Reducing the Cost of Precise Types

By

Nicolas Frisby

Submitted to the graduate degree program in
Electrical Engineering & Computer Science
and the Graduate Faculty of the University of Kansas
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Committee members

Dr. Warren Perry Alexander,
Chairperson

Dr. Andy Gill

Dr. Prasad Kulkarni

Dr. Bo Luo

Dr. Sara Wilson

Date defended: 28 August 2012

The Dissertation Committee for Nicolas Frisby certifies
that this is the approved version of the following dissertation :

Reducing the Cost of Precise Types

Dr. Warren Perry Alexander, Chairperson

Date approved: 28 August 2012

Abstract

Programs involving precise types enforce more properties via type-checking, but precise types also prevent the reuse of functions throughout a program since no single precise type is used throughout a large program. My work is a step toward eliminating the underlying dilemma regarding type precision versus function reuse. It culminates in a novel traversal operator that recovers the reuse by automating most of each conversion between "similar" precise types, for a notion of similarity that I characterize in both the intuitive and technical senses. The benefits of my techniques are clear in side-by-side comparisons; in particular, I apply my techniques to two definitions of lambda-lifting. I present and implement my techniques in the Haskell programming language, but the fundamental ideas are applicable to any statically- and strongly-typed programming functional language with algebraic data types.

Page left intentionally blank.

Acknowledgements

It's been a long time comin'! After seven years in the "fast track" PhD program, I've archived here only a sliver of what I've learned and discovered. The knowledge I have omitted — knowledge about type theory, term representations, DSLs, optimization, hardware/software codesign, perseverance, practicality, communication, decision making, team work, morality, meaning, community, friendship, family, love, and beyond — would dwarf this dissertation's 150 pages. As a proxy, I can only thank those who have enabled me to learn and those who have learned alongside me.

Thank you, Perry. As my advisor, you've provided a stupefying degree of support through monies, understanding, and sagacity. It has been a pleasure to work with and for someone with whom I share the most important beliefs about life. All of those stories you share add up to reflect a man who believes in family, kindness, and the pursuit of knowledge — I am lucky to have had your mentorship for these eleven years!

Thank you, Andy, Prasad, Bo, Sara, Dave Andrews, Gary Minden, Douglas Niehaus, Gunes Ercal, and Ron Sass. From deep technical discussion all the way to last-minute committee membership, you've each affected me and my research, and I appreciate it.

Thank you, lab mates! Andrew Schmidt, Philip Weaver, Ilya Tabakh, Rory Petty, Matt Cook, Garrin Kimmell, Ed Komp, Jason Agron, Wesley Peck, Mark Snyder, Jeni Lohoefener, Megan Peck, Evan Austin, Andrew Farmer, Brigid

Halling, Michael Jantz, Neil Sculthorpe, and Kevin Matlage: it's been a pleasure to share so many classes, presentations, and discussions with you and to tackle our problems side-by-side. The lab would have seemed much more like the bare-ceilinged and concrete-floored room it actually is if you all had not been in it with me. Good luck to you. Same to Eddy Westbrook and Bill Harrison: it was a pleasure to work with you, learn from you, and just hang out.

Thank you, friends from college all the way back to childhood. I won't name one if I can't name all, and I certainly can't do that here. I've learned the most with you so far and it's going to be that way forever; I have no doubt. You will always be my most cherished teachers.

Thank you, family. Thank you for fostering my love for learning and science but also making it clear that it's just a fraction of who I am as a flesh and blood person. Your constant love for me and for our family and friends will always guide me.

Thank you, Jessica. This dissertation has become quite a milestone in my life — a rather elusive objective, finally fulfilled. But having summited this huge peak, I can now see past it with a spectacular view, and I'm most affected by the new clarity I have looking out over my future with you.

Contents

Listings

1 Introduction

In this dissertation, I make it easier for programmers working in statically- and strongly-typed functional languages to use more *precise* types, *i.e.* those that contain fewer unintended values. This is a step towards making large programs easier to write, because precise types simplify function definitions by eliminating unintended exceptional cases and enforce some of the properties expected at various points in a program. For example, a precise codomain of a function implementing let-elaboration would intentionally not be able to represent let-expressions. The most compelling motivation for my work is that precise types are currently prohibitively expensive to use in large statically- and strongly-typed programs; their benefits are out-of-reach when needed the most.

Ideally, programmers could readily define numerous precise types throughout a large program, each enforcing the respective properties that characterize the intended data within a given part of the program. However, while each precise type itself is relatively cheap to define, the functions converting between precise types become expensive to write. In particular, the usual technique of defining a traversal operator and reusing it throughout a program is no longer viable: because of the precise typing, there is no single data type used pervasively throughout a program. Because the precise types themselves are necessarily distinct types, a simple traversal operator cannot be reused for multiple precise—but otherwise conventional—data types.

As a result of these costs of precise types, developers of large programs favor the reusability of simpler types over the correctness benefits of precise types. For example, the source of the Glasgow Haskell Compiler (GHC) includes many functions that immediately raise fatal run-time errors for some of their domain type's constructors. The GHC implementers document these functions' related preconditions using type synonyms of the inexact types, indicative constructor names, and comments that explain which of the constructors are expected as input and output. Specialized precise types are avoided and reuse made possible, but at the

cost of some enforced correctness. For data types with more than a hundred constructors, like the GHC representation of external syntax, this is a fair engineering choice. My work, though, is a step toward eliminating the underlying dilemma regarding type precision versus reuse.

My technical work in this dissertation culminates in a novel reusable traversal operator that can automate most of the conversion between "similar" precise types. I characterize this notion of similarity later in both the intuitive and technical senses and demonstrate several ways that such similar types naturally arise in programs. My operator eliminates a major cost of precise types: it makes the conversions between them as cheap to write as the corresponding function operating on a single imprecise type would be. I give a spectrum of concrete motivating examples in the next section. The comparisons of these examples' uniquely expressive solutions enabled by my operator against their imprecisely-typed-and-reusable and nonreusable-but-precisely-typed counterparts serve as the basis of my evaluation.

I work within two research areas: *generic programming* and *precisely-typed programming*. The first terminology is well-established in the literature, which includes my main publications [51, 16, 17]. The second is my own terminology for programming practices that prioritize leveraging the type system in order to assure program correctness (*e.g.* my other relevant publication [53]). Precisely-typed programming is an existing, seemingly implicit, notion in the broad literature regarding programming with statically-checked types, but I do not know of existing terminology with the same connotation as *precisely-typed programming* nor any published work that focuses directly on it. I refer to the large body of existing work relevant to precisely-typed programming as if it were a recognized classification within the field. In that sense, my work lies in the intersection of generic programming and precisely-typed programming: I develop new generic programming techniques in order to define my traversal operator, which then reduces the cost precisely-typed programming.

Generic programming techniques enable extremely reusable definitions. I focus on a major branch of generic programming that attains its genericity by mapping back and forth between a data type and a quasi-canonical representation of that data type built from a finite set of representation types. If a generic value is defined for each of the representation types, then it can be used for any data type that can be represented. Common generic functions include structured-based ordering, (un)serialization, pretty-printing, and uniform traversals. Generic programming simplifies definitions (often to a few function calls) and reduces their coupling to the involved data types by consolidating that correlation within in each data type's representation.

I contribute to generic programming by enriching a major branch with lightweight support for programming with anonymous subsets of a data type's constructors, *i.e. constructor subsets*. This allows the programmer to easily use more precise intermediate types within value-level definitions. Lacking this capability, some case expressions involve "impossible" cases that the programmer hopes will never arise at run-time but the type-checker still requires to be handled and well-typed. Usually, the programmer discharges the impossible cases by raising an exception exclaiming that "the impossible has happened!". There is, however, a more nuanced and costly scenario in strongly-typed languages: the well-typedness of impossible cases may introduce nonsensical and even unsatisfiable type constraints. In Haskell, this sometimes forces the programmer to either declare nonsensical and misleading type class instances or to obfuscate the case expression, function, module, or even program in order to prevent the troublesome constraints from being introduced. With my support for constructor subsets, some impossible cases can easily be eliminated altogether.

In precisely-typed programming, the programmer prioritizes the use of types to assure program correctness. The two essential elements of precisely-typed programming are 1) the precise types that strongly characterize the data at particular points in a program and 2) the ubiquitous *precise conversions* between such types. The use of precise types eliminates down-

stream impossible cases, so that the type-checker can statically confirm that a case expression or function definition handles all possible alternatives. This assures the absence of run-time errors due to non-exhaustive patterns and also avoids nonsensical and unsatisfiable type constraints.

A statically- and strongly-typed programming language is essential for precisely-typed programming, but the converse is not true. It is always possible to program within a statically- and strongly-typed programming language without actually relying on the type-checker. For example, one could represent all data with strings or even with slightly-structured trees that contain only strings within each node. Within such programs there are many invariants that could be encoded in the types but are not; the type system is ignored. While this sort of practice seems odd, it does yield some practical benefits over precise types, such as simplicity, uniformity, and portability. Indeed, "slightly-structured trees" is the basic structure of S-expressions and XML, which certainly exhibit those virtues. This identifies a trade-off involved in choosing a level of precision for the types in a program. Very precise types assure program correctness, but are less practical. Statically- and strongly-typed programming languages accommodate many degrees of type precision, where precisely-typed programming is one extreme. My main objective is to recover some of the practical benefits of less structured data when using precisely-typed programming.

I contribute to precisely-typed programming by enriching generic programming techniques to be usable within the definitions of precise conversions. Because a precise conversion's domain and codomain are unequal types, there is no automated support for expressing conversions between them. In particular, existing generic programming techniques cannot express the useful default behaviors. While programmers can define precise types and conversions without generic programming, they usually choose not to because manually defining precise conversions is only tractable in small programs with small data types. For large programs, with many data types with many constructors and many derived precise types, maintaining

the numerous precise conversions becomes untenable. My enriched generic programming techniques reduce the cost of precise conversions by automating the tedious cases that "just do the obvious thing". My approach relies on a precise conversion's domain and codomain having mostly isomorphic constructors. This precondition is not limiting in practice because programs are already often organized as a series of small passes. Because precise conversions can be simplified via my generic programming techniques, programmers get the benefits of precisely-typed programming without having to write nearly as much tedious code.

The methods I develop for using constructor subsets and generic precise conversions together reduces the cost of precise types. Constructor subsets are a variety of precise type that is useful within a single function definition. In particular, constructor subsets are essential for my application of generic programming techniques to precise conversions. In turn, generic precise conversions make it more cost-effective for the programmer to write the conversions between precise types necessary for precisely-typed programming across multiple functions. Both varieties of precise types, in the small and in the large, provide the benefits of simpler and less coupled definitions, resulting in programs that are easier to document, test, and verify. The principal benefit is that programmers can attain the benefits of precise types—both in a single definition and in the broader form of precisely-typed programming—without having to write as much tedious code.

My primary technical contribution is the novel *traversal operator for generic precise conversions*. Effective use of the operator requires *cheap constructor subsets*, which is a secondary technical contribution because it can also be independently useful. Both serve my broader agenda of simplifying large programs by reducing the cost of precise types. I include small running examples while developing the basic approach and adding support for more sophisticated data types. The enrichments make the techniques applicable to most data types commonly used in practice.

I claim that my work makes it easier for programmers working in statically- and strongly-typed functional languages to use more precise types. Specifically, I show that my traversal operator makes it easy to define precise conversions. Since programmer effort is difficult to usefully quantify, my evidence for this claim is qualitative and relies on many comparative examples. The benefits of generic programming and precisely-typed programming are clear in side-by-side comparisons. Generic programming makes definitions simpler and less coupled. By "simpler", I mean fewer explicit cases (and so usually less code) and more uniform structure. In particular, generic programming tends to focus on reusable operators that correspond to well-understood semantics (*e.g.* catamorphism) and precisely-typed programming eliminates many side-conditions. The evaluation consists, therefore, of juxtaposition of definitions without generic programming and precisely-typed programming against their variations using generic programming and precisely-typed programming via my techniques. My culminating examples use my traversal operator to define two definitions of lambda-lifting, which is a well-studied pass in functional language compilers.

I present and implement my techniques in the Haskell programming language, but the fundamental ideas are applicable to any statically- and strongly-typed programming functional language with algebraic data types. I have implemented my ideas in a Haskell library available at `http://hackage.haskell.org/package/yoko`. The details of that implementation are discussed more concisely in my paper [17].

2 Motivation

Researchers working on static type systems strive to balance two opposing objectives. They want type systems to catch as many errors as possible and to help structure programs, but they want just as much to avoid getting in the programmer's way. In this chapter, I use examples to motivate how my techniques will augment this balance with respect to algebraic data types. I give the solutions to the smaller examples here in order to provide a feel for the resulting interface. In the later chapters, I explain the details of the solutions to all examples.

I begin with a definition of *precise* type. The term is inherently relative. It indicates a comparison on the spectrum of *adequacy* central to the work of Harper et al. [19]. For example, the following generalized algebraic data type [48], when wrapped with `STLC`, is inhabited only by values encoding closed and well-typed terms of the simply-typed lambda calculus, like `identity = STLC $ Lam Var`.

```
data Tm :: (* → *) → * → * where
  Var :: v a → Tm v a
  Lam :: (v a → Tm v b) → Tm v (a → b)
  App :: Tm v (a → b) → Tm v a → Tm v b

newtype STLC ty = STLC {unwrap :: ∀v. Tm v ty}
```

Such encodings are *adequate*: well-typed values of the data type correspond exactly to the objects the data type is intended to represent [19]. An adequate type is the ultimate precise type, but a precise type is not necessarily adequate. Without the specific context of the Logical Framework as used by Harper et al. [19], adequacy is too strong a notion for general purpose programming. The *precise* terminology is more relaxed. It also tends to be used as a comparison, where adequacy in its intended context is all-or-nothing. For example, the following data type only gaurantees that the represented lambda calculus terms are well-scoped.

```
data Tm v = Var v | Lam (v → Tm v) | App (Tm v) (Tm v)

newtype ULC = ULC {unwrap :: ∀v. Tm v}
```

The `ULC` type is an adequate encoding of the untyped lambda calculus, but not of the simply-typed lambda calculus. However, it is a more precise encoding of either the untyped or the simply-typed lambda calculus than the most common first-order representation using `String`s for binders and variable occurrences. "More adequate" would be just as meaningful as "more precise", but it connotes more formality than I intend and constrasts with the usual all-or-nothing semantics of "adequate".

2.1 In-the-Small

While easier precisely-typed programming is my main goal, my solution also involves support for programming with constructor subsets. This is independently useful but not well-supported in many strongly-typed languages.

Example 1: Serializing Data Containing Cached Functions. The `C` constructor of the data type `T` uses its second field as a cache for a function determined by its first field. In general, the caching could be a significant optimization. All other types in `T` are assumed to not involve functions and to be easy to serialize.

```
data T = ... | C X (X → X) -- the first field determines the second
```

The objective is to serialize values of this data type, using the following interface.

```
class Serialize t where serialize :: t → [Bool]
```

The salient charateristic of this example is the use of a default handler (*e.g.* a generic serializer) that would be ill-typed if applied to a `C` value. Without automation, the handling of the non-`C` constructors could be tedious.

End of Example ■

The basic notion of serialization is "emarassingly generic"; it has almost nothing to do with the details of the data type. However, the generic programming technique I have adopted as a basis for my work cannot apply its generic definition of serialization to `T`. I discuss the technique in depth in Section 3.3, but it is important to know now that it is a leading generic programming: this example exacerbates one of its few comparative faults. The generic serializer defined with it as `ig_serialize`.

```
ig_serialize :: (...) ⇒ t → [Bool]
ig_serialize = ... -- defined in Section 3.3
```

I mention this now because it first occurs in this example's definition. In Haskell, the word "field" usually refers only to the arguments of a constructor that were declared with names using Haskell's lightweight support for record syntax. I instead use "field" to mean "an argument to a constructor". Record syntax is irrelevant to this dissertation but is occasionally used for convenience without ambiguity.

The problem with applying `ig_serialize` to a value of type `T` is the function space in the field of `C`. The definition of `ig_serialize` requires all types occuring within its argument's type to be themselves serializable. This usually just requires the programmer to indicate that a few auxiliary serialization instances should also be generically automated, but there is no meaningful instance of `Serialize` for the → type. Thus the simplest first attempt at serializing `T`

```
instance Serialize T where serialize = ig_serialize   -- ill-typed
```

fails with an error complaining about the lack of a `Serialize` instance for `X → X`.

It is insightful to understand why the naïve next attempt also fails in the exact same way.

```
instance Serialize T where              -- also ill-typed!
  serialize t = case t of
    C c _ → False : serialize c
    x     → True  : ig_serialize x
```

I add the Booleans to preserve the essential bijection of serialization. The intent of this definition is to capitalize on the fact that the second field of `C` is determined by the first, so only the first need be serialized. The omitted unserialization would rebuild the second field. This successfully avoids ever trying to serialize a function. However, because of the imprecise types involved in general Haskell pattern matches, this instance is ill-typed with the same error message: no `Serialize` instance for `X → X`. The crux of the matter is that while our explicit manual handler for the `C` case eliminates the actual run-time need for the serialization of the function, the argument `x` of the `ig_serialize` function still has type `T`. Since my generic programming is type-directed and the type of `x` does not encode the simple property that it is not constructed with `C`, the type error remains.

The common solution to this problem is to use the naïve instance above and declare a vacous `Serialize` instance for the function type → that does not actually define the `serialize` method. Because the `T` instance successfully guards against invoking `serialize` on a function, there will be no run-time error. However, this is exactly the kind of side-condition that precisely-typed programming is supposed to supplant! It seems simple enough to manage the side-conditions at this point, but during the life-time maintenance of the program, the side-condition can easily be overlooked in the documention or simply forgotten about. Worse still, there is the nonsensical instance of `Serialize` for functions, which other programmers might unknowingly rely on.

Constructor subsets neatly solve this problem, preventing the need for side-conditions and avoiding the nonsensical instance. In particular, they allow an instance of `Serialize` for `T` that is just a syntactic variant of the previous one but does not need the nonsensical `Serialize` instance for the → type. The following example solution imports my library as `Data.Yoko`.

```
import qualified Data.Yoko as Y

Y.yokoTH ''T  -- Template Haskell derives necessary declarations

instance Serialize T where
  serialize t = Y.precise_case (λx → True : ig_serialize x) t $ Y.one $
    λ(C_ c _) → False : serialize c
```

The basic improvement is that `x` in the new instance has an inferred type that indicates it is not constructed with `C`. Its type is therefore the subset of `T` constructors that are not `C`. Since this intuitive idea is now actually formalized in the Haskell type system, the nonsensical constraint `Serialize (X → X)` is never generated. The type-checker validates that all cases are handled in a well-typed manner without the programmer having to manage any side-conditions. The precise constructor subset type enables precisely-typed programming.

Other than the occurrence of the `one` function, this instance has nearly the exact same syntactical components as the previous naïve instance. The `precise_case` function, which simulates a more precisely-typed case expression, is defined in Section 4.2. Note that the underscore at the end of the name `C_` in the pattern is not a typo. It belongs to a type automatically derived from `T` which includes only the `C_` constructor. Thus, the lambda is total. This derived data type is generated by the Template Haskell splice `yokoTH`.

This example of serializing values of the `T` data type just involved the single definition of the `serialize` method. In this localized case, it explained how precise types can be used to simplify functions. The programmer can easily handle the troublesome `C` case and leverage the type-directed generic programming library to automatically serialize the rest of the constructors. The next section introduces an example where the concerns are less localized.

2.2 In-the-Large

My main goal is to make it easier to use a particular kind of precise type throughout large programs. Specifically, I make it cheap for the programmer to define ad-hoc variantions of data types with relatively few differing constructors. By encoding invariants in these more precise ad-hoc data types, the programmer leverages the type-checker to actually enforce the invariants. These precise types are ideally ubiquitous, but require at least as many precise conversions between them. With current technology, these conversions' definitions are untenably tedious to maintain in such large numbers. My techniques eliminate the tedium and therefore prevent precise types, a crucial element of precisely-typed programming, from getting in the programmer's way. I adapt a small example from Chakravarty et al. [8].

Example 2: Losing Weight. The objective is to define a function `rmWeights` to remove weights from weighted trees.

```
module WTree where

data Tree a w = Leaf a | Fork (Tree a w) (Tree a w) | WithWeight (Tree a w) w
```

The range of `rmWeights` must not contain any `WithWeight` constructors.

End of Example ■

The definition of `rmWeights` given by Chakravarty et al. uses an imprecise type.

```
rmWeights :: Tree a w → Tree a w                -- adapted from [8]
rmWeights t = case t of
  Leaf a          → Leaf a
  Fork l r        → Fork (rmWeights l) (rmWeights r)
  WithWeight t _  → rmWeights t
```

While it is true that the range of `rmWeights` contains no `WithWeight` constructors, its codomain still does. If any downstream functions require the absence of `WithWeight`, they would rely on a side-condition that `rmWeights` has been applied, even perhaps applying it locally. A

more precise codomain for `rmWeights` enables precisely-typed programming, eliminating the side-condition.

```
module Tree where

import qualified WTree as W

data Tree a = Leaf a | Fork (Tree a) (Tree a)

precise_rmWeights :: W.Tree a w → Tree a
precise_rmWeights t = case t of
  W.Leaf a          → Leaf a
  W.Fork l r        → Fork (precise_rmWeights l) (precise_rmWeights r)
  W.WithWeight t _  → precise_rmWeights t
```

In the definition of `precise_rmWeights`, the codomain actually enforces the absence of weights in the range. Downstream functions that cannot handle weights could use this more precise `Tree` data type as their domain and thus avoid side-conditions and potentially redundant applications of `precise_rmWeights`.

Superficially, the only cost of employing precisely-typed programming in this example is the additional declaration of the more precise `Tree` data type. While this is an "in the large" example because it is concerned with the flow of types and values between definitions throughout the program, its ramifications are diminished by its small size. In larger programs, involving richer data types, more constructors, more fields, and many properties to be encoded as more precise variants, the definition of functions like `precise_rmWeight` is untenable because there are many of them and each involves so many tedious cases. For `precise_rmWeights`, the tedius cases are for `Leaf` and `Fork`—it is obvious to the programmer what these cases ought to do. Unfortunately, mainstream languages and existing generic programming techniques cannot provide the necessary default behavior. With the larger examples, this means that precise types in the large are not a workable solution. On the other hand, existing generic programming techniques can automate the tedius cases within

the imprecisely-typed `rmWeights` function. Because of this opportunity cost, precisely-typed programming is too costly to use.

For example, the technique at the foundation of my work can define the reusable operator `compos` [7], which handles the tedius cases of `rmWeights'`. Since the traversal is side-effect free, the `pure_compos` variant suffices. I discuss the `compos` operator in depth in Section 3.2.

```
pure_compos :: (...) ⇒ (t → t) → t → t
pure_compos = ... -- defined generically in Section 3.2

rmWeights' :: Tree a w → Tree a w
rmWeights' t = case t of
  WithWeight t _ → rmWeights' t
  x              → pure_compos rmWeights' x
```

Because `pure_compos` is defined generically, the `Leaf` and `Fork` cases are effectively automated in the definition of `rmWeights'`. This definition exhibits the same default handler that is the main characteristic of Example 2.1. Thus, as I enrich my generic programming approach to be applicable to `precise_rmWeights` in the following, I take extra care in order to avoid the sort of type error demonstrated in Example 2.1. In this way, my techniques for precise types in the large also require the constructor subset types that provide precision in the small.

The eventual solution for Example 2.2 is the `precise_rmWeights'` function. It uses the `hcompos` function from my library that I show here for convenient comparison with `compos`. The "h" is for *heterogenous*, since the domain and codomain of `hcompos` can be inequal unlike `compos`. Note that `precise_rmWeights'` in the following example solution does indeed rely on `precise_case` just as does the solution of Example 2.1.

```
import qualified Data.Yoko as Y
import qualified WTree as W
import Tree

Y.yokoTH ''W.Tree
Y.yokoTH ''Tree

-- hcompos generalizes to compos to handle distinct types;
-- Invariant: soc is some subset of the constructors of s
pure_hcompos :: (...[1]) ⇒ (s → t) → soc → t

pure_hcompos = ... -- defined generically in Section 5

precise_rmWeights' :: (...) ⇒ W.Tree a w → Tree a
precise_rmWeights' t = Y.precise_case (pure_hcompos precise_rmWeights') t $
  Y.one $ λ(WithWeight_ t _) → precise_rmWeights' t
```

This generic definition of `precise_rmWeights'` fits into the same schema as all of the previous generic functions: it has a default case that needs to exclude a particular constructor. Modulo the use of `precise_case` and `one`, it is syntactically analogous to the generic definition of the less precisely typed `rmWeights'`.

In Table 2.1 I organize the four functions for solving Example 2.2 in a grid. The columns indicate if the function is precisely-typed and the rows indicate if it has a concise and modular definition. Existing generic programming methods allow the programmer to improve the definition of `rmWeights` to `rmWeights'`. As a principal benefit of this dissertation, I enrich generic programming to permit that same improvement for the precisely-typed `precise_rmWeights`; the bottom right cell is part of my objective.

	imprecise	**precise**
complicated & immodular	`rmWeights`	`precise_rmWeights`
concise & modular	`rmWeights'`	`precise_rmWeights'`

Table 2.1: Solutions to Example 2.2 organized on a grid indicating precision of type and concision/modularity of definition.

[1]The context I have left out is an artifact of my simulation of a decidable type-level equality test. This artifact is necessary for using my technique with polymorphic types like `W.Tree a w` and `Tree a`. Based on communication with the GHC implementers, I believe these constraints will soon not be necessary with upcoming GHC versions. I explain them in detail in Section 3.5.

Though the required programmer effort for precisely-typed programming becomes untenable without generic programming only in larger programs, I make Example 2.2 intentionally small so as to focus on the main idea behind enabling precisely-typed programming with precise types. In the next section, I summarize a well-known instantiation of precisely-typed programming from the literature and adopt part of its larger program as an example in which generic programming is essential for precisely-typed programming to be practical.

2.3 Nanopasses in Haskell

Sarkar et al. [39] summarized their investigation of building a compiler as a series of "nanopass". They begin with a *micropass* compiler that is composed of minimal passes, each of which implements an individual logical step of the compiler. Their micropasses qualify as precisely-typed programming because they use precise types between some passes to enforce the intermediate program grammars (though not statically). Sarkar et al. claim numerous benefits derived from their micropass formulation of precisely-typed programming. They summarize those beneifts as easier to understand (*i.e.* simpler) and easier to maintain. They further refine the micropasses into *nanopass* by reducing the size of passes' definitions. This is accomplished by their novel *pass expander*, which automates the tedious cases much like `compos` does. In this dissertation, I translate the essence of nanopasses from their syntax-based extension of Scheme to Haskell using modern generic programming foundations. This culminates in an enrichment of `compos` that generalizes the pass expander.

The nanopass (and micropass) compiler of Sarkar et al. [39] includes many *simplification* and *conversion* passes such as the elaboration of pattern matching syntax and the conversion of higher-order functions into closures. Simplification passes transform from one grammar to a smaller grammar that excludes the simplified syntacitc forms. Conversion passes transform between two mostly equivalent grammars. The grammars' similarity enables the major improvement of nanopasses over micropasses: the programmer can omit tedius cases from conversions because the pass expander infers them.

The nanopass work of Sarkar et al. is a well-known example of the benefits of precisely-typed programming. Yet they based their work in the untyped programming language Scheme! To achieve precisely-typed programming in Scheme, they extended the Scheme language using the `syntax-case` mechanism of Dybvig et al. [14]. The extension most essential to precisely-typed programming that they added was support for enforced grammars (though not statically). This involves a `define-language` syntax for declaring grammars and related syntaxes for pattern-matching and for the "templates" that are used to construct terms. Haskell's native support for data types provides comparable mechanisms; such restricted grammars are straight-forward to encode as a mutually recursive data types. Haskell is therefore a more natural place to start when investigating precisely-typed programming.

The `define-language` syntax does, however, provide some mechanisms without counterparts in Haskell, such as declared classes of pattern variables (*i.e.* metavariables) enforcing, *e.g.*, that pattern variables like $\texttt{x}_n$ can only match occurrences of variable names, $\texttt{e}_n$ can only match occurrences of expressions, and $\texttt{n}_n$ can only match integer literals. They also provide mechanisms comparable to view patterns [1, §7.3.5], including a special type-directed implicit syntax comparable to the *implicit calculus* of d. S. Oliveira et al. [11]. However, these mechanisms' benefits are minor compared to the basic strong-typing of data types inherent to Haskell and the pass expander that I implement using IG.

The pass expander underlying nanopasses is less recognized as an early instance of generic programming. Since Haskell provides many other elements of nanopasses, the pass expander is the more technically interesting part with respect to this dissertation. I take the rest of that work as evidence supporting the benefits of precisely-typed programming; it is the inspiration for my dissertation. I compare the details of the techniques demonstrated by Sarkar et al. to my Haskell implementation within my evaluation in Section 7.4.

Because the simplification and conversion passes considered by Sarkar et al. are so Scheme-centric, I will also solve the more generally interesting example of lambda lifting, which would be consideration a conversion pass.

Example 3: LAMBDA-LIFTING. Implement a precisely typed lambda lifting function.

End of Example ■

I solve this example twice, once in Section 5.5 and again in Section 7.3. The first solution demonstrates a concise lambda lifter over a de Bruijn indexed term representation. The use of de Bruijn indices exacerbates one of the restrictions imposed by the `hcompos` interface. When discussing that example, I focus on how its concision follows from using my technical contributions and also on the demonstrated difficulty of accomodating the `hcompos` interface. In the second solution, I reimplement a lambda lifter from the literature, specifically that of Jones and Lester [23]. They designed that lambda lifter to be modular, so it is not as dramatically concise as my first solution. However, by using multiple steps, the example shows precisely-typed programming in action. In my implementation, I use precise types to enforce invariants they specifically lament not being able to encode in the type system. Without `hcompos`, the use of those types would be untenable, which is something those authors implicitly expressed in their paper. The generic programming benefits of `hcompos` and the precisely-typed programming benefits it enables yield a dramatic side-by-side comparison of the old code and my new code.

Jones and Lester [23, §4] also discusses the lack of support for precisely-typed programming in Haskell (without using that terminology, of course). They share some design decisions regarding the data type over which they define their lambda lifter. They begin with `Expression` where `Name` and `Constant` are language-specific.

```
data Expression
  = EConst Constant -- e.g. 7 or True or +
  | EVar Name
  | EAp Expression Expression
  | ELam [Name] Expression
  | ELet IsRec [Definition] Expression

type Name = String
type IsRec = Bool
type Definition = (Name, Expression)
```

They note that this data type's fixed use of `Name` for binders lacks the flexibility that is important for later passes like type inference. They quickly advance to `Expr`.

```
data Expr binder
  = EConst Constant
  | EVar Name
  | EAp (Expr binder) (Expr binder)
  | ELam [binder] (Expr binder)
  | ELet IsRec [Defn binder] (Expr binder)

type Defn binder = (binder, Expr binder)

type Expression = Expr Name
```

They point out that `Expression` is an instance of `Expr` while the data type used for type inference could be `Expr (Name, TypeExpr)`. Note that `Expression` is now just one type in the parametric `Expr` family of types[2]. We shall see in the next chapter than conversions between instances of a parametric family of types are still beyond the reach of `compos`.

[2]I follow the convention from the Logical Framework of using "family of types" to refer to any type of higher-kind. I will only use "type family" to refer directly to the Haskell feature of the same name.

Jones and Lester are still not satisfied with their data type because they intend to annotate nodes in the tree other than just lambdas. They end up using both `Expr` and `AnnExpr`.

```
type AnnExpr binder annot = (annot, AnnExpr' binder annot)

data AnnExpr' binder annot
  = AConst Constant
  | AVar Name
  | AAp (AnnExpr binder annot) (AnnExpr binder annot)
  | ALam [binder] (AnnExpr binder annot)
  | ALet IsRec [AnnDefn binder annot] (AnnExpr binder annot)

type AnnoDefn binder annot = (binder, AnnExpr binder annot)
```

The mutual recursion between `AnnExpr` and `AnnExpr'` guarantees that each node in the tree is paired with a value of `annot`. At this point, the authors lament that `AnnExpr` and `Expr` duplicate the constructors. And while `Expr binder` is almost isomorphic to `AnnExpr binder ()`, they say it is too tedious to deal with the ubiquitous `()`s when constructing and destructing `Expr`s.

Finally, they define their lambda lift function with the following signature.

```
lambdaLift :: Expression → [SCDefn]

type SCDefn = (Name, [Name], Expression)
```

They remark that the `Expression`s in `SCDefn` will never be constructed with the `ELam` constructor and that in the Haskell type system (of 1991) there was no way to express/enforce this fact. There still is no best mechanism to do so in a general way. My contribution for subsets of constructors leverages current generic programming fundamentals as a lightweight solution to this problem. Where they mention adding "yet another new variant of `Expr`" without the `ELam` constructor, they do not even stop to consider that option—it is implicitly and immediately dismissed as impractical. My techniques make it viable. In particular, Jones

and Lester could indeed define another such data type as I do in Section 5.5, or they could even define `SCDefn` as follows.

```
import qualified Data.Yoko as Y

Y.yokoTH ''Expr  -- recall that Expression = Expr Name

type SCDefn = (Name, [Name], Y.DCs Expression Y.:-: Y.N (ELam_ Name))
```

In my library, `DCs` is a type family that maps a data type to the set of all its constructors, `:-:` is a set difference operator, and `N` builds singleton sets. In this example, `ELam_` is the name of both the data type and its only constructor that is derived from the `ELam` constructor. Thus the type literally reads as *the constructors of* ***`Expression`*** *except* ***`ELam`***. Either way, using yet another variant or explicitly using a constructor subset, the definition of `lambdaLift` can use the `hcompos` function to handle the tedious cases.

I have recounted the design considerations of Jones and Lester because it is a concrete testimony that the concerns of precise types arise in practice with even well-known functions and extremely capable programmers. The work of Sarkar et al. further testifies that relying on mechanisms to directly address these concerns pays off.

2.4 My Work on the Rosetta Toolset

While this chapter has so far motivated my work by referring to established ideas and examples from the literature, the actual research unfortunately happened backwards. I invented my ideas to solve an engineering problem and then backed up to find the relevant literature. I will overview the engineering problem and sketch the solution because it demonstrates how my techniques can influence even the high-level design of programs.

I am an implementor of the Rosetta specification language [3]. Rosetta is intended to be a industrial strength language in wide-spread use, and its intended users are distributed along multiple spectrums from design engineers "on the front lines" to formal methods

practioners proving more general theorems. In order to contend with its various peers along these spectrums (like VHDL, PVS, Verilog, Coq, *etc*), ROSETTA must include a cornucopia of powerful language features. Even just the features of its type systems that are well-understood in isolation, such as dependent types, implicit subtypes, first-class modules, and type reconstruction, lead to open problems in the respective fields when integrated [42].

The ROSETTA team has decided to adopt my techniques in order to gradually diffuse the complexity of the full ROSETTA feature set. In particular, some members of the implementation team are researching the integration of subsets of the aforementiong features while others are researching the novel language constructs specific to ROSETTA. As a result, the intermediate representation of the full language is in flux. This incurs the software engineering problem of constantly updating parts of the toolchain, like the parser and the pretty-printer. Moreover, these changes to the intermediate representation tend to be experimental, since we are conducting research on open problems. My techniques make it cheap for an implementor to create a secondary intermediate representation that excludes the troublesome features that are irrelevant to that implementor's current research. Prior to my techniques, this was prohibitively expensive, since the abstract syntax for the full ROSETTA language at the time of writing this disseration contains 17 Haskell data types with 92 constructors.

The major engineering issue is that when a ROSETTA implementor forks the abstract syntax, they usually need to also fork other functionality of the tool set, such as the parser and sometimes even the type-checker (which is itself experimental!). This is, for the most part, tedious busy work—most of the syntax can still be parsed in the same way. My work mitigates much of this work by making it inexpensive to translate between the full abstract syntax and the implementors' experimental forks of it.

That is how my work will help the ROSETTA implementors in their day-to-day research: by automating the tedious parts of converting between large intermediate representations data types. However, the ROSETTA team's long-term plan is actually to provide a suite of tools

that each handles only a subset of the ROSETTA feature set, akin to the PVS tool suite [37]. This is a practial approach to the complex open problems inherent in the combination of all those rich language features; we do not anticipate a single monolithic tool capable of supporting all valid specifications within the entire ROSETTA language. Thus, my work is not only useful for the current research that is devoloping ROSETTA tools, it will remain within the code that interfaces between the various tools in the suite.

2.5 Limitations

I claim my work only reduces of the cost of precise types, as opposed to eliminating the cost, for three reasons. As I discuss during the evaluation of my work, the time and space overhead of my techniques could prohibit their use in some programs. There is, though, no obvious theoretical limitation; continued research on optimizing generic programming techniques in Haskell will likely reduce my library's overhead. Similarly, the type errors incurred by mistaken uses of my API have a poor signal-to-noise ratio, and this is again a drawback of concern common to the field of generic programming. Finally, as I explain when relating my work to that of Bahr and Hvitved [4], my techniques address only one detriment of precise types. In particular, my techniques are not a solution to Wadler's *expression problem* [50].

My techniques could be extended to address the expression problem, but that has not been my objective in this work. It could even be simpler to make my library interoperable with approaches like that of Bahr and Hvitved that do address the expression problem. Even without the reuse specific to the expression problem, my work significantly reduces the costs of using precise types in large Haskell programs.

2.6 Summary

In this chapter, I defined and explained the benefits of precise types in the small and in the large. This results in three unsolved examples, one of which focuses on in-the-small (*i.e.*

constructor subset) and the other two on the more important in-the-large (*i.e.* precisely-typed programming). Two of the examples are from existing literature [8, 23].

Having here motivated and established some concrete objectives, I explain the technical foundations of my general approach in the next chapter. The novel technical work comprising my contributions is developed in the following three chapters. I extend the foundation with cheap constructor subsets and generic precise conversions, first on only simple data types and then in the presence of richer data types. These chapters include solutions to all of my concrete examples as well as many ad-hoc examples that assist with the development and explanation of the ideas.

To evaluate my work, I compare and constrast the solutions enabled by my techniques against the conventional alternatives. My arguments are merely qualitative, since legitimate empirical studies are beyond my resources. Even so, I make a compelling argument for the worthwhileness of my library's unique capabilities for a widely-recognized subset of Haskell programmers: those prioritizing correctness and rapid development. I conclude that I have made it easier for programmers working in the statically- and strongly-typed Haskell language to use more *precise* types. Moreover, the underlying ideas are not deeply entangled with Haskell; they can be ported to other languages that have users with the same cultural priority on correctness and comparable algebraic data types.

3 Background

As a review of the technical definitions that I build on, I recall the definition and usage of applicative functors [34], the `compos` operator [7], the `instant-generics` generic programming approach [8], and the automated embedding and projection into sums types [44]. I then summarize my use of type-level programming in Haskell. I assume that the reader is familiar with Haskell, including type classes, monads, and the more recent type families [40].

3.1 Applicative Functors and Traversable Functors

The standard `Control.Applicative` and `Data.Traversable` modules are inspired by the work of McBride and Paterson [34], which shows how their interface precisely factors a widespread class of data structure-centric algorithms.

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

(<$>) :: Applicative i ⇒ (a → b) → i a → i b
(<$>) = fmap
```

Applicative functors are *not necessarily* computations that yield values (*i.e.* monads), but such computations are a useful subset of applicative functors. The `pure` function corresponds to monadic `return`, and the `<*>` function is a weaker version of `>>=` with its own related laws. For any monad, `f <*> m = f >>= λf → m >>= λm → return (f m)`. On the other hand, there exist applicative functors that are not monads. The useful `Const` data type is an example.

```
newtype Const a b = Const {unConst :: a}

instance Monoid a ⇒ Applicative (Const a) where
  pure _ = Const mempty
  Const a1 <*> Const a2 = Const $ a1 `mappend` a2
```

This data type emphasizes the major difference from monads. The continuation argument of `>>=` allows later side-effects to depend on the value of previous computations. Such a value would be impossible to provide with the `Const` data type—it cannot be a monad. Where the monadic interface allows the computational structure to dynamically depend on the computed values, the applicative interface is restricted to static computations.

The `Applicative` interface becomes much more useful in the presence of the `Traversable` class. Its `traverse` method is a structure-preserving map like `fmap` that uses `Applicative` to also correctly thread side-effects in a "left-to-right" manner.

```
class Traversable t where
  traverse :: Applicative i ⇒ (a → i b) → t a → i (t b)
```

Instances of the `Traversable` class specify how a particular functor `t` can distribute with any applicative functor. It creates an applicative computation that is structured similarly to the input value of type `t a`. Its type doubly generalizes the standard `mapM` and `sequence` functions: the `Monad` restriction is relaxed to `Applicative` and the list type is generalized to any `Traversable` type. Therefore, it can be used with non-monadic effects like that of `Const` and operate or more interesting structures like trees.

```
import Tree (Tree(..)) -- unweighted trees from Section 2.2

instance Traversable Tree where
  traverse f (Leaf a)   = Leaf <$> f a
  traverse f (Fork l r) = Fork <$> traverse f l <*> traverse f r
```

For example, these instances for `Const` and `Tree` can be combined to flatten a `Tree` into a list. The list is the monoid carried by the `Const` applicative functor. Other monoids are defined in `Control.Applicative` that are often useful: `Any` and `All` for the disjunctive and conjunctive monoids over Booleans and the `Sum` and `Product` monoids for number types.

```
flatten :: Tree a → [a]
flatten = unConst ∘ traverse Const
```

Or a `Tree` could be traversed with the `State` monad in order to uniquely number its elements. This is assuming that the `State` monad has been declared directly as an instance of `Applicative`. For some monad that has no such instance, one could use the `Control.Applicative.WrappedMonad` newtype.

```
renumber :: Tree a → Tree (Int, a)
renumber = flip runState 0 ∘ traverse gensym
  where gensym a = get >>= λi → put (i + 1) >> return (i, a)
```

As McBride and Paterson show with more examples like these two, the `Applicative` interface enables the `traverse` operator to model the shape of many diverse traversal functions. In next section, I discuss another `Traversable`-like class that use `Applicative` so that it can to model a distinct class of traversal functions. In the following, it will be useful for the reader to observe that some of my main classes have similar semantics to `Traversable`.

3.2 The `compos` Operator

Bringert and Ranta [7] introduce the `compos` operator for defining "almost compositional functions". In this context, *compositional* means that the result of a function for a constructor of a data type is determined by that constructor's non-recursive content and the result of that same function on all the immediate recursive occurences of the constructor. The `compos` operator provides just the necessary interface to a data type to use compositionality as a definitional mechanism. It has the following signature, assuming that `t` is a singly recursive data type.

```
class Compos t where compos :: Applicative i ⇒ (t → i t) → t → i t

pure_compos :: Compos t ⇒ (t → t) → t → t
pure_compos f = runIdentity ∘ compos (pure ∘ f)
```

The operator is useful for defining the uninteresting cases of bottom-up transformations. For example, I used the pure variant to define the `rmWeights` function from Example 2.2 without

explicitly handling the tedius cases of leaves and branches. Those cases are tedius because they include no weights. However, because they contain recursive occurrences into which the transformation must be propagated, they still need definitions. Indeed, those cases ought to be compositional, so `pure_compos` suffices. We can define a similar function that applies an adjustment to the weights instead of removing them.

```
adjustWeight :: (Double → Double) → WTree Double a → WTree Double a
adjustWeight adjust = f where
  f (WithWeight t w) = WithWeight (f t) (adjust w)
  f x = pure_compos f x
```

Without `compos`, this definition would be written with the following two cases, which are equivalent to the relevant inlining of `pure_compos`.

```
  f (Leaf a) = Leaf a
  f (Fork l r) = Fork (f l) (f r)
```

Bringert and Ranta call functions like `rmWeight` and `adjustWeight` *almost compositional* because most of their cases are compositional. As a result, there are significant benefits to replacing those cases with `compos`. The relative concision of the definition prevents those bugs due to the inattentive programming invited by the tedius cases (*e.g.* `Fork (f l) (f l)`). It also simplifies reasoning, since all compositional cases are handled uniformly and are explicitly identified as compositonal. Finally, it reduces the coupling with the definition of `WTree`. In particular, this definition of `adjustWeight` no longer explicitly depends on any details of those constructors which should behave compositionally. Since changes in those details need not be reflected in the definition of `adjustWeight`, maintainance is easier. For example, the `Fork` constructor could be changed to allow n-ary branching without any change to `adjustWeight`. Compositionality is the semantics of most cases for many interesting functions.

Beacuse the type of `compos` preserves the input type, the function cannot be used, for example, to change the type of the weights in a tree. Even though the following definition

of `changeWeight` is so similar to `adjustWeight`—the tedious cases are even syntactically identical—the type of the `compos` operator is too strict for it to be used.

```
import WTree

changeWeight :: (w → w') → Tree a w → Tree a w'
changeWeight f = work where
  work (WithWeight t w) = WithWeight (work t) (f w)
  -- work x = pure_compos work x -- intuitively meaningful but ill-typed!
  work (Leaf a)   = Leaf a
  work (Fork l r) = Fork (work l) (work r)
```

From this example, it is evident that `compos` cannot be used to automate conversions between similar data types, even different instances of the same parametric data type. Applying GP to such conversions is the main technical challenge I overcome in this dissertation. The eventual solution is a generalization of the `Compos` class. For the linear presentation of the technical development, however, I first introduce the starting point of my GP techniques. In particular, the next section includes a generic (*i.e.* very reusable) definition of `compos`.

3.3 The `instant-generics` Approach

This section summarizes the `instant-generics` approach [8], the foundation of my generic programming. It is expressive [33] and performant [8, §5]. Existing generic programming techniques are sufficient for generically defining the `compos` function of Bringert and Ranta [7]. These same techniques convey the majority of the reusability of my techniques; my extensions just enable more flexible use.

The `instant-generics` framework derives its genericity from two major Haskell language features: type classes and type families. I demonstrate it with a simple example type and two generically defined functions in order to set the stage for my extensions. In doing so, I introduce my own vocabulary for the concepts underlying the `instant-generics` Haskell declarations.

Listing 3.1: The core `instant-generics` interface.

```
-- 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 Representable a where
  to :: Rep a → a
  from :: a → Rep a

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

The core `instant-generics` declarations are listed in Listing 3.1. 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. [8]), `Rec`, `U`, `:*:`, `C`, 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 `Representable` class converts between a type and its `Rep` structure. 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.

The Sum-of-Products Representation Types

Each representation type models a particular structure common in the declaration of data types. 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. 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 [55, 32], 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 [13], 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)

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 added to the sum-of-products representation types only to define another class of generic values, such as `show` and `read`. I call these types *constructor types*. They are analogous to a primary component of my cheap constructor subsets, and so will be referenced in Section 4.1. 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 `Representable` instance for `ULC` is almost entirely determined by the types.

```
instance Representable 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
```

Two Example Generic Definitions

The `instant-generics` approach can generically define the `compos` operator for singly recursive types.

```
class Compos t where compos :: Applicative i ⇒ (t → i t) → t → i t
```

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

For the `Rec` case, the original function is applied to the recursive field, but `compos` itself does not recur. As shown in the definition of `rmWeights'` at the end of Section 2.2, the programmer, not `compos`, introduces the recursion. With this generic definition in place, the `Compos` instance for `ULC` is a straight-forward delegation.

```
ig_compos :: (Applicative i, Representable a, Compos (Rep a)) ⇒
             (a → i a) → a → i a
ig_compos f = fmap to ∘ compos (fmap from ∘ f ∘ to) ∘ from

instance Compos ULC where compos = ig_compos
```

Further, we can generically define the `serialize` method of the `Serialize` class from Section 2.1.

```
instance Serialize a ⇒ Serialize (Dep a) where serialize (Dep x) = serialize x
instance Serialize a ⇒ Serialize (Rec a) where serialize (Rec x) = serialize x
instance                Serialize U       where serialize _       = []
instance (Serialize a, Serialize b) ⇒ Serialize (a :*: b) where
  serialize (x :*: y) = serialize x ++ serialize y
instance Serialize a ⇒ Serialize (C c a) where
  serialize (C x)     = serialize x
instance (Serialize a, Serialize b) ⇒ Serialize (a :+: b) where
  serialize (L x)     = False : serialize x
  serialize (R x)     = True  : serialize x
```

With these instances, the `ig_serialize` generic function is immediate. As Chakravarty et al. [8, §5] show, the GHC inliner can be compelled to optimize away much of the representational overhead.

```
ig_serialize :: (Representable t, Serialize (Rep t)) ⇒ t → t
ig_serialize = serialize ∘ from
```

As demonstrated with `compos` and `serialize`, 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 `to` and `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.

Limitations

The `instant-generics` approach is not the perfect solution. Specifically for my interests, there are data types useful for precise types that `instant-generics` cannot fully represent with the representation types in Listing 3.1. *Generalized Algebraic Data Types* (GADTs) are the most appealing variety of data type that cannot be faithfully represented. The use of GADTs was demonstrated in Section 2 by the example of adequate encodings of terms in the simply-typed lambda calculus.

The two data types from that section both use the parametric higher-order abstract syntax [9], which is a well-studied representational technique for binding. An alternative well-scoped encoding of the untyped lambda calculus can be defined instead using *nested recursion* [5].

```
data Tm free = Var free | Lam (Tm (Maybe free)) | App (Tm free) (Tm free)
```

In this data type, the parameter corresponds to the type of free variables that can be used in the term representation. It is not the general type of free variables, like `String`, but rather the precise type of usable names, like `{0, 1, 2}` for a term with three free variables. The application of `Maybe` within the `Lam` constructor achieves this semantics and is characteristic of nested recursion. `Tm` could also be enriched to be a GADT that enforces simply well-typedness.

GADTs and nested recursion are not supported by `instant-generics`. They are, however, better supported by two extensions of IG: `generic-deriving` [30] and the work presented by Magalhães and Jeuring [29]. `generic-deriving` represents `* → *` types like `instant-generics` represents `*` types. Since nested recursion takes place in the parameter, a representation that tracks the parameter can better handle it. Magalhães and Jeuring [29] can represent those GADTs that only use existential types in indices (*i.e.* as phantom type variables), which includes well-typed lambda calculi representations. I believe that my

extensions can also be applied to these more expressive variants. I discuss the details further in Section 6.3.

Summary

The `instant-generics` approach is a successful GP approach because it relies on two major features of GHC Haskell: type classes and type families. By using the major features, it inherits GHC's strengths. This includes performance, as the optimizer is aggressive with dictionaries and case expressions, as well as extensibility, because classes and families are *open* definition mechanisms. This is why I adopt it as the basis of my GP techniques.

There are alternative approaches that might better support more precise types, but `instant-generics` is a solid and simple foundation with which to start this line of technical work. In the next section, I discuss a well-known approach unrelated to `instant-generics` from which I will also adopt some concepts in the next chapter. In particular, these concepts make it easier to program with constructor subsets.

3.4 Automatic Embedding and Projection

Swierstra [44] discusses a different way to represent data types: two-level data types, in which the recursion and the sum of alternatives are introduced independently. The two-level data type for the untyped lambda calculus is defined as `Mu SigULC`. Data types like `SigULC` are called *signatures*.

```
newtype Mu f = In (f (Mu f)) -- the standard type-level fixed-point operator

data SigULC r = Var Int | Lam r | App r r
type ULC = Mu SigULC
```

Because the recursion is factored out via `Mu`, the `r` parameter of a signature stands for what would have been its recursive occurrences. This key idea is discussed by Meijer et al. [35].

The two-level approach can implement the `compos` function as a specialization of `traverse`.

```
compos :: (Traversable f, Applicative i) ⇒ (Mu f → i (Mu f)) → Mu f → i (Mu f)
compos f (In x) = In <$> traverse f x
```

The equivalence follows from the semantics of `compos` and the `r` parameter. Because the type of `traverse` is more general than that of `compos`, the `traverse` operator can be used more flexibly. As Sheard and Pasalic [41] demonstrate, two-level types yield many software engineering benefits. But the approach does inherently provide any genericity. Though `compos` can now be defined as `traverse`, two-level types provide no means for specifying a generic definition of `traverse` comparable to the `instant-generics` definition of `compos`.

The `regular` approach of van Noort et al. [46] merges the fundamentals of `instant-generics` and two-level types by using `* → *` versions of the core `instant-generics` declarations listed in Listing 3.1. For example, instead of `:+:`, the `* → *` sum is

```
data (:++:) f g a = L (f a) | R (g a)   -- c.f. :+: from the previous section
```

but the essential semantics is the same.

Sheard and Pasalic [41] do not use any representation types like `Mu`, `:++:`, `:**:`, or so on. They instead introduce a fresh data type isomorphic to `Mu` for each new data structure and declare signatures and explicit sums of those signatures. This is presumably a trade-off of reuse to make the code more accessible.

Swierstra [44] uses `Mu` for recursion and `:++:` for sums but not `:**:` for products. Instead of summing products representing fields, Swierstra follows Sheard and Pasalic and sums user-defined signatures. The difference is that Swierstra reuses `:++:` to combine the signatures, and so he uses the `Sub` class to hide the representational details.

```
class Sub sub sup where
  embed   :: sub a → sup a
  project :: sup a → Maybe (sub a)

embedIn :: Sub sub sup ⇒ sub (Mu sup) → sup (Mu sup)
embedIn = In ∘ embed
```

For example, `SigULC` might be decomposed into a signature for lambdas and applications and a separate signature for variables, since variables are often used independently of the other two.

```
type SigULC = V :++: LA
data V  r = Var Int
data LA r = Lam r | App r r
```

The identity function is represented as `In (R (Lam (In (L (V 0)))))`. This immediately motivates abstracting over the `L` and `R` constructors, since further uses of `:++:` to add other signatures to the sum quickly amplifies the noise. This is remedied by the `Sub` class. With the appropriate instances, the identity function becomes `embedIn (Lam (embedIn (V 0)))`. This value inhabits any sum of signatures that includes `V` and `LA`; it therefore has the type `∀f. (Sub V f, Sub LA f) ⇒ Mu f`. Uses of `project` incur similar types.

The common instances of `Sub` use the overlapping instances GHC extension in order to check for type equivalence. This also requires that `:++:` is only nested in a right-associative way and that sums are terminated with `Void`. The `:++:` operator is effectively treated like a cons for type-level lists of signatures.

```
data Void r

-- NB overlapping instances; all sums must be right-nested and Void-terminated
instance Sub f (f :++: g) where
  embed = L
  project (L x) = Just x
  project _ = Nothing
instance Sub f h ⇒ Sub f (g :++: h) where -- NB f /= g, b/c overlap
  embed = R embed
  project (R x) = project x
  project _ = Nothing
```

My extensions to `instant-generics` and Swierstra and the further developments in Section 4.2 involve non-trivial type-level programming. Thus I first introduce the newer Haskell features I use as well as some conveniences that I assume for the sake of presentation.

3.5 Type-level Programming in Haskell

Promotion of data types to *data kinds* is a recent extension of GHC [55]. The definitions in this paper use the genuine `Bool` data kind where `True` and `False` are also types of kind `Bool`. I also uses the following straight-forward declarations of the type-level conditional and disjunction as the `If` and `Or` type families.

```
type family If (b :: Bool) (t :: *) (f :: *)
type instance If True  t f = t
type instance If False t f = f

type family Or (b1 :: Bool) (b2 :: Bool) :: Bool
type instance Or False b2 = b2
type instance Or True  b2 = True
λnoindent

I explicitly simluate the ‘Maybe‘ data kind, since some of my operations (λieλ
type families) over the ‘Maybe‘ kind would require kind variables, which are
not yet fully supported by GHC.

λbegin{haskellq}
data Nothing
data Just a

type family MaybeMap (f :: * → *) (x :: *) :: *
type instance MaybeMap f (Just x) = Just (f x)
type instance MaybeMap f Nothing = Nothing

type family   MaybePlus (a :: *) (b :: *)
type instance MaybePlus Nothing  b = b
type instance MaybePlus (Just a) b = Just a
```

I use `MaybeMap` like a type-level `fmap` and `MaybePlus` uses the `Maybe a` monoid for backtracking in type-level computation. These families would ideally have the respective kinds `(∀k1 k2. (k1 → k2) → Maybe k1 → Maybe  k2)` and `(∀ k. Maybe k → Maybe k)`.

The type-level programming necessary for my generic programming extensions is only partially supported in Haskell. For clarity of presentation, I assume throughout this dissertation that these features are already available:

1. a type family implementing decidable type equality, and

2. direct promotion of strings to the type-level.

My current implementation simulates these features without exposing them to the user. The user cannot observe type-level strings at all, and the type equality test is only exposed as the occasional redundant-looking constraint `Equal a a` on some type variable.

The GHC implementers are currently discussing how to implement these features. In particular, the decidable type equality will likely be defined using *closed* type families, with the common fall-through matching semantics from value-level patterns.

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

This type family definition incurs the following table of example type equalities, assuming that the `a` and `b` type variables are both in scope. The $\perp$ cells result in a type error due to that fact that `Equal` is undefined for those indexes; essentially, more information is needed before the equality can be decided.

A	B	Equal A B
Int	Int	True
Int	Char	False
Int	a	⊥
a	b	⊥
a	a	True
[Int]	[Char]	False
[a]	[a]	True
Maybe Int	[Int]	False

I simulate this definition of `Equal` in a way that requires all potential arguments of `Equal` to be mapped to a globally unique type-level number. The `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 [24, `#TTypeable`] uses the same mapping. This simulation of `Equal` is undefined for some arguments for which the ideal `Equal` is defined. One example case is when both arguments are the same polymorphic variable. I add a column to the previous table showing the result using the simulated `Equal`.

A	B	Equal A B	simulated Equal A B
Int	Int	True	True
Int	Char	False	False
Int	a	⊥	⊥
a	b	⊥	⊥
a	a	True	⊥
[Int]	[Char]	False	False
[a]	[a]	True	⊥
Maybe Int	[Int]	False	False

Note that the simulation is strictly less than the ideal `Equal` with respect to a partial order where $\perp$ is the least element and `True` and `False` are incomparable. In particular, the simulation can only determine concrete types to be equal; it is incapable of identifying two uses of equivalent type variables in its arguments. Thus a `a ~ b` constraint would imply that the ideal `Equal` is `True` at `a` and `b` but not so for the simulated one. However, if each ~ constraint were to be accompanied by a corresponding `Equal` constraint with `True`, such as `Equal a b ~ True`, then the simulation becomes otherwise entirely accurate. I occasionally use such constraints in this dissertation, but they will no longer be needed once upcoming versions of GHC support closed type families.

Furthermore, the simulation is only defined for concrete types that have been reflected with `yoko`'s bundled Template Haskell, which I tacitly assume for all of my examples. This is a meager assumption since Template Haskell is always (within this dissertation) used to derive the generic structure of the data types anyway.

In the next chapter, I extend `instant-generics` to support constructor subsets. The key ingredient is an intermediate representation that uses `:++:` sums to represent subsets without using `:**:` products to represent constructor fields. In the same way as Swierstra uses `Sub`, I use analogous classes to let the programmer manipulate sums without exposing their irrelevant representational details. However, I define more complicated instances that respect the symmetry of `:++:` as a sum; I do not cast it as a type-level list. Furthermore, I define a class that permutes sums in order to interpret them as unordered type-level sets.

Page left intentionally blank.

4 Constructor Subsets

I must extend `instant-generics` in order to generically define `hcompos`. Existing generic programming techniques cannot in general be used to define consumer functions with unequal domain and codomain types, which is an essential quality of `hcompos`. 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, I 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; I define it in Section 5 below. This improved restriction is enabled by my two extensions to `instant-generics`.

4.1 Disbanding Data Types

My first extension is the basis for clear and modular use of `hcompos`. 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 `hcompos`. Specifically, this extension makes it possible for the programmer to use individual constructors independently of their siblings from the data type declaration. I therefore call the resulting generic programming approach `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 of these mechanisms are defined in Section 4.2 below.

My second extension enables the generic definition of `hcompos` 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 (iii) because they do not reflect enough information about data types. My

extension reflects constructor names at the type-level, which is how my generic definition of `hcompos` automatically identifies corresponding constructors.

Delayed Representation

My 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. For example, the `ULC` data type needs three fields types, one for each constructor.

```
data Var_ = Var_ Int
data Lam_ = Lam_ ULC
data App_ = App_ ULC ULC
```

As will be demonstrated in Section 5.5, programs using the `hcompos` approach use the fields types directly. It is therefore crucial that fields types and their constructors be predictably-named.

The `yoko` interface for data type reflection is listed in Listing 4.1. 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 *disbanded* 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 of its `Generic` class. The inverse mapping, from a fields

Listing 4.1: The `yoko` interface for data type reflection. I presume type-level strings and reuse the `Generic` class from `instant-generics` (imported here as `IG`).

```
-- analog of instant-generics' C representation type
newtype N dc = N dc

-- maps a type to a sum of its fields types
type family DCs t

class DT t where disband :: t → DCs t

-- maps a fields type to its original type
type family Codomain dc
class (IG.Generic dc, DT (Codomain dc)) ⇒ DC dc where
  rejoin :: dc → Codomain dc

-- maps a fields type to its tag
type family Tag dc :: String
```

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 t = case t of
    Var i     → L    (Var_ i)
    Lam e     → R (L (Lam_ e))
    App e0 e1 → R (R (App_ e0 e1))

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

instance DC Var_ where rejoin (Var_ i)     = Var i
instance DC Lam_ where rejoin (Lam_ e)     = Lam e
instance DC App_ where rejoin (App_ e0 e1) = App e0 e1
```

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, though, that the `Rep` instances for fields types never involve sums. Because every fields type has one constructor, 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 straight-forward 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 `:+:` type is its own representation. However, where `Rep` is homomorphic with respect to `:+:`, it eliminates the `N` structure.

```
type instance Rep (N dc) = Rep dc
instance Generic dc ⇒ Generic (N dc) where
  to         = N ∘ to
  from (N x) = from x
```

With the instances of `Rep` for the representation types for sums, applying `Rep` to a disbanded data type 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; I omit this for brevity. In this way, the delayed representation could preserve the `instant-generics` structural interpretation. In general, with an equivalence $\cong$ that ignores the `C` type, $\forall$`t. Rep t` $\cong$ `Rep (DCs t)`.

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. I define the `Tag` type family (bottom of Listing 4.1) 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`, I declare the following `Tag` instances for the corresponding fields types. The `Constructor` class's `conName` method could be defined using an interface to the type-level strings that supports *demotion* to the value-level.

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

4.2 Implicit Partitioning of Disbanded Data Types

This section develops the implicit partitioning of disbanded data types, which builds on the delayed representation extension in order to enable modular and clear use of `hcompos`. 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 the solution for Example 2.1 as given in Section 2.1, the `precise_case` function implicitly partitions the constructors of the `T` type in the definition of `serialize`. Recall that the following definition of `serialize` for `T` would be ideally concise, but it is ill-typed.

```
instance Serialize T where serialize = serialize_T

serialize_T :: T → [Bool]
serialize_T t = case t of
  C c _ → False : serialize c
  x     → True  : ig_serialize x   -- NB ill-typed
```

The problem is that the type of `x` in the third case is `T`. Thus the type of `disband x` is `DCs T`, which includes the `C_` fields types. Because `ig_serialize` is applied to the `x`, the generic definition of `ig_serialize` will eventually apply the `serialize` method to all the fields of `C_`,

having mapped it to its `instant-generics` `Rep`resentation. Since it is applied to a function value, there will be a type error about the missing instance of `Serialize`. The crucial insight motivating my entire approach is that the `C_` fields type will never occur at run-time as a value of `x`, since it is guarded by the first case of `serialize_T`. 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 sums of fields types.

```
type instance DCs T = N C_ :+: ...    -- assuming ‘C_‘ is the leftmost

-- NB speculative: well-typed, but immodular
serialize_T :: T → [Bool]
serialize_T t = case disband t of
  L (C_ c _) → False : serialize c
  R x        → True  : ig_serialize x
```

This second definition is well-typed but unacceptably immodular because it exposes extraneous details of `T`'s representation as a sum to the programmer. Specifically, the `L` and `R` patterns depend on how the `:+:` type was nested in `DCs T`. 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 `serialize_T` 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 `instant-generics` `C` representation type that indexes the `Tag` type family by `instant-generics`'s constructor types. The above definition of `serialize_T` would still need to avoid the type-error by partitioning the constructors. However, where the fields types' syntax conveniently imitates the original constructors, the `instant-generics` `C` type would compound the immodularity 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

Listing 4.2: Interface for implicitly partitioning disbanded data types.

```
-- embedding relation
class Embed sub sup where embed_ :: sub → sup

embed :: (Range sub ~ Range sup) ⇒ sub → sup
embed = embed_

-- ternary partitioning relation
class Partition sup subL subR where
  partition_ :: sup → Either subL subR

-- set difference function
type family (:-:) sum sum2
partition :: (Range sub ~ Range sup, Partition sup sub (sup :-: sub) ⇒
  sup → Either sub (sup :-: sub)
partition = partition_

-- a precisely typed eliminator
precise_case :: (Range dcs ~ t, Range (DCs t) ~ t,
                 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
(|||) :: (Range l ~ Range r) ⇒ (l → a) → (r → a) →   l :+:   r → a
(||.) :: (Range l ~ Range r) ⇒ (l → a) → (r → a) →   l :+: N r → a
(.||) :: (Range l ~ Range r) ⇒ (l → a) → (r → a) → N l :+:   r → a
(.|.) :: (Range l ~ Range r) ⇒ (l → a) → (r → a) → N l :+: N r → a
```

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 Listing 4.2. Its implementation interprets sums as sets. Specifically, `N` constructs a singleton set and `:+:` unions two sets. The `Embed` type class models the subset relation with elements identified by the `Equal` type family from Section 3.5. 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.

Some of the definitions implementing this interface use the `foldPlus` function to fold values of the `:+:` type. This provides a nice structure to my code that manipulates sums. I use the simple `unN` function as an analog for the `N` type.

```
foldPlus :: (l → l') → (r → r') → l :+: r → l' :+: r'
foldPlus f g x = case x of
  L x → f x
  R x → g x

unN :: N a → a
unN (N x) = x
```

The following instances are complicated and dense. But they are never seen by the user and need only be extended with more instances if a power user adds representation types for sums alongside `N` and `:+:`. This is unlikely as long as the focus is on representing data types in Haskell.

Embedding

I define the `Embed` and `Partition` classes using auxiliary type-level programming that encodes and manipulate paths through sums of types. This is how I avoid the technique used by Swierstra [44] that treats sums like type-level lists. Paths are represented by the `Here`, `TurnLeft`, and `TurnRight` types.

```
data Here a
data TurnLeft path
data TurnRight path
```

The `Locate` family constructs a path by finding a given type in a given sum. For example, the type `Locate Int (N Int)` is `Just (Here Int)`, the type `Locate Char (N Int)` is `Nothing`, `Locate Char (N Int :+: N Char)` is `Just (TurnRight (Here Char))`, and `Locate Int (N Int :+: N Char)` is `Just (TurnLeft (Here Int))`.

```
type family Locate a sum

type instance Locate a (N x) = If (Equal x a) (Just (Here a)) Nothing
type instance Locate a (l :+: r) =
  MaybeMap TurnLeft  (Locate a l) `MaybePlus`
  MaybeMap TurnRight (Locate a r)
type instance Locate a Void = Nothing
```

In the case of a singleton set, it uses the `Equal` equality test to determine if the only element is the target element. If so, it returns a result in the `Maybe` monad with `Just`, otherwise is fails with `Nothing`. The case for the union of two sets uses both the functorial properties of `Maybe` and the monoid properties of `Maybe a`. It recurs into the two sets and then tags the result via `MaybeMap` with the appropriate turn. Then `MaybePlus` combines the two results with a preference for paths in the left set. As a result, `Locate` will only return one path if a type occurs at multiple paths within a sum. While `Locate` could be enriched to return all possible paths, this is unnecessary for my interests. The sets correspond to sums of constructors and should never include two equivalent types.

Given a value of type `a` and a path, the `injectAt` function builds a value of any sum that contains `a` at that path. The path is passed in using a proxy argument: `data Proxy t = Proxy`.

```
class InjectAt path a sum where injectAt :: Proxy path → a → sum

instance InjectAt (Here a) a (N a) where injectAt _ = N

instance InjectAt path a l ⇒ InjectAt (TurnLeft path) a (l :+: r) where
  injectAt _ = L ∘ injectAt (Proxy :: Proxy path)
instance InjectAt path a r ⇒ InjectAt (TurnRight path) a (l :+: r) where
  injectAt _ = R ∘ injectAt (Proxy :: Proxy path)
```

If the path is constructed with `Here`, then the sum is a singleton set built with `N`. If it is a left turn, then the sum must be a union and the type `a` can be injected into the left set using the rest of the path. The case for a right turn is analogous. The occurrences of `N` and `:+:` types in the instance heads could be relaxed using equality constraints in the instance contexts to add more robustness to these instances. However, this is unnecessary here because the

path argument will always be built with by passing the sum to `Locate`—the structure of the union is always already fixed and so need never be refined by a ~ constraint. Instantiation at incompatible types incurs in a type error.

The definition of `Embed` relies on `Locate` and `InjectAt`. In particular, the case for singletons uses `Locate` to find the element in the superset and then injects its value correspondingly with `injectAt`. The instance for a union tail-recurs through the unioned subsets.

```
instance (Locate x sup ~ Just path, InjectAt path x sup) ⇒ Embed (N x) sup where
  embed_ = injectAt (Proxy :: Proxy path) ∘ unN
instance (Embed l sup, Embed r sup) ⇒ Embed (l :+: r) sup where
  embed_ = foldPlus embed_ embed_
```

In order to use these methods, the structure of both sets must be fully concrete. This is always the case for disbanded data types because the entire structure is determined by the data type declaration as statically encoded by instances of the `DCs` type family.

The `embed_` method generalizes the `inj` function of Swierstra [44]. Where `inj` can insert a signature in the correct place within a type-level list of signatures, the `embed_` function can insert each type in a set (encoded as a sum) into a superset (encoded as a sum). While `inj` could be enriched towards `embed`, its use of overlapping instances and reliance on the type-level list structure of `:+:`s makes it more difficult to do so. The use of the `Equal` type family in the definition of `Embed` makes the general definition the natural one.

The `embed` function is defined as a specialization of the `embed_` method in order to encode the intuitive notion that constructors from various disbanded data types should not be mixed. This is the purpose of the `Range sub ~ Range sup` constraint. While this is not a required constraint, it may help catch some bugs. Furthermore, for higher-kinded types, such as list, this constraint propagates type parameters between arguments. I discuss this issue further at the end of the section.

Partitioning

The `Partition` class is more complicated than `Embed`. In particular, the class is defined as a three-place relation between a set and two subsets that it can be partitioned into. However, when used to implement `precise_case`, it is more convenient to think of `Partition` as a function from a set and one of its subset to the rest of the set. Hence the `partition_` method is separate from the `partition` function, which is equal but has a type specialized to the case of interest using the `:-:` type family to compute the third set.

```
partition :: Partition sup sub (sup :-: sub) ⇒ sup → Either sub (sup :-: sub)
partition = partition_
```

The `Partition` class cannot be defined with its method having this more specific signature because the more specific relation it encodes does not hold throughout the type-level traversal of the type-level encodings of the set that the instances implement. I will return to this point after defining the instances in order to explain that they are simpler this way.

The instances of the `Partition` class rely on `Embed` and an auxiliary class `Partition_N` that handles the `N` case.

```
class Partition_N bn x subL subR where
  partition_N :: Proxy bn → N x → Either subL subR
```

As with `InjectAt`, the `Partition_N` class is only instantiated with arguments that satisfy a particular relation. Specifically, the `bn` type variable indicates whether or not the type `x` was found in `subL` by `Locate`. It is equal to `Elem x subL`.

```
type Elem a sum = IsJust (Locate a sum)
```

The `Elem` family implements a decidable membership relation by testing if `Locate` was successful. The `partition_N` method injects single element into the left or right subset accordingly.

```
instance Embed (N x) subR ⇒ Partition_N False x subL subR where
  partition_N _ = Right ∘ embed

instance Embed (N x) subL ⇒ Partition_N True  x subL subR where
  partition_N _ = Left  ∘ embed
```

Using `Partition_N`, the `Partition` class is defined like `Embed` was with an interesting instance for `N` and a merely tail-recursive instance for `:+:`. Note that the instance for `N` invokes `Partition_N` with the expected arguments.

```
instance (Partition_N (Elem x subL) x subL subR)
   ⇒ Partition (N x) subL subR where
  partition_ = partition_N (Proxy :: Proxy (Elem x subL))

instance (Partition a subL subR, Partition b subL subR)
   ⇒ Partition (a :+: b) subL subR where
  partition_ = foldPlus partition_ partition_
```

It is because of the tail-recursive `:+:` instance that the `partition_` method must have a more relaxed type than the `partition` function. If the method requires that its third type argument be the set difference of its first two, then the recursion in the `:+:` instance would be ill-typed. The first argument is getting smaller while the other two stay the same, which means that it might be losing some elements that are always present in the third argument. This is an example where programming with type-level relations is simpler than programming with type-level functions. The `partition` function also adds an equality constraint on its type variables `Range`s that is comparable to the one that `embed` adds to `embed_`.

The set difference operator `:-:` is much easier to define than the `Embed` and `Partition` classes. It also uses the `Elem` membership predicate.

```
type instance (:-:) (N a)     sum2 = If (Elem a sum2) Void (N a)
type instance (:-:) (a :+: b) sum2 = Combine (a :-: sum2) (b :-: sum2)
```

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 is reasonable since the empty set represents a data type with no constructors and these offer no potential simplification with GP.

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

Because the `Embed` and `Partition` instances 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 Swierstra [44]. However, my instances have two advantages over existing work. They avoid overlapping instances and therefore respect the `:+:` operator as a closed binary operator instead of treating it like a type-level cons. I use type-level programming to avoid overlap for reasons discussed by Kiselyov [24, #anti-over]. I also use the decidable `Equal` type family and its derivatives, such as `Locate` and `Elem`, to explicitly distinguish between instances that would otherwise overlap. Kiselyov [24, #without-over] develops a similar method.

The `partition` method generalizes the `prj` function of Swierstra [44]. Where `prj` can isolate a signature from within a type-level list of signatures, the `partition` function can divide a superset (encoded as a sum) into two subsets (encoded as sums), one of which is specified by the programmer. Where `inj` could be partially enriched towards `embed`, `partition` is strictly more powerful than `prj` since it returns the smaller subset of the other part of the superset. There is no way to recover this functionality with the `prj` interface. On the contrary, `prj` can be easily defined in terms of `partition` by using a singleton set as the first argument and forgetting the extra information via `either Just (const Nothing)`.

Simulating a Precise Case Expression

The last part of the `yoko` interface is comprised of the `one` function and the family of disjunctive operators assemble functions consuming fields types into larger functions. These larger functions' domain is a subset of constructors. The `|||` operator is directly analogous to the `Prelude.either` function. Its derivatives are defined by composition in one argument with `one`.

```
one (N x) = x
(|||) = foldPlus
f .|| g = one f ||| g
f ||. g = f ||| one g
f .|. g = one f ||| one g
```

When combined with the `disband` and `partition` functions, these operators behave like an extension of the Haskell case expression that supports precise types. Note again that the type of `precise_case` introduces equality constraints over `Range` that are not technically required.

```
precise_case x g f = either f g $ partition $ disband x
```

With `precise_case`, the well-typed but immodular definition of `serialize_T` at the beginning of this section can be directly transliterated to the ideal (*i.e.* both precisely typed and concisely defined) `Serialize` instance for the `T` data type that solved Example 2.1, repeated here for convenience.

```
instance Serialize T where serialize = serialize_T

serialize_T t = precise_case (λx → True : ig_serialize x) t $ one $
  λ(C_ c _) → False : serialize c
```

The main difference is that the default case occurs as the first argument to `precise_case`; I abandon the allusion to gravity for "fall-through" instead of requiring that the exceptional cases be wrapped in parentheses.

To observe the `|||` operators in action, consider if `T` had four constructors like `C` that need special treatment.

```
data T = ... | C0 W (W → W) | C1 X (X → X) | C2 Y (Y → Y) | C3 Z (Z → Z)

instance Serialize T where
  serialize t = precise_case (λx → True : ig_serialize x) t $
    (λ(C1_ c1 _) → False : False : serialize c1) .||
    (λ(C2_ c2 _) → False : True  : serialize c2) .||
    (λ(C3_ c3 _) → True  : False : serialize c3) .|.
    (λ(C4_ c4 _) → True  : True  : serialize c4)
```

The operators' fixities are declared so that they could also be used in this example in the order `.|.`, `|||`, `.|.` without having to add any more parentheses. Because `Partition` handles any permutation it does not matter which shape the programmer chooses. In this way, the `|||` family of operators help the programmer use disbanded data types in a clear way. As demonstrated in the above example, 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 precisely-typed case expression. They also have the specialized constraints required that the `Range` types be equivalent.

Partitioning Higher-Kinded Data Types

Because the `yoko` generic view is focused on `*` types, higher-kinded data types require special consideration. For example, the `yoko`-specifc part of the generic representation of `Maybe` is as follows.

```
data Nothing_ a = Nothing_
data Just_    a = Just_ a

type instance DCs (Maybe a) = N (Nothing_ a) :+: N (Just_ a)

instance (EQ ~ Compare a a) ⇒ DT (Maybe a) where
  disband x = case x of
    Nothing → L ∘ N $ Nothing_
    Just x  → R ∘ N $ Just_ x
```

```
instance (EQ ~ Compare a a) ⇒ DC (Nothing_ a) where rejoin_ _         = Nothing
instance (EQ ~ Compare a a) ⇒ DC (Just_    a) where rejoin_ (Just_ x) = Just x
```

There are two main observations to make. First, the `Compare` constraints on the instances of `DC` and `DT` are the artifact necessitated by my simulation of the ideal `Equal` type family as discussed in Section 3.5. They enable the simulation to conclude that `a` is indeed equal to `a`. The second observation is the major benefit of the extra `Range` constraints that pervade the interface in Listing 4.2. While previously they had been explained as an enforcement of the intuitive notion that sums of constructors should all have the same `Range`, they more importantly provide an avenue for the propagation of actual type parameters. For example, without the `Range` constraints, the inferred type of a the following function would be too polymorphic.

```
(λNothing_ → mempty) .|. (λ(Just_ a) → a)
```

Without the `Range` constraints on `.|.`, this function has the polymorphic type `∀a b. Monoid a ⇒ (Nothing _b) :+: (Just _a) → a`. Note that the type argument to `Nothing_` might differ from the argument to `Just_`. This is more polymorphic than intended, and may lead to type errors caused by ambiguity, such as the infamous `show . read` type error.

The `Range` constraints on `embed`, `partition`, `precise_case`, and the disjuctive operators—*i.e.* any type in the interface where two sums of fields types are combined or in a subset relationship—perform the same unification of type arguments. Especially because of the type-level programming involved behind-the-scenes, extraneous type variables can lead to incomprehensible types that quickly overwhelm the programmer. It is often hard to identify where an ambiguity type error is coming from, especially when it the error itself is obfuscated by the type-level programming's delayed computations. And once found, it is awkward to add the necessary ascription that elimates the ambiguity. Without the (intuitive!) `Range` constraints, the `yoko` library would be unusable.

4.3 Summary

My first extension to `instant-generics` *delays* the structural representation of a data type by intermediately 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 a predictably similar name. Any sum of fields types is called a *disbanded data type*. My 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 chapter. The last extension hides the the representation of sums so the user can maniuplate them without coupling code to their irrelevant structural details. Together with fields types, this extension prevents uses of `hcompos` from obfuscating the surrounding definitions.

Page left intentionally blank.

5 A Pattern for Almost Homomorphic Functions

I explain more precisely how the `hcompos` function generalizes the `compos` function of Bringert and Ranta [7], and then generically define `hcompos` accordingly. I begin by developing a schema for the homomorphisms I am interested in. This schema serves as a specification for `hcompos` and one specialization of the schema also specifies the behavior of `compos`. This emphasizes their shared semantics and demonstrates how `hcompos` generalizes `compos`. The use of `yoko`'s reflection of constructor names to add support for heterogeneity to `compos`.

5.1 Homomorphism

A *homomorphism* is a function that preserves a semantics common to both its domain and codomain. For example, the function from lists to their length is a homomorphism from lists to integers that preserves monoidal semantics. Recall that a monoid is determined by a unit element `mempty` and a binary operator `mappend` that together satisfy the monoid laws. Lists with `mempty = []` and `mappend = (++)` form a monoid, as do integers with `mempty = 0` and `mappend = (+)`. The `length` function is a monoid homomorphism because it preserves the monoidal semantics: it maps lists' `mempty` to integers' `mempty` and lists' `mappend` to integers' `mappend`.

```
length mempty        = mempty
length (mappend x y) = mappend (length x) (length y)
```

Note that the `mempty` and `mappend` on the left-hand sides operate on lists while those on the right-hand sides operate on integers.

The two equations above are an instantiation of a more general schema that expresses the homomorphic property. It is simple to abstract the `length` function out of the equations. Doing so results in a schema is only specialized to monoids; any function `f` satisfying the following two equations is a *monoid homomorphism*, assuming that the involved `Monoid` instances for the domain and codomain of `f` do actually satisfy the monoid laws.

```
f mempty        = mempty
f (mappend x y) = mappend (f x) (f y)
```

It is more difficult to abstract over the part of this schema that is specific to monoidal semantics. Semantics for constructions from abstract algebra, like monoids, tend to be encoded in Haskell with type classes and informal side-conditions for those associated laws that cannot be encoded directly in the Haskell type system. This is the case for the `Monoid` type class, which does not actually guarantee that `mempty` is a unit and `mappend` is associative. The natural option, therefore, is to define the general form of the homomorphism schema in terms of the methods of a type class by requiring one equation per method. A function `f` is a *`TC` homomorphism* (in general, a *type class homomorphism*) for some class `TC` if `f` satisfies the following equation for each method `m` that takes n occurrences of the the class's type parameter. In this equation, I assume without loss of generality that the $\vec{\mathtt{x}}$ variable includes all arguments of `m` that do no involve the class's type parameter.

```
f (m x⃗ r_0 ... r_n) = m x⃗ (f r_0) ... (f r_n)
```

The assumption about the $\mathtt{r}_i$ values having the class's parameter as their type precludes the use of classes that have methods with richly structured arguments, like a list of the class parameter, for example. I add support for some rich arguments in Section 6. The assumption primarily helps the presentation, since it allows me to write $\mathtt{f\_\_r}_i$ instead of having to map `f` through whatever compound structure is containing the recursive occurrence.

This form of homomorphic schema, however, is too general for my purpose. I am specifying `hcompos` as an operator that the programmer can use in order to invoke the homomorphic schema as a definitional mechanism. The `compos` operator serves the same purpose for compositional cases of a function. The programmer must be able to *define* the tedious (*i.e.* either compositional or homomorphic) cases of functions simply by appeal to `compos` or `hcompos`. The above schema is problematic when interpreted as a definition of a case of `f` since it includes a method within a pattern. The arguments to `f` on the left-hand side of the

schema equations must only be valid Haskell patterns in order to be executable. Moreover, the patterns must be exhaustive, and a type class's methods do not necessarily partition all values in its parameter type. For this reason, I instead define the general case of the homomorphic schema using data types to encode semantics.

Analogous to the assumption in the class-based schema that $\vec{\texttt{x}}$ contains no occurrences of the class's parameter, I assume here without loss of generality that a constructor's non-recursive fields are all to the left of its recursive fields. I also assume that its recursive occurrences are all simple fields without compound structure like lists. Again, this second requirement is primarily for simplicity of the presentation; I relax it in Section 6. A function `f` with domain `T` is a *`T` homomorphism* (in general, a *data type homomorphism*) if it satisfies the following equation for each constructor `C` of the data type `T`, which may have type arguments.

```
f (C x⃗ r0 ... rn) = C x⃗ (f r0) ... (f rn)
```

This switch to data types instead of type classes refines the schema closer to a valid Haskell definition. However, I am not actually interested in defining homomorphisms; I am interested in defining *almost* homomorphic functions. Thus the schema equation need not apply to all constructors `C` of the domain, just enough of them for the programmer to benefit from automating them via the `hcompos` operator. So I am actually interested in data type homomorphic cases of functions as opposed to data type homomorphisms.

Though the use of constructors instead of methods successfully forces the schema equation for data type homomorphism to be a valid Haskell function case definition, it also introduces a problem with the types. In the schema equation based on type classes, the `m` method occurs with different types on the left- and right-hand sides in the general case. Simply replacing `m` with a constructor `C` resulted in a schema that is too restrictive: it requires that the defined function map one instantiation of a polymorphic data type to another. This schema requires

that both the domain and codomain of `f` are an instantiation of the same data type `T`. I will relax this requirement shortly.

Note how similar the above schema is to a potential schema for the `fmap` method of the `Functor` type class. The schema for `fmap` can be derived from the above schema by focusing on the occurrences of the functor's type parameter in addition to recursive occurrences. Under the assumption that the new $\texttt{p}_i$ variables correspond to all occurrences in the `C` constructor of the functor's type parameter, the following schema characterizes valid instances of `fmap`. An `fmap` function `f` whose domain is an instantiation `T a` extends some some function `g` whose domain is the type `a` by applying it to all occurrences of `a` in `C` and recurring structurally.

$$\texttt{f g (C } \vec{\texttt{x}} \texttt{ p}_0 \texttt{ ... p}_m \texttt{ r}_0 \texttt{ ... r}_n\texttt{) = C } \vec{\texttt{x}} \texttt{ (g p}_0\texttt{) ... (g p}_m\texttt{) (f g r}_0\texttt{) ... (f g r}_n\texttt{)}$$

If the `p` values are removed from both sides of this schema equation (including the applications of `g` on the right-hand side), the resulting equation would just be a specialization of the previous schema. It would express that the `C` case of the function `f g` is `T` homomorphic. This is in accord with the semantics of *nonvariant functors*, those that do not include any occurrences of their type parameter. The `fmap` operator for a nonvariant functor `T` must be the unique `T`-homomorphism. Similarly, for any covariant functor data type `F`, the case in its `fmap` function for a constructor `C` that does not include occurrences of the type parameter must be `F` homomorphic.

I have compared the data type homomorphic schema to the schema for `fmap` for two reasons. First, it shows that it is not unusual to model a semantics with a data type when considering a homomorphism's preservation of that semantics. Second, I hope it prevents the reader from conflating data type homomorphisms with their intuitions about the functorial extension. An entire `fmap` function is only a `T`-homomorphism if the data type `T` is nonvariant. However, it is likely that many cases of that `fmap` function are data type homomorphic, such as the case for the empty list, for `Nothing`, for the `Fork` constructors of both types `Tree.Tree`

and `WTree.Tree`, and so on. Indeed, `hcompos` can be used to automate these cases of `fmap` since the final homomorphic schema I am about to define generalizes the one for data type homomorphism.

The last observation about the data type homomorphic schema is that it is implied by the compositionality of cases defined with `compos` (*c.f.* Section 3.2). The `compos` function uses the data type homomorphic schema as a definitional mechanism for `f`, under the severe additional constraint that `f` has equivalent domain and codomain. Thus, each default (*i.e.* compositional) case of an almost compositional function `f` is trivially data type homomorphic: it does not change the constructor or even its type arguments. All compositional cases for a constructor `C` of functions like `f` satisfy the data type homomorphic schema equation.

By specification, the `hcompos` function is applicable for the definition of cases of functions like `fmap`, in which the `compos` function cannot be used: when the domain and codomain are unequal. To refine the notion of homomorphism encoded by the previous schema towards an implementation of this specification of `hcompos`, I generalize the data type homomorphic schema. As I said before, restricting the domain and codomain to be instantiations of the same parameterized data type is too restrictive for my purposes. So instead of using the constructor itself as the semantics-to-be-preserved, each default case handled by `hcompos` is defined to preserve the constructor's *programmer-intended semantics*. In Example 2.2 from Section 2, the `Tree.Leaf` and `WTree.Leaf` constructors correspond to one another because they have the same intended semantics, namely they both construct the leaves of a tree.

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 `instant-generics`, the only available property that could suggest a correspondence between two constructors in separate data types is the structure of their fields. However, this is too abstract to be unambiguous. Specifically, it is common

for a data type to have multiple constructors with the same fields. For example, consider the following two data types representing arithmetic expressions.

```
module PN where

data E = Lit Int | Plus E E | Negate E

module PNM where

data E = Lit Int | Plus E E | Negate E | Mult E E
```

Any programmer would expect that the `PN.Plus` and `PNM.Plus` constructors have the same intended semantics. It would be frustrating for the programmer to have to specify that `PN.Plus` is mapped to `PNM.Plus` instead of mapped to `PNM.Mult`.

I encode this intuition as the specification of `hcompos`. Specifically, I approximate the programmer's intended semantics by interpreting the name of a constructor as its intended semantics. This is crude but practical because it is lightweight and in accord with the innocuous assumption that constructors with comparable intended semantics have similar names. The interpretation is also not impractically severe because it is in accord with the common practice of using indicative names when programming.

Since the default cases of the almost homomorphic functions built with `hcompos` are defined by their preservation of the programmer's intended semantics, the `hcompos` operator implements a function that maps constructors in one type to constructors in another type that have the same name. It is simple to extend the data type homomorphic schema with this new flexibility. Each homomorphic case of a function `f` maps a domain constructor C_D with non-recursive fields $\vec{\texttt{x}}$ and n many recursive fields to the codomain constructor C_R which has the same name and, modulo the types that are being converted (*i.e.* D and R), the same fields:

`f (`C_D $\vec{\texttt{x}}$ `r`$_0$ `... r`$_n$`) = `C_R $\vec{\texttt{x}}$ `(f r`$_0$`) ... (f r`$_n$`) where Tag `C_D ` ~ Tag `C_R`.`

The compatibility of the two constructors' fields is not an explicit side-condition because it is implied by the well-typedness of this equality. Any such case of a function is called *tag homomorphic*. The benefit of tag homomorphic cases over data type homomorphic cases is that tag homomorphic cases do not require the domain and codomain to be instantiations of the same polymorphic data type. Instead, they can be any two data types that have a constructor with the same name. I refer to this relaxed constraint between domain and codomain as the *heterogeneity* of `hcompos`. It is what the `h` stands for: `hcompos` is a heterogeneous `compos`.

Because compositionality implies data type homomorphicness, and data type homomorphicness implies tag homomorphicness, the `hcompos` operator generalizes the `compos` operator.

5.2 Constructor Correspondences

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

The `FindDC` type family defined in Listing 5.1 implements the `FindDC` algorithm. An application of `FindDC` to a fields type `dc`, modeling the constructor C, and a data type `dt` uses

Listing 5.1: The `FindDC` type family.

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

type family FindDC_ s dcs
type instance FindDC_ s (N dc)    = If (Equal s (Tag dc)) (Just dc) Nothing
type instance FindDC_ s (a :+: b) = MaybePlus (FindDC_ s a) (FindDC_ s b)
```

the auxiliary `FindDC_` family to find a fields type in the `DCs` of `dt` with the same `Tag` as `dc`. The instances of `FindDC_` query `dt`'s sum of constructors, using the type equality predicate and the `Maybe` data kind discussed in Section 3.5. The result is either the type `Nothing` if no matching fields type is found or an application of the type `Just` to a fields type of `dt` with the same name as `dc`. The `hcompos` definition below enforces the compatibility of corresponding constructors' fields with an equality constraint.

Because the names of corresponding constructors must match exactly, I have so far been explicit about defining data types in distinct modules. For example, in the discussion of Example 2.2, the two `Tree` data types declared in the `Tree` and `WTree` modules both have constructors named `Leaf` and `Fork`. In the solution of this example, the use of `hcompos` actually requires `FindDC` to map the `Leaf` constructor of `WTree.Tree` to the `Leaf` constructor of `Tree.Tree` and similarly for the `Fork` constructors.

The inconvenience of needing to define similar data types in separate modules is actually one of the major detriments of my approach. Therefore, I plan to implement a more robust `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. The risk is that the definition of `FindDC` might imply that two constructors correspond that the programmer did not intend to correspond. This sort of accident is currently very unlikely because `FindDC` requires the names to be equivalent. One possible alternative is to factor `FindDC` out of the definitions and allow the programmer to provide an algorithm for determining constructor correspondence. Only time will identify the appropriate balance of expressiveness, convenience, and the possibility for confusion within the interface. For the rest of this dissertation, I will continue to define data types with equivalently named constructors and explicitly use separate modules to do so.

Listing 5.2: The `hcompos` function generalizes `compos`.

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

pure_hcompos :: HCompos a dcs b ⇒ (a → b) → dcs → b
pure_hcompos f = runIdentity ∘ compos (pure ∘ f)
```

5.3 The Generic Definition of `hcompos`

The generic homomorphism is declared as the `hcompos` method in Listing 5.2. To support heterogeneity, the `HCompos` type class adds the `dcs` and `b` parameters along side the original `a` parameter from the `Compos` class. The `dcs` type variable will be instantiated with the sum of fields types corresponding to the subset of `a`'s constructors to which `hcompos` is applied. The `dcs` parameter is necessary because it varies throughout the generic definition of `hcompos`. The `b` type is the codomain of the conversion being defined.

The generic definition of `hcompos` relies on the generic function `mapRs`, which extends a function by applying it to every recursive field in a product of fields.

```
-- q is p with the fields of type a mapped to b
class MapRs a b p q where
  mapRs :: Applicative i ⇒ (a → i b) → p → i q

instance MapRs a b (Rec a) (Rec b) where mapRs f (Rec x) = pure Rec <*> f x
instance MapRs a b (Dep x) (Dep x) where mapRs _         = pure
instance MapRs a b U       U       where mapRs _         = pure
instance (MapRs a b aL bL, MapRs a b aR bR) ⇒
  MapRs a b (aL :*: aR) (bL :*: bR) where
  mapRs f (l :*: r) = pure (:*:) <*> mapRs f l <*> mapRs f r
```

The instance for `Rec` is the only one that uses the supplied function; it applies that function to all recursive occurrences. All the other instances behave homomorphically (natch). The `mapRs` function is another example of the theme where recursive occurrences are treated specially like the `Traversable` class treats its parameter's type parameter. In fact, in some generic programming approaches—those that represent a data type by the fixed point of its

pattern functor—the `mapRs` function would correspond to a use of `fmap`. The `mapRs` function also exemplifies the fact that sums and products are not mixed when representing data types: `mapRs` handles only products while `hcompos` handles sums.

The `hcompos` function, like most generic functions, handles sums directly with the `foldPlus` operator. Note how the `dcs` parameter varies in the head and context of the following instance; it hosts this type-level traversal. This is the reason for its existence as a parameter of the class.

```
instance (HCompos a l b, HCompos a r b) ⇒ HCompos a (l :+: r) b where
  hcompos cnv = foldPlus (hcompos cnv) (hcompos cnv)
```

The instance for `N` uses the enhanced data type reflection of `yoko`. This is where the real work is done.

```
instance (Generic dc, MapRs a b (Rep dc) (Rep dc'),
          Just dc' ~ FindDC dc b, DC dc', Codomain dc' ~ b
         ) ⇒ HCompos a (N dc) b where
  hcompos cnv (N x) = fmap (rejoin ∘ (id :: dc' → dc') ∘ to) $ mapRs cnv $ from x
```

The instance converts a fields type `dc`, with possible recursive occurrences of `a`, to the type `b` in three steps. First, it completes the representation of `dc` by applying `from`, eliminating the delay of representation. Second, it applies the `mapRs` extension of `cnv` in order to convert the recursive fields. The result is a new product, where the recursive fields are of type `b`. This use of `mapRs` is well-typed due to the second constraint in the instance context, which enforces the compatibility of the two constructors' fields. Thus, the final step is to convert the new product to the new fields type found by `FindDC` in the third constraint and then insert it into `b` via `rejoin`. The insertion requires the last two constraints in the context and relies on the guarantee of the `MapRs` constraint that the result of the `mapRs` function is equal to the product of fields that represents the `dc'` fields type. The two instances of `HCompos`

traverse the sums of constructors and then delegate to `MapRs` in order handle the products of fields.

I have now explained the `yoko` generic programming approach and shown how its implicit partitioning enables the clean use of `hcompos` via the `precise_case` combinator in examples like Example 2.2 . I have also generically defined the `hcompos` operator by using my novel delayed representation. In the rest of this chapter, I discuss some more examples, including the lambda-lifting of Example 2.3. In the next chapter, I enrich `hcompos` to be applicable to more interesting data types. It cannot currently be used, for example, with mutually recursive data types.

5.4 Small Examples

The `hcompos` operator is more widely applicable than the `compos` operator. In the discussion of `compos` in Section 3.2, I explained how it could not be used to convert between two instantiations of the same parametric data type. The particular example was the function that maps a function on weights to a function on weighted trees without changing the structure of the tree.

```
import WTree

changeWeight :: (w → w') → Tree a w → Tree a w'
changeWeight f = work where
  work (WithWeight t w) = WithWeight (work t) (f w)
  -- work x = pure_compos work x -- intuitively meaningful but ill-typed!
  work (Leaf a)   = Leaf a
  work (Fork l r) = Fork (work l) (work r)
```

The `pure_compos` operator is not applicable because its type requires its resulting function to have an equivalent domain and codomain. This is not the case here, since it must map from `Tree a w` to `Tree a w'`. This requires data type homomorphicness, not just compositionality.

Two different instantiations of the same parametric data type are the most prevalent example of data types that have constructors with the same names. As such, `hcompos` can be readily

applied, since tag homomorphicness implies data type homomorphicness. Indeed, in this example, it works where `compos` fails. However, it cannot just be plugged in directly; the following naïve attempt is still ill-typed, with a comparable type error.

```
changeWeight :: (w → w') → Tree a w → Tree a w'
changeWeight f = work where
  work (WithWeight t w) = WithWeight (work t) (f w)
  work x = pure_hcompos work $ disband x -- still ill-typed!
```

The new type error is intuitively being caused by the fact that this definition uses `hcompos` to construct a homomorphic case for `WithWeight`. Since this homomorphic case does not alter the weight field—homomorphic cases only structurally recur—the resulting type error correctly explains that `Tree a w` is not equal to `Tree a w'`. Once again the `precise_case` combinator is required. Translating the normal case expression to the simulation is as straightforward as always.

```
changeWeight :: (w → w') → Tree a w → Tree a w'
changeWeight f = work where
  work t = precise_case (pure_hcompos work) t $ one $
    λ(WithWeight t w) → WithWeight (work t) (f w)
```

This last definition is finally well-typed, except for the omitted `EQ/Compare` constraints necessary for the simulation of `Equal`. With those constraints, this function could be used as the `fmap` method in a `Functor` instance for `WTree.Tree`.

A function for changing the `a` parameter of `Tree` instead of the weights is just as simple as `changeWeight`.

```
mapTree :: (a → b) → Tree a w → Tree a' w
mapTree f t = w where
  w t = precise_case (pure_hcompos w) t $ one $ λ(Leaf_ a) → Leaf $ f a
```

And so is the definition of `map` for lists.

```
mapList :: (a → b) → [a] → [b]
mapList f = w where
  w t = precise_case (pure_hcompos w) t $ one $ λ(Cons_ a as) → f a : w as
```

Maintaining effects is just a matter of dropping the `pure_` prefix and using the `Applicative` combinators instead of standard application in the alternatives for each specific fields type. The resulting variants of `changeWeight` and `mapList` could be readily used as instances of `Traverse` for `WTree.Tree` and lists.

I have shown with these simple examples that functions for converting between two instantiations of the same parametric data type are prime candidates for the using homomorphicness as a definitional mechanism. The `hcompos` operator provides the convenience of `compos`, which would otherwise be out of reach because of strong typing. When using `hcompos` in such definitions, the programmer need only explicitly handle the constructors involving occurrences of the parameters that are being changed; the nonvariant cases can be automated with `hcompos`. For many data types, this means many cases can be omitted.

I have now developed the two technical cores for the two small examples Example 2.1 and Example 2.2 and discussed them in detail. In the next section, I use `hcompos` to solve the lambda lifting example, Example 2.3. The following two chapter enriches the set of types that `hcompos` can be used with.

5.5 Example: Lambda Lifting

I solve Example 2.3 in this section in order to demonstrate the utility of fields types, `hcompos`, and implicit partitioning on a realistic example. Lambda-lifting (also known as closure conversion; see, *e.g.*, [38, §14]) 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 automate approximately 35 out of 40 constructors. Also, lambda-lifting with the term

representation I have chosen here involves sophisticated effects and therefore demonstrates the kind of subtlety that can be necessary in order to encapsulate effects with an applicative functor.

The Types and The Homomorphic Cases

The lambda-lifting function lifts lambdas out of untyped lambda calculus terms without changing the meaning of the program. 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 juxtaposed 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. I call the second kind of variable *captive* variables, as the lambdas' bodies 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. I encode this property in precise type derived from `ULC`; the `SC` data type is used for the bodies of top-level functions, the resulting *supercombinators*.

```
data SC = Top Int [Occ] | Occ Occ | App SC SC
data Occ = Par Int | Env Int
```

Instead of modeling lambdas, `SC` data type models occurrences of top-level functions, invocations of which must always be immediately applied to the relevant captives. This is the meaning of the new `Top` constructor. The data type also distinguishes occurrences of formals

and captives. I also assume, as with `ULC`, that `SC` has instances for the `yoko` type families and classes.

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

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

Each function declaration specifies the sizes of its two sets of variables: the number of captives and the number of formals. Because this data type has no constructor corresponding to lambdas, it guarantees that the values returned by lambda-lifting replace all lambdas with occurrences of top-level function definitions.

Because both of the `ULC` and `SC` data types have a constructor named `App`, the definition of lambda-lifting delegates the case for applications to `hcompos`. If constructors for any other syntactic constructs unrelated to binding were added to both `ULC` and `SC`, the definition of lambda-lifting below would not need to be adjusted. This is because non-binding constructs have tag homomorphic cases for lambda-lifting. As mentioned above with the GHC example, this can lead to significant savings in real-world data types.

The `lambdaLift` function is defined by its cases for lambdas and variables. I define the monad `M` next.

```
lambdaLift :: ULC → Prog
lambdaLift ulc = Prog tlfs body where
  (body, tlfs) = runM (ll ulc) ((i, IntMap.empty), 0)
  i = IntSet.findMax $ freeVars ulc

ll :: ULC → M SC
ll tm = precise_case (hcompos ll) tm $ llVar .|. llLam

llVar :: Var_ → M SC   ;   llLam :: Lam_ → M SC
```

This is the principal software engineering benefit of `hcompos`—tag homomorphic cases are completely automated. In fact, the rest of the code in this example is largely independent of the use of `hcompos`; this code already exhibits the punchline: the whole function can be defined via a case for variables, a case for lambdas, and a use of `hcompos` for everything else. However, the rest of the code is indirectly affected by the use of `hcompos` because the operator requires that a monad be used. As will be discussed below, this is not a trivial matter given the chosen encoding of `SC`.

This small code listing demonstrates a further benefit of the delay representation: fields types make it convenient to handle specific constructors separately. In previous examples, I have written the alternatives directly in the last argument of `precise_case` in order to emphasize how `precise_case` simulates the normal case expression. In this example, though, the `llLam` case is going to be more complicated than previous alternatives, so I break it out as a separate definition. I do the same for the `llVar` alternative because it involves details of the monad that I have not yet introduced. Note, however, that the `llVar` and `llLam` functions have types that strongly indicate their meaning. This helps me clearly present the structure of my lambda-lifting function in this abbreviated code listing because the names `llVar` and `llLam` are indicative but so are the types. Fields types help the programmer in this way to write code that is easier to understand.

The traversal implementing lambda-lifting must collect the top-level functions as they are generated and also maintain a renaming between `ULC` variables and `SC` occurrences. The monad `M` declared in Listing 5.3 provides these effects. They are automatically threaded by the `hcompos` function since every monad is also an applicative functor.

The `Rename` type includes the number of formal variables that are in scope and a map from the captives to their new indices. It is used as part of a standard monadic environment, accessed with `ask` and updated with `local`. The list of generated declarations is a standard monadic output, collected in left-to-right order with the `[]`/`++` monoid, and generated via the `emit`

Listing 5.3: The monad for lambda-lifting.

```
-- # of formal variables and the mapping for captives
type Rename = (Int, IntMap Int)
newtype M a = M {runM  :: (Rename, Int) → (a, [TLF])}

instance Monad M where
  return a = M $ λ_         → (a, [])
  m >>= k  = M $ λ(rn, sh) →
    -- NB backwards state: a and w' are circular
    let (a, w)  = runM m     (rn, sh + length w')
        (b, w') = runM (k a) (rn, sh)
    in (b, w ++ w')

-- monadic environment
ask   :: M Rename
local :: Rename → M a → M a

-- monadic output and its abstraction as backward state
emit            :: TLF → M ()
intermediates   :: M Int
ignoreEmissions :: M a → M a
```

function. The final effect is the other `Int` in the environment, queriable with `intermediates` and reset to `0` by `ignoreEmissions`. It corresponds to the number of emissions by *subsequent* computations; note the circularity via the `w'` variable in the definition of `>>=`. This *backwards state* [49, §2.8] is crucial to maintaining a de Bruijn encoding for the occurrences of top-level functions.

Because monads are now a familiar method of structuring side-effects, the `hcompos` interface was designed to support them in the same way that the standard `Traversable` class and the `compos` function do. This makes the `hcompos` function applicable in many more situations; were it entirely pure, it would rarely be useful. However, this example was designed to also demonstrate that it can be difficult to formulate a function using monads. The backwards state is essential here, but it is considered a rare and confusing monadic formulation. As will be discussed in Section 7.3, this means that architecting some functions so that `hcompos` can be used in their definition may require clever techniques.

The Interesting Cases

The variable case is straight-forward. Each original variable is either a reference to a lambda's formal argument or its captured lexical environment. The `lookupRN` function uses the monadic effects to correspondingly generate either a `Par` or a `Env` occurrence.

```
-- llVar :: Var_ → M SC
llVar (Var_ i) = (λrn → lookupRN rn i) <$> ask

lookupRN :: Rename → Int → Occ
lookupRN (nLocals, _) i | i < nLocals = Par i
                        | otherwise = case IM.lookup (i - nLocals) m of
  Nothing → error "free␣var"
  Just i → Env i
```

The occurrence of `error` is required because the `ULC` data type does not guarantee that all of its values correspond to well-scoped terms; it is not a precise data type. It would be reasonable to enrich the monad `M` to handle exceptions and raise one in this case, but I prefer not to do so for the sake of the presentation. Also, in a program doing lambda lifting, an ill-scoped input term is likely a fatal error, so there is not necessarily a benefit to complicating the monad with exceptions. If `ULC` did guarantee the well-scopedness of its values, it would no longer be in `yoko`'s generic universe. I discuss this as future work in Section 6.3.

The case for lambdas is defined in Listing 5.4. It uses two auxiliary functions: `freeVars` for computing free variables and `peel` for peeling a sequence of lambdas off the top of a `ULC` term. I omit the definition of `freeVars` because it is familiar and can be sufficiently automated using `instant-generics`.

```
freeVars :: ULC → IntSet

peel :: ULC → (Int, ULC)
peel = w 0 where
  w acc (Lam tm) = w (1 + acc) tm
  w acc tm       = (acc, tm)
```

Listing 5.4: The lambda-lifting case for lambdas.

```
-- llLam :: Lam_ → M SC
llLam lams@(Lam_ ulcTop) = do
  -- get the body; count formals; determine captives
  let (k, ulc) = peel ulcTop
  let nLocals = 1 + k
  let captives = IntSet.toAscList $ freeVars $ rejoin lams

  -- generate a top-level function from the body
  do let m = IntMap.fromDistinctAscList $ zip captives [0..]
     tlf <- ignoreEmissions $ local (const (nLocals, m)) $ ll ulc
     emit (IntMap.size m, nLocals, tlf)

  -- replace lambdas with an invocation of tlf
  rn <- ask
  sh <- intermediates
  return $ Top sh $ map (lookupRN rn) $ reverse captives
```

Though `peel` is a property-establishing function—the resulting `ULC` is not constructed with `Lam`—I do not encode the property in the types because I do not actually depend on it. If `peel` were redefined with a `precise_case`, it could be given the codomain `(Int, DCs ULC :-: N Lam_)`.

The `llLam` function in Listing 5.4 uses `peel` to determine the sequence of lambdas' length and its body, called `nLocals` and `ulc` respectively. The captives of a sequence of lambdas are precisely its free variables. The body is lambda-lifted by a recursive call to `ll` with locally modified monadic effects. Its result, called `tlf`, is the body of the top-level function declaration generated by the subsequent call to `emit`. Since `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 the `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 `Top` reference and the intended top-level function. This is the reason for the backwards state in `M`. The corresponding circularity

in the definition of `>>=` is guarded here (*i.e.* it does not lead to infinite regress) because `tlf` is emitted without regard to the number of subsequent emissions, though its value does depend on that number. In other words, the length of the list of `TLF`s in the monad's monadic output depends only on the input `ULC` term. In particular, it does not depend on the part of the monadic environment that corresponds to how many emissions there will be later in the computation. Indeed, the length of the output actually determines that value, even though the `TLF`s in the output rely on it.

The recursive call to `ll` must use a new monadic environment for renaming that maps captives variable occurrences to the corresponding parameters of the new top-level function; `local` provides the necessary environment to the subcomputation. The recursive call must also ignore the emissions of computations occurring after this invocation of `llLam`, 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 `intermediates` because the emissions of those ignored computations do not end up between `tlf` and the other top-level functions it invokes.

Summary

This example evidences the software engineering benefits of `hcompos` for defining non-trivial functions. Lambda-lifting establishes the absence of lambdas, which I encode with a precise type. It is also an almost homomorphic function, because only two cases are not homomorphic. My `yoko` approach therefore enables an concise and modular definition of a precisely-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.

6 Enriching the Generic Universe

All of the data types in the previous sections have been easy to represent generically; they are just sums of fields types, each of which can be mapped to a product of fields that are either `Dep` or `Rec`. Large programs, however, usually involve richer data types. In this section, I enable the use of `yoko` with two additional classes of data types in order to make `hcompos` more applicable. I add support for mutually recursive data types and for those data types with *compound recursive fields*. Compound recursive fields contain applications of higher-kinded types, such as `[]`, `(,)`, and `((,) Int)`, to recursive occurrences. Such fields are only partially supported by `instant-generics` but are common in large programs. My solution is still partial, but much less so, and is not specific to `hcompos`. I discuss support for even more sophisticated data types like GADTs in the last section of this chapter.

I have used simpler data types up to this point in order to focus on the semantics of `hcompos`. Adding support for mutually recursive data types makes the type of `hcompos` less clear. Similarly, I have been presenting the `compos` operator that only handles singly recursive data types. Since mutually recursive data types are common and important, Bringert and Ranta [7] also define a version of `compos` that handles mutually recursive data types. I will begin with their solution. The other kind of data type for which I add support, compound recursive fields, has no effect on the interface of `hcompos` or `compos`. It is simply a disadvantage of the `instant-generics` representation types. I support such fields by adding additional representation types. Since the `instant-generics` approach was designed to be extensible, my solution is simple to implement.

6.1 Mutually Recursive Data Types

The previous declaration of `hcompos` and `mapRs` cannot be used with mutually recursive data types because of the type of their first argument is too restrictive. In particular, the type of the function argument constrains that function to convert only one data type. In order to support mutually recursive data types, this function must instead be able to convert

each type in the mutually recursive family; it must be polymorphic over those types. There are at least two established ways in the generic programming literature to handle mutually recursive data types. I will show how to apply both techniques to the `HCompos` class and the generic definition of the `hcompos` operator.

Encoding Relations of Types with GADTs

The first generalization of `hcompos` supports mutually recursive families using the same essential idea as Bringert and Ranta. This approach uses *generalized algebraic data types* (GADTs) [48] 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. Instead of supporting mutually recursive data types, Bringert and Ranta demonstrate how any mutually recursive family of data types can be redefined as a single GADT without losing any expressiveness. The GADT is indexed by a set of empty types that are in one-to-one correspondence with types in the original family. Each constructor from the original family becomes a constructor in the GADT with an index that corresponds to the constructor's original range. As a result, the `Compos` function is redefined as follows by just making room for the new index and requiring the function argument to handle any index.

```
class Compos t where compos :: (∀a. t a → i (t a)) → t a → i (t a)
```

The rank-2 polymorphism guarantees that the input function can handle all constructors in the GADT. This is comparable to handling all types from the original mutually recursive family.

The essence of this technique is separated into more elementary components in the `multirec` generic programming approach [54]. The `multirec` approach uses GADTs to encode sets of types instead of combining all of those types into a single monolithic GADT. For example, the following `Odd` and `Even` data types are a pair of mutually recursive data types for even

and odd natural numbers with special cases for addition by two. The set of types {`Odd`, `Even`} is encoded by the `OddEven` GADT.

```
module OEWithDouble where

data Odd  =        SuO Even | SuSuO Odd
data Even = Zero | SuE Odd  | SuSuE Even

data OddEven :: * → * 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, and a function with type `∀s. OddEven s → s → a` can consume precisely either of those types. I prefer this approach to the one used by Bringert and Ranta. Their formulation would require the user to define only a monolithic `OddEven` GADT instead of using the full `Odd` and `Even` data types separately.

```
module NotUsed where

data Odd
data Even

data OddEven :: * → * where
  SuO   :: OddEven Even → OddEven Odd
  SuSuO :: OddEven Odd  → OddEven Odd
  Zero  :: OddEven Even
  SuE   :: OddEven Odd  → OddEven Even
  SuSuE :: OddEven Even → OddEven Even
```

Since the full use of `Odd` and `Even` in the `OEWithDouble` module is more conventional and the small `OddEven` GADT in that module ultimately provides the same power as the larger one in the `NotUsed` module, I prefer the `multirec` approach. Indeed, `OEWithDouble.Odd` is isomorphic to `NotUsed.OddEven NotUsed.Odd` and the same relationship holds for the `Even` types.

Using the `multirec` approach to that of Bringert and Ranta, I redefine `Compos`.

```
class Compos phi where
  compos_phi :: (∀a. phi a → a → i a) → phi a → a → i a

instance Compos OddEven where
  compos_phi f = w where
    w :: OddEven a → a → i a
    w Odd  (SuO   e) = SuO   <$> f Even e
    w Odd  (SuSuO o) = SuSuO <$> f Odd  o
    w Even Zero      = pure Zero
    w Even (SuE   o) = SuE   <$> f Odd  o
    w Even (SuSuE e) = SuSuE <$> f Even e
```

In their work, Yakushev et al. refer to those GADTs like `OEWithDouble.OddEven` that encode the set of types in the mutually recursive family as a *family*. Since the terminology "family" is already overloaded, I will not use it. I will, however, adopt the convention of abstracting over such families with a type variable named `phi`.

The second occurrence of the `phi` variable in the type of `compos_phi` is a nuisance, as can be seen in the occurrences of `f` within the definition of `w` in the instance for `OddEven`. Considering the context of those applications, the first argument to `f` should be inferable. The `multirec` approach therefore also provides a class that automatically selects the correct GADT constructor to witness the membership of a given type in a given set.

```
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. It enables the following definition of `compos`.

```
compos :: (Compos phi, Member phi a) → (∀a. phi a → a → i a) → a → i a
compos f = compos_phi f memb
```

This `compos` operator can be used much like the singly recursive one from the previous chapters, though it is often simpler to use `compos_phi` directly.

My example function for the rest of this section will be the function that inlines the special-case `SuSu` constructors as two applications of the relevant `Su` constructors. It is defined as follows using the `compos_phi` operator.

```
import OEWithDouble

forget_double :: OddEven a → a → a
forget_double = w where
  w :: OddEven a → a → a
  w Odd  (SuSuO o) = SuO $ SuE $ w Odd  o
  w Even (SuSuE e) = SuE $ SuO $ w Even e
  w oe    x        = compos_phi w oe x
```

Since the `forget_double` function guarantees that no `SuSu` constructors occur in its range, it could have a more precise codomain. As with all early examples, this unequal domain and codomain would then make the above definition ill-typed because `compos` requires its domain and codomain to be equivalent. I therefore extend `hcompos` to support mutually recursive data types in the exact same way as I have extended `compos` to do so.

The `multirec` approach uses GADTs to encode the set of types in a mutually recursive family so that the types of operators like `compos` can abstract over the whole family by abstracting over the appropriate GADT. GADTs can also be used to similarly encode the relation between types that is required for `hcompos`. The `HCompos` and `MapRs` classes can be parameterized over a GADT-encoded relation as follows, where C is `HCompos` or `MapRs` and m is `hcompos` or `mapRs`, respectively.

```
class C rel x b where
  m :: Applicative i ⇒ (∀a b. rel a b → a → i b) → x → 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 `N` invokes `FindDC` *etc.* and the auxiliary `MapRs` class. The `MapRs` class still applies the conversion function to the recursive fields, but it must now also provide the conversion function with its `rel` argument in order to invoke the part of the conversion that handles the specific pair of types being converted by that particular invocation of `mapRs`. So the `Rec` instance must use the `Related` type class, analogous to the `Member` class, to determine the first argument to the conversion function. The code for the other instances of `HCompos` and `MapRs` do not need to be changed.

```
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) = Rec <$> cnv rel x
```

This encoding of relations requires the user to define the GADT. Otherwise, it behaves just like the `hcompos` for singly recursive types. In particular, the `forget_double` function can now still be defined generically even with the precise codomain. This new codomain is defined in the `OEWithoutDouble` module.

```
module OEWithoutDouble where

data Odd  =        SuO Even
data Even = Zero | SuE Odd

data OddEven :: * → * where
  OddT  :: OddEven Odd
  EvenT :: OddEven Even
```

The `forget_double` function can be defined with `hcompos` as follows. First, the programmer must define a GADT specifying the intended relationship between data types. The encoded relation does not need to be functional, as it is in this example.

```
import qualified OEWithDouble    as WD
import qualified OEWithoutDouble as WoD

data Rel :: * → * → * where
  ROdd  :: ROdd  WD.Odd  WoD.Odd
  REven :: REven WD.Even WoD.Even

instance Related Rel WD.Odd  WoD.Odd  where rel = ROdd
instance Related Rel WD.Even WoD.Even where rel = REven
```

This GADT can then be used to structure the cases of `forget_double`.

```
forget_double :: Rel a b → a → b
forget_double = w where
  w :: Rel a b → a → b
  w ROdd  = precise_case (hcompos w) $ one $
    λ(WD.SuSuO_ o) → WoD.SuO $ WoD.SuE $ w ROdd  o
  w REven = precise_case (hcompos w) $ one $
    λ(WD.SuSuE_ e) → WoD.SuE $ WoD.SuO $ w REven e
```

In this example I have used `precise_case` twice instead of also extending it to handle mutually recursive data types directly. I have not done so for two reasons. First, this demonstrates that `precise_case` can be used with the `multirec` technique without needing to be changed. Second, I ultimately prefer another solution for mutually recursive data types, so adjusting `precise_case` would just be extra work.

The `multirec` approach to handling mutually recursive data types works well for `compos` and `hcompos`. In particular, it remains easy to convert from an imprecisely-typed function that was defined with `compos` to a more precisely-typed function that uses `hcompos`: just switch `compos` to `hcompos` and switch case expressions to uses of `precise_case` as necessary. The major downside of this approach is that it needs the GADT-encoded relation to be separately provided by the user. While this example makes it seem like it could be easy to

automatically derive the relation and its GADT encoding (perhaps with Template Haskell), there are important degrees of freedom that this example avoids for simplicity of presentation. If I wanted to preserve the look and feel `compos`, I would investigate how to automate this relation. However, I ultimately do want the programmer to have to provide the information that is contained in this relation, but I find the redundancy in this approach to be unacceptable. Fortunately, the conventional `instant-generics` approach for supporting mutually recursive data types integrates the relation much more seamlessly.

Encoding Relations of Types with Type Classes

The second generalization of `hcompos` uses the established `instant-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 semantics of the `hcompos` operator does not rely on the closedness of the relation, the type class approach suffices.

In this encoding, I introduce another type class with the same signature as `HCompos`. This is the class the programmer will instantiate in order to simultaneously define both the conversion relation between types and and the value-level implementation of the conversion. I call this class `Convert`.

```
type family Idiom (cnv :: * → *) :: * → *
class Applicative (Idiom cnv) ⇒ Convert cnv a b where
  convert :: cnv → a → Idiom cnv b
```

The main beneift of the `Convert` class is that it accomodates an intuitive semantics for its three types parameters. This is important because it is the interface that the programmer uses. Each instance of `Convert` for a given `cnv` type specifies that a value of the `cnv` type can be used to convert between the second two parameters of `Convert`. In this way, the instances determine the relation of types previously encoded with the `Related` class.

The applicative functor could also be provided as a type class parameter, but I have not done so in order to reserve the instance head for specifying the relation. The explicit encoding of the applicative functor necessary for a given conversion is the major downside of this approach; in the previous approach it was implicit. However, the requisite instance of the `Idiom` type family is less burdensome for the programmer to define than is the GADT that encodes a relation in the previous section's approach.

The `Convert` class helps handle mutually recursive data types because it provides a means for a single value to handle each type in the family in a different way. This is evident in the new definition of the `forget_double` function converting from the `Odd`/`Even` types in `OEWithDouble` to those in `OEWithoutDouble`. In this definition, the `OddEven` GADTs are unnecessary.

```
data Forget = Forget
type instance Idiom Forget = Id   -- no effects needed for this conversion

instance Convert Forget WD.Odd  WoD.Odd where
  convert cnv o = precise_case o (hcompos cnv) $ one $
    λ(WD.SuSuO_ o) → (WoD.SuO ∘ WoD.SuE) <$> convert cnv o
instance Convert Forget WD.Even WoD.Even where
  convert cnv e = precise_case e (hcompos cnv) $ one $
    λ(WD.SuSuE_ e) → (WoD.SuE ∘ WoD.SuO) <$> convert cnv e

forget_double :: Convert Forget oe oe' ⇒ oe → oe'
forget_double = unId ∘ convert Forget
```

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 becomes a consequence of the necessary instances of `Convert` for a given conversion. Thus, the `Convert` class subsumes the `Related` class.

In order to integrate the `Convert` class into the definition of `hcompos`, the `HCompos` and `MapRs` classes are redefined with the same signature as `Convert`.

```
class Applicative (Idiom cnv) ⇒ C cnv a b where m :: cnv → a → Idiom cnv b
```

As with `Convert`, this declaration indicates that `hcompos` is now a mapping from the type `cnv` to a conversion between the types `a` and `b`. Since the `Convert` instances for a given `cnv` type also defines the relation characterizing that conversion, the `MapRs` instance for `Rec`, which had used the `Related` class in the previous section, must call `convert` instead of `rel`.

```
instance Convert cnv a b ⇒ MapRs cnv (Rec a) (Rec b) where
  mapRs cnv (Rec x) = Rec <$> convert cnv x
```

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

The `Forget` value used in the last definition of `forget_double` and its instances of the `Convert` class subsume both the `w` function and the `Rel` data type from the `multirec`-based definition of `forget_double`. This combination makes the type class encoding of relations more concise, which is the reason for adding the new `Convert` class.

Semantically, the `Convert` class is similar to the `Apply` class popularized by Kiselyov et al. [25].

```
class Apply fn dom cod where apply :: fn → dom → cod
```

Instances of this class specify both the standard action that a function-like value of type `fn` has on values of the domain as well as the less conventional action in which the type `fn` also determines the relationship between the `dom` and `cod` types. For example, standard functions have the following `Apply` instance.

```
instance (a ~ x, b ~ y) ⇒ Apply (a → b) x y where apply = ($)
```

One way in which `Apply` generalizes normal application is that a given function type `fn` can be ad-hoc polymorphic, which allows it to effectively reify a type class. This is why the `Convert` class is useful for adding support for mutually recursive data types to `hcompos`. Indeed, the only change that was required for the generic definition of `hcompos` was the switch

from standard application to application via the `convert` method in the `MapRs` instance for the `Rec` type. This allows the first argument of `hcompos` to handle multiple types non-parametrically—like a type class. Since its former argument used the → type directly, it could only be applied to a single specific domain/range pair, which is evident from the above instance of `Apply` for the → type.

Ultimately the two techniques for handling mutually recursive data types are also mutually definable. As I have said, I consider the second one preferable because it is more concise. Instead of defining a GADT and inventing a constructor name for each pair of types in the relation that the GADT encodes, the programmer need only invent a name for the function-like type implementing the conversion and specify each pair of related types in an anonymous instance of `Convert` for that conversion. The extra instances of the `Idiom` type family is a comparatively small price to pay. Moreover, each function-like type could be reused for various conversions if it has a general enough semantics, and the encoded relation has the added benefit of being defined with an open mechanism—if it is unnecessary, closedness risks becoming a maintenance burden.

Finally, two clever instances let the second approach subsume both the first approach as well as the tidy interface for singly recursive data types that is used in early chapters. First, the instance of `Apply` for the → type can be imitated for the `Idiom` family and the `Convert` class.

```
type instance Idiom (a → i b) = i
instance (Applicative i, a ~ x, b ~ y) ⇒ Convert (a → i b) x y
  where convert = ($)
```

Because of this instance, if a given occurrence of `hcompos` only requires that its argument be used to convert bewteen a single pair of types then that argument can just be a properly typed function. This condition is always met when converting between single recursive data types. Thus the code examples in the previous chapters still work with this definition of `HCompos`.

The second clever instance embeds the `multirec` approach into this section's approach.

```
newtype MultiRec rel i = MultiRec (∀a b. rel a b → a → i b)

instance (Applicative i, Related rel a b) ⇒ Convert (MultiRec rel i) a b where
  convert (MultiRec f) x = f rel x
```

Because of these two instances, the programmer has many options for how to define the ad-hoc type-indexed behavior of a conversion. The reverse embedding, of the multiparameter type-class based approach into the GADT-based approach is vacuous: it just uses a GADT that guarantees an suitable `Convert` constraint is available. It uses the same idea that is fundamentally captured by the `Dict` GADT, which uses the recent `ConstraintKinds` extension to GHC to reify type class dictionaries.

```
data Dict :: Constraint → * where Dict :: con ⇒ Dict con
```

With this data type, or a specialization of it, the programmer can encode a set as a GADT by actually encoding the set as a type class (*e.g.* `Convert`) and then reifying that class with the GADT. This is more convoluted than the opposite encoding via the `MultiRec` newtype, so I prefer the type class approach though both are equally powerful, in some sense.

6.2 Data Types with Compound Recursive Fields

All three definitions of `hcompos`, for one for singly recursive data types and the two for mutually recursive data types, require that the `Rec` representation type is only applied to simple recursive occurrences. Thus `hcompos` is inapplicable to data types with compound recursive fields. Since data types commonly have such fields, I extend `instant-generics` to handle them.

Consider the data type `T` with two constructors `Tip` and `Bin` with compound recursive fields as follows. These two instances of `Rep` follow the implicit conventions for representing compound

recursive fields that are established in the `instant-generics` and `generic-deriving` papers [8, 30].

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

The `hcompos` operator 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. These two representations, however, use `Rec` for types that are not just recursive occurrences: the arguments of `Rec` are types other than just `T`.

This issue cannot be solved by adding an `HCompos` instance for `(a, b)`. For example, such an instance 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`. I consider requiring the user to declare such ad-hoc and degenerate instances is unacceptable.

The real problem is the imprecision of the conventional `instant-generics` representation types. However, `instant-generics` is designed to be extensible, so it is easy to 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 $\texttt{Arg}_1$ and $\texttt{Arg}_2$ 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 $\texttt{Arg}_n$ types indicate which of their arguments need conversion, they do not introduce degenerate `HCompos` constraints. Because these types are only used as representations of fields, their addition to the representation types only require additional 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

instance (BiTraversable ff, MapRs cnv a1 b1, MapRs cnv a2 b2) ⇒
  MapRs cnv (Arg2 ff a1 a2) (Arg2 ff b1 b2) where
  mapRs cnv (Arg2 x) = Arg2 $ bitraverse (mapRs cnv) (mapRs cnv) x
```

The instance for $\texttt{Arg}_1$ simply uses the `traverse` method to propagate the `hcompos` conversion within the compound structure that the $\texttt{Arg}_1$ representation type delineates. The instance for $\texttt{Arg}_2$ is analogous to the one for $\texttt{Arg}_1$, but instead uses the `BiTraversable` class and two C constraints. The `BiTraversable` class is like `Traverse`, but handles mappings that involve two parameters.

```
class BiTraversable t where
  bitraverse :: Applicative i ⇒
    (a1 → i b1) → (a2 → i b2) → t a1 a2 → i (t b1 b2)
```

It is a simple extension of `Traversable` that handles an additional type parameter.

The pattern indicated by $\texttt{Arg}_1$ and $\texttt{Arg}_2$ extends smoothly to data types with more parameters. It is easy to add similar representation types for data types with greater arities as long as their parameters are of kind `*`. In order to support data types with higher-kinded parameters, `yoko` would need to be extended in the same manner that `instant-generics` was extended in the development of `generic-deriving` [30]. All representation type would need to be of a higher kind, say `* → *`. This is a pervasive change, and though it is semantically orthogonal to delayed representation, there two extensions will likely require some integration.

The dimension of extensibility towards higher-kinds has been problematic in the `instant-generics` branch of generic programming, but it is commonly accepted that higher-

kinded parameters—especially beyond the `* → *` kind supported by `generic-deriving`—are relatively rare in practice. As a consequence, there are likely to be only a few instances of the `BiTraversable` class. The usual candidates are `(,)` and `Either`.

I have motivated the addition of the $\texttt{Arg}_n$ family of representation types by observing that the generic definition of `hcompos` requires that the `Rec` representation type only be used in a certain way. In particular, it must only be applied directly to recursive occurrences. Most generic functions do not have such a requirement. I suppose this is why the $\texttt{Arg}_n$ types were not initially included in `instant-generics`. Generic definitions can usually just use ad-hoc instances on a per-functor basis. For example, the common instance `Eq a ⇒ Eq (Maybe a)` seamlessly handles compound recursive fields without the need for the $\texttt{Arg}_n$ representation types. The reason for this is that instances of the `Eq` class have a uniform behavior for both representation types and genuine types. This is not the case for `HCompos`, so it requires the $\texttt{Arg}_n$ types.

I hypothesize that any generic function that behaves differently in the `Dep` and `Rec` cases would benefit from the $\texttt{Arg}_n$ types. My experiments with the `hasRec` function from the work of Chakravarty et al. [8] is one such example where use of the $\texttt{Arg}_n$ types enables the related `:+:` instance of `Enum` to avoid divergence in some cases where the standard `hasRec` definition causes it to diverge. More similarly to `HCompos`, the generic implementation of `fmap` that uses the `Var` representation type of Chakravarty et al. [8] also assumes that `Rec` is only applied directly to recursive occurrences. Such generic functions would benefit from the $\texttt{Arg}_n$ types in the same way that `hcompos` does: they would be applicable to more types.

As I said above, the $\texttt{Arg}_n$ types can only represent the compound structure of fields that involve data types with parameters of kind `*`. If the parameters are of a higher-kind, then all of the represntational machinery must be expanded to represent types of that kind. The `generic-deriving` library represents `* → *` kinds in this way. Thus the $\texttt{Arg}_n$ types are a simple extension that is sufficient for representing (fully saturated) applications of higher-

kinded types, but `generic-deriving` is required to represent abstraction at higher-kinds. However, the more complicated `generic-deriving` approach is usually considered necessary even for just handling applications. I have shown here that this is not the case: the simpler $\texttt{Arg}_n$ types are enough for applications.

6.3 Future Work: GADTs and Nested Recursion

The most precise types currently possible in Haskell require the use of GADTs, or at least nested recursion. The poster child example is the GADT that adequately represents well-typed (and hence well-scoped) terms of the simply-typed lambda calculus.

```
data Tm :: (* → *) → * → * where
  Var :: v a → Tm v a
  Lam :: (v a → Tm v b) → Tm v (a → b)
  App :: Tm v (a → b) → Tm v a → Tm v b

newtype STLC ty = STLC {unwrap :: ∀v. Tm v ty}
```

A similar data type can also be defined using de Bruijn indices instead of PHOAS [9] for binding.

```
data Elem :: * → * → * where
  Here :: Elem (gamma, a) a
  Next :: Elem gamma a → Elem (gamma, b) a

data Tm :: * → * → * where
  Var :: Elem gamma a → Tm gamma a
  Lam :: Tm (gamma, a) b → Tm gamma (a → b)
  App :: Tm gamma (a → b) → Tm gamma a → Tm gamma b

newtype STLC ty = STLC {unwrap :: Tm () ty}
```

In this case, a `Tm` represents an open term that is well-typed in the context modelled by `gamma`.

As adequate types, these types are the most precise possible kind of data types. In particular, they are more precise than the kind of types I intended to help programmers use in large

programs, since they are practially dependently typed. However, this sort of programming seems to be arriving in Haskell via GADTs and singleton types [15]. While I have not chosen to support it now, it does deserve mention as important future work.

My techniques are based on `instant-generics` and are thus similarly amenable to GADTs. In particular, my experimentation with integrating the approach of Magalhães and Jeuring [29] into `yoko` shows promise. The outstanding technical issues come from poor interactions with the type-level machinery for avoiding overlapping instances that I discussed in Section 3.5. Even so, the techniques from Magalhães and Jeuring can almost directly translated to `yoko`. Unfortunately, even those authors' original extension of `instant-generics` is cumbersome. In particular, it requires an instance of each generic class (*e.g.* `Show`, `Serialize`, my `Convert`) for every index used within the declarations of the GADT's constructors. If the de Bruijn-indexed `Tm` data type above was extended with a base type like `Integer` or `Bool`, then its generic instantiation for `Show` would require two instances, one for the `Tm` with an index of the $\rightarrow$ type and one with an index of `Integer`. In short, `instant-generics` can be economically extended to represent GADTs, but instantiating generic functions for them remains cumbersome. It seems that applying that approach directly would `yoko` would only make it a little more complicated.

Similar approaches to `instant-generics` but with representation types of kind `* → *` like `multirec` have fared better. They benefit from the fact that the kind `* → *` admits a representation type that is itself a GADT. For indexed programming, as Magalhães and Jeuring [29] discuss, the sort of type equality that can be encoded with a simple GADT is all that is needed. This suggests that shifting my extensions from `instant-generics` to `multirec` or `generic-deriving` might be a simpler means to better handle GADTs than integrating the extension of `instant-generics` by Magalhães and Jeuring into `yoko` itself.

6.4 Summary

In the above, I have refined the `HCompos` class and extended its generic definition to work for an additional family of representation types. Together, these changes add support for using the `hcompos` operator on mutually recursive families of data types and also on data types that include compound recursive fields. While both considered mechanisms for supporting mutually recursive data types are at least rooted in the literature, the approach to compound recursive fields might be a simpler alternative than what is present in existing literature about the `instant-generics` branch of generic programming. Since both mutually recursive data types and compound recursive types are in common use, the additions to the universe of types representable with `yoko` are crucial for the `hcompos` operator to be practical. Support for more expressive data types like GADTs is deferred to future work; the generic programming field currently offers multiple unsatisfying options for representing them.

7 Evaluation

In this evaluation section, I support four claims.

1. The `hcompos` operator provides the programmer with more automation than does the `compos` operator of Bringert and Ranta [7].

2. Enhanced with the `hcompos` operator, standard Haskell mechanisms support the static and strong typing and automation of the nanopass methodology better than does the Scheme implementation of Sarkar et al. [39]. I make no claim about relative efficiency.

3. The `hcompos` operator can be used to define non-trivial functions.

4. The benefit of `hcompos` as summarized by the other three claims is greater than its detriments.

These claims imply that my technical contributions as presented in this dissertation make it easier for programmers working in statically- and strongly-typed functional languages to use more *precise* types, specifically by providing a novel degree of useful automation.

I assume that automating function cases, especially ones the programmer finds tedious, eliminates bugs and makes the code easier to understand and reason about. I am not attempting to evidence that belief; it is my main premise and motivation. It is an instantiation of the DRY modern coding maxim: Don't Repeat Yourself. Similarly, I consider it accepted that the `compos` operator and the pass expander are useful largely because they underly this automation. My general claim is that my operator improves upon the other two in this capability.

It is straight-forward to evidence the first two claims. For the first claim, I have given numerous examples throughout the previous chapters showing that `hcompos` supplants `compos`. The second claim is to a large extent immediately evidenced by the simple fact that my operator is a genuine Haskell function while the nanopass implementation uses Scheme definitions

augmented with some basic compile-time checks. Because Haskell obviously supports static and strong typing better than does Scheme and the `hcompos` operator function places no restrictions on the rest of the program, Haskell with `hcompos` is more useful to a programmer working with static and strong typing. However, since the two operators are defined in separate languages, it is more difficult to show that they provide similar automation. I therefore compare the two approaches on a per-feature basis in Section 7.4.

I demonstrate the validity of the third claim by refining the definition of an important function from the literature. In particular, I add the precise types and automation uniquely supported by the `hcompos` operator to the lambda lifter of Jones and Lester [23]. Again, I consider the importance of lambda lifting and the competence of the original authors as an accepted fact.

The fourth claim is the most difficult. In particular, I have not focused on the efficiency and ease-of-use of `hcompos`. Compared to hand-written code—code enjoying none of the benefits of my first three claims—definitions that use `hcompos` are less efficient and also generate type errors for easy mistakes that are harder to understand (but ultimately contain the important information). I discuss the details of performance and type errors in Section 7.2. Ultimately, the loss of efficiency and clarity of type errors does not eclipse the benefits of the automation and precise types enabled by `hcompos`. At the very least, the operator is advantageous for experienced Haskell programmers rapidly developing prototypes.

My evaluation throughout is predominantly qualitative. The sort of quantities I do have, such as automating 13 out of 15 cases or reducing the definition from 39 to 25 lines of code (−36%), are anecdotal. I include them anyway because I believe they are reflective of the benefits in the general case. I could only empirically evidence this claim with expensive longitudinal studies of teams of professional programmers. The legitimate execution of such experiments is simply beyond my means. I therefore settle for *drawing parallels between my methods and*

existing research that has already been accepted by the community and highlighting ways in which my approach obviously surpasses the previous ones.

In the rest of this chapter, I first review and summarize the comparisons from previous chapters of my `hcompos` operator and the `compos` operator of Bringert and Ranta (Claim 1). I consider the detriments of `hcompos` by summarizing the efficiency issues in terms of other generic programming libraries and by discussing the type errors incurred by possible mistakes within one of the simpler examples (Claim 4). I update the function definitions within the lambda lifter of Jones and Lester and explain the improvements (Claim 3). Finally, I breakdown the nanopass framework into its major features and show that they are supported by a combination of existing Haskell features and my `yoko` library and `hcompos` operator. Almost all of their features are totally reproduced; a few of the minor features have only partial analogs (Claim 2).

7.1 Comparison to `compos`

Claim *The `hcompos` operator provides the programmer with more automation than does the `compos` operator of Bringert and Ranta [7].*

There is a single essential difference between the `compos` operator of Bringert and Ranta [7] and my `hcompos` operator. While their operator requires that the function being defined has the same domain and codomain, my operator requires only that each constructor in the domain has a matching constructor in the the codomain with the same name and fields. Since this property is trivially satisfied when the domain and codomain are equal, the type of `hcompos` is strictly more flexible than that of `compos`. Semantically, `hcompos` can always replace `compos`; it is strictly more expressive.

My solution for Example 2.2 demonstrates that `hcompos` generalizes `compos`. The `compos` operator can be used to automate most cases in the definition of a function that discards the weights within a weighted tree–the programmer only needs to explicitly handle the interesting

case for the `WithWeight` constructor. Even though the `WithWeight` constructor never occurs in the range of the function, this cannot be enforced by the the codomain, since `compos` requires it to be the same as the domain. In order to enable a more precise codomain in which the `WithWeight` constructor no longer exists without sacrificing the benefits of generic programming, the `compos` operator must be replaced by my `hcompos` operator and a simple invocation of the `precise_case` function.

My first solution to Example 2.3 offers a more sophisticated example. I define a lambda lifting function with a precise codomain. In particular, it guarantees that the range of the function includes no anonymous lambdas, which is an intended property of any lambda-lifter. If the domain were reused as the codomain, then the `compos` operator could provide the same genericity. However, the type of the lambda lifting function would no longer reflect and enforce its characteristic invariant.

Because of its extra flexibility, the `hcompos` operator is also useful for common functions which the `compos` operator could never be used for. In particular, I showed that the `hcompos` operator can be used to automate the tag homomorphic cases of functorial maps like `fmap`. In these functions, many cases are tag homomorphic, but they are not compositional in the sense required by `compos` because the domain and codomain are inherently unequal. An embedding conversion is another kind of function for which `hcompos` can be used but `compos` would be ill-typed. These functions embed one data type into another data type that includes the same constructors but also has additional ones. For example, the precise codomain of the function that removes weights from trees could be embedded into the original weighted tree data type using just the function `fix $ λw → hcompos w . disband`.

These examples evidence my first claim. The `compos` operator is a widely applicable mechanism for automating compositional functions that simplifies the definition of important functions with potentially no overhead [7]. My examples show that `hcompos` can both re-

place `compos` as well as be used in definitions where `compos` would never have been well-typed; it provides strictly more automation than does `compos`.

7.2 The Detriments of `hcompos`

Claim *The benefit of `hcompos` as summarized by the other three claims is greater than its detriments.*

As discussed in the beginning of this chapter, the use of `hcompos` incurs costs. In particular, there are three potential ways for it to be detrimental. I list these issues in order of increasing severity.

1. The need for the `precise_case` function instead of just a normal Haskell case expression and the rich type of `hcompos` slightly obfuscates those definitions that rely on them.
2. The `hcompos` operator can be less efficient than the the `compos` operator.
3. The type-level programming in the definitions of `precise_case` and `hcompos` can result in some type errors that are either difficult to understand or very large.

Code Obfuscation

The issue about `precise_case` is the least severe because it is so similar to the structure of a normal case expression. The obfuscation is slight and the programmer likely internalizes the difference after seeing a few examples. In fact, minimal syntactic sugar would solve the issue almost entirely. Language support for constructor subsets is a viable consideration for future adoption; see my comparison with polymorphic variants in Section 8.

In my first solution of Example 5.5, I observed that the use of de Bruijn indexing for references to the resulting supercombinators required that the monad—necessary for the use of `hcompos`—use the rare backwards state feature. I do not consider this reflective of the general case for two reasons. First, monads have become pervasive in Haskell (and other languages too), and so the same awkward feature would be required without the `hcompos`

interface. Second, though it might be possible to redefine the entire lambda lifter to avoid using a monad with the de Bruijn indices in such a way that requires backwards state, the exotic monadic feature can be considered an "elegant" solution. This indicates that the overall structure is correct, and so organizing a definition so that the `hcompos` operator can be used may actually encourage a definition that better reflects its essential semantics. In other words, the need for backwards state is odd, but not necessarily obfuscating.

It is a simpler argument for me to observe that the `compos` operator and the pass expander have similar interfaces to `hcompos` and they have not impaired programmers. Only the subtle distinction between a fields type and its constructors and the slight syntactic noise of `precise_case` versus a natural case expression make `hcompos` any harder to use than programming by hand. And as my examples have shown, those differences are slight enough to dismiss.

Inefficiency

The issue of efficiency is concerning for two reasons. First, uses of `compos` can be straightforwardly optimized by hand-writing some code instead of relying on generic definitions while uses of `hcompos` cannot. Any instance of the `Compos` class for a user data type could be defined by hand, and with suitable annotations for inlining, the GHC compiler can then simplify any definition that relies on that instance (*i.e.* uses `compos`) to code that is equivalent to the conventional definition as it would have been written by hand not using `compos`. Since the generic definition of `hcompos` is more complicated than that of `compos`, there are in fact more places a user could write instances by hand. However, the fact that these classes (*i.e.* `MapRs`, `HCompos`, `Partition`, and `Convert` in the fully expressive case) are paramterized by two types (*i.e.* roughly: the source and target of the conversion) means that the user interested in writing code by hand would end up writing the tedius part of conversion by hand and not using `hcompos` (or `precise_case`) at all. Thus, the efficiency of the code using `hcompos`

will always rely on the GHC optimizer to eliminate the overhead introduced by the generic definition of `hcompos` via the `yoko` interface (*i.e.* `DT`, `DC`, *etc*).

The second reason is that the expected current overhead of `yoko` is a factor of two. I am supposing that the overhead of `yoko` is comparable to that for `instant-generics`, but I have not done measurements—the basic mechanisms are exactly the same: families and classes for mapping between data types and almost the exact same set of representation types. Generic definitions in `instant-generics`, like the one for `compos` in Section 3.3, have been measured at about twice as slow as hand-written code [31, 2]. Only a recent generic programming approach has done any better; previously existing approaches are usually orders of magnitude slower than `instant-generics` and its derivatives [8, 31, 2].

The new Template Your Boilerplate (TYB) approach of Adams and DuBuisson [2] relies heavily on metaprogramming to program generically with little overhead. The basis of the approach is a set of Template Haskell combinators that directly generate code by inspecting the definition of the involved types. As a result, GHC compiles the generated code almost as effectively as if it were hand written. But use of the TYB interface is rife with quasiquations and splices; it is severely obfuscated. The `instant-generics` and `yoko` interface, on the other hand, is composed of combinators and classes that are better founded in data type semantics. This results in easier to read generic definitions and invocations of those definitions. However, if that trade-off were preferable, I am confident that the semantic concepts of `yoko` and `hcompos` can be transferred to TYB. In fact, I performed my original investigations of the very same ideas in Template Haskell.

There is an important qualification to this efficiency concern. The GHC implementors adopted `generic-deriving` [30] as the basis of GHC-preferred generic programing approach as of the 7.2.1 release in August 2011. The `generic-deriving` approach is a derivative of `instant-generics` that uses the exact same Haskell features: type families and type classes. Because GHC adopted it, and those are two major features of Haskell, I believe that con-

tinued improvements to the GHC optimizer will result in better efficiency of definitions that are automated as-is with `instant-generics`and `generic-deriving`. Morever, `yoko` relies on the exact same Haskell features, so it will also improve. Essentially, I am hitching my wagon to the implementors of GHC, `instant-generics`, and `generic-deriving`, and they have a great record.

Even without the confidence that the efficiency of `yoko` will improve as a consequence of the related efforts by other researchers, the benefit of `hcompos` still outweighs its efficency detrminents in many scenarios. At the very least, it can be used during the prototyping phase or used as a specification for testing the more optimized version. This has been sufficient for my applications, which is why I have not further investigated the efficiency and means of optimization. In short, it will be harder to optimize in the presence of `hcompos` than in the presence of `compos`, but the initial overhead of `hcompos` is not prohibitive for most applications.

Bad Type Errors

This issue of unclear type errors is appeased by roughly the same argument as for efficiency concerns. This is again a prevalent issue in Haskell because it accomodates embedded domain-specific languages (EDSLs) so well; this is a well-attended field of research. I expect its result to benefit `yoko` error messages as it advances because `yoko` (and other `instant-generics`-based approaches) can be cast as EDSLs for programming with the structure of data types. Even so, the type errors resulting from a misapplication of `precise_case` or `hcompos` usually include at least one error that indicates the mistake and its location. The major issue is that the compile-time computation determined by type family instances and reduction of type class constraints via instance selection do not preserve enough information to determine their origin. GHC has cannot differentiate the useful errors from the less useful ones that arise as a cascade, and so it just reports them all.

The important case is when `hcompos` is used to automate a case that is not actually tag homomorphic. This happens in two ways, according to the two conditions required by the schema for tag homomorphism. First, the codomain type might not have a constructor with the same name as one in the domain. With the current definitions, this results is an opaque error message that only successfully indicates the location of the error.

```
Tree.hs:35:27:
    Couldn't match type 'Data.Yoko.MaybeKind.Nothing'
                   with 'Data.Yoko.MaybeKind.Just (N dc')'
    In the second argument of '(.)', namely 'w'
    In the expression: runIdentity ∘ w
    In an equation for 'rmWeights':
        rmWeights = runIdentity ∘ w where
          w t = hcompos w $ disband t
Failed, modules loaded: WTree, MonoHCompos.
```

I generate this type error by removing the special treatment of the `WithWeight` constructor from the definition of `rmWeights'`. The resulting definition just tries to automate the conversion of every constructor by disbanding the input and then immediately invoking `hcompos`. The only thing the error tells the confused programmer is what line the problem is actually on. It gives no domain-specific (*i.e.* `yoko` or `hcompos`-based) reason that the `Nothing` and `Just` types are involved at all. As I said, this is a prevalent problem from EDSLs in Haskell. The only solution I know of is to add extraneous parameters to the involved type families and type classes that preserve enough information to reflect the related errors' origin. In this case, I could enrich the `Nothing` type with an additional parameter and the failure case of `FindDC` in order to generate a type error like the following.

```
Tree.hs:35:27:
    Couldn't match type 'Data.Yoko.MaybeKind.Nothing
                            (CouldNotFindMatchingConstructorFor
                              (WTree.WithWeight_ a))'
                   with 'Data.Yoko.MaybeKind.Just (N dc')'
...
```

This error now correctly specifies where the error originates and why.

The second condition of the tag homomorphic schema that can be violated is the compatibility of the matched constructors' fields, modulo the conversion. For example, again reusing the `rmWeights'` function, if the `Leaf` constructors in the precise codomain had an extra field, say of type `String`, the programmer would see the following error.

```
Tree.hs:35:27:
    Could not deduce (MapRs
                        (W.Tree a w → Identity (Tree a))
                        (Dep a)
                        (Dep a :*: Dep [Char]))
      arising from a use of ‘w’
      at Tree.hs:(35,1)-(58,2)
    Possible fix:
      add (MapRs
             (W.Tree a w → Identity (Tree a))
             (Dep a)
             (Dep a :*: Dep [Char])) to the context of
        the type signature for
          rmWeights :: W.Tree a w → Tree a
      or add an instance declaration for
         (MapRs
            (W.Tree a w → Identity (Tree a))
            (Dep a)
            (Dep a :*: Dep [Char]))
    In the second argument of ‘(.)’, namely ‘w’
    In the expression: runIdentity ∘ w
    In an equation for ‘rmWeights’:
        rmWeights = runIdentity ∘ w where
          w t = precise_case (hcompos w) t $ one $ λ (W.WithWeight_ t _) → w t
Failed, modules loaded: WTree, MonoHCompos.
```

An error like this would be generated for every mismatch in each pair of constructors' fields. This can quickly fill up multiple screens, which intimidates programmers that are unfamiliar with EDSL type errors. While this error actually includes enough information to explain the problem that information is presented in terms of the data types representation and the `MapRs` class, which is supposed to internal to the generic definition of `hcompos`. It is not intended for the common programmer using `hcompos` to understand either the type class

or the representation types. I can improve the problem similarly as last time by adding a parameter to the `MapRs` class that indicates what is going on in domain-specific terms.

```
Tree.hs:35:27:
    Could not deduce (MapRs
                        (W.Tree a w → Identity (Tree a))
                        (TryingToConvertBetween (W.Leaf_ a w) (Leaf_ a w))
                        (Leaf_ a)
                        (Dep a)
                        (Dep a :*: Dep [Char]))
...
```

Presumably, the programmer is now aware that the two leaf constructors have incompatible fields and need to be handled explicitly using `precise_case`. The improved type errors resulting from both varieties of mistaken automation via `hcompos` now actually indicate both the location and details of the mistake. I said before that these are the two important cases; I suspect they are the most common mistakes to make.

Another easy mistake is for the programmer to use an original constructor in the pattern for a `precise_case` special case instead of the corresponding fields type's constructor. Again for the `rmWeights'` function, the programmer might accidentally write `λ(WithWeight t) → ...` instead of `λ(WithWeight_ t) → ...`. This results in two errors. The first of which is both unhelpful and enormous: it is a noisy 762 lines of unreduced applications of type families with fully qualified names, since the involved names are not imported because they are not (supposed to be) part of the programmer-facing `yoko` interface. The second error, though is short and informative.

```
Tree.hs:27:27:
    Couldn't match type ‘Range (W.Tree Int Char)’
                    with ‘W.Tree Int Char’
    In the second argument of ‘(.)’, namely ‘w’
    In the expression: runIdentity ∘ w
    In an equation for ‘rmWeights’:
        rmWeights = runIdentity ∘ w where
          w t = precise_case (hcompos w) t $ one $ λ (W.WithWeight t _) → w t
Failed, modules loaded: WTree, MonoHCompos.
```

If the programmer understands that the `Range` type family is only supposed to be applied to fields types—which seems like a reasonable prerequisite for a progammer choosing to employ `hcompos`—then this error both locates and explains the mistake. It just unfortunately occurs after the other large error. Similar mistakes like using the wrong type's fields type results in the same issue: accurate informative messages that might be hidden among the other messages, some of which are unhelpful and intimidatingly large.

The problem of very large unhelpful errors is inherent to type families' compile-time computation. Because it is a recent and important feature addition to Haskell, I anticipate the development of techniques for dealing with these sort of situations errors. Even a simple interface for summarizing type errors in, say, 20 lines or less while being able to expand them as necessary would vastly improve this situation.

I listed type errors as the most severe issue in the list of detriments at the beginning of this section. I consider bad type error issues more severe than inefficiency because they make it more difficult for programmers to use `hcompos`. On the other hand, the type errors do successfully identify the location and explanation of the mistakes I have made when developing with `hcompos`. Over my own years of EDSL programming, I have learned the lesson that I should investigate the small type errors first and even then look for the one that is easier to solve, since it might have caused the others in a cascade effect. Unfortunately, until other techniques are developed for retaining domain-specificity for EDSL type errors, the adding the extraneous parameters for the sake of provenance and asking for a bit of understanding and patience from the programmer is the best I can do with respect to the `precise_case` and `hcompos` errors I have seen. If the programmer understands even just the major `yoko` and `hcompos` concepts and focuses on the small error messages, they can find the bug.

Summary

The above discussions show that the fourth claim is evidenced, though less absolutely than are the others. While the `hcompos` operator does provide even greater benefits than those of `compos`, its richer type and complex generic definition lead to comparatively worse efficiency and type errors. Neither of these are so extreme as to make it unusable, though: the current efficiency is enough for at least the prototyping phase and the type errors include the essential information among the noise. Also, both issues are largely due to widely-recognized deficiencies of embedding DSLs in Haskell with GHC. Since this is a major use case of Haskell, I anticipate the issues will receive attention. Furthermore, any improvements will likely integrate with `yoko` because it only relies on type families and type classes. Even without this optimistic opinion, the code simplification benefits of `hcompos` outweight the efficiency and type error issues. The generalization from `compos` to `hcompos` worsens the efficiency of the defined function and the possible type errors, but not to such a degree that its benefits are eclipsed.

7.3 Comparison to a Conventional Lambda Lifter

Claim *The `hcompos` operator can be used to define non-trivial functions.*

In this comparison, I solve Example 2.3 for a second time to demonstrate how my approach enables precisely-typed programming with non-trivial functions. My first solution in Section 5.5 showed how expressive `hcompos` is and also how its rigid interface may demand some clever techniques. This solution is instead a reimplementation of the lambda lifting function defined by Jones and Lester [23]. I use the same decomposition as they do, but use `hcompos` to define each part. This makes it cheaper for me to use precise types between each component function. They factored the function into three parts.

1. The `freeVars` function computes all of the nodes' free variables and caches this information on each node as an annotation.

2. The `abstract` function discards the cached information for all nodes except lambdas.

3. The `collectSCs` function extracts those lambdas as top-level supercombinator declarations and replaces them with references to those new declarations. Since it is a thin wrapper, I will instead deal with its main workhorse function `collectSCs_e`.

The lambda lifting function is the composition of the three functions. The involved concepts are familiar from my first solution in Section 5.5. There are three major differences between this algorithm and that one. First, this one uses an atomic representation for names. Because it just uses strings instead of de Bruijn indices, it does not require anything as exotic as a backwards state monadic effect. Second, it is modularly defined as a composition of three smaller functions. This makes room for numerous precise types in the sequence. Third, it is to some extent designed with efficiency in mind. In particular, it involves caching the result of a code analysis in the term representation, which presents yet another compelling use case for `hcompos`.

In this comparison, I will discuss the three function definitions presented by Jones and Lester in turn. For each, I will summarize the original authors' discussion that is relevant to the benefits provided by `hcompos` and juxtapose the new code along-side the old. In order to make the comparison more accurate, I have updated the older definitions to use modern coding techniques, such as monads and applicative functors and their many related combinators. While I have changed a few variable names and fused a few local definitions for the sake of presentation, I have tried to keep the translation minimal. I recommend looking to the original paper [23]—it is enjoyable and informative.

Step Zero: The Term Representation Data Types

In their paper, Jones and Lester allocate an entire section to discussing their choice of data types. They end up using two principal types.

```
data Expr n
  = EConst String
  | EVar n
  | EAp (Expr n) (Expr n)
  | ELam [n] (Expr n)
  | ELet IsRec [(n, Expr n)] (Expr n)

type Expression = Expr Name

data AnnExpr a n = Ann a (AnnExpr' a n)

data AnnExpr' a n
  = AConst String
  | AVar n
  | AAp (AnnExpr a n) (AnnExpr a n)
  | ALam [n] (AnnExpr a n)
  | ALet IsRec [(n, AnnExpr a n)] (AnnExpr a n)
```

They remark that `Expr n` is almost isomorphic to `AnnExpr () n`, but they decide to keep the separate data types. They lament the dilemma: either juggle the duplicate sets of constructors or obfuscate the code with ubiquitous `()` values and patterns. They prefer the sets of constructors, even without the assistance of `hcompos`.

Note how the mutual recursion between `AnnExpr` and `AnnExpr'` annotates each node in the representation with a value of type `a`. Jones and Lester use this in order to cache the results of the free variables analysis within the representation. As part of their modularity theme, they remark that they rely on the cached results in other parts of their compiler as well.

Even though the `AnnExpr` and `AnnExpr'` data types are mutually recursive, the arguments to `hcompos` in the following definitions are simple monomorphic functions like in all the examples prior to the previous section. This is possible because each of the mutually recursive data types only occurs in the other: all recursive occurrences within each data type

are occurrences of the same type. One reasonable variation that would require the support for mutually recursive data types addin the previous section would use a fresh data type instead of tuples to structure each declaration. In that case, the `AnnExpr'` data types would recur both at `AnnExpr` and also at this new data type for declarations; therefore uses of `hcompos` for the `AnnExpr`' that automate the cases for lets would require a polymorphic function type. The other enrichment from Section 6, compound recursive fields, is already present in the let constructors, so the `Par1` enrichment will be used. This will not be evident in the following definitions though, since it is silently handled by the `yoko` Template Haskell. In particular, the let constructors' field for the definitions will be represented with the type `Par1 [] (Par1 ((,) n) (Rec ...))`. Both `[]` and `(,) n` are covariant functors, so the `Traversable` instances required by the `Par1` instance for `hcompos` are available—`hcompos` is readily applicable in the definitions below.

Step One: Caching the Set of Free Variables

The first function in the chain computes a subterm's free variables and caches the result at each node using the `AnnExpr` type. The entire function definition of Jones and Lester [23] is in the top half of Listing 7.1. Since `AnnExpr` is essentially a pair, the function can be understood in two parts. The first part determines a terms' free variables and the second part converts that term in the `Expr` type to an analogous term in the `AnnExpr'` type. This is clear in the cases for `EConst` and `EVar`. For the `EAp` case, it becomes apparent that these two parts happen simultaneously, since only one recursive call is necessary per subterm. Indeed, this case and the `EConst` case are structurally recursive in the first part and tag homomorphic in the second part. The `ELam` and `ELet` cases are more complicated because those syntactic constructors involve binders that affect the computation of free variables. However, they are both still computing two parts.

The part of the `freeVars` definition that computes the free variables could be automated for the `EConst` and `EAp` constructors using just the techniques of `instant-generics`. Because

Listing 7.1: The old and new `freeVars` functions.

```
freeVars :: Expression → AnnExpr (Set Name) Name
freeVars = w where
  w (EConst k)              = Ann Set.empty (AConst k)
  w (EVar v)                = Ann (Set.singleton v) (AVar v)
  w (EAp e1 e2)             = Ann (free1 `Set.union` free2) (AAp e1' e2')
    where e1'@(Ann free1 _) = w e1
          e2'@(Ann free2 _) = w e2
  w (ELam args body)        = Ann (Set.difference bFree argSet) (ALam args body')
    where argSet = Set.fromList args
          Ann bFree body' = w body
  w (ELet isRec ds body)    =
    Ann (dsFree `Set.union` bFree') (ALet isRec (zip binders rhrss') body')
    where binderSet = Set.fromList $ map fst ds
          rhss' = map (w ∘ snd) ds
          rhssFree = Set.unions $ map (λ(Ann a _) → a) rhss'
          dsFree = (if isRec then (`Set.difference` binderSet) else id) rhssFree
          body'@(Ann bFree _) = w body
          bFree' = bFree `Set.difference` binderSet

----------------------------------------

freeVars :: Expression → AnnExpr (Set Name) Name
freeVars = runIdentity ∘ w where
  w e = (λx → Ann (alg x) x) <$> hcompos w (disband e)

  alg (A.Const _)                          = Set.empty
  alg (A.Var v)                            = Set.singleton v
  alg (A.App (Ann free1 _) (Ann free2 _)) = free1 `Set.union` free2
  alg (A.Lam ags (Ann bFree _))            = bFree `Set.difference` argSet
    where argSet = Set.fromList args
  alg (A.Let isRec ds (Ann bFree _))       = dsFree `Set.union` bFree'
    where binderSet = Set.fromList $ map fst ds
          rhssFree = Set.unions $ map (λ(_, Ann a _) → a) ds
          dsFree = (if isRec then (`Set.difference` binderSet) else id) rhssFree
          bFree' = bFree `Set.difference` binderSet
```

the two parts of the function share a recursive component, the generic part would map an `AnnExpr' (Set Name) Name` to `Set Name`; it would be defined based on the structure of `AnnExpr'`, with special cases for the constructors involving with variables: `EVar`, `ELam`, and `ELet`. I omit this automation from my lambda lifter both because it is of little immediate benefit and because this generic programming is not enabled by my technical contributions. Instead, the point I intend to emphasize is that, with `hcompos`, the programmer no longer needs to explicitly map the `Expr` constructors to the `AnnExpr'` constructors one-by-one.

I use the same `Expr` and `AnnExpr` data types in my implementation of `freeVars`, given in the bottom half of Listing 7.1; my function has the same type as does theirs. However, because I am using `hcompos`, I define the data types in separate modules as previously discussed, so that I can give the corresponding constructors the same names. I import the modules qualified; I write, *e.g.* `E.Const` and `A.Const` where Jones and Lester wrote `EConst` and `AConst`. The simple Template Haskell splices `yokoTH ''Expr`, `yokoTH ''AnnExpr`, and `yokoTH ''AnnExpr'` derive all of the necessary fields types and `yoko` instances.

The `yoko` interface and the `hcompos` operator allow me to totally automate the part of the `freeVars` function that converts between the `Expr` and `AnnExpr'` data types. Whereas every case of the old definition maps an `Expr` constructor to the corresponding `AnnExpr'` constructor, my definition only explicitly computes the free variables for each constructor. Specifically, the local definition of `w` uses `hcompos` to simultaneously convert the recursive fields of the input `Expr` value and also convert its constructors to the `AnnExpr'` data type. The `w` function then applies the local function `alg` in order to compute the free variables for the current node from those of its subterms. In `alg`, I explicitly implement the part of the function that computes free variables in the same way as did Jones and Lester. Again, this part could be partially automated without using my specific contributions. I call the function `alg` because it behaves like the algebra argument to a catamorphism[1]. Since `hcompos` recurs

[1]It actually behaves like a paramorphism, though the original subterm is not needed for this part of the function.

into the structure first, my computation of free variables uses only the recursive results at each subterm. The old function could be similarly reorganized, but without using `hcompos` to automate the conversion, it would result in duplicated pattern matching.

Comparing the two definitions of `freeVars` in Listing 7.1 shows yet again that `hcompos` can be used to simplify functions that convert between similar data types. In this example, Jones and Lester designed a variant of a data type that adds annotations to every node. The Haskell type system does not naturally accomodate this sort of derived data type. The original authors lament this fact, but choose to manage the duplicate set of constructors anyway. This evidences another way that such pairs of similar data types arise naturally in practice, and also that it is natural to name their constructors almost identically.

Though `hcompos` simplifies the `freeVars` function significantly, it makes an even bigger difference for `abstract` and `collectSC_e` functions. This is because those functions are less specific to binding, and so all cases except the one for lambda are tag homomorphic.

Step Two: Discarding the Extraneous Caches

After caching the free variables in each constructor, Jones and Lester use that information in the second pass to replace all of the lambdas with β-equivalent applications of supercombinators. Their definition of the `abstract` function is again above my own in Listing 7.2. Excepting the abandonment of the cache, all non-lambda cases are simply tag homomorphic conversions back to the `Expr` data type. Instead of caching the data at every constructor, they could have fused the `freeVars` and `abstract` functions together, using a `Writer` monad to temporarily accumulate the computed free variables. However, the original authors mention that the `freeVars` function was used in other parts of their compiler, where its cached results might not have been immediately discarded.

Defining the two functions separately makes `abstract` burdensome to define. Indeed, the original authors relegate all cases except the lambda case to the appendix of their paper, which reinforces my general claim that programmers find tag homomorphism tedious to

Listing 7.2: The old and new `abstract` functions.

```
abstract :: AnnExpr (Set Name) Name → Expression
abstract (_   , AConst k)           = EConst k
abstract (_   , AVar v)             = EVar v
abstract (_   , AAp e1 e2)          = EAp (abstract e1) (abstract e2)
abstract (free, ALam args body)     = foldl EAp sc $ map EVar fvList
  where fvList = Set.toList free
        sc = ELam (fvList ++ args) (abstract body)
abstract (_   , ALet isRec ds body) =
  ELet isRec (map (second abstract) ds) (abstract body)

----------------------------------------

abstract :: AnnExpr (Set Name) Name → AnnLam (Set Name) Name
abstract = runIdentity ∘ w where
  w (Ann free e) = precise_case (hcompos w) e $ one $
    λ(A.Lam_ args body) → AL.Lam free args <$> w body
```

handle manually! They discuss only the lambda case, as it is the only interesting one. It replaces each lambda with a supercombinator (computed by the recursive call to `abstract`) applied to the required free variables as determined by the cached set of names. This transformation preserves functional equivalence.

My version of `abstract` has a different type that reflects a slight change. It uses the `AnnLam` data type, which is defined in its own module as follows.

```
data AnnLam a n
  = Const String
  | Var n
  | App (AnnLam a n) (AnnLam a n)
  | Lam a [n] (AnnLam a n)
  | Let IsRec [(n, AnnLam a n)] (AnnLam a n)
```

I refer to its constructors as if I imported the module qualified with the prefix `AL`. This derived data type also adds an annotation to the `Expr` data type, but only to the lambda constructor. I use this data type as the codomain of `abstract` instead of converting back to `Expr` since this type is more precise. In the range of the original `abstract`, it is unclear which applications are from the original program and which were added in order to be explicit

about lambda's dependencies on their lexical environment. My version uses the `AnnLam` data type to retain the cached annotation of free variables in order to preserve the distinction between original and generated applications. As the case for lambdas in the old `abstract` function shows, the set of free variables is the nub of the necessary information. So I define my `abstract` as a conversion from the fully annotated `AnnExpr` data type to the minimally annotated `AnnLam` data type.

I defer actually replacing the lambda with an equivalent applied supercombinator until the last step in the lambda lifting chain. This is in accord with the theme of modularity that Jones and Lester maintained in their paper: other functions in the compiler may also benefit from retaining annotations only on lambdas. Those authors may have done the same if it were as easy for them to manage multiple derivatives of `Expr`. I have the additional benefit of `hcompos`, which nearly renders my `abstract` function a one-liner. Note that my version would be no more complicated if I immediately implemented the transformation back to `Expr`; only the right-hand side of the `A.Lam_` alternative would need to change.

Step Three: Extracting the Supercombinators

The final step in lambda lifting function extracts the annotated lambdas out as top-level supercombinator declarations. The old and new definitions of the `collectSC_e` are in Listing 7.3. Both definitions use the same monad, which just provides a supply of fresh names with a state monad effect and collects the generated supercombinator definitions in a list as an output monad effect. I leave out the standard definitions of `newName` and `tell`.

Although the old `ELet` case is slightly obfuscated by the need to traverse the structure of its list of definitions, it is tag homormphic like all other the cases except lambda. Therefore, the simplification due to `hcomps` is again drastic: it automates all but the interesting case. That case is slightly different in the new version only because I updated it to include the transformation that I deferred from the `abstract` function. The lambda case now uses the cached set of free variables to replace the original lambda, as did the old `abstract` function.

Listing 7.3: The old and new `collectSC_e` functions.

```
collectSCs_e :: Expression → M Expression
collectSCs_e = w where
  w (EConst k)              = return $ EConst k
  w (EVar v)                = return $ EVar v
  w (EAp e1 e2)             = EAp <$> w e1 <*> w e2
  w (ELam args body)        = do
    body' <- w body
    name <- newName "SC"
    tell [(name, args, body')]
    return $ EConst name
  w (ELet isRec ds body) = ELet isRec <$>
    mapM (λ(name, rhs) → (,) name <$> w rhs) ds <*> w body

----------------------------------------

collectSCs_e :: AnnLam (Set Name) Name → M (SC Name)
collectSCs_e = w where
  w e = precise_case (hcompos collectSCs_e) e $ one $
    λ(AL.Lam_ free args body) → do
      body' <- w body
      name <- newName "SC"
      tell [(name, free ++ args, body')]
      return $ foldl SC.App (SC.SC n) $ map SC.Var free
```

Each lambda becomes an application of a reference to the newly generated supercombinator with the requisite variables from the original lambda's environment as arguments. As a consequence, the range of `collectSCs_e`—and of the entire lambda lifting function—will not contain any lambdas. Jones and Lester lamented that it would require another data type declaration in order enforce this invariant in the type system. And that was the extent of their discussion: the idea of managing even a third data type was too unbearable to seriously consider. With `hcompos`, though, it is easy to use the following data type as the precise codomain of `collectSCs_e`.

```
data SC n
  = Const String
  | Var n
  | App (SC n) (SC n)
  | SC n
  | Let IsRec [(n, SC n)] (SC n)
```

The `SC` data type is simply a variant of `Expr` in which the `Lam` constructor is replaced with a constructor `SC` that models references to top-level supercombinator declarations.

Summary

I reimplemented the lambda lifter of Jones and Lester [23] using more precise types while also drastically simplifying the definitions. My principal technical contributions, the `yoko` generic view and the `hcompos` operator, made possible both the precise types and the simplification. Jones and Lester originally designed the data types in this example as a minimal but expressive internal representation of functional programs. As such, they are relatively small with five constructors, three of which directly involve variables or binding. Even so, the `hcompos` operator can automate 13 out of 15 cases across all three functions that would have otherwise required manual conversion between the data types. Moreover, lambda lifting is an important and established algorithm, and its definition benefits both from the use of precise types and the automation enabled by `hcompos`.

This example introduced an interesting new scenario that naturally gives rise to data types with similar constructors with similar names. That a total of three such scenarios that I have identified within this dissertation.

1. The most common scenario involves unequal instantiations of a parametric data type. The `fmap` method is a good example of a function that converts between two such data types.

2. The precise codomain of most previous examples have been (manually) derived from the domain by adding or removing a few constructors. In particular, a conversion's precise codomain can gaurantee that a constructor from the domain does not occur in its range.

3. The new scenario involves derived data types that annotate the original constructors. My definition of the `freeVars` function shows that the current definition of `hcompos`

can automate a conversion from a data type to one that is annotated uniformly at each node (`AnnExpr`). My definition of the `abstract` function then showed that the partitioning of `precise_case` can also be used in order to handle a data type that annotates just a few constructors (`AnnLam`). More sophisticated variants of `hcompos` could more fully automate the ad-hoc addition of fields for annotations, but the need for that is not yet clear.

Another important compiler phase that exhibits the new source of tag homorphisms is type reconstruction. Such a pass converts an external syntax that allows some (or all) type declarations, annotations, and instantiation to be omitted into an internal syntax that carries those types as annotations. Both options are appropriate: while every node in the representation could cache its type, the internal syntax could instead cache the type only at a minimum number of essential positions (like lambdas' domains) if a more compact representation were preferred.

These specifically identified scenarios give rise to conversions between data types with many similar constructors. Functions impelementing those conversions can be automated with `hcompos`, which establishes many basic opportunities to use `hcompos`. With the developments of Section 6, these conversions can additionally involve complex data types, including mutual recursion and compound recursive fields, and support for indexed data types looks promising. Thus, the third claim has been thoroughly evidenced by the examples throughout this dissertation; `hcompos` is not just for toy examples.

7.4 Comparison to `nanopass`

Claim *Enhanced with the* `hcompos` *operator, standard Haskell mechanisms support the static and strong typing and automation of the nanopass methodology better than does the Scheme implementation of Sarkar et al. [39]. I make no claim about relative efficiency.*

I observe that the Haskell type system is significantly more expressive than the Scheme type system. My task here is therefore to show that `hcompos` provides automation comparable to that of the pass expander. Sarkar et al. [39] summarize the main features of their extension to the basic notion of micropass with the following sentence.

> A nanopass compiler differs from a micropass compiler in three ways: (1) the intermediate-language grammars are formally specified and enforced; (2) each pass needs to contain traversal code only for forms that undergo meaningful transformation; and (3) the intermediate code is represented more efficiently as records, although all interaction with the programmer is still via the [Scheme] s-expression syntax.

My contributions mimic the first two improvements in Haskell. Sarkar et al. had to use an additional mechanism to specify and statically enforce the grammars within the dynamically typed Scheme language. Because I use Haskell instead of Scheme, I specify grammars naturally as data types and so adherence to those grammars is automatically enforced by the Haskell's static and strong typing. Those authors also develop their *pass expander* in order to simplify the definition of passes from the micro- to the nano-scale by omitting tag homomorphic cases. My technical contributions overcome the challenges of implementing the pass expander in the static and strong typing of Haskell's data types. Though I have not focused on their third difference, efficiency, I partly chose `instant-generics` as a foundation for my techniques because it is comparatively more efficient than other generic programming methods. I briefly discuss the technical details at the end of this section.

In this section, I visit each technical feature that Sarkar et al. explain in their paper and compare to the corresponding feature I have demonstrated in this dissertation. In accord with their own high-level summary of the differences between micropasses and nanopasses, I partition my discussion of their features into two classes: those features that help the

programmer specify and enforce adherence to grammars and those features that help with the definition of functions that consume and produce terms from those grammars.

Specifying and Enforcing Grammars

The developers of nanopasses began by enriching Scheme with a mechanism for declaring intermediate languages. For example, the following is a declaration listed in Figure 2 in their paper [39].

```
(define-language L0 over
  (b in boolean)
  (n in integer)
  (x in variable)
 where
  (Program Expr)
  (e body in Expr
    b n x
    (if e1 e2 e3)
    (seq c1 e2) ⇒ (begin c1 e2)
    (lambda (x ...) body)
    (e0 e1 ...))
  (c in Cmd
    (set! x e)
    (seq c1 c2) ⇒ (begin c1 c2)))
```

The `define-language` construct is their mechanism for encoding grammars in their toolset. The above declaration encodes the following grammar in BNF with the Kleene star.

```
L0      → Program
Program → Expr
Expr    → boolean | integer | var
        | (if Expr Expr Expr)
        | (seq Cmd Expr)
        | (lambda (var*) Expr)
        | (Expr (Expr*))

Cmd     → (set! var Expr)
        | (seq Cmd Cmd)
```

Excepting their use of declared meta-variables to specify occurrences of terminals and non-terminals, this is a transliteration. The remaining difference in information content is the two ⇒ annotations on the `seq` productions. Sarkar et al. call these *properties*; I discuss them below.

Language definitions like this one are used to specify the "input" and "output" languages of each nanopass. Using that information, the nanopass compiler enforces that the result of a nanopass must adhere to the grammar encoded by the pass's output language. Languages serve the same purpose for passes as domain and codomains serve for functions in Haskell.

The correspondence between a language declaration and a Haskell data type declaration is well-understood; grammars are regularly encoded as data types in Haskell [22, §2.6]. The `L0` data type would be declared as follows.

```
type L0 = Program
type Program = Expr

data Expr
  = Boolean Bool | Integer Integer | Variable String
  | If Expr Expr Expr
  | Seq_Expr Cmd Expr
  | Lambda [String] Expr
  | App Expr [Expr]

data Cmd
  = SetBang String Expr
  | Seq_Cmd Cmd Cmd
```

Any pass would be a function using the `L0` data type as its domain or codomain. In this way, Haskell data types and its static and strong typing subsumes the nanopass "templating mechanism" for enforcing that inputs and outputs of passes always adhere to the grammars of the corresponding intermediate languages. Note that this naturally gives rise to mutually recursive data types.

There are four differences between the `define-language` construct and Haskell data types:

1. the use of meta-variables,
2. the allowance of one productions without a leading terminal,
3. the unrestricted reuse of constructor names,
4. the *language inheritance*, and
5. the "properties" (like ⇒ in `L0` above).

The first two of these are just issues of syntactic sugar. In particular, a constuctor name must be provided for each variant of a data type. This is a small inconvenience compared to the naked productions for literal values and applications in the `declare-langauge` declaration. The second two deserve discussion.

I have already discussed that the `FindDC` algorithm used in the generic definition of `hcompos` requires that two constructors in separate data types must have the same exact name if they are supposed to correspond automatically. This is why I have declared data types in separate modules: use of `hcompos` requires the constructor names to be equivalent, but Haskell does not allow the same constructor name to be declared twice in the same module. On the one hand, this again just an issue of convenience. On the other hand, it is a heavyweight inconvience. To allow a similar convenience, the `FindDC` algorithm could be refined with a weaker prerequisite. For example, it could conservatively ignore prefixes or suffixes when comparing one fields type's `Tag` against another.

In order to achieve the full benefits of the nanopass methodology in the original Scheme implementation, the programmer can define similar grammars as extensions of existing grammars. For this reason, a `define-language` declaration can be specified as a series of modifications (*i.e.* deletions, additions, adjustments) to each production of some other language that was

already defined. This mechanism is referred to as *language inheritance* and is just a notational convenience. The pass expander will work for any two similar types, just like `hcompos`. This importantly provides more modularity by reducing coupling. With language inheritance, one of the languages must be identified as a derivative of the other. Without it, neither needs to be considered the ancestor of the other. Thus language inheritance is considered an overspecification. Its only comparative benefit is that it can avoid repeating the declaration of the shared constructors. I have not developed it, but it seems simple to define Template Haskell combinators to provide a comparable interface for the user's convenience.

The issue of properties is the only significant difference between nanopass languages and Haskell data types. Sarkar et al. [39] refer to various kinds of properties, but only demonstrate the $\Rightarrow$ "translates-to" property. In the example language `L0`, this property means that both `seq` constructs have equivalent semantics as the Scheme constructor `begin`; it is just ordering evaluation. Properties like this support features of nanopasses that I am not trying to emulate, such as inferring a interpreter for the entire language. I am interested only in the automation of their pass expander, as I discuss in the next subsection.

Defining Functions over Terms Adhering to Grammars

A nanopass is defined with a `define-pass` declaration. These include three parts: an input language, an output language, and a set of transformation clauses. Each transformatoin clause specifies how the pass transforms a non-terminal from the input language into the output language. Thus, the analog of passes in Haskell are just functions between two data types that model grammars.

There are notable parallels between the Haskell mechanisms for function definition and these three features of nanopasses:

1. the `void` and `datum` languages,
2. the subpatterns for results of structural recursion, and
3. the pass expander.

The `void` and `datum` are primitive languages built-in to the nanopass approach. They are used for the definition of passes that are run only for side-effects or result in some value that need not adhere to a grammar. As an example, a pretty-printing pass might target the `void` language, since it could just use IO effects to print the term to the screen, or `datum` if it built and returned a string value. Programmers comfortable with Haskell would immediately recognize that the `void` language corresponds to the type `m ()` for whicher monad `m` supports the necessary side-effects. And since Haskell functions are not as restricted as the `define-pass` syntax, there is no need for an analog of `datum`, data types encoding languages are just types like any other.

The nanopass system includes syntactic mechanisms for invoking transformations on subterms as part of the pattern matching process. All but one of their variants for this syntax are directly supported by the GHC language extension for *view patterns*, which allow arbitrary functions to be embedded with patterns. The unique syntax not supported allows the programmer to omit the name of the transformation, which is then determined by the involved input and output languages if only one such transformation exists. Since this is an instance of types determining values, it is related to the general notions of Haskell type classes and generic programming. I will only observe that the complete omission of the name is more similar of the recent *implicit calculus* of d. S. Oliveira et al. [11]. Like many other mismatched features, lack of support for this variant is a minor inconvenience. The Haskell

view patterns almost totally subsume this nanopass feature. Moreover, since passes are implemented as genuine Haskell functions, they could be defined with recursion combinators such as the cata- or para-morphism that provide the same code concision and are commonly automated via generic programming.

The only crucial nanopass feature missing from GHC Haskell is the pass expander. In the implementation of Sarkar et al., this is a code generator that fills in missing transformation cases in `define-pass` declaration based on the corresponding input and output languages' `define-language` definitions. It just generates tag homomorphic cases; the `hcompos` operator provides the exact same automation. The only difference is that the `define-pass` syntax looks more natural than the `precise_case` syntax and that the pass expander is invoked implicitly for all uses of `define-pass`. On the other hand, the Haskell nanopasses are just normal functions, so they invoke `hcompos` explicitly and use `precise_case` for the cases that are not tag homormophic in order to avoid type errors. As I have shown throughout this disseration, that is merely an inconvenience.

GHC Haskell in combination with my generic definition of `hcompos` subsumes almost every benefit of the original nanopasses implemention in Scheme. The missing features are all minor inconveniences along the lines of syntactic sugar. This discussion has therefore evidenced the second claim.

7.5 Final Evaluation

I repeat my claims here for ease of reference.

1. The `hcompos` operator provides the programmer with more automation than does the `compos` operator of Bringert and Ranta [7].

2. Enhanced with the `hcompos` operator, standard Haskell mechanisms support the static and strong typing and automation of the nanopass methodology better than does the Scheme implementation of Sarkar et al. [39]. I make no claim about relative efficiency.

3. The `hcompos` operator can be used to define non-trivial functions.

4. The benefit of `hcompos` as summarized by the other three claims is greater than its detriments.

I have designed these claims so that they emphasize the worthwhileness of my `hcompos` operator. The operator enables the established benefits of the nanopass methodology. It surpasses the usefulness of the `compos` operator, which is implemented by many Haskell generic programming libraries. It can be used within defintions of non-trivial functions like lambda lifting without necessarily changing the overall shape of the conventional definition—not to mention the myriad smaller examples from earlier chapters. And though it likely brings inefficiency for run-time and adds noise to the type errors, its detriments are not extreme enough to outweight the benefits of its automation. Furthermore, it will likely benefit from the advances with respect to efficiency and domain-specific type errors that I anticipate for future ESDLs in Haskell, which is still an active field of research.

I have focused on the `hcompos` operator because it is the culmination of my technical contributions. Even so, the delayed representation (*i.e.* fields types) and the `precise_case` operator for partitioning those disbanded data types can be useful independently of `hcompos`. Specifically, Example 2.1 shows that `precise_case` can be used to enable use of `instant-generics` techniques that exhibit fine-grained typing concerns. My first solution of Example 2.3 also indicates that fields types can be used to extricate a complex case from larger definitions as a separate declaration with an indicative type. Recall how just the types of `llVar` and `llLam` reflect the expected semantics of each function's argument.

In this chapter, I have evidenced that programmers can use the technical contributions of my dissertation to simplify their function definitions and that these benefits are not unique to toy examples. My `hcompos` operator generalizes the `compos` operator of Bringert and Ranta [7] in order to support automation of conversions between two similar types. The resulting automation drastically reduces the cost of using more precise types throughout an entire

program, which provides the engineering benefits of the nanopass methodology of Sarkar et al. [39] to Haskell programmers. The underlying ideas make it easier for programmers working in statically- and strongly-typed functional languages like Haskell to use more precise types.

Page left intentionally blank.

8 Related Work

The two works most related to mine are that of Bringert and Ranta [7] and that of Sarkar et al. [39], which are thoroughly discussed in Section 2. Many existing generic programming techniques can generically define `compos`, so programmers can often use it "for free" to improve the modularity of their definitions. I add heterogeneity to `compos` in order to make its benefits available when defining the property-establishing functions that are pervasive in exactly typed programs. The most important resulting benefit is that `hcompos` serves as an analog of the *pass expander* of Sarkar et al..

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

8.1 Exact Types

While the most exact possible types require type dependency, I am interested in the best approximation supported by mainstream non-dependent types. Even so, I believe my basic approach may be adapted to the dependent setting, where a notion similar to type exactness is known as *adequacy* [19]. In the non-dependent setting, however, I have unfortunately found little research on exact types and their software engineering benefits. The major works are Turner's *total functional programming* Turner [45] and generalized algebraic data types (GADTs) [48].

Exact types are essential for Turner's *total functional programming*, in which every function must be total. Specifically, all patterns must be exhaustive and all recursion well-founded. Without termination guarantees, exhaustive pattern matching could be trivially achieved by just diverging in what would be the omitted cases for non-exhaustive patterns. In this dissertation, I have adopted the stance of Danielsson et al. [12] and consider the challenges of exact types without concern for termination. The major benefit of exact types is that they make it straight-forward for patterns to be exhaustive without having to pass around handlers

for the "error" cases, since such cases are eliminated upstream by property-establishing functions.

The investigation of Turner's ideas seems to have faltered. I suspect that the Draconian requirement of termination lead researchers to embrace dependent types, as in Coq or Agda. Once termination is required, dependent typing is a natural addition in order to enjoy the Curry-Howard isomorphism. Moreover, if programmers want to use general recursion instead of being restricted to primitive combinators for recursion (such as catamorphism, paramorphism, anamorphism, apomorphism; see Meijer et al. [35]), the type-checker must then include a termination checker (and productivity checker for codata), which can sometimes be satisfied far more directly with the use of dependent types [6].

A significant advance in the type exactness of functional programs was the recent adoption of GADTs from type theory. A constructor of such a data type can restrict zero or more of the data type's type parameters. Those restricted parameters are called *indexes* while unrestricted parameters are called *uniform* parameters. Conventionally, constructors restrict the indexes by specifying equalities amongst them and possibly other concrete types. In Haskell, however, the restriction mechanism used by GADT constructors is naturally generalized to employ the qualified type mechanism. Accordingly, GADTs and equality constraints were simultaneously added to Haskell.

GADTs simulate type dependency in Haskell by providing *local type refinement*: within the alternative of a case for a GADT constructor, the indexes of that constructor's data type are refined by the constructor's type context. In this way, values (*i.e.* the GADT constructor) determine types (*i.e.* the indexes). Even this restricted form of type dependency allows for much more exact types; *c.f.* the recent work of Lindley [28]. I will discuss GADTs further in the next section, but the basic observation is that my work is an extension of generic programming, and the generic programming support for GADTs is nascent.

In this dissertation, I have focused on a particular kind of type exactness: subsets of constructors. I have found two principal existing approaches to explicit support for subsets of constructors (in statically-typed languages), one exemplified by the Common Algebraic Specification Language (CASL) [36] and one that seems unique to the OCaml programming language [27]. CASL and some other specification languages (*e.g.* PVS [37]) support named declaration of constructor subsets by declaring data types as non-disjoint unions of smaller data types [36, §4]. In a hypothetical CASL-Haskell pidgin language, this corresponds to data type declarations like the following.

```
data List a = Nil | data (NeList a)
data NeList a = Cons a (List a)
```

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 immodularity is unacceptable for programming in the large. In contrast, my 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`, `Codomain`, `DC`, and `DT` type families, and it is made practical by generating those instances as well as the related fields types for the user with Template Haskell, which is a common and accepted dependency for Haskell generic programming.

In OCaml, polymorphic variants allow any name, called a *variant*, to occur as if it were a constructor [18]. 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 compatible fields. Fields types, on the other hand, are associated with an original data type via the `Codomain` type family.

The type-indexed coproducts of Kiselyov et al. [25, §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. [25] specifically models a type-level cons and is therefore not associative.

The recent work on data kinds by Yorgey et al. [55] promotes constructors to types that superficially seem related to my notion of a fields type. These type-level constructors of data kinds, though, already have a notion of corresponding term-level value, the *singleton types* [15].

In summary, interest in exact types has been dominated by dependent types. Turner was interested in absolute correctness so he required termination, but it seems that total programming without dependent types was unfruitful. Moreover, dependent type theory is making headway into even non-dependent language, by ways of mechanisms like GADTs in Haskell. However, as I have shown in this dissertation in the spirit of Sarkar et al. [39], even providing support for subsets of constructors, which is rare and underdeveloped at the language-level, can provide significant software engineering benefits.

8.2 Generic Programming

All of my technical contributions involve generic programming in Haskell, which is an active and rather mature area of research. I will limit this discussion to the approaches most related to `yoko`'s `instant-generics` foundations: I omit the informative historical trajectory along SYB [26], PolyP [20], Generic Haskell [10], and many others. I detailed the `instant-generics` approach itself in Section 3.3.

Generic programming techniques are broadly characterized by the *universe* of type that can represent (*i.e.* manipulate generically). `yoko` has the same universe as `instant-generics`, with more complete support for compound recursive fields. I believe `yoko`'s enhancements

are orthogonal to other extensions of `instant-generics` and will investigate integration as future work. I also believe that some generic programming techniques not based on `instant-generics` also admit extensions comparable to mine. These beliefs are rooted in the simplicity and orthogonality of my basic idea of focusing on constructor names as compared to existing approaches.

Nested Recursion and GADTs

The most exact types supported by Haskell require sophisticated data types not in the `yoko` universe. In particular, nested recursion [5] and GADTs are crucial to encoding many interesting properties, such as well-scoped 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 [29], is investigating more natural representations. One derivative of `instant-generics` that is better suited for these features is `generic-deriving` [30], which was recently integrated with GHC. Most of the `generic-deriving` representation types are simple liftings of the `instant-generics` types from the kind $*$ to the kind $*\rightarrow*$. The `generic-deriving` approach only represents type constructors, but it interprets $*$ types as $*\rightarrow*$ types that do not use the type parameter. Since type parameters are a prerequisite for nested recursion and GADTs, a representation designed for $*\rightarrow*$ types more naturally handles such sophisticated data types. I emphasize again that my essential idea of reflecting constructor names is orthogonal to these libraries' foundations and so I anticipate that they could be easily extended in the same way as `instant-generics`. Indeed, the step from `instant-generics` to the the work of Magalhães and Jeuring [29] is very incremental; the corresponding adjustment to `yoko` was brief and my initial investigations seem promising.

Binding

One important generic programming feature that is less obviously compatible with my extensions is automatic support for binding. Since much of the advanced generic programming research is motivated by providing functionality for the various intermediate representations

within compilers, some generic support for binding constructs is of particular interest. This is another active field of research with many open questions, but the `unbound` [52] library has explored one effective option. It adds a few special-purpose representation types to the basic ones like sums and products. These new representation types serve as a small type-level DSL for specifying binding within the data type definitions. For example, there is a primitive type for variable occurrences and a primitive type for scoped variable bindings. If a data type definition uses these primitives appropriately (and its other structure is conventionally representable), then the `unbound` library provides such nominal functions as capture-avoiding substitution and α-equivalence. Since `unbound` is not rooted in `instant-generics`, it is less obvious if it is amenable to my `yoko` extensions. In addition, I am not sure how `yoko` would interact with the notion of primitive, non-structural representation types (*e.g.* those for names and binders). Other (non-generic programming) approaches to binding also have similarly primitive data types, such as the `Var` constructor central to PHOAS [9] and the name and multi-bindings of Hobbits [53]. If these approaches to binding representation were to be combined with a generic programming technique similar to `instant-generics`, the result would involve non-structural representation types as in `unbound`. Thus, the combination of `yoko` with non-structural representation types is an important direction for future investigation.

Abstracting over Recursive Occurrences

One criticism by reviewers of my work has called for further justification of my choice of `instant-generics` as a generic basis. In particular, there are many similar approaches based on the notion of viewing regular data types as fixed points of functors [35, 41, 16, 46, 54], which the reviewers correctly observed would more naturally accommodate some of the `yoko` definitions. For example, in the fixed point of a functor approach, the `MapRs` class would instead just be `Traversable`. In short, my response to this criticism is that the development based on `instant-generics` is more accessible and best emphasizes the crucial features of

the `yoko` extensions. Also, knowing that nested recursion (which is not "regular") could lend further exactness, I wanted to avoid a technique that was fundamentally opposed to it.

The fixed point of functors approach to regular data types is educational and powerful. Viewing data types in this way explains some common functional programming concepts, such as folds, in a fundamentally general framework. Moreover, the *open recursion* (which is not actually recursion) inherent to the approach has many well-documented modularity benefits [4, 43, 47]. This open recursion and its various reflections at the value-level are the basic technique that provides the characteristic extensibility of *two-level data types* [41], which are data types that are explicitly defined as the fixed point of a functor. In particular, functors can be combined with the functor sum operator before being closed with the fixed point.

The most developed work along this vein is by Bahr and Hvitved [4], who improve upon the techniques summarized by Swierstra [44], making them more robust and usable in non-trivial programs beyond the academic examples of Wadler's "expression problem" [50]. In particular, they demonstrate how some extensible definitions of functions (*i.e.* algebras) can be almost directly reused on multiple similar data types within a hypothetical compiler pipeline. Moreover, their principle contribution is to identify a subset of algebras, which they call *term homomorphisms*, that exhibit some more useful properties. For example, term homomorphism compose just like functions. They also integrate well with annotation techniques common to the fixed point of functors approaches. The characteristic example of a term homomorphism is an elaboration, such as the one converting a let to an applied lambda.

For my comparative purpose, it is most interesting to note that term homomorphisms can define property-establishing functions. For example, the let elaborator guarantees the absence of the lets in the range. This looks as follows in a simplified form using a simple algebra instead of an explicit term homomorphism.

```
newtype Fix f = In (f (Fix f))
cata :: Functor f ⇒ (f a → a) → Fix f → a
data (:+:) f g a = L (f a) | R (g a) -- as in instant-generics

data Let r = Let String r r

data ULC r = Var String | Lam String r | App r r

elab :: Fix (Let :+: ULC) → Fix ULC
elab = cata $ λe → case e of
  L (Let s rhs body) → App (In (Lam s body)) rhs
  R ulc → ulc
```

Note in particular that `elab`'s codomain is an exact type, unequal but similar to its domain. Using an improved version of the injection and projection techniques summarized by Swierstra [44], this function can be given a polymorphic type, so that it becomes generic in way comparable but distinct from the genericity of `instant-generics`. With that type, `elab` would be applicable to any two-level type that included both `Let` and `ULC` as summand functors, and it would effectively remove the `Let` summand. Because of this functionality, two-level data types with overloaded injection and projection seems comparable to `yoko`. The reviewers' criticism becomes: "you should explain why you did not build on top of the two-level types techniques". My response is two-fold.

- Two-level data types are burdensome and obfuscated to work with compared to directly-recursive data types. I wanted to avoid that noise when presenting my ideas.
- I have simultaneously shown that two-level types are not a prerequisite for my techniques; `yoko` works with directly-recursive data types.

That last point is crucial. In particulary, every functor is also a conventional data type, and so my techniques can potentially be used in conjuction with two-level data types! For example, consider if the user had factored the syntactic functors differently, choosing to group variable occurrences with lets instead of with lambdas and applications. (Note that

this pairing of lets and variables would be quite natural in anticipation of a lambda-lifting stage.)

```
data Let2 r = Let String r r | Var String

data ULC2 r = Lam String r | App r r

elab2 :: Fix (Let2 :+: ULC2) → Fix ???
```

With this factoring, the `elab2` function can no longer just remove the `Let2` summand; the lambdas still need variable occurrence in the terms. The use of two-level data types requires the user to factor the intended conventional data type into summand functors a priori. This is comparable to the short-coming of the CASL approach where the user had to identify the subsets of constructors a priori. This is avoided by my fields type, which are the ultimate and atomic factors.

The observation that fields types are the atomic factoring highlights the fact that my disbanded data types—the sums of fields types—are analogous to two-level types. Users of two-level types could adopt the same extreme and only put one constructor in each summand functor. Now the main difference is the fact that two-level types are built as fixed points of "openly recursive" summed functors while disbanded data types are simply sums of fields types that have the recursion built-in. The only major consequence is that every fields type is bound to a particular data type because that data type occurs directly as the recursive components of the fields type. For a summand functor, the recursive is abstract, so a particular functor does not determine a two-level data type.

The independence of a summand functor from any particular two-level data type is key to the modularity benefits of the two-level approach. In particular, the same summand functor can occur in multiple two-level data types *and* functionality (*e.g.* as algebras) specific to just that summand functor can be applied to those two-level data types. For example, Bahr and Hvitved show how to reuse pretty-printer algebras across multiple two-level data types

that include some of the same summand functors. This capability addresses a major part of Wadler's expression problem.

As implemented in my work, `yoko` is not intended to the expression problem. I learned during my experiences with the expression problem (*e.g.* [16]) that the obfuscation inherent to two-level types is not obviously worth the extra modularity. I wanted `yoko` to be a more lightweight mechanism that solves the similar problem of reducing costs of converting between similar data types a là Sarkar et al. [39]. Even so, I invented a use of type families to abstract over recursive occurrences in field types without lifting to functors. I later found a similar use of type families in the work of Jeltsch [21, §3.2]. This is a more lightweight solution compared to conventional two-level types, but the typing concerns become delicate. It is an interesting avenue of future work.

Still, instead of enriching `yoko` to address the expression problem, I can apply `yoko` to two-level data types. While the current implementation can not directly convert between a fixed point involving the `Let2` summand and one involding `Let`, the basic automation—matching on constructor names—still applies. I am very confident the small technical mismatches (*e.g.* dealing with `Fix` and `:+:`) could be overcome. It would be fruitful to combine `yoko`ś genericity with that of two-level types, since the two provide incomparable functionalities. In particular, the reuse of a single algebra (*e.g.* for pretty-printing) at multiple data types within the pipeline is a kind of reuse not currently provided by `instant-generics` or `yoko`.

9 Conclusion

I have explained my technical contributions and shown that they satisfy my stated objective: they make it easier for programmers working in statically- and strongly-typed functional languages like Haskell to use more precise types. The key idea is to encode the programmer's intended correspondence between the constructors of similar data types in order to provide a very reusable operator for homomorphically converting between the two types. This reduces the cost of precisely-typed programming by recovering the level of reusability that was previously confined to imprecise types. In particular, my work provides a level of datatype-agnostic reuse characteristic of generic programming. As a result, Haskell programmers need no longer face the trade-off between reusability and assurance when leveraging precise types in a large program. The conversion definitions are concise, modular, and cheap to write as needed.

I have presented my techniques and implemented them in the Haskell programming language, but the fundamental ideas are applicable to any statically- and strongly-typed programming functional language with algebraic data types. Haskell's rich type features do, however, allow an implementation of my ideas mostly within the language—implementations in other languages may require more, possibly external, metaprogramming. I have implemented my ideas in a Haskell library available at `http://hackage.haskell.org/package/yoko`.

With respect to the field of generic programming, my contributions are three-fold. First, I provide lightweight support for constructor subsets. Beyond its own independent benefits, the constructor subset mechanism enables the use of my second contribution, a reusable operator for tag homomorphisms. The generic homomorphism `hcompos` factors out a pattern in the definition of functions that convert between data types with analogous constructors: constructors that have the same intended semantics. My generic programming technique is the first that can convert between distinct but similar genuine Haskell data types; it is a generalization of the `compos` operator of Bringert and Ranta [7] to support unequal domain

and codomain. It thereby adds precisely-typed programming as a compelling application area for generic programming. My work specifically allows the fundamental ideas of generic programming approaches like `instant-generics` [8] and `compos` to be applied in the context of precisely-typed programming without requiring obfuscation as severe as the two-level type approaches like that of Bahr and Hvitved [4].

With respect to the field of precisely-typed programming, my contribution is to port the *pass expander* of Sarkar et al. [39] from the dialect of Scheme to Haskell. This enables the improved assurance of precise types without sacrificing maintainability, which usually prevents precise types from being used in large programs. I demonstrated the technique with many small examples and two large examples of lambda-lifting functions that encode the resulting absence of lambdas with their precise codomain type. Because of the use of my `hcompos` operator, these definitions are concise and modular, referencing only constructors for variables and binders (*i.e.* variables and lambdas in my examples).

I plan two main avenues for future work. As I discussed in Section 7.2, the viability of using my existing `yoko` library in actual programs would be drastically increased by two improvements that are already of independent interest to the general haskell generic programming community. First, the time and space overhead of methods based on the fundamental ideas of `instant-generics` needs to be better reduced by the GHC optimizer. Second, with the advent of type-level programming in GHC Haskell, as used by my `yoko` library, type errors are exhibiting a poor signal-to-noise. Improving the overhead and quality of type errors due to type-level programming are both active research problems that would also benefit the users of my techniques. The second avenue is specific to my work. The benefits of `yoko` are in many ways orthogonal to solutions to Wadler's expression problem, such as the work of Bahr and Hvitved [4]. Enhancing `yoko` with comparable features and/or developing means to use both approaches simultaneously would yield a more comprehensive framework for precisely-typed programming in Haskell.

Bibliography

[1] The GHC user's guide. `http://www.haskell.org/ghc/docs/7.4.1/html/users_guide/`, 2012.

[2] M. D. Adams and T. M. DuBuisson. Template Your Boilerplate: Using Template Haskell for efficient generic programming. In *5th ACM SIGPLAN Symposium on Haskell*, 2012.

[3] P. Alexander. *System-Level Design with Rosetta*. Morgan Kaufmann Publishers, Inc., 2006.

[4] P. Bahr and T. Hvitved. Compositional data types. In *7th ACM SIGPLAN Workshop on Generic Programming*, pages 83–94, 2011.

[5] R. Bird and L. Meertens. Nested datatypes. In *3rd International Conference on Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer–Verlag, 1998.

[6] A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.

[7] B. Bringert and A. Ranta. A pattern for almost compositional functions. *Journal of Functional Programming*, 18(5-6):567–598, 2008.

[8] M. Chakravarty, G. Ditu, and R. Leshchinskiy. Instant generics: fast and easy. At `http://www.cse.unsw.edu.au/~chak/papers/CDL09.html`, 2009.

[9] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *13th ACM SIGPLAN International Conference on Functional Programming*, pages 143–156, 2008.

[10] D. Clarke and A. Löh. Generic Haskell, specifically. In *IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 21–47, 2002.

[11] B. C. d. S. Oliveira, T. Schrijvers, W. Choi, W. Lee, and K. Yi. The implicit calculus: a new foundation for generic programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–44, 2012.

[12] N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 206–217, 2006.

[13] N. G. de Bruijn. Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser Theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[14] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1992.

[15] R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *5th ACM SIGPLAN Symposium on Haskell*, 2012.

[16] N. Frisby, G. Kimmell, P. Weaver, and P. Alexander. Constructing language processors with algebra combinators. *Science Computer Programming*, 75(7):543–572, 2010.

[17] N. Frisby, A. Gill, and P. Alexander. A pattern for almost homomorphic functions. In *8th ACM SIGPLAN Workshop on Generic Programming*, pages 1–12, 2012.

[18] J. Garrigue. Programming with polymorphic variants. In *ACM SIGPLAN Workshop on ML*, 1998.

[19] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of ACM*, 40(1):143–184, 1993.

[20] P. Jansson and J. Jeuring. PolyP - a polytypic programming language. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[21] W. Jeltsch. Generic record combinators with static type checking. In *12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, 2010.

[22] J. Jeuring and D. Swierstra. *Grammars and Parsing*. Open Universiteit, 2001.

[23] S. L. P. Jones and D. R. Lester. A modular fully-lazy lambda lifter in Haskell. *Software, Practice and Experience*, 21(5):479–506, 1991.

[24] O. Kiselyov. `http://okmij.org/ftp/Haskell/typeEQ.html`, 2012.

[25] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *8th ACM SIGPLAN Workshop on Haskell*, pages 96–107, 2004.

[26] R. Lämmel and S. P. Jones. Scrap Your Boilerplate: a practical design pattern for generic programming. In *ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 26–37, 2003.

[27] X. Leroy. Objective Caml. `http://caml.inria.fr/ocaml`, 2000.

[28] S. Lindley. Embedding F. In *5th ACM SIGPLAN Symposium on Haskell*, 2012.

[29] J. P. Magalhães and J. Jeuring. Generic programming for indexed datatypes. In *7th ACM SIGPLAN Workshop on Generic programming*, pages 37–46, 2011.

[30] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for Haskell. In *3rd ACM SIGPLAN Symposium on Haskell*, pages 37–48, 2010.

[31] J. P. Magalhães, S. Holdermans, J. Jeuring, and A. Löh. Optimizing generics is easy! In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 33–42, 2010.

[32] J. P. Magalhães. The right kind of generic programming. In *8th ACM Workshop on Generic Programming*, 2012.

[33] J. P. Magalhães and A. Löh. A formal comparison of approaches to datatype-generic programming. In *4th Workshop on Mathematically Structured Functional Programming*, pages 50–67, 2012.

[34] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18:1–13, 2008.

[35] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *5th ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991.

[36] T. Mossakowski, A. E. Haxthausen, D. Sannella, and A. Tarlecki. CASL the common algebraic specification language. In D. Bjrner and M. C. Henson, editors, *Logics of Specification Languages*, Monographs in Theoretical Computer Science, pages 241–298. Springer Berlin Heidelberg, 2008.

[37] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *8th International Conference on Automated Deduction*, volume 607, 1992.

[38] S. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[39] D. Sarkar, O. Waddell, and R. K. Dybvig. A nanopass infrastructure for compiler education. In *9th ACM SIGPLAN International Conference on Functional Programming*, pages 201–212, 2004.

[40] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *13th ACM SIGPLAN ACM SIGPLAN International Conference on Functional Programming*, pages 51–62, 2008.

[41] T. Sheard and E. Pasalic. Two-level types and parameterized modules. *Journal Functional Programming*, 14(5):547–587, 2004.

[42] M. Snyder. *Type Directed Specification Refinement*. PhD thesis, University of Kansas, 2011.

[43] M. V. Steenbergen, J. P. Magalhães, and J. Jeuring. Generic selections of subexpressions. In *6th ACM Workshop on Generic Programming*, 2010.

[44] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[45] D. Turner. Total functional programming. *Journal of Universal Computer Science*, 10: 187–209, 2004.

[46] T. van Noort, A. Rodriguez, S. Holdermans, J. Jeuring, and B. Heeren. A lightweight approach to datatype-generic rewriting. In *4th ACM Workshop on Generic Programming*, pages 13–24, 2008.

[47] S. Visser and A. Löh. Generic storage in Haskell. In *6th ACM Workshop on Generic Programming*, 2010.

[48] D. Vytiniotis, S. Peyton Jones, T. Schrijvers, and M. Sulzmann. OUTSIDEIN(X) : modular type inference with local assumptions. *Journal of Functional Programming*, 21 (4-5):333–412, 2011.

[49] P. Wadler. The essence of functional programming. In *19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, 1992.

[50] P. Wadler. The expression problem. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`, 1998.

[51] P. Weaver, G. Kimmell, N. Frisby, and P. Alexander. Constructing language processors with algebra combinators. In *6th International Conference on Generative Programming and Component Engineering*, pages 155–164. ACM Press, 2007.

[52] S. Weirich, B. A. Yorgey, and T. Sheard. Binders unbound. In *16th ACM SIGPLAN International Conference on Functional Programming*, 2011.

[53] E. M. Westbrook, N. Frisby, and P. Brauner. Hobbits for Haskell: a library for higher-order encodings in functional programming languages. In *4th ACM SIGPLAN Symposium on Haskell*, pages 35–46, 2011.

[54] A. R. Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *14th ACM SIGPLAN International Conference on Functional Programming*, pages 233–244, 2009.

[55] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 53–66, 2012.