inr . pvar

cd :: (Lift a, Free algebra (Code a), algebra (Code a), FreeExt_C algebra a) ⇒
      FreeExt algebra a → Code a
cd = eva tlift (`pbind` id)
```

Fig. 5. *frex*'s generic interface to partially-static data

3.5.1 Using frex with Existing Instances. Frex includes pre-defined free extensions for a number of structures, including sets, monoids, commutative rings, distributive lattices, abelian groups, and F-algebras. If *frex* already defines a free extension for a structure and some other library provides instances of the structure for types in the program, using *frex* is typically simply a matter of inserting sta, dyn and cd.

For example, since the standard library provides a **String** instance for Monoid, nothing more is needed to write programs involving partially-static strings:

$$\text{cd ((dyn } [\![x]\!] \text{ `mappend` sta "abc") `mappend` (sta "def" `mappend` dyn } [\![y]\!]))$$
$$\rightsquigarrow \quad [\![x \text{ `mappend` "abcdef" `mappend` y}]\!]$$

3.5.2 Creating New Instances to Use with frex. Similarly, no special work is needed to add new instances to *frex* when the free extension is already defined. For example, adding a CMonoid instance for () (building on the existing Monoid instance) is sufficient to enable *frex*'s commutative monoid simplifications:

instance CMonoid ()

3.5.3 Adding New Classes to frex. Finally, adding new structures to *frex* is a matter of defining suitable Coproduct and *Free* instances; again, no modifications to *frex* internals are needed. §4 provides a number of examples, building on the discussion in §2.

4 INSTANCES AND APPLICATIONS

With the general interface in place (Fig. 5), we now turn to the implementation of free extensions for common algebraic structures: monoids (§4.1), commutative monoids and abelian groups (§4.5), sets (§4.8), commutative rings (§4.11), distributive lattices (§4.13), and F-algebras (§4.15).

In many cases the general coproduct for the structure and the free object can be defined separately, then combined to give the free extension. However, in some cases (e.g. §4.11) where it is not possible to give a general form for the coproduct, we define the free extension directly.

```
class Monoid t where              𝟙 ⊛ x  ≡  x  ≡ x ⊛ 𝟙
  𝟙 :: t
  (⊛) : t → t → t             x ⊛ (y ⊛ z)  ≡ (x ⊛ y) ⊛ z
```

Fig. 6. Monoids and their laws

```
data AorB = A | B
data Alternate :: AorB → * → * → * where
  Empty :: Alternate any a b
  ConsA :: a → Alternate B a b → Alternate A a b
  ConsB :: b → Alternate A a b → Alternate B a b
```

Fig. 7. An alternating sequence of a and b elements

```
instance (Monoid a, Monoid b) ⇒ Coproduct Monoid a b where
  data Coprod Monoid a b where M :: Alt _ a b → Coprod Monoid a b
  inl a = M (ConsA a Empty)
  inr b = M (ConsB b Empty)
  eva (f :: a → d) (g :: b → d) (M c) = eva' c
   where eva' :: Alternate start a b → d
         eva' Empty = 𝟙
         eva' (ConsA a Empty) = f a
         eva' (ConsB b Empty) = g b
         eva' (ConsA a m) = f a ⊛ eva' m
         eva' (ConsB b m) = g b ⊛ eva' m
```

Fig. 8. The coproduct of monoids

4.1 Coproduct of Monoids

Fig. 6 defines a Monoid class. The free extension for monoids is an instance of the more general coproduct of monoids, and a slight variant of the partially-static structure for semigroups (§2).

Fig. 8 gives the Coproduct instance for the Monoid class constraint, built from Monoid instances a and b. As with Semi, the Coprod type is defined as a sequence of alternating a and b elements (Fig. 7), indexed by the type of the first element in the list, and with the index hidden by an existential (here M) to allow either a-prefixed or b-prefixed sequences. Unlike semi, the sequence may be empty, since Monoid adds an identity element. The inl and inr injections create singleton sequences, and eva is a fold over the sequence, using the ⊛ operation of the target monoid to combine the results. As a small optimization, eva does not map the Empty constructor to 𝟙 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.

Fig. 9 defines the Monoid instance for Coprod Monoid a b, where a and b also have instances of Monoid.

The 𝟙 and ⊛ operations respectively construct an empty sequence and concatenate two sequences. Prepending an a element to an a-prefixed sequence combines the element with the head of the sequence using the ⊛ operation of the a monoid, and similarly for b, *mutatis mutandis*.

```
instance (Monoid a, Monoid b) ⇒ Monoid (Coprod Monoid a b) where
  𝟙 = M Empty
  M l ⊛ M r = l `mul` r
   where mul :: (Monoid a, Monoid b) ⇒
              Alternate s a b → Alternate s' a b → Coprod Monoid a b
         mul l Empty = M l
         mul Empty r = M r
         mul (ConsA a m) r | M m' <- mul m r = M (consA a m')
         mul (ConsB b m) r | M m' <- mul m r = M (consB b m')
         consA :: Monoid a ⇒ a → Alternate s a b → Alternate A a b
         consA a Empty = ConsA a Empty
         consA a (ConsA a' m) = ConsA (a ⊛ a') m
         consA a r@(ConsB _ _) = ConsA a r
         consB :: Monoid b ⇒ b → Alternate s a b → Alternate B a b
         consB b Empty = ConsB b Empty
         consB b (ConsB b' m) =  ConsB (b ⊛ b') m
         consB b r@(ConsA _ _) = ConsB b r
```

Fig. 9. The coproduct of monoids is a Monoid

```
instance Free Monoid x where
  newtype FreeA Monoid x = P [x] deriving (Monoid)
  pvar x = P [x]
  P [] `pbind` f = 𝟙
  P xs `pbind` f = foldr ((⊛) . f) 𝟙 xs
```

Fig. 10. Free instance for Monoid
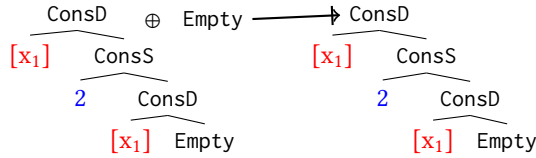
Fig. 11. Partially-static monoid: dropping 𝟙
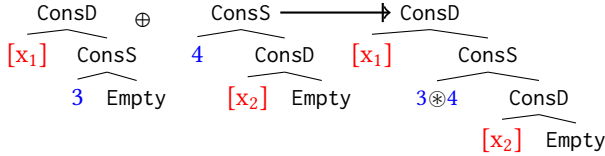
Fig. 12. Partially-static monoid: coalescing adjacent static values

4.2 Free Monoids and the Free Extension

Fig. 10 shows the Free instance for Monoid. 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:

$$(\text{pvar } x_1 \circledast \text{pvar } x_2 \circledast \ldots \circledast \text{pvar } x_n) \; `pbind` \; f \quad \leadsto \quad (f \; x_1 \circledast f \; x_2 \circledast \ldots \circledast f \; x_n)$$

Type class resolution combines the Free and Coproduct instances to form the free extension FreeExt.

```
class Format f where
  type Acc f a :: *
  lit :: String → f a a
  cat :: f b a → f c b → f c a
  int :: f a (Acc f Int → a)
  str :: f a (Acc f String → a)
  sprintf :: f (Acc f String) a → a
```

Fig. 13. Format signature

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

Fig. 14. An unstaged Format implementation

```
instance Format FmtS where
  type Acc FmtS a = Code a
  lit x = FmtS $ \k s →k ⟦ $s ++ x ⟧
  f `cat` g = FmtS (fmtS f . fmtS g)
  int = FmtS $ \k s x →k ⟦ $s ++ show $x ⟧
  str = FmtS $ \k s x →k ⟦ $s ++ $x ⟧
  sprintf p = fmtS p id ⟦ "" ⟧
```

Fig. 15. An staged Format implementation

```
instance Format FmtPS where
  type Acc FmtPS a = Code a
  lit x = FmtPS $ \k s →k (s ⊛ sta x)
  f `cat` g = FmtPS (fmtPS f . fmtPS g)
  int = FmtPS $ \k s x →k (s ⊛ dyn ⟦ show $x ⟧)
  str = FmtPS $ \k s x →k (s ⊛ dyn x)
  sprintf (FmtPS p) = p cd 𝟙
```

Fig. 16. A partially-static Format implementation

Figs. 11 and 12 illustrate how *frex*'s partially-static monoid performs static reductions using the monoid laws.

4.3 Example: Improving `printf` with Partially-Static Monoids

To show the Monoid free extension in action, we consider an example from the functional programming literature [Asai 2009; Danvy 1998]: typed sprintf. It is straightforward to use staging to turn sprintf from a function into a code generator [Yallop and White 2015]; 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:

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

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

$$⟦ \; λx \; y \; →(((("" \; ++ \; \textbf{show} \; x) \; ++ \; "a") \; ++ \; "b") \; ++ \; \textbf{show} \; y) \; ⟧$$

Since strings form a monoid under catenation, switching to *frex*'s partially-static operations generates the following more efficient code:

$$⟦ \; λx \; y \; →\textbf{show} \; x \; ++ \; ("ab" \; ++ \; \textbf{show} \; y) \; ⟧$$

(In fact, as we shall see, *Frex* can also generate the more efficient code that makes a single call to an *n*-ary catenation function).

Fig. 13 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 sprintf instantiation. The following three operations construct format strings: lit s is a format string that accepts no arguments and prints s; cat 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 Fig. 13 in continuation-passing style, using an accumulator:

```
newtype Fmt r a = Fmt {fmt :: (String → r) → String → a }
```

With this implementation, a format string `Fmt` is a function accepting a continuation argument of type `String` → r and an accumulator of type `String`. Both `lit` and `int` call k directly, passing an extended string; `cat` is simply function composition. The function `sprintf` passes the identity function as a top-level continuation along with an empty accumulator (Fig. 14).

Staging `sprintf` is straightforward (Fig. 15). We treat format strings statically; arguments and, consequently, the accumulator, are dynamic. The `cat` function is unchanged, and the rest of the implementation is annotated in accordance with the assignment of static and dynamic classifications:

> `newtype` FmtS r a = FmtS {fmtS :: (Code `String` → r) → (Code `String` → a)}

The generated code (shown at the beginning of this section) is sub-optimal because the staging is simplistic: every catenation is delayed, even when both operands are statically available.

Staging using *frex*'s partially-static monoid is also straightforward (Fig. 16). The steps are as follows, starting from the unstaged implementation: replace `String` with FreeExt Monoid `String`, replace cat 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:

> `newtype` FmtPS r a = FmtPS { fmtPS :: (PS Monoid `String` → r) → PS Monoid `String` → a }

This implementation statically constructs a canonical representation before residualizing, eliminating nesting and redundant catenations with 𝟙; the result (shown at the beginning of this section) contains only two catenations rather than the original four.

§4.4 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.

4.4 Residualization for Monoids

It is often possible to give a more efficient residualization function for a particular instance of a structure. Here we sketch how to give an alternative residualization function for the partially-static monoid of strings. There is no need to step outside the framework provided by *frex*; it is sufficient to instantiate the residualizing call to eva with an alternative Monoid instance.

The Monoid interface (Fig. 6) exposes nullary and binary constructors 𝟙 and ⊛. However, for some monoids it is more efficient to combine more than two elements in a single operation. The partially-static monoid structure generates the following code for the printf example in §4.3:

$$\llbracket\; s_1 \;{+\!+}\; \$d \;{+\!+}\; s_2 \;\rrbracket$$

However, depending on the representation of strings, it may be more efficient to generate a single call to an *n*-ary catenation function. (For example, *n*-ary catenation is more efficient in OCaml, where strings are strict arrays, but not in Haskell, where strings are lazy lists by default.)

$$\llbracket\; \texttt{concat}\; [s_1,\; \$d,\; s_2] \;\rrbracket$$

It is straightforward to write an alternative to cd specialized to the monoid of strings that generates this more efficient code.

The opportunity to improve code generation at the point of residualization is one of the advantages of the free extensions view over existing ad-hoc approaches to partially-static representations.

The interface to partially-static data in earlier work (e.g. Kaloper-Meršinjak and Yallop [2016]) typically provides cd as the only way to inspect partially-static data. The coproduct view presented here 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.

§6 shows how *frex*'s simplifications to the code generated by the staged printf function example lead to significant performance improvements.

```
                                                                    𝟙 ⊛ x ≡ x
class Monoid m ⇒ CMonoid m                                   x ⊛ (y ⊛ z)  ≡  (x ⊛ y) ⊛ z
class CMonoid c ⇒ CGroup c where cinv :: c → c                    x ⊛ y ≡ y ⊛ x
                                                                cinv x ⊛ x ≡ 𝟙
```

Fig. 17. Commutative monoids and abelian groups

```
instance (CMonoid a, CMonoid b) ⇒ Coproduct CMonoid a b where
  data Coprod CMonoid a b = C a b
  inl a = C a 𝟙
  inr b = C 𝟙 b
  eva f g (C a b) = f a ⊛ g b
```

Fig. 18. The coproduct of commutative monoids

Practical Considerations: Canonicity. Ideally the partially-static representation should be *canonical*: expressions that are statically equivalent under the laws of the algebra should have the same representation in PS.

Unfortunately, it is not always possible to achieve full canonicity; for example, ConsS can store empty monoid elements, even though these could be eliminated according to the unit elimination laws. This kind of deviation from canonicity is sometimes unavoidable, 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.

4.5 Coproducts of Commutative Monoids and Abelian Groups

Fig. 17 shows the interface to commutative monoids CMonoid and abelian groups CGroup. The CMonoid class inherits the methods of Monoid and adds a commutativity law. The only difference between Monoid and CMonoid is the set of laws tacitly associated with the class.

However, adding the commutativity law to the monoid interface leads to quite a different coproduct structure (Fig. 18). As with the transition from semigroups to commutative semigroups (§2), applying commutativity to the alternating sequence structure of the monoid coproduct allows the elements of each constituent monoid to be brought together and coalesced. The alternating sequence consequently collapses into a two-element sequence — i.e. a Cartesian product of sets. (The coproduct of abelian groups is isomorphic to the commutative monoid coproduct, and not shown.)

4.6 Free Commutative Monoid and Abelian Group

Fig. 19 shows implementations of free algebras for commutative monoids and abelian groups.

The CMonoid and CGroup free algebras are multisets (bags) and finite maps (dictionaries) with integer values, respectively:

```
pvar x ⊛ cinv (pvar y) ⊛ pvar x ⊛ cinv (pvar y)    ⤳    { x ↦ 2, y ↦ -2 }
```

In each case the implementation of pbind maps f over each element in the representation and combines the results using mappend; the pbind implementation for abelian groups additionally applies cinv to each output element when the count associated with the element is negative:

```
(pvar x ⊛ cinv (pvar y) ⊛ pvar x ⊛ cinv (pvar y)) `pbind` f
     ⤳   f x ⊛ f x ⊛ cinv (f y) ⊛ cinv (f y)
```

```
instance Ord x ⇒ Free CMonoid x where
   newtype FreeA CMonoid x = CM (MultiSet x)
   pvar = CM . singleton
   CM b `pbind` f = emit (toList b)
     where emit [] = mempty
           emit [x] = f x
           emit (x:xs) = f x `mappend` emit xs

instance Ord x ⇒ Free CGroup x where
   newtype FreeA CGroup x = CG (Map x Int)
   pvar x = CG (singleton x 1)
   CG b `pbind` f = emit (toList b)
     where emit [] = mempty
           emit [(x,1)] = f x
           emit ((x,0):xs) = emit xs
           emit ((x,n):xs) | n < 0 = cinv (f x) `mappend` emit ((x,n+1):xs)
                           | otherwise = f x `mappend` emit ((x,n-1):xs)
```

Fig. 19. Free instances for commutative monoids and abelian groups

As for monoids, the Free and Coproduct instances combine to form the free extension.

4.7   Example: power with Partially-Static Commutative Monoids

We are ready to revisit the staged power function from the introduction. The naively-staged power function generates code with a linear sequence of multiplications that concludes with an unnecessary multiplication by 1:

$$\llbracket \ \backslash x \rightarrow \$(\text{power } \llbracket x \rrbracket \ 6) \ \rrbracket \quad \leadsto \quad \llbracket x * (x * (x * (x * (x * (x * 1))))) \ \rrbracket$$

Here is an implementation of power suitable for use with *frex*:

```
power :: CMonoid m ⇒ m → Int → m
power x 0 = 𝟙
power x n = x ⊛ power x (n - 1)
```

This new definition illustrates two benefits of *frex*. First, there are no more low-level staging annotations; it is sufficient to make the code more polymorphic by defining power for an arbitrary commutative monoid, then instantiate with *frex*'s predefined free extension. Second, *frex*'s simplifications improve the generated code. With a cd function specialized to commutative monoids, in the same vein as the specialized monoid cd of §4.4, the six multiplications in the naively-staged version can be reduced to three:

$$\text{cdPower (power (dyn } \llbracket x \rrbracket) \ 6) \leadsto \llbracket \textbf{let } y = x*x \textbf{ in let } z = y*y \textbf{ in } y*z \ \rrbracket$$

As in the previous example, these improvements depend only on a specialized cd; the implementation of power and the partially-static representations are untouched. The specialized cd can be implemented in terms of eva by supplying a suitable CMonoid instance, or directly on the CoProd representation.

4.8   Coproduct of Sets

One special case of an algebraic structure is Set, the structure with no equations.

```
                                          instance Coproduct Set a b where
                                            data Coprod Set a b = Inl a | Inr b
    class Set a                             inl = Inl
    instance Set a                          inr = Inr
                                            eva f g (Inl x) = f x
                                            eva f g (Inr y) = g y
```

Fig. 20. The coproduct of structures with no equations

```
                        instance Free Set x where
                          newtype FreeA Set x = F x
                          pvar = F
                          F x `pbind` k = k x
```

Fig. 21. Free instance for Set

Fig. 20 shows the instance for Coproduct in the category of sets. The Set class has no constraints or methods, and a single instance that encompasses every Haskell type. Consequently, the Coproduct instance for Set has no constraints, since all the Coproduct class constraints are satisfied by the single Set instance. The associated Coprod type is simply the familiar type of binary sums (called **Either** in the Haskell standard library), with inl and inr as its two constructors, and eva corresponding to the familiar **either** function[1].

4.9 Free Algebra for Sets

Fig. 21 shows the free algebra for sets, where a value is a single variable, and pbind is application.

Although the Set free extension is a rather impoverished structure, it is perhaps the most frequently used representation for partially-static data in the multi-stage programming literature. The Set free extension does not take advantage of any equations; nevertheless, switching from a binary binding-time classification, in which each expression is fixed as always-static or always-dynamic, to the *possibly-static* world of the Set free extension, in which an expression may switch between static and dynamic on different executions, is a significant improvement for many applications.

4.10 Example: Possibly-Static Data

Here is a simple example of possibly-static data. The **isDigit**$_{PS}$ function classifies a possibly-static character using the standard **isDigit** function. If the character is static then the classification is performed immediately; if it is dynamic then the classification is deferred to the next stage.

```
            isDigitPS :: FreeExt Set Char →FreeExt Set Bool
            isDigitPS (Inl c) = Inl (Char.isDigit c)
            isDigitPS (Inr c) = Inr ⟦Char.isDigit $c ⟧
```

Without partially-static data it would be necessary either to have two functions for the static and dynamic cases, or to convert every character to a dynamic value, losing the opportunity to perform further static computation on the result.

The idea naturally generalizes to lift arbitrary functions to operate on possibly-static values.

4.11 Free Algebra and Free Extension of Commutative Rings

Fig. 22 shows the interface Ring and the axioms for commutative rings.

---

[1]https://hackage.haskell.org/package/base-4.10.0.0/docs/Data-Either.html

```
                                             (a ⊕ b) ⊕ c  ≡  a ⊕ (b ⊕ c)
        class Ring a where                   a ⊕ b  ≡  b ⊕ a      a ⊕ 𝟘  ≡  a
           (⊕), (⊗) :: a → a → a             a ⊕ rneg a  ≡  𝟘
           rneg :: a → a                     (a ⊗ b) ⊗ c  ≡  a ⊗ (b ⊗ c)
           𝟘, 𝟙 :: a                         a ⊗ b  ≡  b ⊗ a      a ⊗ 𝟙  ≡  a
                                             a ⊗ (b ⊕ c)  ≡  (a ⊗ b) ⊕ (a ⊗ c)
```

Fig. 22. Commutative rings

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

instance Ord x  ⇒  Free Ring x where
    newtype FreeA Ring x = RingA (Multinomial x Int) deriving (Ring)
    pvar x = RingA (MN (Map.singleton (MultiSet.singleton x) 𝟙))
    RingA xss `pbind` f = evalMN initMN f xss
```

Fig. 23. Free commutative rings

```
instance (Ring a, Ord x)  ⇒  Coproduct Ring a (FreeA Ring x) where
    newtype 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 (CR c) = evalMN f (g . pvar) c
```

Fig. 24. The coproduct of a commutative ring and a free commutative ring

Fig. 23 defines the free commutative ring on a set x, defined as multinomials — i.e finite sums of products of variables in x with integer coefficients. The call pvar x builds a representation of the trivial polynomial x, whose exponent and coefficient are both 𝟙. The call evalMN initMN f in the definition of pbind interprets both the variables and the coefficients in an arbitrary ring, and combines the results using the ring operators. The extended version of the paper gives further details.

It is not always possible to give a closed form for the coproduct of general algebras. However, we can still sometimes define the free extension, as is the case with commutative rings. The coproduct of a commutative ring A with a free commutative ring consists of multinomials with coefficients in A (Fig. 24).

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 ⊕... ⊕𝟙) (n times, or using ⊖ for negative n). As with the pbind operation for the free commutative ring, eva evaluates the multinomial using the ring operations for addition and multiplication:

$$\text{eva f g } (a + bx^2y) \quad \leadsto \quad \text{f a} \oplus (\text{f b} \otimes \text{g x} \otimes \text{g x} \otimes \text{g y})$$

A free commutative *semi*ring — i.e. an algebra for the operations and axioms of a ring except those involving ⊖ — is the same but with natural number coefficients instead of integers, and the free extension is defined analogously.

4.12 Example: Linear Algebra with Partially-Static Rings

Linear algebra offers many opportunities for optimization via multi-stage specialization and numerical simplification such as the Fast Fourier Transform [Kiselyov et al. 2004], Gaussian elimination

```
dot  [sta 1, sta 0, sta 2, dyn ⟦x⟧]  [dyn ⟦x⟧, dyn ⟦y⟧, dyn ⟦z⟧, dyn ⟦x⟧]
⤳ (sta 1 ⊗ dyn ⟦x⟧) ⊕ (sta 0 ⊗ dyn ⟦y⟧) ⊕ (sta 2 ⊗ dyn ⟦z⟧) ⊕ (dyn ⟦x⟧ ⊗ dyn ⟦x⟧)
⤳ 1x¹ + 2z¹ + x² ⤳ x + 2*z + x*x
```

Fig. 25. Reducing partially-static commutative ring expressions

[Carette and Kiselyov 2011a], and matrix-vector multiplication [Aktemur et al. 2013]. The inner product illustrates the general 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:

$$⟦ (1 * x) + (0 * y) + (2 * z) ⟧$$

There are clear opportunities for improvement: the multiplication by one in the first summand can be omitted, and the multiplication by zero should annihilate the middle summand altogether.

As with power (§4.7), switching from hand-inserted staging annotations to *frex*'s high-level approach means that dot can be written as a polymorphic function with no mention of staging:

```
dot :: Ring r ⇒ [r] → [r] → r              sumr :: Ring r ⇒ [r] → r
dot xs ys = sumr (zipWith (⊗) xs ys)       sumr = foldr (⊕) 𝟘
```

Fig. 25 illustrates the behaviour of dot when the Ring constraint is instantiated to *frex*'s free extension instance: the call to dot constructs multinomials that residualize to multiplications and additions. Returning to the example at the beginning of this section, *frex*'s ring simplifications lead to the following simpler code:

$$⟦ x + (2 * z) ⟧$$

§6 considers further examples involving linear algebra, and shows that *frex*'s simplifications lead to significant performance improvements.

4.13 Coproduct of Distributive Lattices

A distributive lattice is a commutative semiring $(A, 𝟘, ⊕, 𝟙, ⊗)$ with additional absorption rules:

$$a ⊗ (a ⊕ b) ≡ a \qquad a ⊕ (a ⊗ b) ≡ a .$$

For example, booleans form a distributive lattice with && 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 multinomials 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 Fig. 24 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.

4.14 Example: all and any with Partially-Static Distributive Lattices

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

The **all** function takes a predicate p and a list l, and returns true iff every element of l satisfies p. *Frex* 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:

```
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) (Coprod Set a b))
  inl x = L (fmap Inl x)
  inr y = L (fmap Inr y)
  eva g h (L e) = e `pbind` eva (g . pvar) (h . pvar)
```

Fig. 26. The coproduct of two free F-algebras

$$\text{all}_{PS} \ \textbf{even} \ (\text{sta} \ [2, \ 4] \ ++ \ \text{var} \ [\![x]\!] \ ++ \ \text{sta} \ [3])$$
$$\rightsquigarrow \ \text{sta} \ (\textbf{even} \ 2) \circledast \text{sta} \ (\textbf{even} \ 4) \circledast \text{dyn} \ [\![\ \textbf{all even} \ x \ ]\!] \circledast \text{sta} \ (\textbf{even} \ 3)$$
$$\rightsquigarrow \ \text{sta false}$$

The all_{PS} function operates on *frex*'s representations, turning a partially-static monoid (the input list) into a partially-static distributive lattice (the output boolean). The distributive lattice laws, used in *frex*'s free extension instance, reduce the expression to the value false, even though one element in the sequence is dynamic. The dual function any_{PS} can be defined similarly.

4.15 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 [Jones et al. 1993; Kaloper-Meršinjak and Yallop 2016; Sheard and Diatchki 2002] that is discussed further in §7.

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. Fig. 26 shows this coproduct for 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

$$\text{IList X} := \mathbb{1} + \textbf{Int} \times \text{X}$$

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

4.16 Example: 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].

The free extension for F-algebras in *frex* can be used to define a variant of **sum** that operates on partially-static lists by traversing the initial portion of a list, leaving traversal of the dynamic tail for later:

$$\text{cd} \ (\textbf{sum}_{PS} \ (1 :_s 2 :_s 3 :_s \text{dyn} \ [\![t]\!])) \rightsquigarrow [\![6 + \textbf{sum} \ t]\!]$$

4.17 Practical Considerations: Duplicating and Discarding 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, duplication and discarding are even more crucial to avoid. However, in some of our examples, such as power and dot, quoted expressions injected with dyn may appear either several times or not at all in the output of 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 **let**-bound [Carette and Kiselyov 2011b; Kiselyov 2014; Yallop 2017] using a function (commonly named genlet) that accepts a dynamic expression e, inserts a **let**-binding for e at some higher point in the code, and returns the bound variable:

$$\text{g (genlet } [\![\ \mathsf{f}\ \mathsf{x}\ ]\!]) \ \leadsto\ [\![\mathbf{let}\ \mathsf{y}\ =\ \mathsf{f}\ \mathsf{x}\ \mathbf{in}\ \ldots\ \$(\mathsf{g}\ [\![\mathsf{y}]\!])]\!]$$

Automatic conversion to ANF form in the LMS multi-stage programming framework [Rompf 2016] serves a similar purpose.

let-insertion combines straightforwardly with partially-static data; however, we have omitted it from the exposition for simplicity. The MetaOCaml implementation described in the extended version of the paper uses **let**-insertion to avoid duplication.

5 UNIVERSALITY: FREE EXTENSION OF ALGEBRAS

§3 and the examples in §4 show that free extensions, i.e. coproducts with free algebras, provide a natural representation for partially-static data structures. Here we justify more formally why this representation is valid in terms of the universal property of free extensions. We first recall some basic universal algebra, which allows us to discuss classes of algebraic structures uniformly.

5.1 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: consisting of a set O_Σ of *operation symbols* where each symbol is assigned a natural number called its *arity*. For example, monoids use the signature where $O_{\mathbf{mon}} := \{\mathbb{1}, \circledast\}$, $\mathbb{1}$ has arity 0, and $\circledast$ has arity 2. Given a signature Σ, the functional specification is given by a set of *axioms*: equations between terms built from the operation symbols in Σ and according to their corresponding arities. For example, the monoid axioms $\text{Ax}_{\mathbf{mon}}$ are given in Fig. 6.

Put together, the description of an algebraic structure is called a *presentation* $\mathcal{P}$, given by a signature $\Sigma_{\mathcal{P}}$ and a set $\text{Ax}_{\mathcal{P}}$ of axioms over this signature. The example signature and axioms above form $\mathbf{mon}$ — the presentation of monoids.

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 of arity n in $\Sigma_{\mathcal{P}}$, an n-ary function $f_A : |A|^n \to |A|$, such that all the axioms in $\text{Ax}_{\mathcal{P}}$ hold. For example, noting that a nullary function is a constant, a $\mathbf{mon}$-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 $\mathbf{mon}$-homomorphism $h : A \to B$ is a function that satisfies $h(\mathbb{1}_A) = \mathbb{1}_B$ and $h(x \circledast_A y) = h(x) \circledast_B h(y)$, i.e. the familiar notion of a monoid homomorphism.

For each presentation $\mathcal{P}$, the collection of $\mathcal{P}$-algebras and $\mathcal{P}$-homomorphisms 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 $F_{\mathcal{P}} : \mathbf{Set} \to \mathcal{P}\text{-Alg}$, the *free $\mathcal{P}$-algebra functor*. 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 described in §4.2. 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 $\ggg=^{\mathcal{P}} f : F_{\mathcal{P}}X \to A$, 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 integers with multiplication:

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

The categories $\mathcal{P}\text{-Alg}$ have coproducts $A \oplus B$, and their concrete structure is given as follows. The carrier $|A \oplus B|$ is the $\Sigma_{\mathcal{P}}$-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 $f(\iota_1 a_1, \ldots, \iota_1 a_n) \equiv \iota_1 f_A(a_1, \ldots, a_n)$ for every f of arity 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 $h_i(x)$. For example the coproduct of monoids A and B has as its carrier the set of sequences of alternating elements of $|A|$ and $|B|$.

A *free extension* of an algebra A by a set X is the coproduct of the algebra A with the free algebra over X, namely $\text{FreeExt}(A, X) := A \oplus F_{\mathcal{P}}X$. The coproduct injection $\iota : F_{\mathcal{P}}X \to \text{FreeExt}(A, X)$ corresponds under the adjunction to a function $\iota_X : X \to |\text{FreeExt}(A, X)|$. So combining the universal properties of coproducts and adjunctions, a free extension is characterised by an algebra $\text{FreeExt}(A, X)$ together with a homomorphism $\iota_A : A \to \text{FreeExt}(A, X)$, and a function $\iota_X : X \to |\text{FreeExt}(A, X)|$, 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 $\text{eva}(h, e) : \text{FreeExt}(A, X) \to C$ satisfying

$$\text{eva}(h, e) \circ \iota_A = h \qquad |\text{eva}(h, e)| \circ \iota_X = e.$$

5.2 Conceptual Justification

Now suppose the algebra A stands for a static datatype, and the set X stands for a collection of dynamically-known values. §3 sets out the minimum requirements we would want for the corresponding partially-static datatype: an algebra $\text{ps}(A, X)$ together with inclusions

$$\text{sta} : A \to \text{ps}(A, X) \qquad \text{dyn} : X \to \text{ps}(A, X)$$

such that sta is an algebra homomorphism, and a residualization map

$$\text{cd} : \text{ps}(A, X) \to X$$

satisfying $\text{cd} \circ \text{dyn} = \text{id}$ and $\text{cd} \circ \text{sta} = \text{lift}$ where lift is the function $A \to X$ lifting static values to dynamic ones.

Certainly the free extension $\text{FreeExt}(A, X)$ meets these requirements when X is the algebra `Code A`, defining

$$\text{sta} := \iota_A \qquad \text{dyn} := \iota_X \qquad \text{cd} := \text{eva}(\text{lift}, \text{id})$$

as described in 3.4. With just the requirements above it is not the only possible choice. However, there are at least two ways in which we could impose reasonable extra conditions on the partially-static datatype which would only be satisfied by the free extension.

Firstly, we could ask that in addition to residualization partially-static data should allow post-processing. As $ps(A, X)$ is an algebra, we can consider homomorphisms from $ps(A, X)$ into other algebras C. It is natural to expect that a homomorphism $h : A \to C$ and a function $e : X \to |C|$ should lift to a homomorphism $eva(h, e) : ps(A, X) \to C$ acting as expected on purely static and dynamic values. The homomorphism should be unique for a minimal representation of the data. This corresponds exactly to the characterization of $FreeExt(A, X)$ above.

Alternatively, we could impose a uniformity condition on the collection of partially-static datatypes for *all* algebras A. The datatype should store representations of static elements in a uniform way so that homomorphisms of static data lift to homomorphisms of partially-static datatypes. So assume that given a set X,

- partially-static datatypes with sta and dyn exist for every algebra A,
- for any instantiation $e : X \to A$ of variables in X as elements of A, there is a unique homomorphism $eva(id, e) : ps(A, X) \to A$ extending e and preserving static values,
- for every pair of algebras A, B and homomorphism $h : A \to B$, there is a unique homomorphism $ps(h, X) : ps(A, X) \to ps(B, X)$ which acts as h on static values and leaves dynamic values unchanged.

Then an algebraic argument shows that $ps(A, X)$ together with sta, dyn and $eva(h, e) := eva(id, e) \circ ps(h, X)$ has the universal property of the free extension $FreeExt(A, X)$.

6 PERFORMANCE EVALUATION

The central contribution of this paper is a unification of various existing optimizations based around partially-static data, although many of the structures and several of the examples in our study are novel. Our focus up to this point has been on a rationalised interface to partially-static data, on the representations for particular algebraic structures, and on the simplifications that they produce in code generated by multi-stage programs. It is reassuring to discover that the simplifications introduced by *frex* lead directly to improved performance over both the original unstaged program and naively staged versions that make no use of partially-static data.

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

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 using the `-O3` optimization flag, and OCaml code with BER MetaOCaml n104.

6.1 Matrix Multiplication

Fig. 27 shows the performance of four implementations of 10x10 matrix multiplication in Haskell. The naive implementation is a one-line function based on a list-of-lists representation of matrices:

```
mmmul m n = [[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 function above with appropriate instances. The naive staging unrolls the loop, turning the list traversal into an arithmetic expression. The partially-static version takes a static and a dynamic input vector, and converts both to lists of lists before passing them to `mmul`:

```
mmmul [ [sta s₁, sta s₂, ...]..] [[dyn ⟦d!0!0⟧, dyn ⟦d!0!1⟧ ...]..]
```

Instantiating `mmul` with *frex*'s free extension instance results in automatic algebraic simplification.

The graph shows measurements for various sparsities (i.e. for matrices with various proportions of zero elements). As the sparsity of the matrix increases, the algebraic simplifications performed by the partially-static version significantly increase its advantage over naive staging.
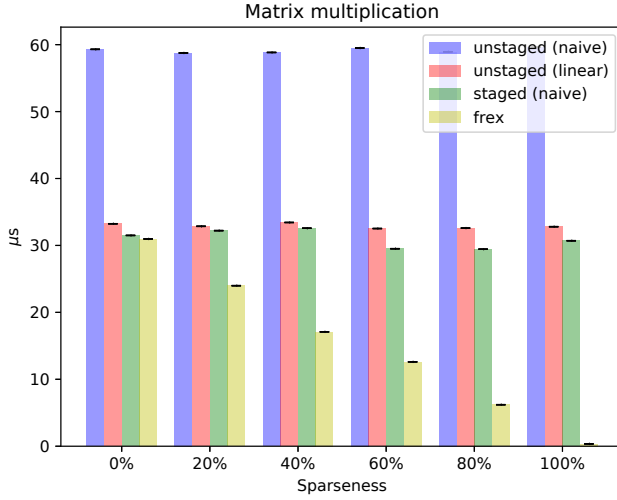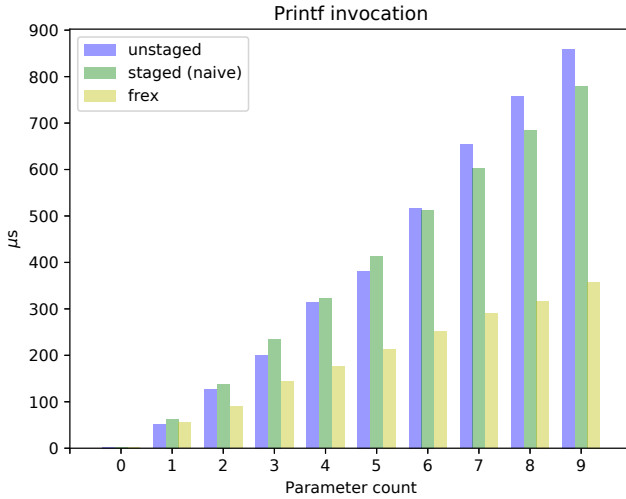
Fig. 27. Matrix multiplication improvements

Fig. 28. Printf performance improvements

6.2 Printf

Fig. 28 shows the performance of three implementations of the printf function in MetaOCaml — a standard unstaged version, a naively staged version, and a partially-static version that uses *frex*'s free extension for monoids with improved residualization (§4.4).

In each case the benchmark measures the time to run a call to printf with the given number of parameters and a format string that parenthesizes each parameter in the output:

```
sprintf ((lit "(" ++ str ++ lit ")") ++ (lit "(" ++ str ++ lit ")"))
```

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 and the number of adjacent static strings to merge grows.

7 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 [Jones et al. 1993] informally describes how to generalize partially-static list representations to arbitrary recursive types. Sheard and Diatchki [2002] describe a similar, but more concrete, scheme for deriving the staged versions of particular datatypes in the multi-stage programming language MetaML [Taha and Sheard 1997]. Kaloper-Meršinjak and Yallop [2016] turn 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 an 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 [2013] considers *partially-static operations*, which incorporate algebraic laws, in a partial evaluation context. Thiemann's vision of specialization that considers algebraic structure is an inspiration for this work. However, the implementation is quite different: Thiemann's design operates via repeated rewriting, whereas the structures described in the present work are reduced using the evaluation mechanism of the host language. It appears unlikely that this approach to rewriting is suitable for direct use in multi-stage programming.

Partially-static data in partial evaluation. Partially-static data has been used in partial evaluation from early times. Mogensen [1988] introduced the concept in a study of partially-static lists, and several authors followed suit (e.g. [Hughes 1999; Jones et al. 1993]). Sheard and Diatchki [2002] note that the term acquired the more specific meaning of static containers with dynamic elements.

Glück et al. [1996] describe the application of a partial evaluator for Fortran with support for partially-static structures to a variety of mathematical algorithms including cubic splines interpolation, Romberg integration and Chebyshev approximation.

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. [2009] uses 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. Similar unit simplifications for vectors are implemented as smart constructors in Rompf et al.'s [2013] staging-based compiler optimization framework.

Inspired by abstract interpretation, [Kiselyov et al. 2004] build a staged FFT implementation that uses partially-static representations that distinguish values from computations and that support simplification using various laws, including the distributive property, and trigonometric identities.

Carette and Kiselyov [2005, 2011b] describe a modular decomposition of Gaussian Elimination that abstracts over staged and unstaged implementations of numeric signatures and other aspects of the algorithm. Partially-static data is used pervasively, primarily in the form of static-dynamic values.

Combining the techniques in this paper with Carette and Kiselyov's [2005] modular approach by instantiating numeric signatures with free extensions is a promising avenue for future exploration.

Yallop [2017] uses 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 similarly to the free extension presented here.

Drawing lessons from supercompilation, Inoue [2014] uses 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.

8 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 (§7). 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 free extensions for partially-static data is evident both from the simplified generated code, and from benchmarks (§6).

In the future we would like to use free extensions of free theories to partially evaluate code using effect handlers [Bauer and Pretnar 2015]. We would also like to investigate the extension of this work to settings with more than two stages, where partially-static structures have already been successfully applied [Glück and Jørgensen 1997].

ACKNOWLEDGMENTS

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.

Jacques Carette and Oleg Kiselyov. 2005. Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code. In *Generative Programming and Component Engineering: 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005. Proceedings*, Robert Glück and Michael Lowry (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 256–274. https://doi.org/10.1007/11561347_18

Jacques Carette and Oleg Kiselyov. 2011a. 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

Jacques Carette and Oleg Kiselyov. 2011b. Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code. *Sci. Comput. Program.* 76, 5 (May 2011), 349–375. https://doi.org/10.1016/j.scico.2008.09.008

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (Sept. 2009), 509–543.

Manuel M. T. Chakravarty, Gabriele Keller, Simon L. Peyton Jones, and Simon Marlow. 2005. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005.* 1–13. https://doi.org/10.1145/1040305.1040306

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

Robert Glück and Jesper Jørgensen. 1997. An Automatic Program Generator for Multi-Level Specialization. *LISP and Symbolic Computation* 10, 2 (01 Jul 1997), 113–158. https://doi.org/10.1023/A:1007763000430

Robert Glück, Ryo Nakashige, and Robert Zöchling. 1996. Binding-time analysis applied to mathematical algorithms. In *System Modelling and Optimization: Proceedings of the Seventeenth IFIP TC7 Conference on System Modelling and Optimization, 1995*, Jaroslav Doležal and Jiří Fidler (Eds.). Springer US, Boston, MA, 137–146. https://doi.org/10.1007/978-0-387-34897-1_14

John Hughes. 1999. *A Type Specialisation Tutorial.* Springer Berlin Heidelberg, Berlin, Heidelberg, 293–325. https://doi.org/10.1007/3-540-47018-2_12

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 (Lecture Notes in Computer Science)*, Michael Codish and Eijiro Sumii (Eds.), Vol. 8475. Springer International Publishing, 86–102.

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.).

Simon Peyton Jones. 2016. Template Haskell, 14 years on. Talk given at the International Summer School on Metaprogramming, Cambridge, UK. (August 2016). https://www.cl.cam.ac.uk/events/metaprog2016/Template-Haskell-Aug16.pptx

Tiark Rompf. 2016. *The Essence of Multi-stage Evaluation in LMS.* Springer International Publishing, Cham, 318–335. https://doi.org/10.1007/978-3-319-30936-1_17

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing Data Structures in High-level Programs: New Directions for Extensible Compilers Based on Staging. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 497–510. https://doi.org/10.1145/2429069.2429128

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

Walid Taha and Tim Sheard. 1997. Multi-stage Programming with Explicit Annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '97)*. ACM, New York, NY, USA, 203–217. https://doi.org/10.1145/258993.259019

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), 29:1–29:29 pages.
https://doi.org/10.1145/3110273
Jeremy Yallop and Leo White. 2015. Modular Macros. (September 2015). OCaml Users and Developers Workshop 2015.