A Pattern for Almost Homomorphic Functions

Nicolas Frisby  Andy Gill  Perry Alexander
Information Telecommunications and Technology Center
The University of Kansas
{nfrisby,andygill,alex}@ittc.ku.edu

Abstract
Modern type systems present the programmer with a trade-off between correctness and code complexity—more precise, or exact, types that allow only legal values prevent runtime errors while less precise types enable more reuse. Unfortunately, the software engineering benefits of reuse and avoiding duplicate code currently outweigh assurance gains of exact types. We factor out a pattern common in conversions that result from using exact types as a reusable function, extending existing generic programming techniques to avoid code duplication and enable reuse.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming

General Terms Algorithms, Design

Keywords Data types; Generic Programming; Type Invariants; Type-Level Programming

1. Introduction
In richly typed languages, programmers can declare types that precisely capture their intended values and nothing more, statically guaranteeing the absence of many runtime errors. Using such precise or exact types is ideal from an assurance point-of-view (c.f. adequacy [6]). In practice such types can have serious maintenance and reuse disadvantages for certain classes of code.

As an example, consider an AST used in a compiler. Some passes cannot handle certain constructors, but for such a pass to use an exact type requires a new type definition. Each exact type is necessarily specific and therefore useful at only a few points in the compiler. Though the number of these types is not itself detrimental, multiple functions for converting between them must be maintained that incur substantial code duplication. When maintaining such code—extending the AST or revising existing features—each of these functions must be accounted for.

As a result, developers of large programs choose code simplicity over exact types. For example, the source of the Glasgow Haskell Compiler (GHC) includes many functions that immediately raise fatal runtime errors for some of their domain type’s constructors. The GHC implementers document these function’s preconditions using type synonyms of the inexact types, indicative constructor names, and comments that explain what constructors are expected as input and output. Specialized exact types and code duplication are avoided, but at the cost of assurance—a fair engineering choice.

In this paper, we introduce novel extensions of existing datatype-generic programming techniques that allow developers to use exact types without duplicating code or sacrificing reusability. Our approach takes advantage of a common pattern found frequently in the definition of property-establishing functions that transform a domain type into a more exact codomain type. In the aforementioned AST example, property-establishing functions transform the AST into an exact type for processing by a downstream pass, protecting it from invalid inputs. This makes such functions crucial when using exact types. We work from the premise that the majority of cases in pragmatic property-establishing functions are homomorphic; that is they structurally recur to map between constructors with compatible fields and corresponding semantics. The explicit use of homomorphism in these cases is the pattern we encapsulate. This premise is legitimate because it characterizes the common practice of defining complex programs, such as compilers [17], as a sequence of minimal transformations. In this context, the exact types approach places exact data types between the transformations that become property-establishing functions.

The homomorphic pattern, when used to define a function with the same domain and codomain type, is well understood and can be factored out with existing techniques [2, 24]. This approach, however, only works for compositional functions, homomorphisms with equal domain and codomain. It cannot be used to define property-establishing functions because their domain and codomain are unequal by definition. We generalize the \texttt{hcompos} function of Bringert and Ranta [2] to the new \texttt{hcompos} that enriches their approach for “almost compositional” functions to support the necessarily heterogeneous domain and codomain types of “almost homomorphic” functions.

We make the following specific contributions.

- We introduce the delayed representation of a data type’s structure, in which constructors are not just products. [8] and [5]
- We introduce constructor name reflection to augment the type-level reflection of data types, thus enabling support of homomorphism. [5]
- We generically define the \texttt{hcompos} operator for almost homomorphic functions using our generic techniques. [5]
- We demonstrate by using \texttt{hcompos} to define lambda-lifting as a property-establishing function with a codomain that admits only top-level function declarations. [7]
- We enrich our generic programming techniques and the definition of \texttt{hcompos} to support more complex data types. [8]

We assume the reader is familiar with Haskell and recent extensions, such as type families [18] and promotion [25].
## 2. Exact Types Defy Existing Generic Techniques

We motivate the technical developments required to generically define our new function `hcompos` by demonstrating its utility on a simple example. The same example function will be defined three ways: first, with exact types and explicit homomorphic cases; second, with implicit homomorphic cases but without exact types; and third, with both exact types and implicit homomorphism via our new function `hcompos`.

The `exact_nnf` function declared in `Figure 1` transforms the `Exp` data type into the more exact `NNF` data type; thereby establishing a negation normal form property where only variables are negated. While the unequal domain and codomain types of `exact_nnf` are essential for the benefits of exact types, they also prevent the use of existing methods for handling tedious cases implicitly. Since `Plus` is not essential to the negation normal property, it would ideally not be present in the definition of `exact_nnf`. Its case simply implements the obvious structural recursion. The only reason we cannot omit this case is because the Haskell language and existing generic programming techniques (19, 20, 21) are blind to the correspondence between the `Plus` and `PlusN` constructors.

Defining the `Plus` case of `exact_nnf` explicitly might not itself seem burdensome. But for large programs, there are multiple properties to be encoded as data types like `NNF`. For example, lambda-lifting establishes the absence of lambda just as `exact_nnf` guarantees only variables are negated. Each property requires both an exact type and a function that establishes the property by converting to that type. To make exact types a workable option in large programs, the definitions of both the exact type and establishing functions must be concise and modular. Non-trivial functions over a data type tend to be more complicated than the data type’s declaration, thus we address these functions directly. Moreover, large data types have dozens of constructors, so each property-establishing function requires dozens of explicit, tedious cases. We generalize `compos` to `hcompos` in order to implement these cases generically.

Existing generic programming techniques cannot automate the `exact_nnf` function; they cannot relate its domain and codomain. The `modular_nnf` function in `Figure 1` has an inexactly typed version of `exact_nnf`. It handles the `Plus` case implicitly via the `compos` operator [2] for such “almost compositional” functions.

```
class Compos a where -- from [2]
  compos :: Applicative i => (a -> i a) -> a -> i a

import Data.Yoko (exact_case, hcompos, (.|.), yokoTH)

yokoTH ''Exp; yokoTH ''NNF -- Template Haskell

best_nnf :: Exp -> NNF
best_nnf = exact_nnf .|.

main = putStrLn $ (λ (Var s) → VarN s) 1

Figure 1. The codomain of `exact_nnf` encodes negation normal form, which the inexact type of `modular_nnf` trades for modularity.
```

Only the essential, non-homomorphic cases of a transformation need be explicit. For negation normal form, this is just variables and negation—not addition.

The delayed representation extension automatically generates a data type for each constructor in the original data type. This generated data type has one constructor, and it has the same fields as the original. Both the type and the constructor are predictably named; we have chosen the arbitrary convention to add an underscore. Thus the patterns of the `Var` and `Neg` alternatives in `best_nnf` are exhaustive. These generated data types are crucial to letting the programmer partition data types into subsets of constructors.

```
import Data.Yoko (exact_case, hcompos, (.|.), yokoTH)

yokoTH ''Exp; yokoTH ''NNF -- Template Haskell

best_nnf :: Exp -> NNF
best_nnf = exact_nnf .|.

main = putStrLn $ (λ (Var s) → VarN s) 1

Figure 2. The preferable definition enabled by our `yokoTH` library.
```

```
Figure 2. The preferable definition enabled by our `yokoTH` library.
```

```
```

## 3. Background: instant-generics

This section summarizes the instant-generics approach as currently available on Hackage, the foundation of our generic programming. It is expressive [13] and performant [9, §5]. Existing generic programming techniques are sufficient for generically defining the `compos` function of Bringert and Ranta [22]. These same techniques convey the majority of the reusability of our `hcompos` function; our extensions just enable its heterogeneity. We believe that our extensions can also be applied to more expressive variants of instant-generics, including generic-deriving and indexed data types (a subset of GADTs) [11].

The instant-generics framework derives its genericity from two major Haskell language features: type classes and type families [13]. We demonstrate it with a simple example type and two generically defined functions in order to set the stage for our extensions.
```
-- set of representation types
data Dep a = Dep a  data Rec a = Rec a
data U = U  data a :+: b = a :+: b
data C c a = C a  data a :+: b = L a | R b

-- maps a type to its sum-of-products structure
type family Rep a
class Generic a where
to :: Rep a → a
from :: a → Rep a

-- further reflection of constructors
class Constructor c where
conName :: C c a → String

data Var; data Lam; data App
instance Constructor Var where conName _ = "Var"
instance Constructor Lam where conName _ = "Lam"
instance Constructor App where conName _ = "App"

The void Var, Lam, and App types are considered auxiliary in instant-generics. They were later added (seemingly by Yaku- shev et al. [24]) to the sum-of-products representation types only to define another class of generic values, such as show and read. We call these types constructor types. They are analogous to a primary component of our delayed representation extension, and so will be referenced in Section 4.2. Each constructor type corresponds directly to a constructor from the represented data type.

The Var constructor's field is represented with Dep, since Int is not a recursive occurrence. The ULC occurrence in Lam and the two in App are recursive, and so are represented with Rec. The entire ULC type is represented as the sum of its constructors' representations—the products of fields—with some further reflective information provided by the C annotation. The definitions of from and to in the Generic instance for ULC are almost entirely determined by their types.

instance Generic ULC where
from (Var n) = L (C (Dep n))
from (Lam e) = R (L (C (Rec e)))
from (App e1 e2) = R (R (C (Rec e1 :+: Rec e2)))
to (L (C (Dep n))) = Var n
to (R (L (C (Rec e)))) = Lam e
to (R (R (C (Rec e1 :+: Rec e2)))) = App e1 e2

3.2 Two Generic Definitions

The instant-generics approach can generically define the compos function of Bringert and Ranta [2]. The Compos class provides the compositional behavior underlying bottom-up traversals.

class Compos a where
compos :: Applicative i ⇒ (a → i a) → a → i a
The compos method also threads effects using an applicative functor [15]. The pure function corresponds to monadic return, and the <$> function is a weaker version of >>=

class Functor i ⇒ Applicative i where
pure :: :: i a → i b
(<<>) :: i (a → b) → i a → i b

For example, the definition of modular_nnf in Figure 1 used the ((→) Bool) applicative functor to track polarity.

The generic definition of the compos method extends its first argument by applying it to the second argument's recursive occurrences. Accordingly, the essential case of the generic definition is for Rec. All other cases merely structurally recur. Note that the Dep case always yields the pure function, since a Dep contains no recursive occurrences.

instance Compos (Dep a) where compos _ = pure
instance Compos (Rec a) where
  compos f (Rec x) = pure Rec <$> f x
instance Compos U where compos _ = pure
instance (Compos a, Compos b) ⇒ Compos (a :+: b) where
  compos f (x ::: y) =
  pure (:++) <$> compos f x <$> compos f y
instance Compos a ⇒ Compos (C c a) where
  compos f (C x) = pure C <$> compos f x
  instance (Compos a, Compos b)
    ⇒ Compos (a :+: b) where
  compos f (L x) = pure L <$> compos f x
  compos f (R x) = pure R <$> compos f x

Figure 3. The core instant-generics interface.
```

In doing so, we introduce our own vocabulary for the concepts underlying the instant-generics Haskell declarations.

The core instant-generics declarations are listed in Figure 3. In this approach to generic programming, any value with a generic semantics is defined as a method of a type class. That method's generic definition is a set of instances for each of a small set of representation types: Dep (replacing the Var type of Chakravarty et al. [3]), Rec, U, :::, :, and :+: . The representation types encode a data type's structure as a sum of products of fields. A data type is associated with its structure by the Rep type family, and a corresponding instance of the Generic class converts between a type and its Rep structure. (We use Generic instead of the actual name Representable for horizontal brevity.) Via this conversion, an instance of a generic class for a representable data type can delegate to the generic definition by invoking the method on the type's structure. Such instances are not required to rely on the generic semantics. They can use it partially or completely ignore it.

3.1 The Sum-of-Products Representation Types

Each representation type models a particular structure in the declaration of a data type. The Rec type represents occurrences of types in the same mutually recursive family as the represented type (roughly, its binding group), and the Dep type represents non-recursive occurrences of other types. The Var type of Chakravarty et al. [3] modeled only occurrences of a data type's parameter. We instead are interested in distinguishing between recursive and non-recursive occurrences, so we have replaced Var with the isomorphic Dep. Sums of constructors are represented by nestings of the higher-order type :::, and products of fields are represented similarly by :+: . The U type is the empty product. Since an empty sum would represent a data type with no constructors, it has no interesting generic semantics. The representation of each constructor is annotated via C's phantom parameter to carry more reflective information. The :::, ::, and C types are all higher-order representations in that they expect representations as arguments. If Haskell supported subkinding to the necessary degree (c.f. [14]), these parameters would be of a subkind of * specific to representation types. Since parameters of Dep and Rec are not supposed to be representation types; they would have the standard * kind.

Consider a simple de Bruijn-indexed abstract syntax for the untyped lambda calculus, declared as ULC.

data ULC = Var Int | Lam ULC | App ULC ULC

An instance of the Rep type family maps ULC to its structure as encoded with the representation types.

type instance Rep ULC =
  C Var (Dep Int) ::: C Lam (Rec ULC) :::
  C App (Rec ULC ::: Rec ULC)
For the Rec case, the original function is applied to the recursive field, but \texttt{compos} itself does not recur. As shown in the definition of \texttt{modular-nnf} in Section 3, the programmer, not \texttt{compos}, introduces the recursion. With this generic definition in place, the \texttt{Compos} instance for ULC is a straightforward delegation.

```
instance Compos ULC where
    compos f = to \circ \text{compos} f \circ \text{from}
```

Further, we can generically define the \texttt{==} method of the Eq class.

```
instance Eq a \Rightarrow Eq (Rec a) where
    Rec x == Rec y = x == y
instance Eq U where \_ == \_ = True
instance (Eq a, Eq b) \Rightarrow Eq (a :+: b) where
    x1 :+: x2 == y1 :+: y2 = x1 == y1 && x2 == x2
instance Eq a \Rightarrow Eq (C c a) where
    C x == C y = x == y
instance (Eq a, Eq b) \Rightarrow Eq (a :+: b) where
    L x == L y = x == y
    R x == R y = x == y
    \_ == \_ = False
```

With these instances, Eq ULC is immediate. As Chakravarty et al. [3, §5] show, the GHC inliner can be compelled to optimize away much of the representational overhead.

```
instance Eq ULC where x == y = \text{from} x == \text{from} y
```

As demonstrated with \texttt{compos} and \texttt{==}, generic definitions—i.e. the instances for representation types—provide a default behavior that is easy to invoke. If that behavior suffices for a representable type, then an instance of the class at that type can simply convert with \texttt{to} and \texttt{from} in order to invoke the same method at the type’s representation. If a particular type needs a distinct ad-hoc definition of the method, then that type’s instance can use its own specific method definitions, defaulting to the generic definitions to a lesser degree or even not at all.

The \texttt{instant-generics} approach does not support the heterogeneity of the \texttt{compos} function. We extend \texttt{instant-generics} in the next section so that \texttt{compos} can be defined generically and also used without introducing obfuscation.

4. Our Generic Technique: yoko

We must extend \texttt{instant-generics} in order to generically define \texttt{compos}. Existing generic programming techniques cannot in general be used to define conversions between similar data types, which is an essential quality of \texttt{compos}. Existing techniques can only define consumer functions with a codomain type that is either (i) the same as the domain, (ii) some type with monoidal properties, or (iii) degenerate in the sense that its constructors are structurally-unique and also subsume the domain’s constructors. In this section, we enable a more feasible restriction: the function must have a subset of homomorphic cases that map a domain constructor to a similar constructor in the codomain. The notion of similarity is based on constructor names; we define it in Section 3 below. This improved restriction is enabled by our two extensions to \texttt{instant-generics}.

Our first extension is the basis for clear and modular use of \texttt{compos}. It emulates subsets of constructors. Thus the programmer can split a data type into the relevant constructors and the rest, then explicitly handle the relevant ones, and finally implicitly handle the rest with \texttt{compos}. Specifically, this extension makes it possible for the programmer to use individual constructors independently of their siblings from the data type declaration. We therefore call the resulting generic programming approach \texttt{yoko}, a Japanese name that can mean “free child”. This extension is the foundation for combining “freed” constructors into subsets and for splitting data types into these subsets; both mechanisms are defined in Section 6.

Our second extension enables the generic definition of \texttt{compos} to automatically identify the similar pairs of constructors in its domain and codomain. Existing generic programming techniques in Haskell can only identify the corresponding constructors under the degenerate circumstances of [13] because they do not reflect enough information about data types. Our extension reflects constructor names at the type-level, which is how our generic definition of \texttt{compos} automatically identifies corresponding constructors.

Both of our extensions, and the further developments in Section 6, involve non-trivial type-level programming. Thus we first introduce the newer Haskell features we use as well as some conveniences we assume for the sake of presentation.

4.1 Background: Type-level Programming in Haskell

Promotion of data types to \texttt{data kinds} is a recent extension of GHC [25]. The definitions in this paper use the genuine \texttt{Bool} data kind where \texttt{True} and \texttt{False} are also types of kind \texttt{Bool}. We omit the straight-forward declarations of the type-level conditional and disjunction as the \texttt{If} and \texttt{Or} type families. Furthermore, the \texttt{Maybe} kind is explicitly simulated, since there is not yet syntax for the kind variables required when defining type families over promoted \texttt{\_ \_ \_ \_} data types.

```
data Nothing  -- data Nothing
    data Just a
    data MaybePlus (a :+: b) (b :+: *)
    type instance MaybePlus Nothing b = b
    type instance MaybePlus (Just a) b = Just a
```

We use the \texttt{Maybe} kind for type-level backtracking.

The type-level programming necessary for our generic programming extensions is only partially supported in Haskell. In particular, for clarity of presentation, we assume throughout this paper that a type family implementing decidable type equality is available. Our current implementation simulates this feature almost without exposing it to the user. It is only exposed as the occasional redundant-looking constraint \texttt{True \_ \_ \_ Equal a a}. These are only necessary when applying \texttt{yoko} to parameterised data types; they will not be present in the rest of this paper.

The GHC implementers are currently discussing how to implement this feature. In particular, the decidable type equality will likely be defined using \texttt{closed} type families, with the fall-through matching semantics familiar from value-level patterns.

```
type family Equal a b :: Bool where
    Equal a a = True
    Equal a b = False
```

We simulate this definition of \texttt{Equal} in a way that requires all potential arguments of \texttt{Equal} to be mapped to a globally unique type-level number. The \texttt{yoko} library provides an easy-to-use Template Haskell function that derives such a mapping for a type according to its globally unique name (i.e. package, version, module, and name); Kiselyov [7 #\texttt{Typeable}] uses the same mapping. This simulation of \texttt{Equal} is undefined for some arguments for which the ideal \texttt{Equal} is defined. One example case is when both arguments are the same polymorphic variable: \texttt{a} and \texttt{a}. The simulation can only determine concrete types to be equal, so it is incapable of identifying two uses of the same type variable in its arguments. Thus \texttt{a \_ \_ \_ \_ a} implies that the ideal \texttt{Equal} is \texttt{True} but not so for the simulated one. However, adding the redundant-looking \texttt{Equal} constraints makes the simulation otherwise transparent.

The simulation of \texttt{Equal} is only defined for concrete types that have been reflected with \texttt{yoko}’s bundled Template Haskell, which we tacitly assume for all of our examples. Similarly, the simulation does not support the type-level string literals added in GHC 7.4.2, but we still use them in the following for the sake of presentation.
4.2 Delayed Representation

Our first extension is the delayed representation of data types. While instant-generics maps a type directly to a sum of products of fields, yoko maps a type to a sum of its constructors, which can later be mapped to a product of their fields if necessary. The intermediate stage of a data type’s representation is the anonymous set of all of its constructors, which the programmer can then partition into the subsets of interest.

Delayed representation requires a type corresponding to each constructor, called a fields type. Fields types are similar to instant-generics’s constructor types. However, constructor types are void because they merely annotate a constructor’s representation, while a fields type is the representation. Accordingly, each fields type has one constructor with exactly the same fields as the constructor it represents. The ULC data type needs three.

- data Var_ = Var_, Int
- data Lam_ = Lam_, ULC
- data App_ = App_, ULC ULC

As will be demonstrated in Section 7, programs using the hcompos approach use the fields types directly. It is therefore crucial that the construction be predictably-named.

The yoko interface for data type reflection is listed in Figure 4. It reuses the instant-generics representation types, except C is replaced by N, which contains a fields type. The DCs type family disbands a data type to a sum of its fields types; any subset of this sum is called a disband ed data type. The DCs mapping is realized at the value-level by the DT class. This family and class are the delayed representation analogs of instant-generics’s Rep family and the from method. The inverse to mapping, from a fields type back to its original type, is specified with the Codomain family and DC class. The ULC data type is represented as follows.

- type instance DCs ULC = N Var_ :+: N Lam_ :+: N App_
- instance DT ULC where
  disband (Var i) = L (Var i)
  disband (Lam e) = R (L (Lam e))
  disband (App e0 e1) = R (R (App e0 e1))

- type instance Codomain Var_ = ULC
- type instance Codomain Lam_ = ULC
- type instance Codomain App_ = ULC

The DC class also requires that its parameter be a member of the instant-generics Generic class. The instances for Var_, Lam_, and App_ are straight-forward and as expected. Note that the Rep instances for fields types never involve sums. Every fields type has one constructor, so sums only occur in the DCs family. Because DC implies Generic, the delayed representation subsumes the sum-of-products representation. In particular, the delay effected by fields type is easy to eliminate. The following instances of Rep and Generic for ::: and N do just that.

- type instance Rep (a ::: b) = Rep a ::: Rep b
- instance Generic (a, Generic b) => Generic (a ::: b) where
to (L x) = L (to x)
to (R x) = R (to x)
from (L x) = L (from x)
from (R x) = R (from x)

The yoko interface for data type reflection. We presume type-level strings and reuse the Generic class from instant-generics (imported here as IG).

With these instances, applying Rep after DCs yields the corresponding instant-generics representation, excluding the C type. This is mirrored on the term-level by the ig_from function.

```
ig_from :: (DT t, Generic (DCs t)) => t -> Rep (DCs t)
ig_from = IG.from . disband
```

The C types’ annotation could be recovered by introducing yet another type family mapping a fields type to its analogous constructor type: we omit this for brevity. In this way, the delayed representation could preserve the instant-generics structural interpretation. In general, with an equivalence ≡ that ignores the C type,

```
∀t. Rep t ≡ Rep (DCs t).
```

4.3 Type-level Reflection of Constructor Names

The instant-generics approach cannot in general infer the correspondence of constructors like Plus and PlusN, because it does not reflect constructor names on the type-level. We define the Tag type family (bottom of Figure 4) for precisely this reason. This type family supplants the constructor type class from instant-generics; it provides exactly the same information, only as a type-level string instead of a method yielding a string. For example, instead of the previous section’s Constructor instances for the constructor types Var, Lam, and App, we declare the following Tag instances for the corresponding fields types. The Constructor class’s cnName method could be defined using an interface to the type-level strings that supports demotion to the value-level, perhaps via singleton kinds [25 §8.2].

- type instance Tag Var_ = "Var"
- type instance Tag Lam_ = "Lam"
- type instance Tag App_ = "App"

4.4 Summary

Our first extension to instant-generics delays the structural representation of a data type by immediately disbanding it into a sum of its constructors. Each constructor is represented by a fields type that has one constructor with the exact fields of the original and predictably similar name. Any sum of fields types is called a disband ed data type. Our second extension maps each fields type to its original constructor’s name, reflected at the type-level. These extensions of instant-generics are both used in the generic definition of hcompos in the next section.
5. The Generic Homomorphism

In this section, we explain how the \texttt{hcompos} function generalizes the \texttt{compos} function of Bringert and Ranta [2], and then generically define \texttt{hcompos} accordingly. We begin with a more rigorous definition of homomorphism and specialize it to both \texttt{compos} and \texttt{hcompos}. This emphasizes their shared structure and motivates the use of yoko’s reflection of constructor names to add support for heterogeneity to \texttt{compos}.

A homomorphism \textit{is} a function that preserves some common structure. For example, the function from lists to their length is a homomorphism from lists to integers that preserves monoidal structure. Recall that a monoid is determined by a unit element \texttt{mempty} and a binary operator \texttt{mappend} that together satisfy the monoid laws. Lists with \texttt{mempty} = [] and \texttt{mappend} = (+) form a monoid, as do integers with \texttt{mempty} = 0 and \texttt{mappend} = (+). The length function preserves the monoidal structure since it maps lists’ \texttt{mempty} to integers’ \texttt{mempty} and \texttt{mappend} to \texttt{mappend}.

\[
\text{length mempty} = \text{mempty}
\]
\[
\text{length (mappend x y)} = \text{mappend (length x)} (\text{length y})
\]

The \texttt{compos} function uses this homomorphic pattern as a definitional mechanism. Each default (i.e. compositional) case of an almost compositional function \texttt{f} is trivially homomorphic because it preserves every possible structure: the constructor is left unchanged. Without loss of generality, we assume throughout this discussion that a constructor’s non-recursive fields are all to the left of its recursive fields. For each constructor \texttt{C} with non-recursive fields \texttt{x} and \texttt{n} recursive fields,

\[
f(C \texttt{x} \texttt{x}_0 \ldots \texttt{x}_n) = C \texttt{x} (f \texttt{x}_0) \ldots (f \texttt{x}_n).
\]

Thus the default cases of almost compositional functions are trivially homomorphic, as they preserve every possible semantics of the constructor \texttt{C} and its non-recursive fields \texttt{x}.

The \texttt{hcompos} function generalizes \texttt{compos} because it results in default cases that preserve a more general semantics. Instead of using the constructor as the semantics, it preserves the constructor’s \textit{programmer-intended semantics}. In the example from Section 2, the \texttt{PlusN} and \texttt{PlusNNF} constructors correspond to one another because they have the same intended semantics, namely addition. The intended semantics is the ideal semantics to preserve, but no algorithm is capable of robustly inferring those semantics from a data type declaration. With existing generic programming techniques like \texttt{instant-generics}, the only available property is the structure of constructors’ fields. This is too abstract to be unambiguous. Specifically, it is common for a data type to have multiple constructors with the same structure. Clearly \texttt{Plus} should map to \texttt{PlusNNF} instead of the hypothetical \texttt{Mult}.

We therefore approximate the intended semantics by assuming that the name of a constructor indicates its intended semantics. This crude approximation is very practical when combined with the innocuous assumption that constructors with comparable intended semantics have similar names. Accordingly, the default cases of the almost homomorphic functions built with \texttt{hcompos} are defined by their preservation of the intended semantics. Each maps a domain constructor \texttt{C}_D with non-recursive fields \texttt{x} and \texttt{n} many recursive fields to the codomain constructor \texttt{C}_R which has the same name and the same fields modulo the types being converted:

\[
f(C_D \texttt{x} \texttt{x}_0 \ldots \texttt{x}_n) = C_R \texttt{x} (f \texttt{x}_0) \ldots (f \texttt{x}_n)
\]

where \texttt{Tag C_D} \sim \texttt{Tag C_R}.

The coherence of the two constructors’ fields is implied by the well-typedness of this equality.

```haskell
-- find a fields type with the same name
type FindDC dc dt = FindDC_ (Tag dc) (DCs dt)

-- convert dcs to b; dcs is sum of a's fields types;
-- uses the argument for recursive occurrences
class HCompos a b where
  hcompos :: Applicative i ⇒ (a → i b) → dcs → i b
```

5.1 Constructor Correspondences

We determine constructor correspondence via an algorithm called \texttt{FindDC}. This algorithm takes two parameters: a fields type and the data type in which to find a corresponding constructor. Applied to a fields type \texttt{C} and a data type \texttt{T}, the algorithm finds the constructor of \texttt{T} with the same name as \texttt{C}. If no such constructor exists, then a type-error is raised; the programmer cannot delegate such cases to \texttt{hcompos}. The role of our first extension, delayed representation, is precisely to help the programmer separate the constructors needing explicit handling from those that can be handled implicitly.

The \texttt{FindDC} family defined in Figure 5 implements the \texttt{FindDC} algorithm. An application of \texttt{FindDC} to a fields type \texttt{dc}, modeling the constructor \texttt{C}, and a data type \texttt{dt} uses the auxiliary \texttt{FindDC_} family to find a fields type in the \texttt{DCs} of \texttt{dt} with the same \texttt{Tag} as \texttt{dc}. The instances of \texttt{FindDC_} query \texttt{dt}’s sum of constructors, using the type equality predicate and the \texttt{Maybe} data kind discussed in Section 4.1. The result is either the type \texttt{Nothing} if no matching fields type is found or an application of the type \texttt{Just} to a fields type of \texttt{dt} with the same name as \texttt{dc}. The \texttt{hcompos} definition enforces the coherency of corresponding constructors’ fields with an equality constraint.

Because the names of corresponding constructors must match exactly, the definition of \texttt{best_nnf} in Figure 2 is actually ill-typed. Fixing it is straightforward, but distracting. Specifically, the \texttt{RNF} data type must be declared in a separate module, presumably also called \texttt{RNF}, with the \texttt{R} suffix removed from its constructors. Thus \texttt{best_nnf} actually requires \texttt{FindDC} to map the \texttt{Plus} constructor of \texttt{Exp} to the \texttt{Plus} constructor of \texttt{RNF}. Where before we wrote \texttt{Var} and \texttt{Plus}, we would actually write \texttt{RVar} and \texttt{RPlus}. We plan to implement a more robust \texttt{FindDC} algorithm that can conservatively match in the presence of disparate prefixes and/or suffixes once GHC adds more support for type-level programming with strings.

5.2 The Generic Definition of \texttt{hcompos}

The generic homomorphism is declared as the \texttt{hcompos} method in Figure 6. To support heterogeneity, its type class adds the \texttt{ds} and \texttt{p} parameters to the original \texttt{a} parameter from the \texttt{compos} class. The \texttt{dc} type variable will be instantiated with the sum of fields types corresponding to the subset of \texttt{a}'s constructors to which \texttt{hcompos} is applied. The \texttt{ds} parameter is necessary because it varies throughout the generic definition of \texttt{hcompos}. The \texttt{p} type is the codomain of the conversion being defined. The generic definition of \texttt{hcompos} relies on generic function \texttt{mapRs}, which extends a function by applying it to every recursive field in a product of fields.
The instance for context of the following instance; it hosts this type-level traversal.

sums directly. Note how the constructors of the best_nnf

Second, it applies the mapRs

but it is ill-typed.

best_nnf

into that subset of interest and the subset of remaining constructors.

This mechanism lets the programmer clearly specify an anonymous subset of a data type's constructors and then automatically partitions a data type into that subset of interest and the subset of remaining constructors.

In Figure 2 the exact_case function implicitly partitions the constructors of the Exp type in the definition of w, the essence of best_nnf. The following definition of w would be ideally concise, but it is ill-typed.

The problem is that the type of e in the third case is Exp. Thus the type of disbanded e is DCs Exp, which includes the Var_ and Neg_ fields types. Because hcompos is applied to the disbanded e, the FindDC algorithm will fail to find a constructor in NNF corresponding to e. The crucial insight motivating our entire approach is that those fields types will never occur at run-time, as they are guarded by the other cases of e. Thus, the type error can be avoided by encoding this insight in the type system. Given the yoko interface up to this point, this can only be done by working directly with the fields types.

\[
\text{-- NB speculative: well-typed, but immovable}
\]

\[
\begin{align*}
& \text{w} : \text{Exp} \to \text{Bool} \to \text{NNF} \\
& \text{w} \ e = \text{case disbanded e of} \\
& \quad \text{L (Var_ s)} \to \text{NNF.Var s} \\
& \quad \text{R (L (Neg_ e))} \to \text{w e \circ not} \\
& \quad \text{R (R e)} \to \text{hcompos w e}
\end{align*}
\]

This second definition is well-typed but unacceptably immodular because it exposes extraneous details of Exp's representation as a sum to the programmer. Specifically, the L and R patterns depend on how the :+: type was nested in DCs Exp. Modularity can be recovered by automating the partitioning of sums that is currently explicit in the L and R patterns. The implicit partitioning of disbanded data types is the final yoko capability.

Beyond motivating implicit partitioning of disbanded data types, the above definition of w is also the original motivation for fields types. Indeed, the hcompos function can be defined—perhaps less conveniently—with a more conservative extension of the C representation type that indexes the Tag type family by instant-generics constructor types. The above definition of w would still need to avoid the type-error by partitioning the constructors. However, where the type fields' syntax conveniently imitates the original constructors, the instant-generics C type would complicate the immorality and even obfuscate the code by exposing the representation of fields. Worse still, the current development of implicit partitioning would require further obfuscation of this definition in order to indicate which summand is intended by a given product pattern, since two constructors might have the same product of fields. It is the fields types' precise imitation of the represented constructor that simultaneously encapsulates the representational details and determines the intended summand.

The implicit partitioning interface is declared in Figure 2. Its implementation interprets sums as sets. Specifically, R constructs a singleton set, and :+: unites two sets. The embedded type class models the subset relation, with elements identified by the equal type family from Section 4. Similarly, the partition type class models partitioning a set into two subsets. Finally, set difference is modeled by the :-: family, which determines the right-hand subset of a partitioning from the left. It enables the type of the value-level partitioning function.

\[
\text{type instance} \ (:::-) \ (\text{N a}) \ \text{sum2} = \\
\text{If} \ \text{Elem a sum2} \ \text{Void} \ (\text{N a}) \\
\text{type instance} \ (:::-) \ (\text{a :+: b}) \ \text{sum2} = \\
\text{Combine} \ (\text{a} :::- \text{sum2}) \ (\text{b} :::- \text{sum2})
\]

The Elem family models the decidable membership relation, again using the Equal type family for identifying elements.

\[
\text{type family} \ \text{Elem} \ a \ :: \ \text{Bool} \\
\text{type instance} \ \text{Elem a} \ (\text{N b}) = \text{Equal a b} \\
\text{type instance} \ \text{Elem} \ a \ (\text{a :+: t}) = \\
\text{Or} \ (\text{Elem a s}) \ (\text{Elem a t})
\]

The Combine type family is only used to prevent the empty set, modeled by Void, from being represented as a union of empty sets. Because Void is used only in the definition of :::-, an empty result of :::- leads to type-errors in the rest of the program. This
-- embedding relation
class Embed sub sup where embed :: sub → sup
-- ternary partitioning relation
class Partition sup subb sub where
  partition_ :: sup → Either subL subR
-- set difference function
type family (:-:) sum sum2
partition :: Partition sup sub (sup :-: sub) ⇒ sup → Either sup (sup :-: sub)
partition = partition_
-- an exactly typed eliminator
exact_case :: Partition (DCs t) dcs (DCs t :-: dcs) ⇒ t → ((DCs t :-: dcs) → a) → (dcs → a) → a
-- assembling fields type consumers
one :: (dc → a) → N dc → a
(ll) :: (1 → a) → (r → a) → l ::*: r → a
(l.l) :: (1 → a) → (r → a) → l ::*: N r → a
(.l.l) :: (1 → a) → (r → a) → N l ::*: r → a
(.l) :: (1 → a) → (r → a) → N l ::*: N r → a

data Void
  type family Combine sum sum2 where
    Combine Void a = a
    Combine a Void = N a
    Combine a b = a :+: b

We omit the Embed and Partition instances for space. Because they only treat the N and :+: types, the programmer need not declare their own. They have the same essential semantics as similar classes from existing work, such as that of Świerstra [20]. However, existing work defines these classes with overlapping instances. We use type-level programming to avoid overlap for reasons discussed by Kiseliov [7] #anti-over. We use the decidable Eq type family and its derivatives, such as Elem, to explicitly distinguish between instances that would otherwise overlap. Kiseliov [7] #without-over develops a similar method.

Finally, the one function and the family of disjunctive operators assemble functions that consume fields types into larger functions. These larger functions’ domain is a subset of constructors. The |(1| operator is directly analogous to the Prelude.either function. Its derivatives are defined by composition in one argument with one. When combined with the disband and partition functions, these operators behave like an extension of the Haskell case expression that supports exact types.

exact_case :: Partition (DCs t) dcs (DCs t :-: dcs) ⇒ t → ((DCs t :-: dcs) → a) → (dcs → a) → a
exact_case x g f =
either f g $ partition $ disband x

With exact_case, the well-typed but immodular definition of $u$ at the beginning of this subsection can be directly transiterated to the ideal $u$ from best_mnf in Figure 2.

The |(| family of operators help the programmer use disbanded data types in a clear way. As demonstrated in best_mnf, they resemble the | syntax used in many other functional languages to separate the alternatives of a case expression. Their inclusion in the yoko interface for partitioning helps simulate an exactly-typed case expression.

7. Example: Lambda Lifting

We demonstrate the utility of fields types, hcompos, and implicit partitioning with a realistic example. Lambda-lifting (also known as closure conversion; see, e.g., [18]) makes for a compelling example because it is a standard compiler pass and usually has many homomorphic cases. For example, if hcompos were used to define a lambda-lifting function over the GHC 7.4 data type for external expressions, it would handle approximately 35 out of 40 constructors implicitly. Lambda-lifting also involves more sophisticated effects than the (→ Bool) applicative functor from Section 2 and therefore demonstrates the kind of subtlety that can be necessary in order to encapsulate effects with an applicative functor.

7.1 The Types and The Homomorphic Cases

The lambda-lifting function lifts lambdas out of untyped lambda calculus terms. As the ULC data type has been a working example in the previous sections, its instances for yoko type families and classes are assumed in this section.

data ULC = Var Int | Lam ULC | App ULC ULC

Lambda-lifting generates a top-level function declaration for each sequence of lambdas found in a ULC term. Each declaration has two sets of parameters: one for the formal arguments of the lambdas it replaces and one for the variables that occur free in the original body. We call the second kind of variable capture variables, as the lambdas capture them from the lexical environment. The bodies of the generated declarations have no lambdas. Their absence is the characteristic property that lambda-lifting establishes. It is encoded in a derivative of ULC, the TLF data type, which is used for the bodies of top-level functions.

data TLF = Top Int [Occ] | Occ Occ | App TLF TLF
data Occ = Par Int | Env Int

Instead of modeling lambdas, TLF models occurrences of top-level functions, invocations of which must always be immediately applied to the relevant captures. It also distinguishes occurrences of formals and captures. Like the ULC type, TLF is also assumed to have instances for the yoko type families and classes.

The codomain of lambda-lifting is the Prog data type, which pairs a telescope of top-level function declarations with a main term.

data Prog = Prog [Dec] TLF
  type Dec = (Int, Int, TLF) -- # captives, # formals

Each function declaration specifies the sizes of its two sets of variables: the number of captives and the number of formals.

Because both of the ULC and TLF data types have a constructor named App, the definition of lambda-lifting can delegate the case for applications to hcompos. If constructors for any other syntactic constructs unrelated to binding were added to both ULC and TLF, the definition of lambda-lifting below would not need to be adjusted.

The lambdaLift function is defined by its cases for lambdas and variables. (We define the monad |@ next.)

lambdaLift :: ULC → Prog
lambdaLift ulc = Prog ds tlf where
  ds = runULC (ll ulc) ((1, IntMap.empty), 0)
  tlf = IntSet.findMax $ freeVars ulc

ll :: ULC → M TLF
ll tm = exact_case tm (hcompos ll) $ llVar .|. llLam

llVar :: Var → M TLF ; llLam :: Lam → M TLF

This is the principal software engineering benefit of hcompos—homomorphic cases are completely implicit. Furthermore, fields types make it convenient to handle specific constructors separately.
The traversal implementing lambda-lifting must collect the top-level functions as they are generated and also maintain a renaming between ULC variables and TLF occurrences. The monad $\mathcal{M}$ declared in Figure 8 provides these effects. They are automatically threaded through the \texttt{hcompose} function: every monad is also an applicative functor. The \texttt{Rename} type includes the number of formal variables that are in scope and a map from the captives to their new indices. It is a standard monadic environment, accessed with \texttt{ask} and updated with \texttt{local}. The list of generated declarations is a standard monadic output, collected in left-to-right order with the [1/++] monad, and generated via the \texttt{emit} function. The final effect is the backwards state and a map from the captives to their new indices. It is crucial to maintaining a de Bruijn encoding for the occurrences of top-level functions.

7.2 The Interesting Cases

The variable case is straightforward. Each original variable is either a reference to a lambda's formal argument or its captured lexical environment. The \texttt{lookupRN} function uses the monadic effects to correspondingly generate either a \texttt{Par} or a \texttt{Env} occurrence.

\begin{verbatim}
11Var (Var_ i) = pure (\rn \rightarrow lookupRN (rn i) \langle> \langle ask \
                     lookupRN :: Rename \rightarrow Int \rightarrow Occ \
                     lookupRN (nLocals, _) i | i \leq nLocals = Par i \
                     | otherwise = \case IM.lookup (i - nLocals) m of 
                          Nothing \rightarrow error "free_var" 
                          Just i \rightarrow Env i 

The case for lambdas is defined in Figure 9. It uses two auxiliary functions: \texttt{freeVars} for computing free variables and \texttt{peel} for peeling a sequence of lambdas off the top of a ULC term. We omit the definition of \texttt{freeVars} because it uses only \texttt{instant-generics}.

\texttt{freeVars :: ULC \rightarrow IntSet} \\
\texttt{peel :: ULC \rightarrow (Int, ULC)} \\
\texttt{peel = u \rightarrow (1 + acc) tm} \\
\texttt{acc tm = (acc, tm)}
\end{verbatim}

Though \texttt{peel} is a property-establishing function—the resulting ULC is not constructed with \texttt{Lam}—we do not encode the property as a data type because we do not actually depend on it.

\begin{verbatim}
instance Monad M where 
  return a = M a 
  m >>= k = M$ m >>= k 
instance Monad M where 
  return a = M a 
  m >>= k = M$ m >>= k
\end{verbatim}

\begin{verbatim}
instance Monad M where 
  return a = M a 
  m >>= k = M$ m >>= k
\end{verbatim}

The \texttt{11Lam} function uses \texttt{peel} to determine the sequence of lambdas’ length and its body, called \texttt{nLocals} and \texttt{ulc} respectively. The sequence of lambdas captures are precisely its free variables. The body is lambda-lifted by a recursive call to \texttt{ll} with locally modified monadic effects. Its result, called \texttt{tlf}, is the body of the top-level function declaration generated by the subsequent call to \texttt{emit}. Since \texttt{tlf} corresponds to the original lambdas, an invocation of it replaces them. This invocation explicitly pass the captives as the first set of actual arguments.

The de Bruijn index used to invoke the newly generated top-level function is determined via the \texttt{intermediates} monadic effect. The index cannot simply be 0 because sibling terms to the right of this sequence of lambdas also generate top-level declarations, and the left-to-right collection of monadic output places them between this \texttt{Top} reference and the intended top-level function. This is the reason for the backwards state in \texttt{ll}. The corresponding circularity in the definition of \texttt{>>=} is guarded here because \texttt{tlf} is emitted without regard to the number of subsequent emissions, though its value does depend on that number.

The recursive call to \texttt{ll} must use a new monadic environment for renaming that maps variable occurrences to the corresponding parameters of the new top-level function; \texttt{local} provides the necessary environment to the subcomputation. The recursive call must also ignore the emissions of computations occurring after this invocation of \texttt{11Lam}, since those computations correspond to siblings of the sequence of lambdas, not to siblings of the lambda body—the body of a lambda is an only child. This is an appropriate semantics for \texttt{intermediates} because the emissions of those ignored computations do not end up between \texttt{tlf} and the other top-level functions it invokes.

7.3 Summary

This example evidences the software engineering benefits of \texttt{hcompose} for defining non-trivial functions. Lambda-lifting establishes the absence of lambdas, which we encode with an exact type. It is also an almost homomorphic function, because only two cases are not homomorphic. Our \texttt{yoko} approach therefore enables an concise and modular definition of an exactly-typed lambda-lifting function. Though the generic case only handles the constructor for function application in this example, the very same code would correctly handle any additional constructors unrelated to binding. This would include, for example, approximately 35 out of 40 constructors in the GHC 7.4 data type for external expressions.
8. Disbanding Complex Data Types

All of the data types in the previous sections have been easy to represent generically, but large programs usually involve richer data types. This section enables the use of `hcompos` with two additional classes of data types: mutually recursive data types and those with compound recursive fields. Compound recursive fields contain applications of higher-kindred types, such as `[]`, `(,)`, and `((,) Int)` to recursive occurrences. Such fields are only partially supported by `instan-generics` but are common in large programs. Our solution is still partial, but much less so, and is not specific to `hcompos`. We discuss support for even more sophisticated data types like GADTs in the next section.

The previous declaration of `hcompos` and `mapRs` cannot support mutual recursion because of the type of their first argument. It constrains the function argument to convert only one data type. For mutually recursive data types, this function must instead convert each type in the mutually recursive family; it must be polymorphic over those types. There are at least two ways to generalize the first argument of `hcompos` to support mutually recursive data types.

8.1 Encoding Relations of Types with GADTs

The first generalization of `hcompos` supports mutually recursive families using the same technique as Bringert and Ranta [24]. This approach uses generalised algebraic data types (GADTs) [24] and rank-2 polymorphism to express the type of polymorphic functions that can be instantiated only for the data types comprising a given mutually recursive family. The technique is separated into more elementary components in the `multirec` generic programming approach [24]. The `multirec` approach uses GADTs to encode sets of types. For example, a pair of mutually recursive data types for even and odd natural numbers with special cases for addition by sets of types. For example, a pair of mutually recursive data types.

We presume for clarity that `mapRs` handles suffixes.

```haskell
data Odd = SuO Even | SuSuO Odd
data Even = Zero | Su Odd | SuSuE Even
data OddEven :: a -> a where
  OddT :: OddEven Odd
evenT :: OddEven Even
```

The index of `OddEven` can only ever be `Odd` or `Even`. It thus emulates a type-level set inhabited by just those types (i.e., a `subkind`), and a function with type `Vs` `OddEven s` -> `s` -> `a` can consume precisely either of those types. The `multirec` approach also provides a class that automatically selects the correct GADT constructor to witness the membership of a given type in a given set.

```haskell
class Member set a where memb :: set a
instance Member OddEven Odd where memb = OddT
instance Member OddEven Even where memb = EvenT
```

The `Member` class is used throughout the `multirec` approach for those generic definitions with value-level parameters that must be instantiable at any type in the mutually recursive family.

GADTs can similarly encode the relation between types that is required for `hcompos`. The `hcompos` and `mapRs` classes can be parameterized over this relation as follows, where `C` is `hcompos` or `mapRs` and `m` is `hcompos` or `mapRs`, respectively.

```haskell
class C rel dcs b where
  m :: Applicative i =>
    (a b, rel a b -> a -> i b) -> dcs -> i b
```

Where the first argument of `hcompos/mapRs` was formerly a function from `a` to `b`, it is now a function between any two types that are related by the GADT-encoded `rel` relation. The former type of the first argument constrained the (implicit) relation to exactly `{a -> b}` and nothing else. The new type removes this excessive constraint.

This variant of `hcompos` has essentially the same instances as declared in [Section 5] the case for the `:+` type folds through the sum and the case for `hcompos` handles the auxiliary `mapRs` class.

8.2 Encoding Relations of Types with Type Classes

The second generalization of `hcompos` uses the established `instan-generics` techniques for mutually recursive data types. The key insight is that multiparameter type classes already encode relations. The only distinction is that type classes encode open relations while GADTs encode closed relations. Since the relation for `hcompos` need not be closed, the type class approach suffices.

The relation between types in the domain and codomain families must still be specified by the user. Instead of being defined with a GADT, however, that relation follows from the instances of `hcompos` for a given conversion. Thus, the `hcompos` class from the following schema subsumes the `Related` class.

```haskell
type family Idiom (cnv :: *) :: a -> *
class C cnv a b where
  m :: cnv a -> a -> (Idiom cnv) b
```

This declaration interprets `hcompos` as a mapping from type `cnv` to a conversion between `a` and `b`. Since the `hcompos` instances for a given `cnv` type also define the relation characterizing that conversion, the `mapRs` instance for `Rec` must call `hcompos` instead of `rel`.

```haskell
class Related rel a b where rel :: rel a b

instance Related rel a b =>
  mapRs rel (Rec a) (Rec b) where
  mapRs cnv (Rec x) = pure (Rec (rel $ x))
```

This encoding of relations requires the user to define the GADT. Otherwise, it behaves just like the `hcompos` for singly recursive types. For example, a user might need to convert the `Odd` and `Even` data types back to a simple data type for naturals.

```haskell
data Nat = Zero | Su Nat

The relation `{Odd => Nat, Even => Nat}` is at the core of this conversion and encoded with the `Rel` GADT. This example does not lose generality by mapping both types to the same type.

```haskell
data Rel :: a -> a -> a
instance Related Rel Odd Nat where rel = RelOdd
instance Related Rel Even Nat where rel = RelEven

The conversion function between the two type families can use `hcompos` with the `rel` relation for any shared constructors. In the `w` function below, the cases for `Zero`, `Su0`, and `SuE` are implicit.

```haskell
newtype Id a = Id (unId :: a)

w :: Rel oe pl -> oe -> Id pl
w RelOdd o = exact_case o (hcompos w) $ one $ 
  λ(SuSu0_ o) → pure (Su0 Su0) <> w rel o
w RelEven e = exact_case e (hcompos w) $ one $ 
  λ(SuSuE_ e) → pure (Su0 SuE) <> w rel e

This variant of `hcompos` has essentially the same instances as declared in [Section 5] the case for the `:+` type folds through the sum and the case for `hcompos` handles the auxiliary `mapRs` class.
instance HCompos cnv a b => 
    MapRs cnv (Rec a) (Rec b) where 
    mapRs cnv (Rec x) = pure Rec <-> hcompos cnv x

Though HCompos and MapRs are now mutually recursive, the user still only declares instances of HCompos.

The conversion from Odd/Even to Nat is expressed as follows. Again, the target type need not be the same.

data OE2N = OE2N ; type instance Idiom OE2N = Id

instance HCompos OE2N Odd Nat where 
    hcompos cnv e = exact_case_e (hcompos cnv) $ one $ 
        λ(SuSu_ e) → pure (Su ○ Su) <-> hcompos cnv e
instance HCompos OE2N Even Nat where 
    hcompos cnv o = exact_case_o (hcompos cnv) $ one $ 
        λ(SuSu_ e) → pure (Su ○ Su) <-> hcompos cnv e

oe2n :: HCompos OE2N oe Nat = oe → Nat
oe2n = unId o hcompos OE2N

The OE2N value subsumes both the v function and the Rec data type from the multirec-based definition of oe2n. This combination makes type class encoding of relations more concise.

8.3 Compound Recursive Fields

All definitions of hcompos, regardless of mutual recursion, require that the Rec representation type is only applied to recursive occurrences. The standard instant-generics practice, though, represents a compound recursive field by applying Rec to the entire field. Since compound recursive fields are common, we extend instant-generics to allow hcompos to handle them.

Consider the following Rep instances for a data type T with two constructors Tip and Bin with one compound recursive field.

    type instance Rep Tip_ = Rec (Int, T)
    type instance Rep Bin_ = Rec (T, T)

The hcompos function cannot be used with data types like T because the MapRs instance for Rec requires that the argument is itself a recursive occurrence and therefore calls hcompos on it. This issue cannot be solved by adding an HCompos instance for (a, b). For example, this cannot distinguish between the (Int, T) and (T, T) fields, but treating them both the same requires an HCompos constraint for converting an Int to an Int. Requiring the user to declare such ad-hoc and degenerate instances is unacceptable.

The real problem is the imprecision of the instant-generics representation. However, instant-generics is designed to be extensible, so we can add new representation types.

    newtype Arg1_ t a = Arg1_ (t a)
    newtype Arg2_ t a b = Arg2_ (t a b) -- etc, as needed

In the Arg1 and Arg2 representation types, the a and b type parameters are representations and the t parameter is not. These types permit a precise representation of compound fields in which Rec is only applied directly to recursive type occurrences.

    type instance Rep Tip_ = Arg1_ ((,) Int) (Rec T)
    type instance Rep Bin_ = Arg2_ ((,) (Rec T) (Rec T)

Because the Arg1 types indicate which of their arguments need conversion, they do not introduce degenerate HCompos constraints. As representations of fields, they only need instances of MapRs.

    instance (Traversable f, MapRs cnv a b) => 
        MapRs cnv (Arg1_ f a) (Arg1_ f b) where 
        mapRs cnv (Arg1_ x) = Arg1_ $ traverse (mapRs cnv) x

The instance for Arg2 is analogous, but involves a BitTraversable constraint and two C constraints. It is easy to add similar types with even greater arity as long as the parameters are of kind *.

These additional representation types permit use of hcompos with compound recursive fields.

9. Related Work

The work most related to ours is that of Bringert and Ranta [2], which defines and demonstrates the compos function. The compos function can be used much like hcompos to handle the homomorphic cases in the definition of function, but it is only applicable when the domain and codomain are the same type. Many existing generic programming techniques can generically define compos, so programmers can use it “for free” to improve the modularity of their definitions. We add heterogeneity to compos in order to make its benefits available when defining the property-establishing functions that are pervasive in exactly typed programming.

We divide the other related work into methods for exact types and generic programming techniques deserving more discussion.

Exact Types

While the most exact possible types require dependency, we are interested in the best approximation supported by mainstream non-dependent types. Even so, we believe our approach may be adapted to the dependent setting. In that context, the notion of exact type is similar to adequacy [6]. A significant advance in the typing of functional programs was the recent adoption of GADTs [22]. We have experimented with using GADTs to model the closed set of fields types instead of enumerating the elements of the set with ::+ and its, but have found it detrimental to the syntax of field types and thus a burden on the user without clear benefit for the examples in this paper.

Exact types are essential for Turner’s total functional programming [21], in which every function must be total. Specifically, all patterns must be exhaustive and all recursion well-founded. Without termination guarantees, exhaustive pattern matching is trivially achieved by just diverging in what would be the omitted cases for non-exhaustive patterns. We adopt the stance of Danielsson et al. [4] and consider the challenges of exact types without concern for termination. Exact types make it straightforward to pattern match on data types without having to pass around handlers for the error cases, since such cases are eliminated upstream by property-establishing functions.

We have found two languages with explicit support for subsets of constructors, the Common Algebraic Specification Language (CASL) specification language [15] and the OCaml programming language [19]. CASL supports named declaration of constructor subsets, by declaring data types as non-disjoint unions of smaller data types [16][4]. This approach requires the subsets of data types to be identified a priori and invasively incorporated into the declaration of the data type itself. For example, the CASL approach cannot be used to characterize subsets of data types defined in libraries, since their declaration cannot be changed. Such immutability is unacceptable for programming in the large. Our approach is applicable to library data types, because constructor subsets are anonymous and do not affect the original data type declaration. This is made possible by the Tag, Co domina1, DC, and DT type families, and it is made practical by our bundled Template Haskell code, which is a common dependency for Haskell generic programming.

In OCaml, polymorphic variants allow any name, called a variant, to occur as if it were a constructor [5]. Both polymorphic variants and yoko’s disbanded data types provide anonymous subsets of constructors. However, polymorphic variants, as a widely-applicable feature, intentionally model subsets with less exact types. In particular, an occurrence of a variant is polymorphic in its codomain. It constructs any data type that has a constructor with the same name and coherent fields. Fields types, on the other hand, are associated with an original data type via the Co domina1 type family.

The type-indexed coproducts of Kiselyov et al. [8][C] are also similar to polymorphic variants. They are a more restricted version of yoko’s sums and provide a capability similar to implicit partitioning. Where ::+ models union of sums in yoko, the operator
of the same name in the work of Kiselyov et al. [8] specifically models a type-level cons and is therefore not associative.

The recent work on data kinds [25] promotes constructors to types that superficially seem related to our fields type. These type-level constructors of data kinds, though, have a distinct notion of corresponding term-level value, the singleton types [19 §3.6].

Generic Programming Generic programming techniques are broadly characterized by the universe of representative types. yoko primarily has the same universe as instan-generics, with more complete support for compound recursive fields. We believe yoko’s enhancements are orthogonal to other extensions of instant-generics and will investigate integration. We anticipate that some generic programming techniques not based on instant-generics also admit extensions comparable to ours.

The most exact types supported by Haskell require sophisticated data types not in the yoko universe. In particular, nested recursion [1] and GADTs are crucial to encoding many interesting properties, such as well-scaled and/or well-typed term representations. While some uses of these features can be forced into the yoko representation types, current research, like that of Magalhães and Jeuring [11], is investigating more natural representations. One derivative of instant-generics is that it is better suited for these features is generic-deriving [12], which was recently integrated with GHC. Most of the generic-deriving representation types are promotions of the instant-generics types from the kind * to the kind *→→*. The generic-deriving approach only represents type constructors, but it interprets * types as *→→* types that do not use the type parameter. Since type parameters are a prerequisite for nested recursion and GADTs, a representation designed for *→→* types more naturally handles such sophisticated data types. We next plan to add the yoko extensions to generic-deriving.

10. Conclusion

The generic homomorphism, introduced in this paper as hcompos, factors out a pattern in the definition of functions that convert between data types with analogous constructors. The technique enables the improved assurance of exact types without sacrificing maintainability. Conversion definitions are concise, modular and cheap to write as needed. We demonstrated the technique by defining a lambda-lifting function that encodes the absence of lambdas with its exact codomain type. Because of hcompos, this definition is modular, referencing only constructors for variables and binders.

Existing generic programming techniques are inapplicable to the hcompos function specifically because of its unequal domain and codomain types. In order to support this heterogeneity, we extend the instant-generics approach with type-level reflection of constructor names and a delayed representation. These generic programming extensions may be more broadly useful, as they provide the fundamental genericity underlying hcompos. In particular, the delayed representation makes it convenient for the programmer to use anonymous subsets of a data type’s constructors.

As an extension of instant-generics, our approach inherits its competitive efficiency and expressiveness. While this implies minimal support for nested data types and GADTs, which are both important for sophisticated exact types, we anticipate our extensions are portable to enrichments of instant-generics that support such data types. Our library, including hcompos and the Template Haskell for deriving all necessary instances, is available at http://hackage.haskell.org/package/yoko

11. Acknowledgments

This work was partially supported by the National Science Foundation under grant CCF-1117569. We thank the anonymous reviewers for their helpful comments.

References