A Syntax for Composable Data Types in Haskell

A User-friendly Syntax for Solving the Expression Problem

Master's thesis in Computer Science and Engineering

FREDRIK ALBERS
ANNA ROMEBORN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

MASTER'S THESIS 2023

A Syntax for Composable Data Types in Haskell

A User-friendly Syntax for Solving the Expression Problem

FREDRIK ALBERS
ANNA ROMEBORN

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

A Syntax for Composable Data Types in Haskell
A User-friendly Syntax for Solving the Expression Problem
FREDRIK ALBERS, ANNA ROMEBORN

Supervisor: Niklas Broberg, Department of Computer Science and Engineering
Examiner: Andreas Abel, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

A Syntax for Composable Data Types in Haskell
A User-friendly Syntax for Solving the Expression Problem
FREDRIK ALBERS, ANNA ROMEBORN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The expression problem is the problem of designing a programming language such that it has both extensible data types and extensible sets of functions over those data types. This means that it should be possible to add new functions, new variants to a data type, and function cases for these new variants, without modification or recompilation of existing code. There exist a number of different approaches to solving the expression problem, which have different qualities in expressiveness or simplicity.

In this thesis, we have designed a syntax, with an accompanying transformation into compilable code, that acts as a solution to the expression problem in Haskell. By designing this syntax, we can make simplifications that would not be possible for solutions written directly in Haskell. In particular, we can, behind our syntax, abstract away some syntactic overhead that can be generated by the transformation.

To demonstrate that our designed syntax is feasible and works as an extension to Haskell, we have implemented the transformation, which transforms code written using our syntax into code that uses the Haskell library `compdata`. That library is an existing solution to the expression problem, and is based on another solution: Data Types à la Carte. Both of these solutions share many similarities in semantics with our syntax.

Similarly to `compdata` and Data Types à la Carte, data types in our syntax are composable; that is, variants of a data type are manually combined to form a type with a fixed set of possible variants. We introduce a new concept, called categories, with the purpose of grouping variants and compositions of data types and simplifying the syntax into one closer to that of regular data types. This means that compositions of variants can only be formed of variants belonging to the same category, as opposed to freely being able to combine any variants. Again, similar to `compdata` and Data Types à la Carte, functions over composable data types are extended automatically in our syntax. This means that a function can be used for a variant of the data type as long as the function has a case defined for the variant. Through the introduction of categories and the abstractions possible through the transformation, we are able to conclude that our syntax provides several improvements compared to existing solutions to the expression problem in Haskell.

Keywords: The expression problem, Haskell, programming language design, programming language, composable, extensible, functional programming.

Acknowledgements

We want to thank everyone that have helped us during the work of this master's thesis. First and foremost, we want to thank our supervisor, Niklas Broberg, for all the guidance and support, especially for all of his thorough feedback on our thesis report. We also want to thank the creators and maintainters of `haskell-src-exts` and `haskell-names`, as these two libraries have given us a good basis for our implementation of the transformation.

On a more personal note, Anna wants to thank the student guidance counsellor, Ingegerd Nilsson, for her continuous support. Last, but not least, we want to thank our friends and families for supporting us through this project, as well as throughout our education as a whole. Thank you for encouraging us to keep going even though you have no knowledge of Haskell!

Fredrik Albers and Anna Romeborn, Gothenburg, February 2023

Contents

1 Introduction

Originally formulated by Philip Wadler, the *expression problem* [1] is a famous dilemma in the domain of programming language design. It concerns the problem of designing a language such that it has both *extensible data types* and *extensible sets of functions* over those data types. A data type is extensible if new variants of the data type can be added outside of its original definition, and if the existing functions over the data type can be extended to handle the new variants. For a set of functions to be extensible, it should be possible to add new functions over a certain data type without needing to change the existing code for the data type.

An example often used for the expression problem is a data type representing an abstract syntax tree of a small expression language, with functions to, for instance, evaluate such expressions. In this thesis report, we use this example of a recursive data type of expressions with three variants: constant integers, addition of expressions, and multiplication of expressions.[1] To begin with, we also define an evaluation function that evaluates an expression to an integer value. To have a solution to the expression problem, it should be possible to add a new variant to the data type, to extend the function to handle this variant, and to extend the set of functions, all of this without modifying existing code. For instance, we may want to add a variant for negation of an expression, and extend the evaluation function to handle that variant, and also add a new function that writes the expression as a string.

Depending on the design paradigm, most existing programming languages are extensible only in one direction, either in data types *or* the set of functions. We will illustrate this here by showing the example mentioned above in both an object-oriented and a functional programming language, and we will see that they offer different directions of extensibility.

Beginning with an object-oriented language, we show the example in Java. Here, we represent the data type as an interface, and the variants of the data type as classes that implement the interface. In the expression language example, this is the interface `Expr` and the classes `Const`, `Add` and `Mul`. The interface has the evaluation method `eval()`, which all of the classes implement. This gives us the following:

[1]The variants for addition and multiplication are thus recursive, which is why the data type itself has to be recursive. Recursive data types is an important aspect of the expression problem, and what we consider in this thesis.

```
interface Expr {
    int eval();
}

class Const implements Expr {
    int value;
    Const(int v) { value = v; }
    public int eval() { return value ; }
}

class Add implements Expr {
    Expr e1, e2;
    Add(Expr a, Expr b) { e1 = a; e2 = b; }
    public int eval() { return e1.eval() + e2.eval(); }
}

class Mul implements Expr {
    Expr e1, e2;
    Mul(Expr a, Expr b) { e1 = a; e2 = b; }
    public int eval() { return e1.eval() * e2.eval(); }
}
```

Note that the blocks for the classes are only dependent on the the interface, and are independent from each other. This means that the classes can be compiled separately. The first direction of extensibility is now to be able to add a new variant, in this example a variant for negation of an expression. We can easily do so by writing a new class as follows:

```
class Neg implements Expr {
    Expr e;
    Neg(Expr a) { e = a; }
    public int eval() { return (-1) * e.eval(); }
}
```

This is added independently from the other classes, and does hence not require any modification or recompilation of existing code. On the other hand, if we want to add a new method, in order to also provide a function `asString()` to write an expression as a string, we must modify the interface and all the classes to handle the new method. Thus, we must edit the existing code, which means that object-oriented languages typically do not offer a solution to the expression problem.

This thesis will primarily consider another programming paradigm, namely functional programming languages. These tend to be extensible in the opposite direction to object-oriented languages; that is, they are extensible in the set of functions, but not in the data types. We will now show the example in the functional programming language Haskell, since that is the language on which this thesis will focus. First, we have the data type for `Expr`, with variants `Const`, `Add` and `Mul`:

```haskell
module Expr where
data Expr = Const Int | Add Expr Expr | Mul Expr Expr
```

The basis of a functional program is to apply and compose functions. This means that we can define several functions over a data type separately from the data type itself. These functions can be compiled independently of each other, as long as they have access to the data type `Expr`. For instance, we can have an `eval` function and an `asString` function written in different modules:

```haskell
module Eval where
import Expr

eval :: Expr -> Int
eval (Const i)   = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

```haskell
module AsString where
import Expr

asString :: Expr -> String
asString (Const i)   = show i
asString (Add e1 e2) = "(" ++ asString e1 ++ " + "
                           ++ asString e2 ++ ")"
asString (Mul e1 e2) = "(" ++ asString e1 ++ " * "
                           ++ asString e2 ++ ")"
```

We can then also later extend the set of functions in a similar way. On the other hand, if we want to extend the data type to also have a variant for negation, we must modify the existing data type to be:

```haskell
data Expr = Const Int | Add Expr Expr | Mul Expr Expr | Neg Expr
```

Furthermore, we must modify all functions over the data type to handle the new variant. This means recompilation of existing code, and shows that the expression problem is typically not solved in functional languages, albeit in a different way from object-oriented languages.

As noted, what we want is to be able to extend both the data type and the set of functions in a modular way, without needing to recompile old code. For an even more modular way of extending a data type, one could aspire to have *composable* data types, instead of just extensible. This means that one can freely combine the variants to form one or more different data types, and thus be able to specify which variants to include or not, rather than them all being included in the extended data type. [2, 3] In Figure 1.1, we show these two ways of constructing data types in flowcharts, to better illustrate the two concepts. In this report, we will use *extensible* as an umbrella term for data types or functions that solve the expression problem, unless it is used as a direct contrast to *composable*.

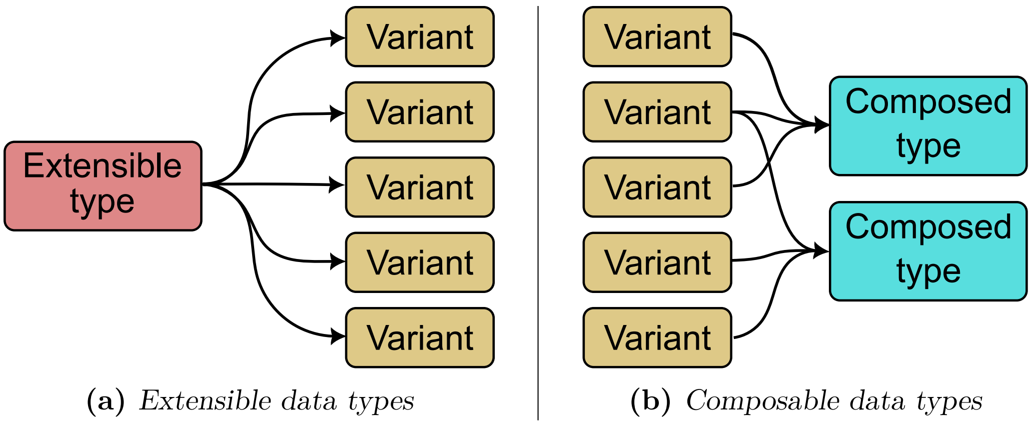

Figure 1.1: *Flowcharts for how to form data types with different variants in two ways: with extensible or composable data types. For extensible data types, we have first a declaration of the extensible data type, which can then be extended with different variants. For composable data types, we instead first declare the different variants, which can then be combined into different composed types. Note that in the second case, we can have different composed types containing just the variants that we want, while in the first case, the extensible data type always contains all variants over which it has been extended.*

Modularity, be it with extensible or composable data types, has many advantages, such as the possibility of keeping related things together and unrelated things separate. Doing so usually means that the code is less prone to contain errors, since there is typically only one place where one needs to make a change, and one has more control over what the change affects. This means that one does not need to consider everything in a project at once. [4, 5] Modularity is also valuable when one wants to extend a project later. For instance, one could have a compiler that could then be extended with plugins from other programmers, without them having to change the original code [6]. In conclusion, a solution to the expression problem would provide many advantages for modularity in a programming language.

While there exist some proposed solutions to the expression problem in Haskell, many of them either build on mechanisms not implemented in a Haskell compiler, or introduce syntactic overhead that offset the benefits of such a solution [2, 5]. The existing solutions seem to often focus on the underlying implementation to provide a system for extensible data types that can be used in many different scenarios, and to be less concerned with the syntax and user-friendliness. Furthermore, by writing the solutions directly in Haskell, the user-friendliness of the solution is also limited by the current syntax and existing mechanisms of Haskell. With this background, this master's thesis project focuses on designing a user-friendly syntax for a solution to the expression problem in Haskell.

1.1 Purpose and Scope

The overarching purpose of this master's thesis project is to investigate what would constitute an elegant syntax for solving the expression problem in Haskell. The concept of an elegant syntax will be defined in Section 1.2. To fulfill the purpose, the project is done with the focus on developing a specific syntax, with the following goals:

- Design an elegant syntax for a solution to the expression problem in Haskell.
- In order to ensure feasibility of the syntax, implement a proof-of-concept language extension that transforms code written with the syntax into compilable Haskell code.
- Evaluate the syntax and implementation based on its expressiveness, that is, the different scenarios it can and cannot handle.
- Compare the user-friendliness and expressiveness of the syntax to existing expression problem solutions.

The primary focus is to investigate what constitutes an elegant syntax in a theoretical way. This means that we do not focus as much on specific choices of keywords and similar, but rather on what is needed overall to provide a syntax that is easy to use in many different scenarios.

Regarding implementing a transformation to compilable code, the goal is again theoretical, meaning that we want to show feasibility of the syntax, rather than to implement a complete language extension ready to be released. Therefore, it is not a priority to solve minor practical limitations related to the implementation, or to optimize performance.

Furthermore, the focus is to improve solutions to the expression problem in Haskell regarding their syntax rather than their expressive power. To focus on the syntax design, the transformation is implemented so that it transforms code written in our syntax into code using an existing solution to the expression problem. We refer to this existing solution as the backend of our project. While this choice may put some limitations on the design of the syntax as it must be transformed to the chosen backend, it relieves the need to implement a solution to the expression problem from the ground up. The choice of backend is further made with care to match the desires for an elegant syntax as well as possible.

1.2 Aim for the Syntax

Before designing and eventually evaluating a syntax, we must establish what constitutes a good syntax. First, we present here some general conditions that must be met.

- **A solution to the expression problem:** The criteria for a solution are elegantly stated by Zenger and Odersky [3], and are only slightly rephrased here:

  - **Extensibility in both directions:** It should be possible to extend both a data type with more variants and the set of functions over said data type. Furthermore, it should be possible to extend all functions to support new data type variants.

  - **Strong static type safety:** It should be impossible to apply a function to a data type variant it cannot handle.

  - **No modification or duplication:** Existing code should not need to be modified or duplicated.

  - **Separate compilation:** Compiling data type extensions or new functions should not require any recompilation of the original data type or existing functions.

  - **Independent extensibility:** It should be possible to combine independently developed extensions so that they can be used together.

  The first four criteria are from the original definition of the expression problem, while the fifth criterion was added by Zenger and Odersky. This is a reasonable addition to capture the nature and purpose of composability, and rejects some early proposed solutions.

- **Extension to Haskell:** The syntax should be usable as an extension to standard Haskell.

- **Self sufficiency:** A user of the syntax should not need to know about or directly use components from any underlying backend, but rather just use our syntax.

- **Independent design:** The syntax should be designed as independently as possible of the interface provided by the backend library, to avoid the need to compromise its elegance due to the specific backend.

Then, we want to establish our specific goals that constitute our view of an elegant syntax. These are the things that we will consider when making a choice between alternatives that all meet the above conditions.

- **Simplicity:** The syntax should not introduce significant syntactic overhead or complexity compared to standard Haskell data types and functions over them. The syntax should stay close to that of standard data types, rather than introducing many new concepts to the user.

- **Familiarity:** Close to the previous goal, the syntax should also be similar to Haskell and keep a functional style, so that it feels familiar to a user.

- **Expressiveness:** The syntax should, beyond being user-friendly, support many different scenarios for using composable data types. There may, however, be a need for compromises between expressiveness and user-friendliness of the syntax.

1.3 Outline

In the rest of this thesis report, we will first describe related work in Chapter 2, where we cover several proposed solutions to the expression problem. We also present more extensive examples of some of these solutions in Appendix C, with accompanying library modules in Appendix D. This related work forms a basis for inspiration to our own syntax, both when it comes to concepts we want to imitate and concepts we want to improve. We then describe our proposed syntax in Chapter 3, with the goal of showing step by step how an elegant syntax can be made that fulfills the criteria of the expression problem. We summarize the syntax in Appendix A, where we present it more formally than described in Chapter 3. In Appendix B, we also show a more extensive example written in our syntax and how it is transformed, similar to the examples of related work. Following the description of the syntax design, we have the transformation into code that uses an existing solution to the expression problem, which we describe in Chapter 4. Then, in Chapter 5, we discuss several ideas for future work, for which we have not fully designed a syntax or implemented a transformation. Finally, we discuss our solution and present our conclusions in Chapter 6.

2
Related Work

There exist several proposed solutions to the expression problem. This chapter will cover a number of these, to provide a basis for our own syntax, both for inspiration and later for comparison and evaluation. More extensive example programs using some of these solutions are given in Appendix C, in order to see the syntax in full programs that also cover a bit more than the examples shown in this chapter.

As this project aims at designing a syntax for Haskell, the focus is on solutions feasible for use in Haskell. Still, other solutions may be relevant for syntactic inspiration and other comparisons. This could also include solutions designed for object-oriented languages, which could be used in Haskell by using one of several proposed extensions to Haskell to provide object-oriented features. The aim of such extensions is much higher than solving the expression problem, and solutions using them lose the functional style that can be kept better with purely functional approaches [4]. Therefore, those solutions are not deemed suitable for a backend of this project, nor in any significant extent for syntactic inspiration. They may, however, be relevant for giving a broader view of different approaches. For this reason, we have chosen to include Section 2.6 on object algebras in this chapter.

There are also language-based solutions to the expression problem, which are based on designing new languages or language features specifically for extensible data types. An example of this is row typing and extensible records and variants [7]. Many of these solutions require significant extensions to the language, for instance by changing its core type or class system [5]. We need to in some way relate our syntax to a solution that is implemented in Haskell, and that can therefore be our backend, which would not be possible with unimplemented solutions. We briefly cover one such solution in Section 2.4, as one of two alternative implementations of Variations on Variants.

Finally, there are solutions that build on existing mechanisms in Haskell. Some of these provide new syntax, while others simply show how the existing mechanisms can be used. This means that we can look to these solutions for inspiration both in syntax and in semantics. Since they are built on existing mechanisms, we will be able to relate our suggested syntax to such a solution directly, to provide functional syntax without the need to implement the backend solution ourselves. Thus, those are the most relevant solutions for choosing a backend for this project and, by extension, also in syntactic inspiration. Therefore, we cover multiple such solutions in this chapter. We discuss two proposed solutions that provide new syntax, Open Data Types and Open Abstract Types, in Sections 2.7 and 2.8. The solutions that is directly usable without new syntax are even more relevant, and we cover several of these in Sections 2.2–2.5, such as Data Types à la Carte and Trees that Grow.

2.1 Running Example

In the following presentations of proposed solutions, we will look at the running example of the abstract syntax tree of a small expression language, as introduced in the introduction. This, with small alterations, is a standard example used when discussing the expression problem. The expression language consists of a data type that looks like this in regular Haskell syntax:

```haskell
data Expr = Const Int | Add Expr Expr | Mul Expr Expr
```

With this data type, one can write an example like the following:

```haskell
threePlusFive :: Expr
threePlusFive = Add (Const 3) (Const 5)
```

It should then be possible to extend this data type with a variant for negation, `Neg Expr`, and have the possibility to allow further extensions. If the added variant simply extends the data type `Expr`, so that it from then on always includes that variant, we say that `Expr` is an *extensible* data type. To instead have a *composable* data type, it should be possible to pick among the building blocks `Const`, `Add`, `Mul` and `Neg` and combine the desired pieces into a composition. Consequently, variants of a composable data type can be combined into multiple different compositions.

It should also be possible to write functions over this data type, as well as to extend said functions for extensions to the data type. In particular, we look at the evaluation function example:

```haskell
eval :: Expr -> Int
eval (Const i)   = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

It should then be possible to extend this with a case for negation, as in:

```haskell
eval (Neg e) = (-1) * eval e
```

The extensibility direction of extending the set of functions for a certain data type must also be considered. This means that we should be able to write, for instance, a show function `asString` on the data type as well. This is especially relevant when looking at object-oriented approaches, since that direction of extensibility is usually the one not fulfilled in such languages, as noted in Chapter 1. However, this thesis focuses on functional languages, where it is easy to add new functions, while it is difficult to extend a data type. Thus, the emphasis here is not on extending the set of functions.

Still, we must consider another function for another reason. The `eval` function above is an example of a *destructive extensible function*. This means that the ingoing type is an extensible type, namely `Expr`, while the result is a simple, non-extensible type, in this case an `Int`. In this way, the function *deconstructs* a value of type `Expr` into an `Int`, based on the structure of the ingoing value, as seen in the pattern matching cases. In particular, this ingoing type is the type over which extensions to the function will be defined. Since the result is an integer, the function does not need to construct any value of an extensible type. On the other hand, another kind of extensible functions is *transformative extensible functions*, where both ingoing and outgoing values are of extensible types. In that case, the function first pattern matches on the ingoing value in the same way as a destructive function, but then it also constructs a value of an extensible type. An example of such a function on the type `Expr` is a desugaring function, which removes some defined syntactic sugar from an expression. In this case, it could remove all negations, returning an expression without the component `Neg`. This would look like the following in regular Haskell syntax:

```haskell
desug :: Expr -> Expr
desug (Const i)    = Const i
desug (Add e1 e2) = Add (desug e1) (desug e2)
desug (Mul e1 e2) = Add (desug e1) (desug e2)
desug (Neg e)      = Mul (Const (-1)) (desug e)
```

In this chapter, we will not show the corresponding `desug` function in full detail when describing different proposed solutions to the expression problem, but will consider how they handle transformative functions. Instead, we show the `desug` function in the complete examples in Appendix C.

There is one final kind of extensible functions, namely *constructive extensible functions*, which construct a value of an extensible type from an input of a regular, non-extensible type. An example could be a parser function that parses a string and constructs an `Expr`. None of the proposed solutions we present in this chapter cover how this would be done, so we will not consider constructive extensible functions any further in this chapter. On the other hand, in Section 5.2 we will discuss how such functions could potentially be implemented in our syntax.

Finally, we refer back to the criteria of being a solution to the expression problem stated in Section 1.2. In addition to supporting extensibility in both directions, as mentioned in the above example, a solution must do so while fulfilling the other four criteria. For instance, it should be possible to write all of the parts in the example above independently in different modules, which can be compiled separately. This means, for instance, that one should be able to have different variants of `Expr`, as well as different extensions to `eval`, written in different modules.

2.2 Data Types à la Carte

The idea of Data Types à la Carte, presented by Swierstra [2], is to make a data type composable by parameterizing the data type so that, at its core, it does not need specific constructors for its different variants. The variants are instead added as their own data types, so that they are independent of each other, which is crucial for being a solution to the expression problem. Furthermore, the solution uses an existing mechanism that already supports composability, namely type classes, to define extensible functions.

This section will show how Data Types à la Carte works on the running example of an expression language, stated in Section 2.1. The example implemented using this solution is also given in its entirety in Appendix C.1. Data Types à la Carte separates the composable data types into a recursive part `Term`, and a signature functor `f`[1] for the different type constructors:

```haskell
data Term f = In (f (Term f))
```

The variants of the data type are now separated from the declaration itself and are defined as follows, as data types themselves:

```haskell
data Const a = Const Int
data Op a = Add a a | Mul a a
```

Note that `Add` and `Mul` are still given in the same data type variant. This is a choice that can be made, so that one can have several variants always contained in the same building block.

The variants can then be combined to form a coproduct, which here is defined as:

```haskell
data (f :+: g) e = Inl (f e) | Inr (g e)
```

Thus, the coproduct is a binary operator that combines the variants in an ordered way, as opposed to a set.[2] Using the coproduct, we can combine `Const` and `Op` with `:+:` to form our specific instance of the data type `Expr`:

```haskell
type Expr = Term (Const :+: Op)
```

Note the use of `Term`, that was defined above, which closes the otherwise open coproduct.[3] This difference between open and closed data types could be considered

[1]As a signature functor, it has the kind `f :: * -> *`, which means that it needs to be applied to a type to get a proper type, similar to how `Maybe` is applied to a type.

[2]In some languages, such as Racket, we can define a union type to combine several types as a union set [8]. This is not available in Haskell, so we have to settle with the binary coproduct operator.

[3]We can also write an open signature, for instance like `type Sig = (Const :+: Op)`, which can be used to also make a further composition, like `type SigWithNeg = Sig :+: Neg`. If the composition is closed through the use of `Term` as in `Expr`, it cannot be used directly in further compositions.

unnecessary for a user to have to consider, and could possibly be abstracted away. The use of the coproduct to combine variants means that it is possible to create different compositions including just the desired variants, so using Data Types à la Carte means having data types that are composable.

The types `Const` and `Op`, as well as their coproduct, can (and should) be declared to be functors. This can be done as follows for operations:

```
instance Functor Op where
    fmap f (Add e1 e2) = Add (f e1) (f e2)
    fmap f (Mul e1 e2) = Mul (f e1) (f e2)
```

Similarly, it can be done like the following for the coproduct:

```
instance (Functor f , Functor g) => Functor (f :+: g) where
    fmap f (Inl e1) = Inl (fmap f e1)
    fmap f (Inr e2) = Inr (fmap f e2)
```

Note that the coproduct instance is general and can be used for coproducts of any functor types. That the types and their coproducts are functors means that we can fold[4] over `Expr` when writing functions on this data type (and more generally on any value of type `Term f`). A fold function is then defined as follows:

```
foldTerm :: Functor f => (f a -> a) -> Term f -> a
foldTerm f (In t) = f (fmap (foldTerm f) t)
```

Then, when we want to define an evaluation function, `eval`, over the data type, we do so in a modular fashion by defining a type class `Eval`, with instances for our variants `Const` and `Op`, as well as the coproduct `f :+: g`. The class function is given as an algebra, which means that it has the type `f Int -> Int`. Thus, we define the class `Eval` as follows:

```
class Functor f => Eval f where
    evalAlg :: f Int -> Int
```

Using this together with the fold function `foldTerm` defined above, we get the evaluation function `eval` as follows:

```
eval :: Eval f => Term f -> Int
eval = foldTerm evalAlg
```

When folding over the algebra in `eval = foldTerm evalAlg`, `evalAlg` is applied to every recursive part, like the subexpressions in `Add` and `Mul`. The fact that `evalAlg` is an algebra, with the type stated above, means that it describes how to handle each recursion step in the folding using `foldTerm`, so the instances do not need to specify the recursive call. This is what we see in the instance for operations `Op` below, where the right-hand side is for example `e1 + e2`, instead of `eval e1 + eval e2`, as it

[4]For more explanation of fold, we refer the reader to the HaskellWiki explanation [9].

would be without an algebra, as in `eval` in Section 2.1. Hence, we get the following instances defined for constants and operations:

```
instance Eval Const where
    evalAlg (Const i) = i

instance Eval Op where
    evalAlg (Add e1 e2) = e1 + e2
    evalAlg (Mul e1 e2) = e1 * e2
```

It is possible to write the functions without the use of algebras as well, and instead specify the recursive calls, an approach which is closer to how one usually writes regular functions. An example of this is given in Appendix C.1.3.

Finally, an instance is also defined for the coproduct, which simply passes the function on to the left or right part of the coproduct, as follows:

```
instance (Eval f, Eval g) => Eval (f :+: g) where
    evalAlg (Inl a) = evalAlg a
    evalAlg (Inr b) = evalAlg b
```

Note that this instance is to be considered as boilerplate, since it would be the same for the coproduct for every class corresponding to such a function.

Next, we will consider how to add a new variant to the data type. This is possible by simply defining the desired data type, and then defining a new composition via the coproduct that includes this variant, as well as the desired variants from earlier. For example, we may add negation by defining a data type for negation as follows:

```
data Neg a = Neg a
```

Then, we define our extended expression type as:

```
type ExprWithNeg = Term (Const :+: Op :+: Neg).
```

To extend a function to also handle the new variant, we simply add a new instance for the variant to the type class corresponding to the function. Hence, we add the `Neg` instance to the class `Eval` as follows:

```
instance Eval Neg where
    evalAlg (Neg e) = (-1) * e
```

Note that, since this is an instance of a type class, it is separate from the declaration of the class itself, and can thus, for example, be written in another module, and be compiled separately. Still, the instance is dependent on the class declaration, which has to be in the same module or in an imported module.

Another important thing to note is that ambiguities can arise if one were to write two extensions to the same function, for the same data type, independently in different modules, and then importing both those modules in a third module. Such ambiguities are a common problem with having orphan instances [10], that is, instances defined separately from their class and data type for which the instance is defined. More generally, this is an inherent problem with extending, in this case, functions in such an automatic way, so that an extension automatically extends the function for all future uses, as opposed to having an explicit composition step, as for composed data types.

Furthermore, since the `eval` function above is general, it can be instantiated to work on any type of `Term f`, both `Expr` and `ExprWithNeg`. Therefore, writing the instance for `Neg` as above is enough to make `eval` usable for handling negation in all future use cases.

Writing example expressions using the constructors of the data type directly is quite cumbersome, even in small examples [2] like the following:

```
threePlusFive :: Expr
threePlusFive = In (Inr (Add (In (Inl (Const 3)))
                             (In (Inl (Const 5)))))
```

However, the insertion of the injections `In`, `Inr` and `Inl` can be mostly automated. This is solved using smart constructors for the functor signatures and defining how they relate to each other. For instance, the smart constructor for addition would have this type:

```
iAdd :: (Op :<: f) => Term f -> Term f -> Term f
```

The constraint `Op :<: f`, called *subsumption*, means any signature `f` that supports the operations addition and multiplication, which in practice means that `Op` is part of the coproduct `f`. Thus, the smart constructor constructs a *polymorphic* composition, which means that it can be used to construct values of different concrete compositions. Furthermore, the type class for the subsumption constraint is used to define a function `inject` alongside the subsumption class. This `inject` function is used to insert the necessary injections `In`, `Inl` and `Inr` to get the correct type, and has the following type:

```
inject :: (g :<: f) => g (Term f) -> Term f
```

This is then used to define the smart constructors and automate most of the injections. For `iAdd`, this is used to define it as follows:

```
iAdd x y = inject (Add x y)
```

With smart constructors `iConst` for constants and `iAdd` for addition, we can write the above example like this:

```haskell
threePlusFive' :: Expr
threePlusFive' = iConst 3 `iAdd` iConst 5
```

Although this example now looks elegant, we need to consider how to use it. If we were to, for instance, evaluate `threePlusFive'`, it would look like this:

```haskell
evalExample :: Int
evalExample = eval threePlusFive'
```

This works as expected and evaluates to 8, but only because we have specified the type of `threePlusFive'` to be `Expr`, the fixed closed data type defined above. If `threePlusFive'` were written without its type signature, the compiler would not be able to resolve it due to ambiguities. This is because the compiler must know the exact composition for the type of the argument in order to choose the evaluation steps. For instance, the type of `threePlusFive'` above could, if not specified, be either `Term (Const :+: Op)` or `Term (Op :+: Const)`, or potentially a larger composition. The type must be specified to one specific such composition in order to evaluate the example without ambiguities. Then, the construction of `threePlusFive'` through the smart constructors builds it by using the correct combination of injections `Inr` and `Inl` depending on the structure of the coproduct type. Another way to make the structure known, if `threePlusFive'` was written without type signature, is to specify it when calling `eval` on it, as follows:

```haskell
evalExample :: Int
evalExample = eval (threePlusFive' :: Expr)
```

The subsumption criterion, `:<:`, is also used when defining transformative functions, to indicate what needs to be part of the result type. For instance, in the `desug` case for negation defined in Section 2.1, `Mul (Const (-1)) (desug e)`, we see that it uses constants and multiplication in the result. This means that those need to be part of the result type, which is indicated by including the constraints `Const :<: g` and `Op :<: g`, where `Term g` is the result type of `desug`. Furthermore, smart constructors are used in the construction of the resulting expression. The case for negation would now be defined as follows, in its instance of the type class `Desug` corresponding to the `desug` function:

```haskell
instance (Functor g, Const :<: g, Op :<: g) => Desug Neg g where
    desugAlg (Neg e) = iConst (-1) `iMul` e
```

For the other variants that are not desugared through the function, we are able to write a default instance:

```haskell
instance {-# OVERLAPPABLE #-} (Functor g, a :<: g)
        => Desug a g where
    desugAlg a = inject a
```

Note that this uses `inject` in the same way as the smart constructors. Using `inject` here is possible because the function is written using an algebra, so the default case does not have to include the recursive call. This means that it can be written in this general way and cover both `Const` and `Op`, and possible other variants that should not be desugared. See the full implementation of `desug` in Appendix C.1.5.

Furthermore, note the use of the pragma `{-# OVERLAPPABLE #-}`, indicating that this instance is allowed to overlap other instances of `desug`. This is needed since the case for `Neg` could also match the default case. Having overlapping instances does not give any errors as long as there is a more specific instance to use where multiple instances would match. In this case, the instance for `Neg` would be used for negation.

In conclusion, writing transformative functions with Data Types à la Carte is not much more complicated than destructive functions, but it has some added complexity in the need for subsumption constraints. The subsumption criterion is, however, useful in order to be able to express polymorphism. This means that the result type of `desug` is only restricted to contain certain types, not specified with a concrete closed composition, like `Expr`. As a result, `desug` is reusable for different compositions.

One final use case of Data Types à la Carte that we will consider here is pattern matching. The separation of data type variants into separate data types, which can then be composed using the coproduct, means that the type of a value is more complex than when using a regular type. This means that, when pattern matching in a regular function, one must pattern match on the complete structure, including the injections from the coproduct. When constructing a value, we can use the smart constructors to avoid the need for inserting all the injections ourselves, which is not as simple when deconstructing, as seen in the following example:

```
evalClosed :: Term (Const :+: Op) -> Int
evalClosed (In (Inl (Const i))   = i
evalClosed (In (Inr (Add e1 e2)) = evalClosed e1 + evalClosed e2
evalClosed (In (Inr (Mul e1 e2)) = evalClosed e2 * evalClosed e2
```

However, analogous to the previously mentioned `inject` function for constructing, Data Types à la Carte provides a function for decomposing: `project`. This is defined as follows:

```
project :: (f :<: g) => Term g -> Maybe (f (Term g))
```

This uses the `Maybe` type to, from a value of a coproduct type, return either `Nothing` or `Just x`, where `x` is a value like `Const i` depending on the sought type. This can then be used instead of direct pattern matching.

Lastly, we note that Data Types à la Carte offers extensibility in both directions in a way that supports separate compilation and does not require any modification of existing code. The data types are composable, meaning that we can write such extensions in a non-linear way and freely combine them, fulfilling the criterion of independent extensibility. Regarding functions, on the other hand, one must note that an extension to a function must be written in the same module as, or in a module importing a module containing, the function declaration, in this case the corresponding type class declaration, but that is to be expected. The important part is that it is possible to extend the *set* of functions, that is, define new functions, independently. Finally, it has strong static type safety since it is impossible to apply a function to a data variant to which a corresponding type class instance is not defined. In conclusion, Data Types à la Carte offers a solution to the expression problem that fulfills all five criteria given in Section 1.2. Still, it can be improved upon syntactically to lessen the syntactic overhead.

2.3 compdata

Bahr and Hvitved [11, 12] have expanded and improved upon Data Types à la Carte. They have implemented a Haskell library, `compdata` [13], for composable data types, as an extended implementation of Data Types à la Carte, using closed type families. This library also provides code generation tools for using Template Haskell to reduce a lot of the boilerplate code involving derivations and type constructors.

In most cases, the usage of `compdata` is identical to that of Data Types à la Carte, so we will not show the full running example in `compdata` here. In Appendix C.2, we do, however, show a full example, which includes the running example along with some other functions.[5] In this section, we will only present the differences from Data Types à la Carte.

An idea brought up by Bahr and Hvitved in their implementation of composable data types is to use the language extension `DeriveFunctor` [17] to derive functor instances for the data type variants automatically, instead of defining them oneself. For the data type variant for operations, it would be used as follows:

```
data Op a = Add a a | Mul a a
    deriving Functor
```

This leads to a reduction of boilerplate code. However, this is not specific to `compdata`, and could be done, for instance, when using Data Types à la Carte as well.

As mentioned above, `compdata` also uses Template Haskell to further reduce the amount of boilerplate code. This is used to generate the code for smart constructors at compile time, and can also be used for other derivations, such as to provide instances for equality checks and `show`. It is written as follows:

```
$(derive [makeEqF, makeShowF, smartConstructors]
         [''Const, ''Op])
```

[5]Bahr and Hvitved also provide some examples in their Haskell library, which in many cases are similar to our example in Appendix C.2 [14, 15, 16].

This code, called a splice, means that the three functions `makeEqF`, `makeShowF` and `smartConstructors` are each used on the signatures `Const` and `Op` to generate the desired code. A similar derivation is made for the class of an extensible function to generate an instance for the coproduct:

```
$(derive [liftSum] [''Eval])
```

This uses `liftSum`, a function defined in `compdata` for lifting a type class (which in this case represents an extensible function) to handle coproducts of functors. In this particular case, this means that an instance for the coproduct is generated for the type class `Eval`.

These derivations through Template Haskell simplify the code compared to the usage of Data Types à la Carte. Still, the derivations could be considered unnecessary for the user to write every time, and make the usage of these composable data types and functions more complicated than using standard Haskell.

Furthermore, Bahr and Hvitved provide some refinement of the subsumption criterion. In particular, this means that `compdata` allow compound signatures on both sides of a subsumption constraint, such as `f :+: g :<: f :+: g :+: h`, while disallowing duplicates such as `f :<: f :+: f`. They also provide an isomorphism constraint, given as `type f :=: g = (f :<: g, g :<: f)`, which is useful to check for signatures which are equal up to order of summands. A final note on both of these constraint is that they only work for ground types, that is, types without variables. [12]

In conclusion, `compdata` is a thorough implementation of Bahr and Hvitved's improvements to Data Types à la Carte, which provides the framework necessary for most use cases of composable data types. As such, `compdata` fulfills the criteria given in Section 1.2 for being a solution to the expression problem in the same way as Data Types à la Carte. However, the usage of `compdata` is still complex in parts for a user and could be further improved.

2.4 Variations on Variants

Morris [5] proposes two implementations of composable data types in a similar manner to Swierstra's Data Types à la Carte, and Bahr and Hvitved's `compdata`, one using instance chains and one using closed type families. Instance chains have been proposed as a useful mechanism to implement extensible data types, but they are not implemented in Haskell. Closed type families, on the other hand, exist in GHC and capture most of the expressiveness of the instance chain implementation, albeit in a somewhat more verbose way.

Since instance chains are not implemented in Haskell, we will not cover the solution based on them in any detail. One can, however, note their usefulness when defining, for example, a membership check for coproducts in a chain-like manner, where the next case is only tested if the previous fail:

```
class In f g

instance f `In` f
else f `In` (g :+: h) if f `In` g
else f `In` (g :+: h) if f `In` h
else f `In` g fails
```

Similar instance chains are defined for other type classes. Morris also provides a type class to denote a difference between types, denoted by the operator `(:-:)`. Here, the predicate `f :-: g = h` holds if `f` is a coproduct that contains `g`, and `h` is what remains when removing `g` from `f`.

The implementation using closed type families works in a similar way and likewise has a remainder operator `:-:`, but defines the constraints as type families. See an implementation of this in Appendix D.3. In Appendix C.3, we also show a complete example program, which covers how to use the implementation in more detail than shown here. The differences between the the instance chains and the type families implementation are limited to how the constraints are defined. In particular, writing data types and functions is done in the same way in both implementations.

Data types are defined in the same way as in Data Types à la Carte, through one data type for each variant, which can be combined using a coproduct, and closed through the use of `Term`. They are also used in the same way when forming examples, in pattern matching and similar. This means that the limitations and potential for improvements are the same here as for Data Types à la Carte regarding the syntax for data types.

Writing functions, on the other hand, is done differently from Data Types à la Carte. We will see how this is done with the running example given in Section 2.1, specifically the `eval` function. Morris expresses this through a function for each case:

```
evalConst (Const i) r = i

evalOp (Add e1 e2) r = r e1 + r e2
evalOp (Mul e1 e2) r = r e1 ∗ r e2
```

These are then combined using a function that uses the type class `In` to unwrap the `Term`:

```
cases cs = f where f (In e) = cs e f
```

The combination to provide an actual evaluation function is then written as follows:

```
eval1 = cases (evalConst ? evalOp)
```

The operator `?` here is a branching combinator, which combines `evalConst` and `evalOp` into a function for their coproduct.

This has the inferred type:

```
eval1 :: (f :-: Const ~ Op) => Term f -> Int
```

Here, `(f :-: Const ~ Op)` is an equality constraint [18], which states that the types `f :-: Const` and `Op` need to be the same. This means that `Const` and `Op` are included in the input type.

Note that `eval1` is a specific composition of the evaluation function, which only supports constants and the two operations addition and multiplication. A new function composition must be defined to provide evaluation for other variants, such as negation. This is different from the type class approach used by Data Types à la Carte, where a function can be used for all variants for which the type class has an instance declaration, which means that, in that case, a new variant can be handled directly by the same function, after defining its type class instance. Writing the functions by composing different cases as in this approach gives a slight advantage in expressive power over the extensible functions in Data Types à la Carte, since one can easily write different cases for the same variant and then use the case one wants in the composition. On the other hand, the composition of function cases may be complex for large compositions.

Transformative functions in Variations on Variants use the subsumption constraint in a similar way to Data Types à la Carte in the functions for each variant of the data type. It is also used in the type signature of the combining function, which also has a constraint using the remainder operator `:-:`, to indicate what is removed through the function. In the case of the `desug` function[6], as specified in Section 2.1, the result type of the function is `f :-: Neg`, where `Term f` is the ingoing type, for instance `ExprWithNeg ~ Term (Const :+: Op :+: Neg)`. This would indicate that `Neg` is removed through the function and not included in the return type. The remainder constraint is useful, but one can achieve a corresponding constraint in `compdata` using its isomorphism constraint. One can then write the constraint in `desug` as `f :=: g :+: Neg`, where `Term f` is the ingoing type and `Term g` is the result type.

Variations on Variants, especially the closed type family implementation, is similar to `compdata` and likewise requires some explicit type annotations to resolve ambiguities, even if not all ambiguities are present in both implementations. A solution is proposed to include a defaulting mechanism in the compiler similar to the mechanism that allows `z = show 1` to type despite its ambiguity of the type of `1`. [5] However, it is important to note that this solution does not have as complete an implementation as the Haskell library for `compdata`.

Lastly, we note that Variations on Variants offers extensibility in both directions, since it is possible to add new data type variants as well as new functions over the data type. It does so without needing any modification or duplication of existing code. Since we have a separate function for each data type variant, which are then

[6]See the full implementation of `desug` using Variations on Variants in Appendix C.3.5.

combined through the use of `cases`, it is clear that a function cannot be applied to a data type variant it cannot handle, which means that the solution offers strong type safety. Furthermore, extensions can be made in different modules, which can be compiled separately. The data types can also be combined freely in the same way as in Data Types à la Carte and `compdata`. In this solution, the same goes for the combination of function cases. Therefore, different extension modules can be written in a non-linear fashion and be combined without problems, meaning that we also have independent extensibility. In conclusion, Variations on Variants fulfills all five criteria given in Section 1.2 for being a solution to the expression problem. Just as with previously shown solutions, there are, however, limitations for practical use as the syntax is complex.

2.5 Trees that Grow

Najd and Peyton Jones [19] give an approach to solving the expression problem that focuses on data types being extensible by having an extra data constructor that is open to being modified when adding new variants. This means that the data types are combined in a chain-like manner rather than the composition of types based on building blocks that we saw in Data Types à la Carte and similar approaches. They also add a similar open field for each data constructor, to be able to extend the constructors' fields as well.[7]

This section will cover how Trees that Grow works on the running example introduced in Section 2.1. A complete example program of this is also given in Appendix C.4. To have it comparable to other related works, we will not consider extension of fields, only extension of more variants. Furthermore, a difference between how we show the example in this section and the other related work is that here we begin with a data type containing all the variants `Const`, `Add` and `Mul`, and only consider how to extend this with a variant for negation, `Neg`. The reason for this is that Trees that Grow requires more verbose code for adding a new variant than previous approaches, and is better suited for larger building blocks than just one variant per block. Hence, this is the data type containing `Const`, `Add` and `Mul` in this approach, written with one open variant:

```
data Expr e = Const Int
            | Add (Expr e) (Expr e)
            | Mul (Expr e) (Expr e)
            | ExprExt (X_ExprExt e)
```

In this data type, we have an extension constructor `ExprExt` with an extension field `X_ExprExt e`. Furthermore, `Expr` has a type index `e`, which is called an extension descriptor. It describes which composition of the data type that is used, for example

[7]Note that the the possibility to extend the fields of constructors as well as extending the data type with new variants could be considered unnecessary if the data types are composable. This is because it would then be possible to just define a new variant that contains the new field, and use that instead of the previous, not extended variant in compositions where it is wanted.

a composition containing negation or not. We will see how this is used to form different compositions of the data type.

The extension field `X_ExprExt e` is defined through a type family as follows:

```
type family X_ExprExt e
```

When defining a specific composition of the data type, this is instantiated to specify an instance of the extension field to use for this composition. For instance, we can define a composition of the data type containing only the variants above. This composition could be written as `Expr UD`, where `UD`, standing for “undecorated”, is its extension descriptor, defined simply through an empty data type `data UD`. To instantiate the extension field for the extension constructor, the constructor should be marked as unusable, since there is no extension. This is done by defining the instance of the extension field as the uninhabited type `Void` [20]. This gives the following instance for the composition `Expr UD`:

```
type instance X_ExprExt UD = Void
```

Writing an example expression using this data type is straightforward and simply uses the constructors as they are:

```
threePlusFive :: Expr UD
threePlusFive = Add (Const 3) (Const 5)
```

Next, we want to extend the data type with a variant for negation of expressions. This is done through defining a new instance of the extension field `X_ExprExt` that specifies the extension we want. The first step is to define the data type for negation, which will be used in the new composition. Since we want to have recursion over the full data type, including both expressions and negations, we define `Expr` to be the ground type, and write negation as an extension to `Expr`, using `Expr e` as its recursive argument. One may initially think to define it simply as:

```
data Neg e = Neg (Expr e)
```

However, if we put this in the extension field for the data type `Expr`, we cannot continue chaining extensions from this, since `Neg` has no open variant. We could have `Neg` at the end of the chain, and put other extensions between `Expr` and `Neg`. To keep it more flexible, we would rather want `Neg` to be an extensible type as well, so that we can have more extensions chained to it. Therefore, we define `Neg` as follows:

```
data Neg e = Neg (Expr e) | NegExt (X_NegExt e)
```

As above, the extension field is a type family, written as follows:

```
type family X_NegExt e
```

To use this in a composition, we define a new extension descriptor as `data WithNeg` and use that to describe the new composition as `Expr WithNeg`. We then need to define type instances for all extension fields. First, we define the extension field of `Expr` to include `Neg`, as follows:

```
type instance X_ExprExt WithNeg = Neg WithNeg
```

Then, analogous to `Expr UD`, we define the type instance for `X_NegExt` to be `Void`, since we have no further extensions:

```
type instance X_NegExt WithNeg = Void
```

In this way, we can combine data type variants in a chain-like manner to form compositions. A composition in this solution can thus be compared to a linked list, rather than a combination with a binary operator like the coproduct in Data Types à la Carte and similar.

It is important to note that `Expr WithNeg` is a different composition from `Expr UD`, and that we would need yet another composition to provide further extensions. For each composition, the type instances for the extension fields must be defined, even if most of them would be the same as in previous compositions. This is similar to how it is necessary in Data Types à la Carte and similar solutions to define a new composition for a new extension, while it is there possible to reuse a coproduct.

To write an example expression that also contains negation, we need to specify negation as the extension to `Expr`, as seen in the use of the constructor `Neg` in the following example:

```
threePlusNegFive :: Expr WithNeg
threePlusNegFive = Add (Const 3)
                       (Expr_Ext (Neg (Const 5)))
```

This could possibly be simplified through having some kind of smart constructors similar to those used in Data Types à la Carte in Section 2.2, although this is not brought up in the paper. Such a smart constructor would then construct a value with the correct chain of extensions. To implement it, one could write an injection function using a type class with instances, similar to the subsumption type class in Section 2.2. However, doing so would require instances for each extensible data type, in this case `Expr` and `Neg`.

To define a function over the data type, we can first write a simple `eval` function for the data type `Expr`, with an extra argument describing how to handle the case of the extension, as the following:

```
eval :: (X_ExprExt e -> Int) -> Expr e -> Int
eval _ (Const i)    = i
eval f (Add e1 e2) = eval f e1 + eval f e2
eval f (Mul e1 e2) = eval f e1 * eval f e2
eval f (ExprExt e) = f e
```

Next, we want to write a function for how to handle the extension of negation in a similar manner. We do so with yet another extra argument describing how to handle the recursive call. This should be a function that describes how to handle the complete composition, not just the case of `Neg`. Therefore, its type is `Expr e -> Int`, covering `Expr` along with any extensions chained to it. This gives us the following function for evaluation of negation:

```
evalNeg :: (Expr e -> Int) -> (X_NegExt e -> Int) -> Neg e -> Int
evalNeg g _ (Neg e)    = (-1) * g e
evalNeg _ f (NegExt e) = f e
```

To then evaluate an expression that could contain negation, we define a function `evalWithNeg`. This would call `eval`, with `evalNeg` as an argument, describing how to handle the extension to `Expr`. In turn, `evalNeg` has `evalWithNeg` as its argument for the recursive call, and `absurd :: Void -> a` as its argument for the extension, since we have no further extensions. Thus, we have the following function for evaluating expressions potentially containing negations:

```
evalWithNeg :: Expr WithNeg -> Int
evalWithNeg e = eval (evalNeg evalWithNeg absurd) e
```

Note that this function is specific to `Expr WithNeg`, the data type containing the original data type and negation, and no further extension, and that another function would be required to evaluate another composition.

A transformative function is written in a similar manner. It is possible to write the return type of such a function in a polymorphic manner, just as the input type in `eval` above. Hence, we can write a `desug` function with the following type:

```
desug :: (X_ExprExt e1 -> Expr e2) -> Expr e1 -> Expr e2
```

Then, we write a similar function for handling negation, `desugNeg`, as follows:

```
desugNeg :: (Expr e1 -> Expr e2)
    -> (X_NegExt e1 -> Expr e2) -> Neg e1 -> Expr e2
desugNeg g _ (Neg e)    = Mul (Const (-1)) (g e)
desugNeg _ f (NegExt e) = f e
```

Note that this is not fully composable in the return type, since we use concrete constructors, as `Mul` and `Const` in the case for `Neg`. In this case, this is no problem since those belong to `Expr`, the ground type, so any composition would use those constructors as such. However, if we want to construct a value of an extension, we must specify the chain of extensions, like `(Expr_Ext (Neg (Const 5)))` in `threePlusNegFive` above. This means that those constructors are specific to compositions with that order of extensions, and `Expr_Ext (Neg ...)` cannot be used when Neg is not the direct extension to Expr. As in the example threePlusNegFive, this could be simplified if one defined some kind of smart constructors.

The functions desug and desugNeg are then used in a concrete function, specifying the input to be of type Expr WithNeg and the output to be of type Expr UD:

```
desugWithNeg :: Expr WithNeg -> Expr UD
desugWithNeg e = desug (desugNeg desugWithNeg absurd) e
```

The output type is Expr UD since that is the type without negation, while it could also be given as a polymorphic type like Expr e. The fact that we can give a polymorphic type here is a difference to approaches like Data Types à la Carte, where a concrete type is necessary when using the function in order to resolve ambiguities. Such ambiguities cannot arise here since we do not use smart constructors similar to those using inject in Data Types à la Carte. Note also the difference between this desugaring function and the one in Data Types à la Carte, where subsumption constraints are required to specify what needs to be included in the result type. Such constraints are not used here, while the result type in desugWithNeg of course needs to contain the variants used for constructing the result.

In this solution, one can perform pattern matching in regular functions, but like in the case of Data Types à la Carte, it requires matching on the exact structure of the data type. In this case, this means matching as in the following:

```
evalClosed :: Expr WithNeg -> Int
evalClosed (Const i)    = i
evalClosed (Add e1 e2) = evalClosed e1 + evalClosed e2
evalClosed (Mul e1 e2) = evalClosed e1 * evalClosed e2
evalClosed (ExprExt (Neg e))     = (-1) * evalClosed e
evalClosed (ExprExt (NegExt e)) = absurd e
```

Note the structure of the extension part, ExprExt (...), matching both on the extension constructor of Expr and the inner structure of Neg. This is unpractical in large compositions. Still, it is useful that pattern matching can be performed at all.

The syntax of Trees that Grow has significant syntactic overhead that could, in some cases, be abstracted away, such as the definitions of the type families. Semantically, Trees that Grow makes it possible to write extensible data types and also several compositions of a data type. One can write a closed data type in a composable way by choosing which data types and extensions to use when defining the type instances.

The fact that extensions use specific constructors in each of the data types leads to some difficulties that are not present with coproduct compositions from Data Types à la Carte. For instance, while smart constructors as mentioned earlier could be defined in a similar manner to those in Data Types à la Carte, it would require a type class instance for each extensible data type, rather than just a few general instances.

Furthermore, it is not possible to combine composed types, since a composed type is a specific composition of the data type, defined by giving type instances for all the extension fields of the extensible data types. Thus, a composition cannot be used to build a new composition. An advantage, on the other hand, is that it is possible to use this approach directly in Haskell, just by using some well-established existing Haskell extensions such as `TypeFamilies`.

Regarding the expression problem criteria mentioned in Section 1.2, Trees that Grow fulfills all of them. Extensibility in both directions is supported, with the possibility to combine independently developed extensions by making a composition of the data type with the desired parts. Such extensions can be written without duplication or modification of existing code, even if the syntax is rather verbose. The compositions are specified for a certain set of data types, which means that functions over the compositions will only work for the included data types, meaning strong static type safety. Finally, the different parts can be written in different modules, which can be compiled separately. Still, there are, as mentioned, limitations in the user friendliness of this solution.

2.6 Object Algebras

Oliveira and Cook [21] propose object algebras as a solution to the expression problem in object-oriented languages such as Java or C#. With object algebras, a set of data type variants is defined as an object algebra interface with each data constructor represented by a function definition.

As object algebras are designed for object-oriented languages, this solution to the expression problem has limited relevance to this project, as mentioned in the beginning of this chapter. Rather, the purpose of this section is to give a wider image of the possible approaches to solving the expression problem.

In this section, we will show how the running example from Section 2.1 would be written using object algebras, in particular, in Java. A complete example program of this is also given in Appendix C.5. As with Trees that Grow in Section 2.5, we will combine the variants for `Const`, `Add` and `Mul` as the initial data type, which is later extended with `Neg`. An object algebra interface for these three initial variants would look as follows:

```java
interface ExprAlg<T> {
    T const(int i);
    T add(T e1, T e2);
    T mul(T e1, T e2);
}
```

Here, each constructor variant is represented by an interface function, which takes arguments representing the constructor fields of the variant to produce some object of the generic type `T`. This type `T` represents the result of applying some function to an expression, similarly to algebras described in Section 2.2, such as `evalAlg :: t a -> a` where `a` fills the same role. It should not be confused with smart constructors from the same section, which instead are parameterized based on a type composition. In other words, `T` is determined not by a composition of variants, but by a specific function.

We will see the type `T` realized when defining a function over the set of data type variants that `ExprAlg` represents. This is done by defining a class that implements the interface `ExprAlg` for a concrete type in place of `T`, and this class is then called an object algebra. A specific object algebra is a representation of an algebraic function over a set of data type variants. Here, we define an object algebra `ExprEval`, and an interface `Eval` to be used in place of `T`:

```
interface Eval {
    int eval();
}
class ExprEval implements ExprAlg<Eval> {
    public Eval const(int i) { return () -> i; }
    public Eval add(Eval e1, Eval e2) {
        return () -> e1.eval() + e2.eval();
    }
    public Eval mul(Eval e1, Eval e2) {
        return () -> e1.eval() * e2.eval();
    }
}
```

In this particular case, we do not actually need to use `Eval`. As `Eval` is an interface with a single function that does not take any argument and does not use mutable state, it would be possible to replace that with the return type, `int`. However, since Java does not allow primitive types as generic parameters, we would use the wrapper type for `int`: `Integer`. This gives the following simplified version of the object algebra:

```
class ExprEval2 implements ExprAlg<Integer> {
    public Integer const(int i) { return i; }
    public Integer add(Integer e1, Integer e2) {
        return e1 + e2;
    }
    public Integer mul(Integer e1, Integer e2) {
        return e1 * e2;
    }
}
```

When instances of our data type are created, it will be done for some specific object algebra, essentially evaluating it immediately. However, since code that creates instances of the data type should not be concerned with the specific type of object algebra that is used, the function for this should instead accept any object algebra that implements the necessary object algebra interface. This gives the following way to write an example:

```
<T> T threePlusFive(ExprAlg<T> alg) {
    return alg.add(alg.const(3), alg.const(5));
}
```

Our previous object algebra can then be used with this function to perform an evaluation like this:

```
threePlusFive(new ExprEval()).eval()
```

With the simplified object algebra, the same evaluation can be written like this:

```
threePlusFive(new ExprEval2())
```

With this setup to define data types and functions, it is straightforward to introduce new functions by defining a new object algebra. For instance, an `asString()` function can be written as follows:

```
interface AsString {
    String asString();
}
class ExprAsString implements ExprAlg<AsString> {
    public AsString const(int i) {
        return () -> String.valueOf(i);
    }
    public AsString add(AsString e1, AsString e2) {
        return () -> "(" + e1.asString()
            + " + " + e2.asString() + ")";
    }
    public AsString mul(AsString e1, AsString e2) {
        return () -> "(" + e1.asString()
            + " * " + e2.asString() + ")";
    }
}
```

We can then use this object algebra in place of `ExprEval` to instead produce a string when creating an expression, as follows:

```
threePlusFive(new ExprAsString()).asString()
```

New data type variants can be defined by creating a new object algebra interface. To add a variant for negation, we write a new object algebra interface `NegAlg` as follows:

```
interface NegAlg<T> {
    T neg(T e);
}
```

To make it easier for functions such as `threePlusFive()` to accept an object algebra that can handle both negations and regular expressions, we could define a new interface to act as the union of the two:

```
interface NegExprAlg<T> extends ExprAlg<T>, NegAlg<T> {}
```

We could also choose to make `ExprAlg` a subset of `NegAlg` by having the latter extend the former, as follows:

```
interface NegAlg2<T> extends ExprAlg<T> {
    T neg(T e);
}
```

To extend a function over these new variants, we define a new object algebra that extends the original object algebra and implements the interface for the new data type variants, as follows:

```
class NegEval extends ExprEval implements NegAlg2<Eval> {
    public Eval neg(Eval e) { return () -> -e.eval(); }
}
```

While it is straightforward to extend an object algebra over additional data type variants with this method, it is not as easy to combine independently developed object algebras. To combine two algebras and create a new object algebra that covers the data type variants of both, one would need to create an object algebra class to act as the union. This new object algebra then needs to implement the interfaces that the union would cover, and dispatching constructor calls to whichever of the two object algebras that can handle that specific constructor call. For example, it is possible to define a combinator to combine an object algebra implementing `ExprAlg<T>` and an object algebra implementing `NegAlg<T>` to create an object algebra that implements `NegExprAlg<T>` and directs calls to either of the other two object algebras. However, such a union object algebra would consist of a significant amount of boilerplate code. A union algebra for `NegExprAlg<T>` could look like this:

```
class NegExprUnion<T> implements NegExprAlg<T> {
    ExprAlg<T> exprAlg;
    NegAlg<T> negAlg;
    ...
    public T const(int i) { return exprAlg.const(i); }
    public T add(T e1, T e2) { return exprAlg.add(e1, e2); }
    public T mul(T e1, T e2) { return exprAlg.mul(e1, e2); }
    public T neg(T e) { return negAlg.neg(e); }
}
```

A limitation of using combinator classes is that they have to be designed for specific object algebra interfaces. For example, `NegExprUnion<T>` is written specifically for combining `ExprAlg<T>` and `NegAlg<T>` into `NegExprAlg<T>`, and cannot be used more generally than that. On the other hand, if we want to combine some algebra extending `ExprAlg<T>` with an algebra extending `NegAlg<T>`, we are able to reuse the previous union algebra `NegExprUnion<T>` by having the new union algebra class extend the previous.

Transformative functions or anything similar are not mentioned in the original paper, though it is possible to write an object algebra for such a function. We demonstrate such an object algebra for the `desug` function in Appendix C.5.5. There, the algebra for desugaring takes a different algebra for the function that should handle the result of the desugaring. Although we can create a new algebra that extends this `desug` algebra, we cannot write a default case where the data type is preserved, as is possible for transformative functions in Data Types à la Carte, as seen in Section 2.2.

Going back to the criteria of the expression problem, object algebras fulfill all of them. We can introduce new data type variants with a new object algebra interface, and we can extend a function of these variants by extending its object algebra and implementing this new interface. We can introduce new functions simply by defining a new object algebra entirely. Strong type safety is ensured as object algebras can only be used for data type variants defined by interfaces that the object algebra implements. New data type variants, new functions and their extensions all involve creating new interfaces or classes, which can always be introduced without modifying the original interfaces or classes, which let these be compiled separately. Still, while it is possible to combine independently developed object algebras using combinator object algebras, it often requires some boilerplate code to do so.

2.7 Open Data Types

Löh and Hinze [4] propose a solution to the expression problem based on open types that can be extended with more variants; that is, this is a proposed solution for extensible, but not composable, data types. They do so through a language feature, by defining new syntax, as opposed to previously mentioned solutions that were using the syntax of existing Haskell mechanisms. The syntax is rather elegant, but the semantics have some problems, which we discuss after showing the syntax.

In this section, we show how the running example from Section 2.1 is written in this syntax. The extensible data types are declared with the use of a keyword `open` and a kind signature. For the data type `Expr`, this looks as follows:

```
open data Expr :: *
```

The variants of the data type are separated into their own declarations by simply writing a type signature for them, as follows:

```
Const :: Int -> Expr
Add :: Expr -> Expr -> Expr
Mul :: Expr -> Expr -> Expr
```

Note the result type in all of these, `Expr`, which marks these as extensions to the extensible data type with the same name. The capital letter at the beginning of the constructor name here creates the distinction between this and a function with a regular type signature. These three constructor declarations are independent of each other, so it is possible to write them in different modules. It is, however, not possible to have multiple variants in the same declaration as was possible in Data Types à la Carte, in Section 2.2.

Writing an example is done in the same way as with regular Haskell data types:

```
threePlusFive :: Expr
threePlusFive = Add (Const 3) (Const 5)
```

Extensible functions are marked similarly with the keyword `open`, as in the following for the `eval` function:

```
open eval :: Expr -> Int
```

Equations to this function for each constructor of the extensible data type are written simply as for regular Haskell functions:

```
eval (Const i) = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

The difference from regular Haskell functions, however, is that these rows are independent of each other and can be written in different modules.

Extensions to a data type would be written by simply defining a new constructor and a new equation for the function. For negation, we would have a constructor `Neg` and an equation of the `eval` function as follows:

```
Neg :: Expr -> Expr

eval (Neg e) = (-1) * eval e
```

Then, this syntax is translated to code without extensible data types and functions. A naive approach of doing this is by combining all modules into a single module `Main`, and then collecting all parts of each extensible data type and function and combining them into regular data types and functions. Löh and Hinze propose some improvements to this naive approach, but it still describes the general concept of how the translation is done.

While Löh and Hinze do not discuss transformative extensible functions, the translation to regular syntax is likely to mean that there would be no difference in writing a transformative function compared to a destructive one, since there is no extra complexity of transformative functions in regular Haskell. An important thing to note, however, is that the type of the transformative function `desug` would be:

```haskell
desug :: Expr -> Expr
```

This means that there is no distinction between the ingoing and outgoing type here. In solutions such as Data Types à la Carte, we are able to express different types using different compositions of variants, and we are also able to express polymorphic types through type constraints such as subsumption. Variations on Variants can further express the distinction between ingoing and outgoing type by specifying removed variants through the remainder constraint, `:-:`. Here, by using `Expr` as both ingoing and outgoing type, we can make no distinction. The only way to make such a distinction would be to define a new extensible data type, but it would then not be possible to reuse variants or functions that were defined for the original data type with the new type. This is a consequence of the data types in this solution only being extensible, and not composable, which is a significant difference from previously mentioned solutions. Thus, while this simplifies the syntax of the data types, it is a significant limitation in the expressiveness of Open Data Types.

Another drawback of Open Data Types is that the process of translating the code into regular Haskell means that there is no separate compilation. The naive approach as described earlier is to combine all modules into one module. This means that everything would have to be recompiled, even with a small change in one module of the original code. Löh and Hinze propose a slightly better transformation, where only the extensible data types and functions are moved to a main module in the transformation, meaning that one can have separate compilation of everything except the main module. However, this move often results in mutual dependencies between the main module and the modules that contained extensible data types [22].

In their paper, Löh and Hinze describe further improvements that can be made to minimize the recompilation needed [4]. Some recompilation is still needed, so these improvements are not enough for Open Data Types to fulfill the fourth expression problem criterion described in Section 1.2. Thus, Open Data Types is not a complete solution to the expression problem.

While collecting all extensible data types and functions into one module means no separate compilation, independent extensibility is straightforward. As all extensions are gathered into one place during the translation, which module an extension originally came from makes little difference. This in turn leads directly to independent extensibility, which is why Open Data Types fulfills the fifth expression problem criterion from Section 1.2.

The first criterion and third criterion are also fulfilled. We see that this is true as it is possible to extend an extensible data type with new variants by simply adding a new constructor declaration. We can also introduce a new function over this data type by declaring an extensible function, and we can also extend this function over new variants by adding a new case to the function. Any of these steps can be done in a new module without changing the source code for any original definitions.

The final criterion from Section 1.2 to discuss is strong static type safety. When a function extension is missing for some data type variant, the extensible function will be translated into a non-exhaustive function. With GHC, this will be detected during compilation. However, GHC will then only issue a warning that a pattern for the data type variant is missing. As this would not be an error that prevents compilation of the function with a missing variant, Open Data Types does not have strong static type safety at this stage. This is not a fundamental problem, as it would be possible for the translation phase to have its own exhaustiveness check and give an error if a variant is missing, and thus guarantee strong type safety. Such a check would, however, be global, rather than on a per-module basis. This is a necessity from the data types and functions being extensible, and not composable, a significant drawback to Open Data Types. This means that extensions from all modules are always contained in the data type or function, and must be considered in an exhaustiveness check.

Related to this, we can discuss how pattern matching works in general with Open Data Types. A common problem when separating the data type variants into independent declarations is that Haskell's usual first-fit order for pattern matching is not available. Löh and Hinze propose a workaround for this, namely to instead use best-fit pattern matching, which means that it would match on more specific cases before more general ones. This makes it possible to write pattern matching in regular functions with this syntax just as with regular Haskell.

In conclusion, Löh and Hinze provide a rather elegant syntax, while its semantics are not fully a solution to the expression problem. Hence, we want to, if anything, take inspiration from the syntax, but improve the semantics.

2.8 Open Abstract Types

Another solution that provides new syntax is one by Seefried and Chakravarty [6], called *Open Abstract Types*. They call this syntax syntactic sugar, because it is translated into a solution to the expression problem written in regular Haskell that could be used directly, while using the syntax makes it easier. The syntax is similar to that of Open Data Types by Löh and Hinze in that it declares the data types as extensible with a keyword `open`, but it also defines the data type with some initial constructors. For the expression data type in the running example defined in Section 2.1, it would look like the following:

```
open data Expr = Const Int | Add Expr Expr | Mul Expr Expr
```

Note that we here keep the three variants of `Const`, `Add` and `Mul` together in an initial data type, just as with, for instance, Object Algebras in Section 2.6, and as opposed to separating them as with, for instance, Data Types à la Carte in Section 2.2. This is because the data type is extensible, so it is extended with new variants rather than by creating a composition of defined variants. Therefore, this initial definition containing the three variants corresponds to the initial composition of the same variants given in previous sections.

Just as with Open Data Types in Section 2.7, the syntax for using this data type looks the same as for regular Haskell data types:

```
threePlusFive :: Expr
threePlusFive = Add (Const 3) (Const 5)
```

Functions are also written as they would be with regular data types. For the evaluation function, it would look like this:

```
eval :: Expr -> Int
eval (Const i)   = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

The data type can later be extended in a different module by using the keyword `extend`, as in the following for a variant for negation:

```
extend data Expr = Neg Expr
```

A restriction put on the data types by Seefried and Chakravarty is that a data type cannot be extended in the same module in which the data type was defined, and it also cannot be extended multiple times in the same module. However, once a data type has been extended, it is possible to add a new equation for the function `eval` to also handle the extended part, unlike functions in regular Haskell. This is done with a syntax like regular Haskell:

```
eval (Neg e) = (-1) * eval e
```

The fact that this is written independently of previously defined equations for `eval` raises the same problem as in Löh and Hinze's solution, that regular first-fit pattern matching cannot be performed. Seefried and Chakavarty avoid this in their paper by disallowing wildcards altogether, while mentioning that an approach with best-fit pattern matching could be possible here as well.

The variants over which a function can be extended within a module are also limited to the variants that are introduced in the same module. In fact, when a new data type variant is introduced, Open Abstract Types requires all functions over the data type to be extended to handle the new variant. A function must therefore be extended over all available variants, even if we do not intend to ever use the function for some variants. This is related to the fact that the extensible data types in this solution are not composable, so an extensible data type always allows all possible variants.

This restriction also ties into independent extensibility, the fifth criterion for being a solution to the expression problem given in Section 1.2. If an extensible function is to be combined with an independently defined data type variant, the function needs to be extended to support this new variant. The function and variant can only truly be independent if the extension to the function is defined in a module that combines the two. In this solution, this is not allowed, since the function extension must be written in the same module as the variant.

As mentioned, the syntax described above is translated into a solution written in regular Haskell. This is true not just for the syntax defined by this solution, such as extensible data types and extensions, but also for some regular components of Haskell that refer to the extensible data type. This includes functions over the extensible data type, but also constructors for variants to the extensible data type.

The translated code is in many ways similar to Data Types à la Carte. The initial extensible data type is translated into a regular data type, and each extension to the extensible data type is translated into a separate data type. Recall that there can only be one extension in each module, meaning that for each extensible data type, there is at most one of these data types per module. Extensible functions are in turn translated into type classes and instances, with one significant difference compared to with Data Types à la Carte: When several extensible functions (for instance `eval` and `asString`) are defined in the same module, they are collected into the same class so that there is at most one of these classes per module. The transformed code of the original data type and the evaluation function would then look as follows:

```
data Expr_0 = Const Int | Add Expr Expr | Mul Expr Expr

class Eval a where
    eval :: a -> Int
instance Eval Expr_0 where
    eval (Const i)    = i
    eval (Add e1 e2) = eval e1 + eval e2
    eval (Mul e1 e2) = eval e1 * eval e2
```

Here, the original data type `Expr` is translated into a data type `Expr_0`.

An extension to this data type would, as mentioned above, be written in a separate module. This module would then be translated to contain a data type for all variants added through that module. For instance, the extension of `Neg` could be translated into being a variant of a new data type, `Expr_1`. The case for negation of the function `eval` would then be simply translated into an instance for `Expr_1` to the type class `Eval`. This gives the following translation:

```
data Expr_1 = Neg Expr

instance Eval Expr_1 where
    eval (Neg e) = (-1) * eval e
```

If the original module had defined both `eval` and `asString`, the class would instead look like this:

```
class Eval a where
    eval :: a -> Int
    asString :: a -> String
```

Next is the complete data type `Expr` that may be made from any variant. The representation of this complete data type is where Open Abstract Types significantly diverges from Data Types à la Carte. In order to represent an extensible type that can accept any data type variant that implements, in this case, `eval`, we define `Expr` as an existential type, as follows:

```
data Expr = forall a. Eval a => Expr a
```

As an existential type, `Expr` is a concrete type, while its constructor can accept any data type `a` as long as it fulfills `Eval a`. This restriction means that only functions defined through `Eval a` can be used when unwrapping this data type. The definition of the data type as above also raises a new problem. By defining the full data type this way, we are unable to properly make use of any new functions. For instance, if we were to add a class constraint for another function, `AsString` to the original definition of `Expr`, we would have to rewrite it like the following:

```
data Expr = forall a. (Eval a, AsString a) => Expr a
```

This means that we would have broken the criterion of separate compilation stated in Section 1.2, as the module with `Expr` would need to be recompiled with the change of the added function. Instead, we could opt for introducing a new full type with the added function class, as follows:

```
data Expr' = forall a. (Eval a, AsString a) => Expr' a
```

This leads to another problem as we would find that `Expr_0`, `Expr_1` and so on would still be using `Expr` and not `Expr'`. The solution to this problem is a technique that Seefried and Chakavarty call retrospective superclassing, which is based on an extension to the work of Läufer [23]. The idea is that, with the version of `Expr` with just `Eval`, we parameterize `Expr` and `Eval` with a type that represents all functions on `Expr` that should be available in addition to `eval`. These functions are then made available through a superclass to `Eval`. A comparison drawn in the original paper is considering if the context to `Eval` were to be parameterized as follows:

```
class ctx a => Eval ctx a where
    eval :: a -> Int

data Expr cxt = forall a. (Eval cxt a) => Expr a
```

However, it is not actually possible to pass a class or context through parameterization in Haskell in this manner. Instead, the class is represented through a type, which gives something more like this:

```
class Sat (ctx a) => Eval ctx a where
    eval :: a -> Int
data Expr cxt = forall a. (Eval cxt a) => Expr a
data Expr_0 cxt = Const Int | Add (Expr cxt) (Expr cxt)
                | Mul (Expr cxt) (Expr cxt)

class Sat a where
    dict :: a
```

This is the fundamental idea for introducing new functions that can be used on `Expr`. There is still some more to do to actually fill and make use of the parameterization. We will not go through that here, and recommend reading the original paper. [6] However, we can summarize the important parts: The translation will insert constraints on the context based on all functions and data type variants that are available in each specific module. The context type itself is then constructed on a per-module basis based on the available functions and data type variants.

Transformative functions with Open Abstract Types leads to the same situation as with Open Data Types. The functions themselves can easily be defined and implemented in the syntax as with any other extensible function. However, the inability to express subsets of a data type through different compositions means that a desugaring function cannot enforce that the result data type does not contain certain data type variants, unlike other solutions that use closed composed types, such as Data Types à la Carte and Variations on Variants. This means that the type of `desug` cannot be more specific in that sense than:

```
desug :: Expr -> Expr
```

Still, Open Abstract Types fulfills the original four criteria for a solution to the expression problem. In particular, in comparison to Open Data Types, it has managed to solve separate compilation, while still offering extensible, rather than composable, data types. However, as noted earlier, it fails to fulfill independent extensibility, which is the fifth criterion. Hence, Open Abstract Types has a simple syntax with a translation to a solution fulfilling the original four criteria for the expression problem, while it is lacking in the expressive power that comes with composable data types, and also does not support independent extensibility.

2.9 Compilation

Table 2.1 shows a compilation of the proposed solutions to the expression problem given in this chapter. All of these are evaluated based whether they fulfill different criteria or not. The criteria are the five criteria for being a solution to the expression problem stated in Section 1.2, and two other properties that distinguish the solutions, namely:

- **Automatic extension of data types:** When a data type is extended with a new variant, the variant is immediately included in the data type. It is thus not possible to use a version of the data type that contains only some of the defined variants.

- **Automatic extension of functions:** An extensible function is extended automatically, meaning that when an extension is written to a function, it is part of all uses of the function.

Having data types or functions being extended automatically tends to mean simpler code, but with less expressive power. For instance, having composable data types, the opposite of automatically extended data types, gives a significant advantage in expressiveness in that one can freely choose which data type variants to include in a composition. The difference in expressive power is, on the other hand, not as prominent regarding if functions are extended automatically or not. Having composable functions rather than automatically extended means that one can write multiple function extensions for the same data type variant, and then choose which to include in the function composition. This is not as clear a practical advantage as having composable data types.

	Data Types à la Carte	`compdata`	Variations on Variants	Trees that Grow	Object Algebras	Open Data Types	Open Abstract Types
Extensibility in both directions	x	x	x	x	x	x	x
Strong static type safety	x	x	x	x	x	–[8]	x
No modification of duplication	x	x	x	x	x	x	x
Separate compilation	x	x	x	x	x	–	x
Independent extensibility	x	x	x	x	x	x	–
Automatic extension of data types	–	–	–	–	–	x	x
Automatic extension of functions	x	x	–	–	–	x	x

Table 2.1: *Compilation of all proposed solutions presented in this chapter. Solutions are evaluated based on the criteria for being a solution to the expression problem given in Section 1.2, as well as two other properties that can distinguish the solutions from each other. Note that the two properties are not necessary to fulfill, and that it is not always obvious whether it is better to fulfill such a property or not.*

[8]Open Data Types as presented in the paper does not fulfill strong static type safety, but it could be possible to implement it in the translation process. See the discussion in Section 2.7.

3

Proposed Syntax

At its core, a syntax for a solution to the expression problem needs syntax for the following:

- defining data types
- combining or extending the data types
- writing operations on the data types that also can be extended to work on new combinations or extensions of the data types

Note that we must also be able to extend the set of functions, as a solution to the expression problem must support extensibility in two directions: the data type variants and the set of functions. However, since we expect to keep a functional style in the syntax, the extension of the set of functions will not be the main concern as direction of extensibility. Recall the difference between functional and object-oriented approaches presented in Chapter 1.

This chapter will describe the syntax we have designed. The purpose of the syntax design is to, in a theoretical way, show how a simpler syntax for solving the expression problem can be designed, as mentioned in Section 1.1. The emphasis is thus not on the exact syntax regarding, for instance, choices of keywords and symbols. Rather, we want to show the parts that need to be included.

We will introduce the development of the syntax step by step with the help of the running example of a small expression language, consisting of constant integers, addition and multiplication, introduced in Section 2.1. Then, we present the final syntax in each section by showing it in a table at the end. A complete compilation of our syntax is done in a table in Appendix A. We also show a complete example of a program written in our syntax in Appendix B.1.

3.1 Notation

Before going into the actual syntax, we present here the notation that we use to describe it. This is based on the notational conventions used to describe Haskell 2010 in its language report [24].

`data`	terminal syntax in typewriter font
pattern	non-terminal syntax in italic font
[*pattern*]	optional (i.e. zero or one)
{*pattern*}	zero or more repetitions
$pattern_1$ | $pattern_2$	choice

We present the patterns in this chapter in a slightly informal way, to showcase the overall syntax rather than showing exactly the syntactic components of which they consist. This is done both by simplifications, to not always show all possible ways of constructing a syntactic component, and by making distinctions between patterns that in reality would be syntactically equal. The syntax is presented more formally in Appendix A.

As an example of a simplification, a type signature can be constructed to be the type signature of several functions, as in the following:

```
f, g, h :: Int -> String
```

Syntactically, this means that the type signature of a function can be written as:

Var {, *Var*} `::` *Type*

Here, *Var* is a variable name, that is, a name starting with a lowercase letter, and *Type* is simply a type. We make the simplification to just show the possible case for one single function:

Var `::` *Type*

Another simplification we make here is that we do not provide all details for every syntactic component that we use, such as *Var* and *Type*, but rather explain them informally. An explanation of such syntactic components is given in Table A.1. Furthermore, a *Var* can be qualified with a module name, and is then instead written as *QVar*. We have chosen to not present this distinction in this chapter, but will do so when giving the formal details in Appendix A.

As an example of a distinction we make in some cases, consider the function example again. We could choose to express the name of the function as *FunctionName* instead, to distinguish it from other uses of the syntactic component *Var*.

3.2 Data Types

To begin formulating a syntax for extensible data types, we first inspect regular data types to identify the crucial parts. We do this with the running example of a small expression language that we introduced in Section 2.1:

```
data Expr = Const Int | Add Expr Expr | Mul Expr Expr
```

The first thing to remember is that a data type can have several different constructors for its different variants, but in the standard syntax it cannot be extended with more variants without recompiling the whole data type. The data type `Expr`, along with any related operations on it, would need to be rewritten and recompiled to add, for instance, negation of an expression, `Neg Expr`. The crucial starting point is thus to decide how to make the data type extensible.

3.2.1 Separation into Pieces and Compositions

Extending a data type with a variant for negation can either be done by a direct extension to an extensible data type, as is done in Open Data Types (Section 2.7) and Open Abstract Types (Section 2.8), or through a composition of building blocks, as is done by solutions like Data Types à la Carte (Section 2.2) and Trees that Grow (Section 2.5). The second approach, to have composable data types, gives a significant advantage of having higher expressive power, since it is possible to form different compositions for different situations. Because of this, we want the data types in our syntax to be composable.[1]

To have composable data types, we want to be able to separate the constructors into their own declarations and later combine them into a *composed type*, or *composition*. We call these parts of a composition *data type pieces*, or just *pieces*, which, for now, can be viewed as single-constructor data types. To begin with, we restrict the pieces to be non-polymorphic; that is, they are not allowed to contain type variables. This gives us the first iteration of our syntax, where the first row is just like a regular non-polymorphic data type with only one data constructor:

$$\begin{array}{rl} \text{PieceDecl:} & \texttt{data}\ \mathit{PieceName}\ \texttt{=}\ \mathit{DataConDecl} \\ \text{ComposedType:} & \texttt{(}\ \mathit{PieceName}_1\ \texttt{|}\ \dots\ \texttt{|}\ \mathit{PieceName}_n\ \texttt{)}\quad n \geq 1 \end{array}$$

Here, a *DataConDecl* is simply a data constructor with zero or more parameters, given as *DataCon* {*Type*}, for instance `Const Int`. Note that the parentheses in the composed type are required. Note also that the first row here is a declaration, whereas the second is a type. This means that the type could be used, for instance, directly in a type signature, or in a type declaration as the following:

$$\texttt{type}\ \mathit{ComposedName}\ \texttt{=}\ \texttt{(}\ \mathit{PieceName}_1\ \texttt{|}\ \dots\ \texttt{|}\ \mathit{PieceName}_n\ \texttt{)}\quad n \geq 1$$

This separation into pieces that are then composed is similar to the idea behind Data Types à la Carte, `compdata` and Variations on Variants, Sections 2.2–2.4. As opposed to the approach in those solutions, however, we make the composition directly, without the two-step process they use with the coproduct and then the wrapping with `Term`. Thus, we make no distinction between open and closed compositions.

[1] It is also worth noting that the two solutions presented in Chapter 2 that have automatic extension of data types both provide their solutions through defining new syntax that is translated into compilable Haskell code, while none of the solutions based on composable data types do so. As we aim at designing new syntax, there is a greater research gap in the existing solutions regarding a syntax for composable data types.

The direct composition of pieces is also simpler than the linked-list composition in Trees that Grow (Section 2.5), which is more verbose as variants are linked together one pair at a time.

As mentioned above, the fact that we compose the pieces explicitly, rather than having data types that are automatically extended as in the approaches by Löh and Hinze (Section 2.7) and Seefried and Chakravarty (Section 2.8), means that we get the ability to freely compose the pieces we want. This gives more control over which pieces are included in a type.[2]

The expression language example `Expr` would then be defined with a data type piece each for `Const`, `Add` and `Mul`, combined in a composed type `(Const | Add | Mul)`. The separation into pieces means that we could extend the original data type, `Expr`, given by the composition `(Const | Add | Mul)`, by simply declaring the data type piece we want to add, `Neg` for negation, and then defining a new composition, like `(Const | Add | Mul | Neg)`.

3.2.2 Multi-constructor Pieces

To make data type pieces more flexible, we could still want to have multiple constructors for one piece, as is done in Data Types à la Carte, if we have parts that we never want to have separate. For instance, this is useful in lambda calculus, where one could have lambda abstractions and applications in the same piece, as they should never be used one without the other. In our expression language example, we could choose to treat basic arithmetic operations, such as `Add` and `Mul`, as a whole, by having them in a single piece `Op`, and thus be able to add them into a composed type just with their unified name. To conclude, we allow multi-constructor data type pieces, which gives us the following syntax:

PieceDecl: `data` *PieceName* `=` *DataConDecl*$_1$ `|` ... `|` *DataConDecl*$_n$ $n \geq 1$

This is a an advantage over having a syntax for pieces like the one proposed by Löh and Hinze, as constructors with a type signature, since we can provide multi-constructor pieces, something that is not possible in their syntax.

3.2.3 Categories for Pieces and Compositions

For the expression language example, with the extension of negation, we can now write the pieces and compositions like the following[3]:

[2]The syntax for a piece can be further compared to that of Open Data Types (Section 2.7). In their syntax, data type variants are written as constructors with a type signature, similar to GADTs (Generalized Algebraix Data Types). Our syntax for pieces is closer to that of regular data types, which gives higher user friendliness due to its familiarity and that if does not require type signatures for the pieces. It could be possible to add the other possible syntax as well, but this is not something we have considered further in this project.

[3]In this report, we will show code written in our syntax contained in a box like this, to distinguish it from other code examples. This includes suggestions for our syntax as well as the actual syntax we have designed.

```
data Const = Const Int
data Op = Add <rec> <rec> | Mul <rec> <rec>
type ExprComp = (Const | Op)

data Neg = Neg <rec>
type ExprCompWithNeg = (Const | Op | Neg)
```

Here, we note the need to be able to make a piece recursive, denoted in the example with a placeholder `<rec>` for the recursive data constructors `Add`, `Mul` and `Neg`, since we have no way of expressing that yet. Data Types à la Carte, and in turn, `compdata` and Variations on Variants, solves the need for recursion by using parameterized data types, to give a parameter to each data type like `data Op a = Add a a | Mul a a`. We could opt for constructing data types in this way in our syntax as well. However, this is not as intuitive for a user as the simple regular data type that we saw in the beginning of this section, where the recursive constructors are formed simply by using the name of the data type itself as the recursive parameter, like `Add Expr Expr`. We want to mimic this simple syntax, rather than using the arguably less intuitive syntax of parameterized data types. It could be possible to for the recursive parameter allow a parametric syntax as well, so that a user can choose between the two. We have not considered that possibility further in this project.

In our case, the separation into several data type pieces means that we have to specify that the recursive parts in, for example `Add`, can be of any piece of the *composed* type, not just a recursion on the piece itself. Furthermore, the data type pieces should be defined independently of any specific composed type, because we could create several compositions from the same pieces, like `ExprComp` and `ExprCompWithNeg` in the example above. This means that we cannot use a fixed composed type as the recursive parameter either, but instead must use a separate name that can refer to any composed type containing the pieces. To still have a connection between the data type pieces and the composition, we define a syntax in which every piece and every composition of pieces belong to a *category*, more specifically also called a *piece category*. We denote this with a special arrow symbol `==>` as follows:

$$
\begin{array}{rll}
\text{PieceDecl:} & \texttt{data}\ \mathit{Category}\ \texttt{==>}\ \mathit{PieceName}\ \texttt{=} & \\
 & \qquad\qquad \mathit{DataConDecl}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \mathit{DataConDecl}_n & n \geq 1 \\
\text{ComposedType:} & \mathit{Category}\ \texttt{==>}\ \texttt{(}\ \mathit{PieceName}_1\texttt{|}\ldots\texttt{|}\ \mathit{PieceName}_n\ \texttt{)} & n \geq 1
\end{array}
$$

This category is simply a *Con* (a name starting with an upper-case letter) and is then used as the name of the recursive part. Note that, while categories are used as types for these recursive parts, they should not be considered being actual types, and cannot be used anywhere a type is normally used outside of piece constructors.

In the expression language example, we can have the data type pieces and their compositions belong to the category `ExprCat`. This means that we can write the recursive data constructors like `Add ExprCat ExprCat`. Hence, we can now write the pieces and compositions described above like the following:

```
data ExprCat ==> Const = Const Int
data ExprCat ==> Op = Add ExprCat ExprCat | Mul ExprCat ExprCat
type ExprComp = ExprCat ==> (Const | Op)

data ExprCat ==> Neg = Neg ExprCat
type ExprCompWithNeg = ExprCat ==> (Const | Op | Neg)
```

To introduce these categories, we also make a distinction between regular data types and data type pieces that can be combined into a composed type, because regular data types should not be a part of any category. Therefore, we mark the pieces with a keyword, `piece`, and further on refer to the data constructors of a piece as *piece constructors*. We also need to denote that a piece constructor can have a category as a parameter for the recursive argument, in addition to being able to have regular *Type* parameters. This is denoted by a *PieceConDecl*, which is given as *PieceCon* $\{$*PieceArg*$\}$, where each *PieceArg* is either a *Type* or a *Category*.[4] This gives us the following iteration of the syntax for piece declarations:

PieceDecl: `data piece` *Category* `==>` *PieceName* `=`
PieceConDecl$_1$ `|` ... `|` *PieceConDecl*$_n$ $\quad n \geq 1$

The categories can be viewed as open data types, like the data types declared with a keyword `open` in Open Data Types (Section 2.7), which can then be explicitly closed through the composition of pieces. In that sense, the use of a category in place of a type is defined to mean a polymorphic composition, that is, some possible composition of pieces in that category. This is exactly what we want by using a category as the recursive parameter in the piece constructors.[5]

Piece categories have further advantages. With them, we have the possibility of distinguishing separate categories for separate data type pieces, such as statements and expressions. The concept of categories could possibly also be expanded to reference other categories, for having data type pieces of one category that contain pieces of another category, such as statements containing expressions; this will be discussed in Section 5.6.1.

On the other hand, the use of categories also limits the expressiveness, since we cannot combine any two pieces belonging to different categories, but only those within the same category. As previously mentioned, however, the categories give several advantages. They give a way to specify the recursive parameter with a category,

[4]In fact, we have defined our syntax so that a *Category* is parsed just as a *Type*. This means that a *PieceArg* could also be a *Type* containing a *Category*, such as `Maybe ExprCat`. To better show the use of a category in the syntax here, we make the simplification to define *PieceArg* to be *Type* | *Category*.

[5]The specific composition which the category represent in that case is then restricted to be the same as the composition for the whole data type. For instance, for an addition value of the composed type `ExprComp`, the category in `Add ExprCat ExprCat` would represent the same composed type `ExprComp`. For an addition value of a different composed type, the category would instead represent the other composed type. We cover this more in Section 3.3, as well as possible ways of relaxing the restriction in Section 5.5.4.

which gives a more intuitive syntax for recursive piece declarations compared to the parametric approach in solutions like Data Types à la Carte. Categories also give the ability to distinguish data type pieces with different categories. This ability can be compared to how a type system distinguishes terms of different types, something that also limits the expressiveness, but is still very useful.

3.2.4 Modules and Category Declaration

To make the data types composable, we need to be able to have different pieces defined in different modules that can be compiled separately. To make sure that the same category can be used over several modules, it needs to be declared independently in one place, and not just indirectly by being used in pieces and compositions. Hence, we define the simple syntax for a piece category declaration as follows:

PieceCatDecl: `piececategory` *Category*

The introduction of piece categories, in particular the category declaration, means that we cannot combine modules as freely as in solutions like Data Types à la Carte. In such solutions, it is possible to define, for instance, `Const` and `Neg` in two separate modules, while our syntax has the extra restriction that the two modules must depend on a module containing the category declaration. This is thus a limitation in the expressiveness of our approach.

We can also compare this category declaration to the extensible data type declarations in Open Data Types and Open Abstract types, specified with the keyword `open`. The pieces defined to belong to a certain category correspond to defining data type variants in those solutions, which all belong to the extensible data type. The main difference is our composition, which is used to select a subset of pieces in a category, instead of using the category as an extensible data type, as `Expr` is used in Open Data Types and Open Abstract Types. This means that the data types in our approach are composable, whereas they are not in those solutions.

3.2.5 Polymorphic Pieces

For our last addition in this section, we note that the pieces we have presented so far are non-polymorphic; that is, they contain no type variables. We can generalize our piece declaration to allow polymorphic pieces by writing type variables to the right of the *PieceName*, imitating the syntax for polymorphic data types. These type variables can then be used in place of types for piece constructors just as with regular data constructors. We would then also need to update the syntax for composed types to be able to there specify the type variables introduced in the piece declaration. This gives the following syntax:

PieceDecl: `data piece` *Category* `==>` *PieceName* {*TypeVar*} `=`
PieceConDecl$_1$ `|` ... `|` *PieceConDecl*$_n$ $\quad n \geq 1$
ComposedType: *Category* `==>` `(` *PieceRef*$_1$ `|` ... `|` *PieceRef*$_n$ `)` $\quad n \geq 1$

A *PieceRef* is given by *PieceName* { *Type* }, where the *Type*s are separated by space.

To see an example of this, we imagine a modified version of `Const`, which has been made to accept any parameterized type, rather than just an `Int`. This could then be written as follows:

```
data piece ExprCat ==> ConstP c = ConstP c
```

This piece can then be instantiated, when using the piece to work for different types, for instance `ConstP Int` and `ConstP Double`. The parameterized type `c` needs to be specified in this way when using the piece in a composition. For instance, we can have a composed type for expressions based on integer constants, `ExprInt`, which would then be isomorphic to the previously defined type `ExprComp`. This would be defined as follows:

```
type ExprInt = ExprCat ==> (ConstP Int | Op)
```

3.2.6 Summary of Syntax for Data Types

We have now presented all of our proposed syntax regarding data types, which we summarize in Table 3.1.

PieceCatDecl	`piececategory` *Category*
PieceDecl	`data piece` *Category* `==>` *PieceName* { *TypeVar* } `=` *PieceConDecl*$_1$ `|` ... `|` *PieceConDecl*$_n$ $n \geq 1$
ComposedType	*Category* `==>` `(` *PieceRef*$_1$ | ... | *PieceRef*$_n$ `)` $n \geq 1$
PieceConDecl	*PieceCon* { *PieceArg* }
PieceArg	*Type* | *Category*
PieceRef	*PieceName* { *Type* }

Table 3.1: *Our proposed syntax for composable data types, containing declarations of categories and pieces, and a type for composed types. Some syntactic components of the three are also specified further.*

Finally, we show the full representation of the expression language example in our syntax. Every declaration here is independent and could thus be compiled separately if put into different modules, with correct module handling. We present the example here with the original definition of the data type in one module, `Expr`, and the extension of negation in a separate module, `Negation`. Recall that our choice of keywords is not important for the purpose of this project, and should thus not be considered final. What is important is the existence of these syntax declarations and the purpose that they serve.

```
module Expr where

piececategory ExprCat

data piece ExprCat ==> Const = Const Int
data piece ExprCat ==> Op = Add ExprCat ExprCat
                          | Mul ExprCat ExprCat

type ExprComp = ExprCat ==> (Const | Op)
```

```
module Negation where
import Expr

data piece ExprCat ==> Neg = Neg ExprCat

type ExprCompWithNeg = ExprCat ==> (Const | Op | Neg)
```

3.3 Constructing Values of Composed Types

To be able to use the composable data types introduced in the previous section, we need a way to construct examples. For instance, we would want to be able to form examples of an expression like the following written for a regular data type using its data constructors:

```
twoMulThreePlusFive :: Expr
twoMulThreePlusFive = Const 2 `Mul` (Const 3 `Add` Const 5)
```

We want to be able to form examples with composable data types using the piece constructors just like regular data constructors, as is done with the extensible data types in Open Data Types (Section 2.7) and Open Abstract Types (Section 2.8). Thus, we state that the constructors defined in a piece declaration can be used directly by name to create a value, as in the following example:

```
constTwo = Const 2
```

While a regular data type constructor would create a value of the data type, a data type piece in our syntax is not considered a type of which a piece constructor can create a value. Rather, these piece constructors should create a value of the extensible data type, which in our syntax is represented by composed types.[6] In particular, we say that the constructor creates a value of some composed type that contains the piece to which the constructor belongs. To make the constructor usable in a variety of situations, we state this more generally: A piece constructor

[6]This is different from Data Types à la Carte and similar solutions, where a constructor to a data type variant does not create a value of a complete composed type, but instead creates a value of a data type specific to the variant. Instead, smart constructors are used there to create a value of a composed type.

can create a value of any composed type that contains the piece to which the constructor belongs. For example, `Const 2` could be of type `ExprCat ==> (Const)` or `ExprCat ==> (Const | Op)`, but not of type `ExprCat ==> (Op)`. Thus, the use of our piece constructors corresponds to the use of smart constructors defined in Data Types à la Carte in Section 2.2, and also derived and used in `compdata` in Section 2.3. The difference is that we want to use the constructors directly as constructors without changing their names, and not use smart constructor functions, which require different names starting with a lower-case letter.

For the piece constructors with a recursive part, such as `Add`, the recursive parts represent the data type that contains the piece. We represent that here by stating that the type of a recursive part of a constructor is the same composed type as the type of the composition created by the constructor.[7] For example, if we have two expressions with the respective types `e1 :: ExprCat ==> (Const | Op)` and `e2 :: ExprCat ==> (Const | Op)`, then the addition of those has the type `(Add e1 e2) :: ExprCat ==> (Const | Op)`.

Now, let us return to `constTwo`. As mentioned above, we can state its type, for instance, like the following:

```
constTwo :: ExprCat ==> (Const)
constTwo = Const 2
```

Next, we consider the larger example `twoMulThreePlusFive` in our syntax, which needs to have a type containing the pieces `Const` and `Op`, for instance as in the following:

```
twoMulThreePlusFive :: ExprCat ==> (Const | Op)
twoMulThreePlusFive = Const 2 `Mul` (Const 3 `Add` Const 5)
```

The types of both of these examples are correct. Their types are composed types, and the examples only use constructors belonging to pieces in said composed types. However, because of the concrete and inflexible composed types in the type signatures, the use of these values becomes limited in a larger context. For instance, if `constTwo` were to be used instead of `Const 2` in `twoMulThreePlusFive`, `constTwo` cannot have the type `ExprCat ==> (Const)`, but must have a type containing `Op` as well, namely `ExprCat ==> (Const | Op)`. We will come back to how this can be alleviated in Sections 3.5.2 and 5.6.4.

3.4 Introduction to Extensible Functions

The next step of defining a syntax for composable data types is that we need a way to perform operations over the data types, that is, define and use functions. To have functions over a composable data type, we need to be able to extend the functions later to act on compositions with new pieces of the data type. In the related work

[7] We discuss ways of relaxing this restriction in Section 5.5.4.

in Chapter 2, we saw two main approaches for doing this: either to have automatic extension of functions or not. In the first alternative, used for instance by Data Types à la Carte, we have some kind of declaration of the function, which is extended by separate cases to handle different variants, all of which automatically extend the function for all future uses. In the second alternative, used for instance by Variations on Variants, separate cases are written which are then combined through an explicit composition.

The first alternative means a generally simpler syntax and also that it is possible to reuse a function directly for different compositions. On the other hand, it also means losing the possibility of picking which extensions to constitute the function. Especially, this can lead to problems when having incompatible, independently developed extensions. Still, we believe that the benefits of a simpler syntax in this case weighs more than the drawbacks, and choose to follow the approach by Data Types à la Carte to have automatic extension of functions. In particular, we likewise take inspiration from an existing extensible mechanism in Haskell, namely type classes.

To begin formulating our syntax for extensible functions, let us consider our running example from Section 2.1. If we write an evaluation function over the expression language that evaluates constants, additions, and multiplications to an `Int`, we need to be able to extend it with a new case for negation if we want to evaluate expressions containing such operations. For a regular, non-composable Haskell data type `Expr`, we would have an evaluation like this:

```
eval :: Expr -> Int
eval (Const i)    = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Neg e)      = (-1) * eval e
```

An extension to handle negation means that we want to write the last row independently from the others, so that the old code does not need to be recompiled. To be able to write the rows of the functions in different modules and compile them independently, we need to make sure that they are still connected, so that the compiler knows that when writing, for instance, `eval (Const i) = i`, it is referring to the `eval` function with type `Expr -> Int`. Similar to the introduction of the category declaration in Section 3.2, we see the need for a declaration of a function. This would define the name and type of a function.

We keep the syntax of this *extensible function declaration* similar to a regular type signature since they are much alike. However, it is not strictly a type signature, because the declaration is required, as opposed to the oftentimes optional type signature. Furthermore, a more important difference is that the function is an extensible function, meaning that it must be defined for a composable data type. Therefore, we use a different symbol in the syntax of the declaration of an extensible function [8], but otherwise keep the syntax as a type signature:

$$\text{CompFunDecl:}\quad \textit{FunctionName}\ \texttt{-:}\ \textit{Type}$$

The *Type* here is a function type, which means that it is a type written on the form $Type_1$ `->` ... `->` $Type_n$, where $n \geq 1$. At least one of the $Type_k$ in this must be a type that can be instantiated to a composed type when calling the function, because otherwise this would just be the type of a regular, non-extensible function. As defined in Section 3.2, this is expressed with a category. In the case of the `eval` example, this is `ExprCat`, so the type of `eval` would be as in the following declaration:

```
eval -: ExprCat -> Int
```

Here, the category serves the same purpose of specifying the argument as in the recursive parameters of the data types, instead of having a polymorphic type variable. We can compare this to the type signature for the `eval` function written with Data Types à la Carte in Section 2.2:

```
eval :: Eval f => Term f -> Int
```

The `Term f` here serves a similar purpose as our category, but is instead specified to support evaluation through the context `Eval`, rather than specifying the category of pieces supported. Note also that what we have written in our syntax is a declaration, and not a type signature. In fact, as for now, we cannot write an actual type signature for `eval`, and it is thus not possible to infer the type of the function. This is because the category `ExprCat` represents a polymorphic type variable, that is, any composition of category `ExprCat`, and we are not yet able to express this with an actual type. We will come back to this in Section 3.5.1.

In the evaluation example above, we have an extensible function that takes a value of some composed type, of a certain category, as its argument and deconstructs it to a simpler type, in this case an `Int`. This is thus a *destructive extensible function*, as also mentioned in Section 2.1. As such, the function deconstructs a value, which can be done based on its structure via pattern matching, and that it does not need to construct any value of a composed type, as do transformative and constructive extensible functions. This means that destructive extensible functions are the simplest. Therefore, we introduce the syntax of such functions in this section, and discuss transformative functions in Section 3.6.4 and constructive functions in Section 5.2.

[8]The use of the symbol `-:` could easily be exchanged for the use of a keyword, such as `open` in the proposed solution of Löh and Hinze (Section 2.7). The importance is that it is distinguished from a regular, non-extensible function. It could also be possible to have the syntax identical to the syntax of regular functions, as does Open Abstract Types (Section 2.8), and instead only make the distinction when transforming the code to regular Haskell, depending on the type of the function. However, since the extensible and non-extensible functions behave differently, it is useful to have a distinction which is clear to the user.

For destructive extensible functions, such as `eval`, we now state the general syntax for the function declaration:

CompFunDecl (destructive): *FunctionName* `-:` *Category* `->` *Type*

The *Type* in this declaration could be any type, including a function type if more arguments are wanted. In fact, it could contain more composed type as arguments as well, but the function will only concern deconstruction of the first argument, which is also the argument over which one can define extensions to the function.

This syntax means that we restrict the first argument to always be a category. This is because we want to make the distinction that this argument is handled in a special way, namely to be deconstructed via this function, and the only possible position to put such a distinction is the first. Instead of using a category for the composable type argument, one could express it with polymorphism through type variables, as does Data Types à la Carte, but that would not yield the same distinction that it is a special argument. The use of a category also makes the syntax more intuitive for a user, similar to the argument for using categories as recursive parameters in the data types in Section 3.2. Furthermore, the use of the category here also means that it could be possible to encode a way to ensure that the types used in the function are actually pieces belonging to said category.

To then define the function, we need to define it for every possible case. In a destructive extensible function, this means that we need to define a case for every possible piece constructor that can match the input argument via pattern matching. Piece constructors that belong to a multi-constructor piece, such as `Op` with constructors `Add` and `Mul`, do so because they should not be separated. Therefore, we want to define the extension cases by *piece*, rather than by constructor, analogously to how we create the type compositions by piece. Therefore, we define a syntax where we write an extension for a piece such that it contains pattern-matching cases for all its constructors. The syntax for this is inspired by that of instance declarations for type classes, since those have a similar purpose.

CompFunExt: `ext` *FunctionName* `for` *PieceRef* `where`
{*FunctionName PieceConPat* {*Pattern*} `=` *FunctionDef*}

Here, *PieceRef* is a piece with zero or, in the case of a polymorphic piece, more type parameters, given as *PieceName* {*Type*}, where the *Type*s are separated by space. If the piece is non-polymorphic, it is thus simply a *PieceName*. For instance, it could be `Op`. Moreover, *PieceConPat* is a pattern that matches a piece constructor including potential parameters, given as *PieceCon* {*Pattern*}. In particular, the piece constructor should be a constructor for a piece belonging to the category specified as the first argument in the function declaration. For instance, a *PieceConPat* could be `Add e1 e2`. Furthermore, {*Pattern*} in the extension is zero or more patterns for potential more arguments to the function. Lastly, *FunctionDef* is the right-hand-side implementation of the function, such as `eval e1 + eval e2`.

Note here the similarity between the components in place in this syntax and the syntax provided by Data Types à la Carte and similar approaches, which are solutions based on type classes with instances for each data type variant. The keywords are different, but we can note a similarity in semantics. Furthermore, we have chosen to represent the functions without the use of algebras, and instead make the recursion explicit. The syntax would look like this for the evaluation example:

```
ext eval for Const where
    eval (Const i) = i

ext eval for Op where
    eval (Add e1 e2) = eval e1 + eval e2
    eval (Mul e1 e2) = eval e1 * eval e2
```

Note here the explicit recursion seen in the equations for the piece `Op`, for instance `eval e1 + eval e2`, as opposed to the syntax with algebras in Data Types à la Carte as just `e1 + e2`. The explicit recursion makes the syntax more intuitive and similar to the syntax shown in the regular Haskell example in the beginning of this section.

Now, we can summarize our proposed syntax for simple, destructive extensible functions, see Table 3.2. This will be modified in the following sections to handle different kinds of functions.

CompFunDecl	*FunctionName* `-:` *Category* `->` *Type*
CompFunExt	`ext` *FunctionName* `for` *PieceRef* `where` {*FunctionName PieceConPat* {*Pattern*} `=` *FunctionDef*}
PieceRef	*PieceName* {*Type*}
PieceConPat	*PieceCon* {*Pattern*}

Table 3.2: *Our proposed syntax for simple, destructive extensible functions, containing a declaration and a function extension. This syntax will have additions to handle more complicated cases in the following sections. Some syntactic components in the extensions are also specified further.*

Last, we show how this fits with the example from Section 3.2.6. We define a new module for evaluation, `Eval`, and add the evaluation of negation into the module `Negation`, which now also imports `Eval`. This means recompilation of `Negation`, although that could be avoided by simply putting the evaluation of `Neg` in a new module as well, but for this example we see them as a joint extension module.

```
module Eval where
import Expr

eval -: ExprCat -> Int

ext eval for Const where
    eval (Const i) = i

ext eval for Op where
    eval (Add e1 e2) = eval e1 + eval e2
    eval (Mul e1 e2) = eval e1 * eval e2
```

```
module Negation where
import Expr
import Eval

...

ext eval for Neg where
    eval (Neg e) = (-1) * eval e
```

For a simple example, we can use the function right away. Consider the negation of the expression example from Section 3.3:

```
negTwoMulThreePlusFive :: ExprCat ==> (Const | Op | Neg)
negTwoMulThreePlusFive =
    Neg $ Const 2 `Mul` (Const 3 `Add` Const 5)
```

To evaluate this, one would simply write:

```
> eval negTwoMulThreePlusFive
> -16
```

3.5 Constraints on Composed Types

For regular types, we can represent them with type variables in polymorphic functions. We can then put constraint on those variables by writing a context containing the desired constraints. For instance, we can write the following to denote that the type `e` is an instance of the `Show` class:

```
showExpr :: Show e => e -> String
```

We want to be able to write similar such constraints for our pieces and composed types. This means that we can use parametric polymorphism for a composed type, which is useful in order to allow different compositions without needing to specify them exactly. Note that this is only necessary since we have composable types, meaning that we are able to have different compositions, rather than just extensible types as in Open Data Types and Open Abstract Types (Sections 2.7–2.8), where the type is the same regardless of how it is used. We will present three different kinds of such constraints in the following subsections.

3.5.1 Function Constraints

To illustrate the first kind of constraint, we show an example where we want to write a polymorphic function that calls an extensible function. Consider an extensible function with a type as in the following declaration:

```
evalCond -: ExprCat -> Bool -> Int
```

If we were to define a helper function that calls `evalCond`, with a specific `Bool` value, it might look like this:

```
evalFalse :: <some composed type to call evalCond on> -> Int
evalFalse a = evalCond a False
```

In order to call an extensible function on a certain composed type, the function must be extended over all pieces of the composed type. We express this as a constraint and allow this to be written in the context of a type signature, in the same way as regular type constraints. The syntax for the function constraint is defined simply as follows:

FunctionConstraint: *Type* `with` *FunctionName*

With this, we can use parametric polymorphism in the type of the helper function, and write it like the following:

```
evalFalse :: e with evalCond => e -> Int
evalFalse a = evalCond a False
```

As seen in Section 3.4, we introduce an extensible function with a declaration similar to a type signature. It was, however, not possible to express the actual inferred type of the function. With the function constraint, that is now possible. For example, the type of the extensible function `eval` can be expressed as:

```
eval :: e with eval => e -> Int
```

Note here the similarity to the type of `eval` in Data Types à la Carte (Section 2.2): `eval :: Eval f => Term f -> Int`. Our constraint `e with eval` corresponds, in essence, to the constraint `Eval f`.

3.5.2 Piece Constraints

In Section 3.3, we saw that it was difficult to express the type of, for example, `constTwo = Const 2` in a general way in order to make it reusable in larger examples that also contain other pieces. What we want is to express the necessary pieces in a composed type, while still allowing it to contain more pieces. The need for this is even more present in functions that in some way create a value of a composed type, such as transformative extensible functions, where we need to express the necessary pieces in the result composed type.

Before considering such functions, we look back at the example in Section 3.3. First we have the constant 2, given with a type signature as:

```
constTwo :: ExprCat ==> (Const)
constTwo = Const 2
```

We want to reuse it in the larger example given with a type signature as:

```
twoMulThreePlusFive :: ExprCat ==> (Const | Op)
twoMulThreePlusFive = constTwo `Mul` (Const 3 `Add` Const 5)
```

This is not possible when `constTwo` is specified to have the concrete composed type `ExprCat ==> (Const)`, since it must have the type `ExprCat ==> (Const | Op)`, as that is the type to be constructed by `Mul`. This could be solved by converting the type of `constTwo` into `ExprCat ==> (Const | Op)`, a possibility which we will discuss in Section 5.5.4. Without type conversions, we want to express the type of `constTwo` in a more general way, so that it is usable in the larger example, without specifying the type to contain any more pieces than the `Const` it has.

To do so, we want to write it with a type variable and a constraint, specifying the type variable to be any composition containing at least the piece `Const`. We express this constraint written in the context of a type, with the following syntax:

PieceConstraint: *PieceRef* `partof` *Type*

Here, a *PieceRef* is, as before, given as *PieceName* {*Type*}, to allow for a polymorphic piece.

For `constTwo`, this gives the following type signature:

```
constTwo :: Const partof e => e
```

This is now directly usable in `twoMulThreePlusFive`, since the type can be instantiated to `ExprCat ==> (Const | Op)`. To make it possible to reuse the larger example in other examples as well, we can express that too with a polymorphic type, as follows:

```
twoMulThreePlusFive :: (Const partof e, Op partof e) => e
```

In Data Types à la Carte and similar approaches, subsumption corresponds to this constraint. For instance, `constTwo` would be written as:

```
constTwo :: Const :<: f => Term f
constTwo = iConst 2
```

As noted in Section 3.3, the use of piece constructors here corresponds to the smart constructors used in Data Types à la Carte and similar approaches, with the difference that we keep the constructor syntax `Const` rather than writing it as a function `iConst` or similar.

As another example, consider the following function:

```
mulTwo expr = expr `Mul` Const 2
```

This takes an expression and creates a new expression that is the old one multiplied by two. With a piece constraint, we can now write it as follows:

```
mulTwo :: (Const partof e, Op partof e) => e -> e
mulTwo expr = expr `Mul` Const 2
```

Note that both the argument and result value must have the same type variable, with the same constraint. This is because the argument is part of the result, so it cannot be free to, for instance, contain pieces not allowed in the result. It is further restricted by the use of the constructor `Mul`, which is defined to take two arguments of the same type and give a result of the *same* type. The input `expr` must thus be of the same type as the result. A more general way to express this would be to specify the argument type to be a subtype of the result type. This would require both an addition of a type constraint and a change to alleviate the restriction caused by the constructors. We have not designed a syntax for this in this project, but we will discuss the possibility in Sections 5.5.4 and 5.6.4.

One may also wonder if we could replace the type variable with a category name, and use that name as a form of polymorphic type similarly to how the category name was used in the piece declaration. Such an approach would be similar to the extensible data types in Open Data Types (Section 2.7) and Open Abstract Types (2.8), where the data type is referred to on a type level directly by name. However, this approach does not remove the need for piece or function constraints. In particular, using a category as a polymorphic type would not introduce a variable on which restrictions can be put. Using categories in place of type variables would mean that there would be no distinction between different compositions of the same category. As a result, this approach would limit the expressiveness in the same way as with extensible data types, as seen in the approaches of Open Data Types and Open Abstract Types.

With this constraint, we can also express the inferred types of piece constructors. As mentioned in Section 3.3, the constructors do not just construct the pieces themselves, but some possible composed type containing the pieces. For the constructors of the expression language example, their types would be as follows:

```
Const :: (Const partof e) => Int -> e
Add :: (Op partof e) => e -> e -> e
Mul :: (Op partof e) => e -> e -> e
Neg :: (Neg partof e) => e -> e
```

3.5.3 Category Constraints

We now have constraints for expressing necessary extensible functions and pieces for a polymorphic composed type. The third part of our syntax that could be expressed as a constraint is the categories, to express that a composed type belongs to a certain category. Looking back at the example in the previous section, we have the following function:

```
mulTwo :: (Const partof e, Op partof e) => e -> e
mulTwo expr = expr `Mul` Const 2
```

A category constraint could then be used here to express an extra guarantee that the composition of the pieces `Const` and `Op` are defined to belong to a certain category, namely `ExprCat`. In fact, such a constraint would then also indicate that the two pieces belong to the category, as the composition, defined for a certain category, should only be possible to form with pieces of said category.

We write the constraint as usual in the context of a type, with the following syntax:

CategoryConstraint: *Category* `==>` *Type*

Note that the *Type* here is referring to a composition, not a piece.[9] In the example, this constraint would be included as follows:

```
mulTwo :: (ExprCat ==> e, Const partof e, Op partof e) => e -> e
mulTwo expr = expr `Mul` Const 2
```

Here, `e` is a type variable representing a composition, which is restricted through one category constraint and two piece constraints.

As the pieces are already restricted to a certain category, this constraint is implied for any type that uses piece constraints. This means that a category constraint here does not provide anything more than a notation of expected behavior. The situation is the same when a category constraint is paired with function constraints, since extensible functions are also defined for a certain category. However, when having extensible functions that also construct a composed type, a category constraint may be useful, as we will see in Section 3.6.2.

[9]While we use the symbol `==>` to indicate category belonging both for pieces and for compositions (see the syntax for the piece declaration and the composed type in Table 3.1), it could be sensible to differentiate the two via different symbols. With such a differentiation, the use of the symbol in the category constraint would more clearly indicate a constraint on a *composition*, not on a piece.

3.5.4 Summary of Syntax for Constraints

The syntax for the three constraints we have defined in the above three subsections is summarized in Table 3.3.

FunctionConstraint	*Type* `with` *FunctionName*
PieceConstraint	*PieceRef* `partof` *Type*
CategoryConstraint	*Category* `==>` *Type*

Table 3.3: *Our proposed syntax for constraints on composable types. These are to be written in the context of a type.*

3.6 Polymorphic Extensible Functions

We have now introduced the possibility of constraints on composed types in regular functions, by adding them to the context of a type, in particular in a type signature. We would also like to have such constraints, as well as regular type constraints such as `Show a`, in extensible functions. For instance, we want to be able to write a polymorphic evaluation function, with a type variable for the return type. This would mean a declaration like the following with a constraint specifying the return type to be numeric:

```
evalNum -: (Num e) => ExprCat -> e
```

This shows that we want to add the possibility of having a context in our function declaration.

Another example is that we want to be able to write a desugaring function. This would mean that we want to write extensions like the following, similar to the instances given for `Desug` in Data Types à la Carte and `compdata`:

```
ext (Const partof a, Op partof a) => desug ...
```

We do not show the full implementation here since this includes syntax that is not yet introduced, but we note the need for a possibility of having constraints in function extensions as well. Having constraints in function declarations is covered in the next section, and having constraints in extensions is covered in Section 3.6.2.

3.6.1 Polymorphic Function Declarations

For the declaration, we look back at the example of `evalNum`:

```
evalNum -: (Num e) => ExprCat -> e
```

As seen here, we want to have the possibility for constraints written in a context of a declaration of an extensible function, similar to how constraints are written for regular functions. This context could contain both regular type constraints, such as `Num e` above, and constraints on composed type, such as `Const partof e`. This gives us the following updated syntax for function declarations:

CompFunDecl: *FunctionName* `-:` [*Context* `=>`] *Category* `->` *Type*

Next, we want to make sure that such constraints and type variables written in the function declaration are usable when defining extensions to that function. For instance, we want to write extensions to `evalNum` given above. One may then want to write extensions for a specific numeric type, for instance to evaluate to an `Int`, which would allow for ad-hoc polymorphism over the variable `e` in the declaration. This means that we want to, in a function extension, specify the type of a type variable in the function declaration, with some way to relate the two. This correspondence between the type variable in the declaration and the type in the extension is done with inspiration from explicit type applications from the `TypeApplications` extension [25], by indicating the type in the extension with the symbol `@`, as `@`*Type*. We will further on refer to this simply as *type applications*. In the example, this would look like the following:

```
ext evalNum @Int for Const where
    evalNum (Const i) = i
```

We could then also define a separate extension for another type, for example `Double`, as follows:

```
ext evalNum @Double for Const where
    evalNum (Const i) = fromIntegral i
```

Then, we can write similar extensions for operations:

```
ext evalNum @Int for Op where
    evalNum (Add e1 e2) = evalNum e1 + evalNum e2
    evalNum (Mul e1 e2) = evalNum e1 * evalNum e2

ext evalNum @Double for Op where
    evalNum (Add e1 e2) = evalNum e1 + evalNum e2
    evalNum (Mul e1 e2) = evalNum e1 * evalNum e2
```

It would be desirable to write this in a polymorphic manner instead, to only have one extension for operations, which can then be used no matter the type to which the constants are evaluated. To do so, we need a constraint in the extension specifying the return type to be numeric, just as in the declaration. We will come back to this in the next section.

We could also have several type applications in the same extension, each corresponding to a variable in the declaration. Consider a function with the following type:

```
evalNum2 :: (Num e1, Num e2) => ExprCat -> e1 -> e2
```

We would then have two type applications in the extensions, like the following:

```
ext evalNum2 @Double @Int for Const where
    ...
```

In concordance with the previous use of explicit type applications, we make them correspond in order of appearance, meaning that the first type application refers to the first variable in the declaration, and so on. This means that `@Double` above instantiates the variable `e1`, and `@Int` instantiates `e2`.

In general, the syntax for function extensions now looks like the following:

CompFunExt: `ext` *FunctionName* {*TypeApp*} `for` *PieceRef* `where`
{*FunctionName* *PieceConPat* {*Pattern*} `=` *FunctionDef*}

Here, each *TypeApp* is a type application given by `@`*Type* and if having multiple, they are separated by space.

3.6.2 Polymorphic Function Extensions

In the example of `evalNum` from the previous section, we noted a need for a constraint in the extension for `Op` in order to make it polymorphic in the return type. We achieve this by allowing contexts in function extensions, and allowing a type application `@`*Type* to contain a type variable, not only an instantiated type like `Int`. For the `Op` extension, this looks like the following:

```
ext (Num n) => evalNum @n for Op where
    evalNum (Add e1 e2) = evalNum e1 + evalNum e2
    evalNum (Mul e1 e2) = evalNum e1 * evalNum e2
```

Here, the variable `n` refers to the variable `e` in the declaration:

```
evalNum -: (Num e) => ExprCat -> e
```

This means that we are able to, through type applications, relate type variables written in an extension to type variables in its function declaration.

In general, we can now state our syntax for function extensions as follows:

CompFunExt: `ext` [*Context* `=>`] *FunctionName* {*TypeApp*} `for` *PieceRef* `where`
{*FunctionName* *PieceConPat* {*Pattern*} `=` *FunctionDef*}

Each *TypeApp* is given by `@`*Type* and, if having multiple, they are separated by space.

3.6.3 Updated Function Constraints

Another situation where we want to refer to type variables written in a function declaration is in the function constraints introduced in Section 3.5.1. For instance, consider that we want to write a function that uses `evalNum` defined in Section 3.6.1, but with a specified result type, for instance `Int`. As before, we write this with a function constraint to specify that `evalNum` can be used on the input argument. This means that we want a function like the following:

```
evalNumInt :: (e1 with evalNum) => e1 -> Int
evalNumInt = evalNum
```

Since `evalNum` contains a type variable, we need to write an instantiation of the type variable in the function constraint here. Similar to how it was done in function extensions, we do this with type applications. This gives the following type:

```
evalNumInt :: (e1 with evalNum @Int) => e1 -> Int
```

Now, `@Int` instantiates the variable `e` in `evalNum`.

This means a change of the syntax for the function constraint to also possibly include a type application, as follows:

FunctionConstraint: *Type* `with` *FunctionName* {`@`*TypeApp*}

As before, each *TypeApp* is given by `@`*Type* and if having multiple, they are separated by space.

In the example above, the *Type* is a concrete type, `Int`, while it could also be a type variable. For instance, we could write a similar function for real numbers as:

```
evalNumReal :: (e1 with evalNum @n, Real n) => e1 -> n
evalNumReal = evalNum
```

3.6.4 Transformative Extensible Functions

An important example of where polymorphic function extensions are needed is in transformative extensible functions, functions that both deconstruct an argument of a composed type and construct a value of a composed type as its result. This could be a desugaring function, as presented in Section 2.1, which removes some syntactic sugar, in this case negation, from an expression. This would be a function that takes an expression of category `ExprCat` as input, and returns an expression, still of category `ExprCat`, but of a composed type that does not contain the piece `Neg`. We specify this result type in the function declaration by use of a type variable and a category constraint, which gives the following declaration:

```
desug -: (ExprCat ==> e) => ExprCat -> e
```

The input argument is given by the category `ExprCat` in the same way as in a destructive extensible function, since that is the argument that is specified to be pattern matched over and decomposed in the function.[10] This is why we do *not* express the type of `desug` as:

```
desug -: (ExprCat ==> e1, ExprCat ==> e2) => e1 -> e2
```

However, it could be possible to extend the syntax to allow a user to write the type like that as well.

To be more specific about the result type, one could want to express it so that it is the input type without the part that was removed via the desugaring function. In Variations on Variants (Section 2.4), this is expressed as a type class denoted by `(:-:)`, while this is not something that is implemented in Data Types à la Carte nor in `compdata`. We have not designed a syntax for this in this project, but we will discuss it in Section 5.6.4.

Note, however, that the ability to be specific about the result type at all, and in particular to make a distinction between input and output, is only possible with composable types. With extensible data types, such as in Open Data Types and Open Abstract Types (Sections 2.7–2.8), the input and output types would be the same, and cannot be anything other than the extensible data type, `Expr`. As a result, those solutions are less expressive, even if they could be considered somewhat simpler in their syntax.

We continue to how this would be used in the function extension for the case of negation. An expression containing a negation, `Neg expr`, would be desugared to ``Const (-1) `Mul` expr``. This means that the pieces `Const` and `Op` must be part of the result type of the desugaring function. This is indicated by two piece constraints in the extension, `Const partof a` and `Op partof a`. Furthermore, we have a type variable for the result type in the declaration of `desug`, which we refer to using a type application in the extension, as follows:

```
ext (Const partof a, Op partof a) => desug @a for Neg where
    desug (Neg e) = Const (-1) `Mul` (desug e)
```

The type application `@a` connects the type variable `a` here to the variable `e` in the declaration.

Similar instances are added for the other pieces, `Const` and `Op`, which simply leave the operation as it is, and recursively call `desug` in the subexpressions. These also need piece constraints to indicate that those pieces are contained in the resulting composed type. Thus, we have the following extensions:

[10]This means that we specify transformative functions in our syntax to dispatch on the input rather than the output, similar to destructive functions. One could consider other possibilities, such as dispatching on the combination of input and output, or on the output alone. Dispatching on the output is discussed regarding constructive extensible functions in Section 5.2.

```
ext (Const partof a) => desug @a for Const where
    desug (Const i) = Const i

ext (Op partof a) => desug @a for Op where
    desug (Add e1 e2) = Add (desug e1) (desug e2)
    desug (Mul e1 e2) = Mul (desug e1) (desug e2)
```

As mentioned, the type variable `a` in the extensions, describing piece constraints for each extension, corresponds to the type variable `e` in the category constraint in the declaration. This means that a piece constraint in an extension should only be fulfilled for one of the intended pieces, that is, a piece in the defined category. In this way, one could ensure that the result of each extension is in the desired category. This is the primary situation in which a category constraint could possibly express something beyond the other two forms of constraints; function constraints and piece constraints.

As a final note, the desugaring function is just one example of a transformative extensible function. One could also have a function where the input and output types are of different categories. For instance, one could have a function like the following, transforming an expression into a value of a composed type of a new category called `Text`:

```
transform -: (Text ==> a) => Expr -> a
```

This function would then have extensions in a similar manner as `desug` above, but constructing values of a composed type of category `Text`.

3.6.5 Summary of Syntax for Extensible Functions

With the parts in this section, we have syntax for extensible functions that works for both destructive and transformative functions, with a possibility to add type constraints in a context, both in declarations and extensions. We also have an updated function constraint, which can now handle type variables. We summarize the proposed syntax for extensible functions and function constraints in Table 3.4.

CompFunDecl	*FunctionName* `-:` [*Context* `=>`] *Category* `->` *Type*
CompFunExt	`ext` [*Context* `=>`] *FunctionName* {*TypeApp*} `for` *PieceRef* `where` {*FunctionName* *PieceConPat* {*Pattern*} `=` *FunctionDef*}
Function-Constraint	*Type* `with` *FunctionName* {`@`*TypeApp*}
TypeApp	`@`*Type*
PieceRef	*PieceName* {*Type*}
PieceConPat	*PieceCon* {*Pattern*}

Table 3.4: *Our final proposed syntax for extensible functions, containing a function declaration and a function extension. The updated function constraint is also included. Some syntactic components of the three are also specified.*

Finally, we also present a full example of a transformative function, `desug`. We show this in a module `Desug` that imports the modules `Expr` and `Negation` from Sections 3.2.6 and 3.4.

```
module Desug where
import Expr
import Negation

-- | Transformative function desug
desug -: (ExprCat ==> e) => ExprCat -> e

-- | Desug extension for constants
ext (Const partof a) => desug @a for Const where
    desug (Const i) = Const i

-- | Desug extension for operations
ext (Op partof a) => desug @a for Op where
    desug (Add e1 e2) = Add (desug e1) (desug e2)
    desug (Mul e1 e2) = Mul (desug e1) (desug e2)

-- | Desug extension for negation
ext (Const partof a, Op partof a) => desug @a for Neg where
    desug (Neg e) = Const (-1) `Mul` (desug e)
```

4 Transformation

We have implemented a proof-of-concept transformation, which transforms code that uses the syntax that we have formulated in Chapter 3 into code that uses an existing solution to the expression problem. By targeting an existing solution, we are able to focus our work on the syntax, without needing to produce an implementation for composable data types from the ground up. We have chosen to use the `compdata` library by Bahr and Hvitved [13], since it provides a thorough implementation of their improved Data Types à la Carte. Furthermore, the semantics are similar between our syntax and `compdata`, which means that the transformation is rather straightforward in many cases.

Our transformation is primarily syntax-driven, which means that it is decided mostly based on the syntactic components. This is implemented using a library called `haskell-src-exts` [26], which provides parsing and pretty-printing for Haskell code as well as several extensions. The choice of `haskell-src-exts` specifically means that we can easily generate output code that uses Template Haskell and other extensions that are important when using `compdata`. By extending this parser to handle our syntax, we can parse code written in our syntax and standard Haskell code to an abstract syntax tree. Our transformation program then transforms this tree into a new syntax tree that no longer uses our syntax and is instead built on `compdata`. This transformed tree can then be printed, using the pretty-printer, to code that can be compiled as a regular Haskell program.

While we have a mostly syntax-driven transformation, we also want to detect some compilation errors at transformation time. In particular, the detection of these errors requires name resolution. An example is checking that when forming a composed type, the pieces to be included can be identified, and belong to the same category, in particular, to the category specified in the declaration of the composed type. Name resolution is also required to be able to distinguish between regular data type constructors and constructors belonging to a data type piece. We get name resolution with the help of a library called `haskell-names` [27], which is used to annotate the initial syntax tree with the result of a name resolution. This annotated syntax tree is then used to check that each piece is in scope and to find out in which category a piece was declared. With a proper way of handling name resolution, we can easily and correctly identify references to pieces, categories, and extensible functions in order to give more accurate error messages if any of these are out of scope.

While we can catch some errors already in the transformation phase, errors arising when compiling the transformed code will refer to the transformed code rather than the code written in our syntax. This means that error messages arising from the transformed code are difficult to understand for a user of our syntax, without knowledge of the transformation or backend. In this project, we have not focused on alleviating this problem, since the transformation is only meant to be proof-of-concept.

In addition to the transformation program, and the extension of the two above mentioned libraries, we have implemented some test cases to check that the transformation works as expected.[1]

The following sections will describe the transformation from code that uses our syntax into code that uses standard Haskell and `compdata`, and will show the correspondence between different syntactic components in the syntax and in the backend. An example of a transformation of a complete program is given in Appendix B.2.

4.1 Avoiding Name Conflicts

The transformation introduces some new names for variables, classes, and other symbols that were not present before the transformation. There are also symbols that are imported from `compdata`, such as `:+:` and `liftSum`. Because of these new and imported names, conflicts may arise with symbols introduced by a user of our syntax. It is also possible that a user uses one of these symbols by accident.

It is straightforward to implement techniques to significantly reduce the impacts of this problem. Names generated by the transformation can be made unusually long, and to have an identifying name, such as `composable_types_recursive_var`. While this is not a fool-proof solution, it makes it unlikely for name conflicts to occur. To ensure that symbols from `compdata` do not overlap with other names, `compdata` can be imported as a qualified module, so that symbols from `compdata` are used for instance as `Data.Comp.liftSum`.

Both of these techniques cause the generated code to become more verbose. This is not a concern, since the generated code is not meant to be read by a user. However, because of the added verbosity, the resulting names from these two techniques will not be seen in the following sections of this thesis report, to increase the readability of examples of generated code. The techniques are still implemented in our proof-of-concept transformation.

[1]The code for the versions of `haskell-src-exts` and `haskell-names` that have been modified to support our syntax, as well as for the complete transformation and its test suite, is available as a project on Github, at: `https://github.com/kirderf1/composable-types`

4.2 Data Types and Categories

Our syntax for data types consists of three parts: category declarations, piece declarations, and composed types. We will show the transformation of these here, using the example of the expression language from previous chapters.

Piece categories are only a tool for expressing components in our syntax and to add compile-time limitations for pieces and composed types. In fact, these categories do not serve a purpose after the initial transformation to standard Haskell. As a consequence, category declarations such as `piececategory ExprCat` need only be read into the transformation phase and then removed from the transformed program. In the transformation phase, on the other hand, the category is used to perform checks, such as checking that the category exists when trying to define pieces belonging to it.

Next, we have the data type pieces, written in our syntax for instance as:

```
data piece ExprCat ==> Op = Add ExprCat ExprCat
                          | Mul ExprCat ExprCat
```

Data type pieces are very similar to regular data types, and we can represent them as such. To represent the recursive type, the transformed data types get a generic variable, just as in `compdata`. Doing so means that a piece is transformed into a data type with a type constructor with one argument for the generic variable, which is also used as the recursive parameter. The piece above is then transformed as follows[2]:

```
data Op a = Add a a | Mul a a
```

The name of the variable `a` here is arbitrary.

Finally, we have the composed types, like `ExprCat ==> (Const | Op)`. The data types in `compdata` are combined through a coproduct, and then closed with a `Term`. Since we want to use the composed type directly in our syntax, it corresponds to the closed type, that is, a `Term` applied to a coproduct over all the pieces included in the composition. Hence, the composed type is transformed into the following:

```
Term (Const :+: Op)
```

As a composed type in our syntax represents a closed composed type in `compdata`, our syntax hides the concept of an open coproduct. We have also chosen to represent the composed type as a list of pieces, which is transformed into a list formed by

[2]To distinguish the transformed code from code written in our syntax and other code snippets, we show the transformed code in this report in a box with a gray background like this.

coproducts.[3] Since the coproduct is a binary operator, we are restricted to represent the composition like an ordered list rather than an unordered set. As a result, equality between composed types only holds if they contain the same pieces *in the same order*. We will come back to this to consider alternative ways of representing composed types in Section 5.4.1.

Data type pieces with additional type variables will, in the transformed code, have their type variables together with the added recursive type variable. To make it easy to transform piece references, the original type variables will remain as the leftmost type variables, while the recursive type variable is added at the rightmost end. The example `data piece ExprCat ==> ConstP c = ConstP c` from Section 3.2 is then transformed into:

```
data ConstP c a = ConstP c
```

The piece constructor with a variable is here a straightforward transformation into a data constructor with the same variable. A piece reference specifying the type variable is similarly transformed to use the data constructor instead of the piece constructor. Hence, the composed type `ExprCat ==> (ConstP Int | Op)` is transformed into:

```
Term (ConstP Int :+: Op)
```

4.3 Constructors and Using the Data Types

An example expression can be created using our syntax as:

```
constTwo = Const 2
```

To specify its type, one can either define it to be of a fixed composed type, like `constTwo :: ExprCat ==> (Const)`, or a parameterized type using constraints, like `constTwo :: Const partof e => e`. The transformation of the composed type was given in the previous section. Now, we will discuss how to transform the constructor, `Const`, which will also cover the transformation of the piece constraint in the parameterized type.

As mentioned in Section 3.3, we want our constructors to function as the smart constructors in Data Types à la Carte and `compdata`, that is, by constructing a value of some composed type containing the necessary pieces. Thus, we want to use the smart constructors of `compdata` when transforming our constructors.

The type of the smart constructor for constants is the following in `compdata`:

[3] We have restricted the composition to be represented as a list, rather than in a nested way like `((`$PieceName_1$` | `$PieceName_2$`) | (`$PieceName_3$` | `$PieceName_4$`))`. This nested composition would still only be a list-like coproduct wrapped in a `Term` in the transformation. When only forming the coproduct from a specific composed type, rather than potentially from a combination of compositions, this is straightforward. However, see the discussion on combining compositions in Section 5.4.2.

```
iConst :: (Const :<: f) => Int -> Term f
```

The corresponding constructor in our syntax has the following inferred type:

```
Const :: (Const partof e) => Int -> e
```

We can see that our piece constraint is closely corresponding to the subsumption constraint of `compdata`, as also mentioned in Section 3.5.2. However, it has one main difference: While the composition in the subsumption is a coproduct, `f`, which is closed only in the type as `Term f`, we have the closed composition directly parameterized as `e`. This means that we cannot directly transform the piece constraint `Const partof e` into a subsumption constraint `Const :<: f`, and consequently not the constructor `Const` into a smart constructor `iConst`. In the next section, we will see how this can be alleviated, before defining our own smart constructors in Section 4.5.

4.4 Parameterized Compositions

The problem indicated in the previous section arises in all cases where we use a type constraint on a composition, since we parameterize compositions as simply a type variable, like `e`, while in `compdata`, it is wrapped in a `Term`, like `Term f`.

One may first try to resolve this simply by detecting type variables for composed types and, when transforming those, wrap them as `Term e` from a variable `e`. This detection can be made using the constraints on composed types, which indicate which variables represent compositions. However, doing so changes the meaning and kind of the type variable `e`, which is difficult to handle in all possible cases. This is especially difficult when only one of two linked type variables should be changed, such as in a type class and an instance.

For instance, consider the following type class and instance written using our syntax:

```
class Transform a b where
    transform :: a -> b
instance Const partof b => Transform Int b where
    transform i = ...
```

Here, we have only one constraint, in the instance declaration. In this example, we would wrap `b` into `Term b` in the transformation of the instance. On the other hand, consider the following case where we have two constraints, one in the class declaration and one in the instance:

```
class (b with eval) => Transform a b where
    transform :: a -> b
instance Const partof b => Transform Int b where
    transform i = ...
```

Now, we only want to wrap one of the variables `b` in a `Term`. Transforming `b` into `Term b` in the class declaration means that the meaning and kind of the variable `b` is changed there. This means that the variable `b` in the instance declaration already corresponds to the `Term b` in the class, and should thus not be wrapped itself. Therefore, it is challenging to correctly wrap the variables only where necessary.

Due to this difficulty, we opt for another approach. Instead of changing the type variables, we leave them be and bridge the gap between the constraints in our syntax and the ones in `compdata` through introducing new type classes and instances to use in the transformed code. For instance, we can write a type class for the piece constraint as follows, using the class for subsumption and its function `inject`:

```
class PartOf f e where
    inject' :: f e -> e
instance f :<: g => PartOf f (Term g) where
    inject' = inject
```

This means that we can transform our piece constraint `f partof e` into `PartOf f e`, and use that instead of a constraint using subsumption `f :<: g`, where `e` represents a closed composed type rather than an open coproduct. We will use a similar approach with another type class in the transformation of functions in Section 4.6.

4.5 Smart Constructors

With the type class `PartOf` defined in the previous section, we can now define the smart constructors. These will be generated in the transformation of each piece. Data Types à la Carte, and in turn `compdata`, defines smart constructors through subsumption, like the following for `Const`:

```
iConst :: (Const :<: f) => Int -> Term f
iConst i = inject (Const i)
```

The result type here is `Term f`, a variable wrapped in a `Term`. The function `inject` is defined in `compdata`. By using the intermediary class `PartOf` instead of `:<:`, we define the smart constructor instead as the following:

```
iConst :: (PartOf Const e) => Int -> e
iConst i = inject' (Const i)
```

This will be generated in the transformation of the piece `Const`. Now, it is possible to use this smart constructor for parameterized types without explicitly wrapping them on the form `Term f`, although the wrapping in `Term` is still an implicit requirement through the class `PartOf`. We also use the function `inject'`, as defined in the class `PartOf`, to serve the same purpose as `inject` defined in `:<:`.

As opposed to `Const`, which takes a constant as its argument, `Add` takes values of the recursive composed type, which we specify to be of the same type as the result type `e`. This gives the following smart constructor, generated in the transformation of the piece `Op`:

```
iAdd :: (PartOf Op e) => e -> e -> e
iAdd e1 e2 = inject' (Add e1 e2)
```

This means that we can now transform an example like the following from Section 3.5.2:

```
twoMulThreePlusFive :: (Const partof e, Op partof e) => e
twoMulThreePlusFive = Const 2 `Mul` (Const 3 `Add` Const 5)
```

This is transformed into the following, using our smart constructors:

```
twoMulThreePlusFive :: (PartOf Const e, PartOf Op e) => e
twoMulThreePlusFive = iConst 2 `iMul` (iConst 3 `iAdd` iConst 5)
```

Note that, since we use `compdata`, we have the same limitation as that approach regarding explicit type annotations. This means that, for instance, when evaluating `twoMulThreePlusFive`, the compiler would not be able to resolve the evaluation due to ambiguities.[4] To be able to evaluate the example above, one needs to specify its type to a concrete composition, like `ExprCat ==> (Const | Op)` in our syntax, or `Term (Const :+: Op)` in the transformed code.

4.6 Destructive Functions

As an extensible function in `compdata` is represented as a type class with instances for each variant of the data type, we want to transform our syntax for extensible functions into type classes and instances. We show it here for a destructive extensible function, `eval`.

First, we have the function declaration:

```
eval -: ExprCat -> Int
```

This defines the type of the extensible function, and acts as the base for any future extensions for specific pieces. This corresponds to the function classes in `compdata`, which looks like the following for the evaluation example given in Section 2.3:

```
class Functor f => Eval f where
    evalAlg :: f Int -> Int
```

The type variable `f` here represents a functor such as a coproduct or a specific piece. In this `compdata` example, the function is written as an algebra, as seen in the type signature `evalAlg :: f Int -> Int`. This means that the recursive value is

[4]See Section 2.2 for the explanation about such explicit type annotations to resolve ambiguities.

evaluated at each step to an `Int`. As stated in Section 3.4, we chose to instead have functions in our syntax written without algebras. Hence, we need a different recursive type, namely some composed type. In particular, it could be any composed type which the extensible function can take as an argument. For `Eval`, we write this as follows:

```
class Eval f where
    eval' :: Eval g => f (Term g) -> Int
```

We then want to use `eval'` in the actual evaluation function, `eval`, just as `evalAlg` is used in the `compdata` example. In the class definition for `Eval`, we assume that the evaluation function `eval` can accept any `Term g` where `Eval g` is fulfilled. We will see that this holds as we continue by defining `eval`. The newly introduced `eval'` accepts a value of a coproduct, that is, a type not closed by `Term`. Meanwhile, `eval` is meant to accept a value of a closed composed type. This means that `eval` can be implemented as `eval' . unTerm`, where `unTerm` is a function in `compdata` that simply unwraps a value from a `Term`. In `compdata`, `eval` is defined exactly as such:

```
eval :: Eval g => Term g -> Int
eval = eval' . unTerm
```

However, doing so in our transformation would lead to the parameterization problem discussed in Section 4.4, as the function here is parameterized on the open coproduct rather than on the closed composed type. Instead, we define `eval` inside a type class that acts as an intermediary class:

```
class Eval_outer t where
    eval :: t -> Int

instance Eval g => Eval_outer (Term g) where
    eval = eval' . unTerm
```

By introducing this class and instance, we can use the outer type class `Eval_outer` in a constraint and thus not need to change any type variables by wrapping them in a `Term`.

The final part to be generated when transforming the declaration of an extensible function is an instance for the coproduct, as with Data Types à la Carte and `compdata`. Just like with `compdata`, this is derived using Template Haskell, in particular using `liftSum` defined in `compdata`. Thus, a line like the following is generated for each extensible function declaration:

```
$(derive [liftSum] [''Eval])
```

Now, we can conclude that the declaration of an extensible function `eval` is transformed into:

- the class `Eval` with its function `eval'`
- the outer class `Eval_outer` containing the actual evaluation function `eval`
- the Template Haskell derivation for the coproduct instance of the class `Eval`

Next, we have the function extensions for the pieces, written for `eval` in our syntax as:

```
ext eval for Const where
    eval (Const i) = i

ext eval for Op where
    eval (Add e1 e2) = eval e1 + eval e2
    eval (Mul e1 e2) = eval e1 * eval e2
```

These correspond, as mentioned, to type class instances to the class `Eval`. This is then a straightforward transformation into the following:

```
instance Eval Const where
    eval' (Const i) = i

instance Eval Op where
    eval' (Add e1 e2) = eval e1 + eval e2
    eval' (Mul e1 e2) = eval e1 * eval e2
```

The function name `eval` is changed into the class name `Eval` in the instance head, along with the name of the piece which takes the place of the variable to the class. Note also the change from `eval` into the class function `eval'` specifically on the left-hand side of the equations.

4.7 Constraints on Composed Types

The constraints we presented in Section 3.5 are transformed into equivalent regular type constraints, written in a context. In each of the following sections, we cover one of those forms of constraints.

4.7.1 Function Constraints

First, we have the function constraint, that a certain function must be supported by a composed type. An example using this was presented in Section 3.5.1 as:

```
evalFalse :: e with evalCond => e -> Int
```

This constraint corresponds in `compdata` to a class constraint, as follows:

```
evalFalse :: EvalCond e => Term e -> Int
```

However, as discussed in Section 4.4, we avoid a transformation of the type variable by instead using an intermediary class, which in this case is the outer class for extensible functions introduced in Section 4.6. This gives the following transformation:

```
evalFalse :: EvalCond_outer e => e -> Int
```

By using the class `EvalCond_outer` rather than `EvalCond`, the type variable does not need to be wrapped in a `Term` as it is in `compdata`.

4.7.2 Piece Constraints

Second, we have the piece constraint, to restrict a composed type to contain certain pieces, which corresponds to the subsumption constraint in `compdata`. For instance, it can be used in the following example from Section 3.5.2:

```
mulTwo :: (Const partof e, Op partof e) => e -> e
```

As mentioned in Section 4.4, a piece constraint is transformed into using an intermediate class, `PartOf`, instead of subsumption directly. This means that the example above is transformed into the following:

```
mulTwo :: (PartOf Const e, PartOf Op e) => e -> e
```

Note that the class `PartOf`, defined in Section 4.4, is not specific to any piece or composition, unlike other code introduced by the transformation. Rather, the same class should be used by any piece constraint in any module. To do so, this class and its instance are defined in a separate library module, and an import statement is generated in each transformed module to import this library module.

4.7.3 Category Constraints

Third, we have the category constraint, that a composed type is of a certain category. This does not have any correspondence in the transformed code written in regular Haskell and `compdata`. We could give it a counterpart by defining yet another type class, with instances for each category, and then transform the constraint into a constraint that the variable should be part of said type class. Then we could use it to set limitations, for instance, on the result type of a transformative function, as mentioned in Section 3.6.2. We have not implemented this in this project, so as of now, a category constraint is simply transformed into an empty context, without any additional checks in the transformation phase. This means that the current state of the implementation does not put any restrictions on type variables based on a category constraint. We discuss the possibility of implementing a more meaningful transformation of a category constraint in Section 5.1.

4.8 Polymorphic Extensible Functions and Type Applications

For polymorphic functions, we may have a context with constraints in a function declaration, for instance like the following from Section 3.6.1:

```
evalNum -: (Num e) => ExprCat -> e
```

There is then both a context and a variable that need to be accounted for in the transformation. The context is simply added to the two class declarations that correspond to the function. Any type variables are gathered in the order that they first appear, and these variables then correspond to type variables in the two aforementioned type classes. In Section 4.6, we introduced a type variable `f` corresponding to the ingoing type of the function, such as a piece or a coproduct. This means that the transformation of `evalNum` needs both the variable `f` as before and the variable `e` as defined in the declaration. We have defined the transformation to have the variable for the ingoing type first, which gives the following transformation for the declaration of `evalNum`:

```
class Num e => EvalNum f e where
    evalNum' :: (EvalNum g e) => f (Term g) -> e

class Num e => EvalNum_outer t e where
    evalNum :: t -> e

instance EvalNum g e => EvalNum_outer (Term g) e where
    evalNum = evalNum' . unTerm

$(derive [liftSum] [''EvalNum])
```

Note that we do not need to make any changes to the Template Haskell derivation of the coproduct when adding new type variables to the class; this is handled through the function `liftSum`.

To then have an extension refer to the type variable written in the declaration, we took inspiration from type applications, as seen in the following extension to `evalNum`:

```
ext evalNum @Int for Const where
    evalNum (Const i) = i
```

Here, `@Int` corresponds to `e` in the declaration. In the transformed code, this extension becomes:

```
instance EvalNum Const Int where
    evalNum (Const i) = i
```

The type `Int` from the type application takes the place of the second variable to the type class, corresponding to the variable `e` in the class definition shown above.

Type applications are also used in function constraints. For example, consider the following example from Section 3.6.3:

```
evalNumInt :: (e1 with evalNum @Int) => e1 -> Int
evalNumInt = evalNum
```

Just as in the transformation of the function extension, the `Int` from the type application here will take the place of the second variable to the type class, resulting in the following:

```
evalNumInt :: (EvalNum_outer e1 Int) => e1 -> Int
evalNumInt = evalNum
```

We are also able to write constraints in function extensions. For instance, say that we have the following declaration for the transformative function `desug`, defined in Section 3.6.4:

```
desug -: (ExprCat ==> e) => ExprCat -> e
```

We can then have the following extension, where the variable `a` in the type application corresponds to the variable `e` in the declaration:

```
ext (Const partof a, Op partof a) => desug @a for Neg where
    desug (Neg e) = Const (-1) `Mul` (desug e)
```

In the transformation, the context here becomes the context for the corresponding instance, and the type application is transformed into a parameter in the corresponding type class, just as with `@Int` above. This means that the function extension above is transformed into:

```
instance (PartOf Const a, PartOf Op a) => Desug Neg a where
    desug' (Neg e) = iConst (-1) `iMul` (desug e)
```

With this, we have covered the transformation of all the syntactic components described in Section 3. A complete example of a transformation is given in Appendix B.2.

5

Future Work

In this chapter, we present various scenarios and potential use cases of composable data types that are not covered by our syntax and transformation implementation. We discuss how these could be integrated into our syntax and transformation, and where potential difficulties may arise.

5.1 Category Constraints

In Section 3.5.3, we presented syntax for a category constraint, which would state that a polymorphic composition belongs to a certain category. We have not implemented this in the transformation, as mentioned in Section 4.7.3. In this section, we revisit the constraint, and discuss how it could be fully implemented.

An example using the category constraint is the `desug` function, given in Section 3.6.4 with the following declaration:

```
desug -: (ExprCat ==> e) => ExprCat -> e
```

The category constraint `ExprCat ==> e` is meant to restrict the result type `e` to be of category `ExprCat`. In particular, this is meant to restrict the result types of extensions defined for this function. As mentioned, this is not an enforcement that we have implemented in the transformation, and the constraint is currently just transformed into an empty context.

To implement an actual enforcement for this constraint, it might be possible to use a similar strategy as for the other constraints, by introducing a type class; see Section 4.4. Just as the type class `PartOf` for the piece constraint, we could define this type class in a library module. The type class could be defined like the following:

```
class InCat e c
```

This would indicate that `e` is of category `c`. For this to correspond to a category constraint, the type variable `e` should represent a closed composed type rather than an open coproduct, since the constraint is on a composed type. Such a composed type can only be created for a certain category if all composed pieces belong to said category. From this, it would be sensible to relate the category constraint class above to a constraint indicating that pieces belong to a certain category. We indicate this with another type class as follows:

```
class InCat_open (f :: * -> *) c
```

The kind signature `(f :: * -> *)` is used since we want this to be defined for pieces, which have that particular kind. This means that we would define instances for the pieces defined in a certain category, for example as:

```
instance InCat_open Const ExprCat

instance InCat_open Op ExprCat
```

The category `ExprCat` here would be defined simply as an empty data type, as follows:

```
data ExprCat
```

Furthermore, it would be sensible to have an instance for the coproduct, like the following:

```
instance (InCat_open f c, InCat_open g c)
    => InCat_open (f :+: g) c
```

This indicates that a coproduct belongs to the category if both of its summands do. To make the connection between `InCat` and `InCat_open`, we define an instance of `InCat` as follows, indicating that the closed constraint is fulfilled for `Term f` if the open constraint is fulfilled for `f`:

```
instance InCat_open f c => InCat (Term f) c
```

Note here the similarity between this and our definition of the intermediary class `PartOf`, which has a similar instance connecting `PartOf`, defined for closed compositions, to the subsumption class, defined for open coproducts. This instance, the instance for the coproduct and both type classes would be included in the library module mentioned above.

The type class `InCat` would then be used when transforming a category constraint, which would result in a constraint like `(InCat e ExprCat)`. For instance, the function `desug` above would be transformed into the following:

```
class (InCat e ExprCat) => Desug f e where
    desug' :: (Desug g e) => f (Term g) ->  e

class (InCat e ExprCat) => Desug_outer t e where
    desug :: t -> e

instance Desug g e => Desug_outer (Term g) e where
    desug = desug' . unTerm

$(derive [liftSum] [''Desug])
```

We also want the category constraint in the declaration of `desug` to restrict the result type in the extensions, which are written using piece constraints. For instance, we have an extension written in our syntax as follows:

```
ext (Const partof a) => desug @a for Const where
    desug (Const i) = Const i
```

In the transformed code, this would look like:

```
instance (PartOf Const a) => Desug Const a where
    desug' (Const i) = iConst i
```

This would not compile, since the compiler cannot deduce `InCat a ExprCat` from just the piece constraint in the instance. However, this deduction is exactly what we want, since it should already be true in practice. If a piece within a certain category is used in a composition, we already require all other pieces, and thus the whole composed type, to also belong to said category. This is also applicable for function constraints, as each extensible function already states the category within which it accepts pieces.

As it is defined so far, a category constraint can only be deduced when all pieces in the composed type are known, as it is only then that the instances to `InCat_open` can be applied. We could consider writing additional instances that state a relation between the constraints directly, such as the following:

```
instance (PartOf f a, InCat_open f c) => InCat a c
instance Eval_outer a => InCat a ExprCat
```

However, these instances are not accepted by a compiler, in part because of the overlap between instances. Another approach may be to modify the function constraints to add the category constraint as a superclass:

```
class InCat a ExprCat => Eval_outer a where
    ...
```

This approach is, however, not immediately applicable to piece constraints, as the category depends on the piece used in the constraint rather than the constraint itself.

In conclusion, we can state that, while deducing a category constraint from a different constraint is difficult to implement, it is still possible to implement just the introduction of type classes for the category constraint as above without this extra constraint deduction.

5.2 Constructive Extensible Functions

In the syntax presented in Chapter 3, we provide support for destructive and transformative extensible functions. What we have not covered is constructive extensible functions, that is, functions that build a composed type from a simple type, and can be extended over additional pieces similarly to the other kinds of extensible functions. An example of such a function could be a parser function that builds a composed type from a `String`. For instance, it could build an expression of the type we have in the running example. This would preferably have a function declaration like the following:

```
parseExpr -: String -> ExprCat
```

Unlike destructive and transformative functions, which are extended for variants of the first argument, constructive functions would be extended for variants of the return type. With the previous extensible functions, only one extension is used for each recursion step, which is chosen to match the variant of the composition that the function takes as its first argument; that is, it chooses an extension based on the input type. The same approach cannot be used for constructive functions, as there is no longer a composed type as a first argument that determines the extension to be used, so the extension would rather be chosen based on the input *value*.

Without a clear way of choosing an extension to be used, one might instead use all extensions to then somehow combine their results. One such approach is evaluation that can fail, such as by producing a result of the `Maybe` type. When calling the extensible function, each extension could be evaluated in order, and the first successful result would be the result for the extensible function. The question is then in which order the extensions should be evaluated. If constructive extensions are linked to pieces in the same way as the original extensions are linked to destructive functions, the order could be made to be the same as the order of pieces in the composed type that is being constructed. Another alternative is to compose constructive functions in the same manner that extensible functions are composed in Variations on Variants in Section 2.4, where the order could be determined by the order of extensions in the composed function.

5.3 Inferences

In some aspects, the syntax is more verbose than necessary. For one, the syntax may require several parts that need to correspond to each other, even if one part may be inferred by another. The category in the syntax for compositions is one such example, as it could be inferred from the pieces included in the composition. In this section, we aim at relaxing some rules to optionally allow a simpler syntax. For some of these additions, a type-aware transformation would be required to implement the addition.

5.3.1 Inferring a Category in Compositions

Consider a composed type such as the following:

```
ExprCat ==> (Const | Op)
```

The category that is used in these types exists for the user to state the intended category, which is then used to check that all pieces in the composition belong to that category. The most important thing, however, is that all the pieces belong to the same category, not that it is any specific category. If two pieces belong to different categories, that should be detected even without stating to which category they are expected to belong. This means that it could be possible to remove the need for writing an intended category, and have the transformation only check whether the pieces belong to the same category. By making the intended category optional, one could write compositions in a simpler manner, as follows:

```
(Const | Op)
```

This syntax would be easy to parse correctly, except when writing a composition of a single piece. There would then be no syntactic distinction between the piece `Const` and the composition consisting of just `Const`, written with an explicit category as `ExprCat ==> Const`. There is, however, a semantic difference, since the piece is open, and the composition is closed, which in the transformation to `compdata` means that the latter is wrapped in a `Term`. This could be resolved by forcing single-piece compositions to have an intended category, or by letting them having a certain syntax. For instance, one could use a syntax like the following:

```
(Const |)
```

This is inspired by single-item tuples in Python [28], written with a trailing comma, as in the following example:

```
singleTuple = 123,
```

A special syntax would be preferred in order to make the syntax similar for single- and multi-piece compositions, while having a clear distinction between a piece and a composition.

5.3.2 Inferring a Piece in Function Extensions

Just as the type system of Haskell can infer the type of bindings under most circumstances, it could be possible to infer the piece over which an extensible function is extended. As an example, consider the following function extension:

```
ext eval for Const where
    eval (Const i) = i
```

Here, the use of the piece `Const` could be inferred in the transformation phase from the pattern with `Const`. This means that one can make it possible to shorten the syntax of the extension to:

```
ext eval where
    eval (Const i) = i
```

Note that this is a suggestion for syntactic sugar, meaning that it would be optional to include the piece or not. The user should still be able to include the piece to make the code clearer.

From here, it could also be possible to simplify an extension with a single case into a single line:

```
ext eval (Const i) = i
```

A multi-constructor piece, on the other hand, has several cases in its extension. This is what the extension for the operations piece, `Op`, would look like without the specified piece:

```
ext eval where
    eval (Add e1 e2) = eval e1 + eval e2
    eval (Mul e1 e2) = eval e2 * eval e2
```

To have this with a single-line syntax, one would need to write an extension declaration for each of the constructors, as follows:

```
ext eval (Add e1 e2) = eval e1 + eval e2
ext eval (Mul e1 e2) = eval e1 * eval e2
```

Since these belong to the same piece, they should belong to the same type class instance when transformed. This means that these rows are not independent from each other, and must be kept together. As a result, allowing this syntax makes the connection between variants that belong to the same piece obscured, and some confusion may arise. Therefore, the by-piece extension syntax is to be preferred for multi-constructor pieces. Thus, the possibility to have multi-constructor pieces limits the simplicity somewhat in this case, while it broadens the expressive power of pieces.

5.3.3 Composition Defaulting

As mentioned in the section on Data Types à la Carte, Section 2.2, solutions based on that have a limitation in that explicit type annotations are needed in some places to resolve ambiguities arising from the coproduct. This is the case for `compdata`, our backend, as well, which means that this is also a limitation of our implementation.

As an example, we can write an expression as follows, with constraints specifying that the pieces `Const` and `Op` are included:

```
expr :: (Const partof a, Op partof a) => a
expr = Const 1 `Add` (Const 2 `Mult` Const 2)
```

When we want to evaluate such an expression, we must specify it to have a fixed composed type, like the following:

```
evalEx :: Int
evalEx = eval (expr :: ExprCat ==> (Const | Op))
```

This is because the compiler cannot otherwise deduce whether the composition is `ExprCat ==> (Const | Op)` or `ExprCat ==> (Op | Const)`, or some even larger composition. Consequently, the compiler cannot deduce the internal structure of the example needed to evaluate it.

A way of resolving such ambiguities is discussed in Variations on Variants (Section 2.4). The proposed solution is to include a defaulting mechanism in the compiler, similar to the mechanism that allows `z = show 1` to type despite the ambiguity of the type of `1`. This is, however, a proposed addition to the compiler, while we will examine here the possibility of solving this in our transformation phase.

As the ambiguity problem is a limitation of `compdata`, it could be possible to include a defaulting mechanism in the transformation, which would infer the composition by defaulting to one of the possible ways to resolve the constraints on the composition. To do so, it would require using a type system to identify which constraints are put on the composition, and then assume the ambiguous type to be the smallest composition that fulfills all constraints. In the example above, this would infer a composition containing the pieces `Const` and `Op`, which could be either `ExprCat ==> (Const | Op)` or `ExprCat ==> (Op | Const)`. The order is not relevant in this example, but an order must be chosen through the defaulting mechanism.

Now, let us consider `evalEx` again, without the fixed composed type:

```
evalEx :: Int
evalEx = eval expr
```

With a defaulting mechanism as described above, this would now be transformed into the following, given that the order was chosen as `ExprCat ==> (Const | Op)`:

```
evalEx :: Int
evalEx = eval (expr :: Term (Const :+: Op))
```

Inferring a composition in the transformation phase would only work in specific circumstances, where the behavior of the program is not altered by the presence or absence of a piece in the composed type. This means that a composed type should not be inferred in a constructive function, to give one example. Therefore, a restriction has to be made for when the composition can be inferred. This would require a type-aware transformation, which could deduce where to infer the composition and insert the type annotation. It is difficult to say how easy it would be to implement this defaulting mechanism in practice.

5.4 Composition Representation

Composed types currently behave as lists of pieces, which have a direct correspondence in composed types in `compdata`. For two of these composed types to match, the order of the pieces must also match. In this section, we consider changing composed types to instead behave as sets, and how this could lay the foundation for a union operator for composed types.

5.4.1 Equivalence Between Composed Types

As mentioned in Section 4.2, composed types are represented as a list of pieces, where for two composed types to match, both the pieces and their order need to match. This means that the following example would give a compile-time error since the types do not match:

```
expr :: ExprCat ==> (Const | Op)
eval :: ExprCat ==> (Op | Const) -> Int
evalEx = eval expr
```

However, the syntax we have presented so far makes no meaningful use of the indexing or ordering of pieces provided by representing composed types like this. Piece constructors are used in the same way regardless of the position of the piece in the composed type, as long as the piece exists somewhere in the composed type. This is also true for function extensions, which are written by piece in no given order. We can compare this to variants of a regular data type, where the order in which the variants are presented does not affect the behavior of the data type. An idea is then to change the composed types so that they are considered equal as long as they contain the same pieces, even if the pieces are ordered differently. Doing so would remove the type error in the example above.

To achieve this, the transformation of a composed type into a fixed coproduct needs to order all pieces in a predictable manner. As a predictable order, one may first think of ordering them alphabetically. For this to work across modules, it would have to be alphabetically over the full, qualified names of the pieces. This is easy to do when each piece reference consists just of a piece name, as in the example above, but it is not as easy with polymorphic pieces. Piece references for polymorphic pieces where all applied types are fixed can be sorted by the name of the first type, followed by the next one, and so on, but that still does not cover all possibilities. Consider the following parameterized type:

```
type ExprP a = ExprCat ==> (ConstP () | ConstP a)
```

When transforming this type, we would sort the pieces by the name of the piece first. As we then have two `ConstP`, we would sort those two based on their first type argument. However, since we do not know which type `a` is, we cannot reliably tell if it should be sorted before or after `()`. Both cases could be true by using different types for `a`.

In order to fully handle equality between composed types, we must consider duplicate pieces in a composition, in addition to ordering of pieces. As of now, this is more of a theoretical problem, since it is not straightforward why it would be practically useful to allow duplicate pieces in a composition declaration at all. On the other hand, handling duplicate pieces will be useful in practice in the next section, where we discuss the possibility of a union operator to combine composed types, potentially containing some overlap in pieces.

We have so far not defined how such duplicates are handled. With the transformation that we have described so far, any duplicates will simply be left as they are, and remain in the transformed coproduct. These duplicates will there give a run-time error when using the subsumption constraint for a piece on the left-hand side of the constraint that exists multiple times in the coproduct on the right-hand side. This is due to a change to the subsumption constraint in `compdata` that disallows such ambiguities. [12]

For our syntax, we have two suggestions on how to explicitly handle duplicate pieces. First, these duplicates could be removed. However, the choice of which piece to remove would make a difference, unless we also sort the pieces as described above, which would render the positioning of duplicated pieces meaningless. The second suggestion is to disallow duplicates completely, and give an error during the transformation phase if any duplicates are detected. Either suggestion is difficult with polymorphic pieces for the same reason as with ordering pieces. In the example with `ExprP` above, we cannot tell if `a` will equate to `()` or not, and thus not whether they are duplicates.

Even if we solve the problem of polymorphic pieces, there would be a drawback to implementing this: By removing the distinction between composed types with different ordering of pieces, we would also remove the possibility for any syntax extensions to make use of the order of pieces for something meaningful. In particular, an order could be useful for constructive functions, as discussed in Section 5.2.

Instead of removing the distinction between compositions based on ordering of pieces, it could be possible to equate composed types that only differ in ordering through using an isomorphism constraint. This constraint is defined in `compdata` as `type f :=: g = (f :<: g, g :<: f)`. This only works for ground types, that is, types with no variables, just as the subsumption constraint. As a result, one needs to give a fixed type for the constraint to be fulfilled. By introducing a similar constraint in our syntax, we could give the possibility to a user for expressing equality between types that only differ in order of pieces.

5.4.2 Union of Composed Types

With the syntax we presented in Chapter 3, one cannot directly extend a composed type, but must write a new composed type containing all the desired pieces. In particular, one must specify all the pieces again, and cannot reuse the old composed type in any way. For instance, consider the type `ExprComp`:

```
type ExprComp = ExprCat ==> (Const | Op)
```

Next, we want to define a new composed type, which extends this type to also contain `Neg`. To do so, we must specify all pieces again as:

```
type ExprCompWithNeg = ExprCat ==> (Const | Op | Neg)
```

To make this simpler, we consider the possibility of adding syntactic sugar that can allow reusing a composed type when defining a new composition, to not have to specify all pieces again. For the example here, we could do so through defining a new operator that combines a composed type and a piece, as follows:

```
type ExprCompWithNeg = ExprComp `extendWith` Neg
```

This operator behaves similarly to the `:+:` operator in `compdata`, but we cannot transform our operator to use `compdata` directly, like this:

```
type ExprCompWithNeg = Term (ExprComp :+: Neg)
```

This will fail because the type `ExprComp` is a closed composition, and not an open composition. Because our syntax lacks a representation of the open coproduct, it is not possible for a user of our syntax to define the coproduct of `ExprComp` separately and use that one instead. Before addressing this problem, we look at a more general operator that also has this problem.

This general operator is a union type operator that would combine two composed types. More specifically, it would return a composed type of the union of the pieces in the two composed types to which it is applied. This means that one could combine two composed types without having to write all the pieces to be included. An example of how this could be written is the following, where `Combined` would contain the pieces `Const`, `Op` and `Neg`:

```
type ExprComp = ExprCat ==> (Const | Op)
type NegComp = ExprCat ==> (Neg)
type Combined = ExprComp +&+ NegComp
```

If we consider using the `:+:` operator to combine the two types, we once more encounter the problem that we cannot use closed composed types in place of coproducts. Our suggestion to solve this problem is to let the transformation look at the two types that are being combined, figure out which pieces they contain, and then transform the use of the union operation directly into a composed type that is built from the same pieces. In the example above, this would mean that, in order to transform `ExprComp +&+ NegComp`, the transformation would first look at the type information for `ExprComp`, see that it is a composition, and get the list of pieces `Const` and `Op`. The transformation would then look at `NegComp` to find the piece `Neg`. These pieces would then be combined to transform the type into:

```
type Combined = Term (Const :+: Op :+: Neg)
```

With this approach, the union operator would behave just as the `:+:` operator in regards to the pieces present in the result, but operating on composed types rather than coproducts. However, since the pieces are obtained from the two composed types being combined during the transformation, we can only do so for composed types that are fixed to a specific composition. Because of this, we cannot apply the union operator to any type that uses type variables, such as in this example:

```
either :: a -> b -> (a +&+ b)
either = ...
```

This is a limitation in the expressiveness of our operator compared to coproducts, but we do not believe that this is a significant drawback. Regardless, we can use this approach also for the above proposed operator that combines a composed type and a piece.

Next, we recall the discussion of the problem of duplicates pieces from the previous section. This problem is relevant for both coproducts and the union operator. To illustrate it, we consider this slightly modified example:

```
type ExprComp = ExprCat ==> (Const | Op)
type NegConstComp = ExprCat ==> (Neg | Const)
type Combined = ExprComp +&+ NegConstComp
```

This combined type will then be transformed into the following:

```
type Combined = Term (Const :+: Op :+: Neg :+: Const)
```

Since `compdata` disallows duplicate pieces, this combination would not be allowed. While this would be no problem when combining disjoint composed types, combining composed types with any shared pieces would not be possible. To allow the union operator to work for these types as well, and thus be an actual *union* operator, we would need to remove any duplicate pieces as described in the previous section. Thus, we consider doing so, together with sorting of pieces. The type `Combined` would then instead be transformed into:

```
type Combined = Term (Const :+: Neg :+: Op)
```

5.5 Function Alternatives

With the syntax we have designed, one can write extensible functions for various kinds of situations. For some kinds of operations, writing them as extensible functions may be excessive. We could then want to provide built-in alternatives to some of these, which could lead to less repeated code or other simplifications. We explore some of these alternatives in this section. The important thing to note here is that these are meant to be simplifications of operations that, to some degree, are *already possible*, but over-complicated, in our syntax.

5.5.1 Pattern Matching on Composed Types

Writing extensible functions over a data type is suitable when providing cases for a growing number of data type pieces. There may be situations where it would be more suitable to write a function in a simpler way, more like a regular function. This could be a function that is not meant to be extended, or perhaps that handles new pieces with a default case. In that case, one could possibly define the function by writing cases through pattern matching instead of writing function extensions for each piece. As this would be written as regular functions rather than extensions, it would not be possible to extend the function later on without recompilation.

As an example, we consider a closed version of the evaluation function, `eval`, written as an extensible function as follows:

```
eval -: ExprCat -> Int
ext eval for Const where
    eval (Const i) = i
ext eval for Op where
    eval (Add e1 e2) = eval e1 + eval e2
    eval (Mul e1 e2) = eval e1 * eval e2
```

If the function is meant only for a fixed composition of a few cases, one may instead want to write the function as a regular, closed function, as follows:

```
evalClosed :: ExprCat ==> (Const | Op) -> Int
evalClosed (Const i)   = i
evalClosed (Add e1 e2) = evalClosed e1 + evalClosed e2
evalClosed (Mul e1 e2) = evalClosed e1 * evalClosed e2
```

As long as the composed type is fixed, we can transform these patterns directly to use `In`, `Inl` and `Inr` depending on where each piece is placed in the composed type. This would then look like the corresponding closed function in Data Types à la Carte in Section 2.2, as follows:

```
evalClosed :: Term (Const :+: Op) -> Int
evalClosed (In (Inl (Const i)))   = i
evalClosed (In (Inr (Add e1 e2))) = evalClosed e1 + evalClosed e2
evalClosed (In (Inr (Mul e1 e2))) = evalClosed e1 * evalClosed e2
```

This is possible as long as the pattern has a fixed composed type that can be processed during the transformation phase. Knowing the fixed type composition, a transformation could observe the order of its pieces, from which it can determine the correct composition of `Inl` and `Inr` patterns. To implement this, the transformation needs to be type-aware, to be able to correctly observe the types of the pieces and the composition and use that to give the correct patterns in the transformed code.

However, this is not possible for a polymorphic type, which does not have a known composition of its pieces. For instance, consider the following function:

```
evalIntClosed :: (Const partof a) => a -> Int
evalIntClosed (Const i) = i
evalIntClosed _         = ...
```

Here, the transformation cannot determine the correct composition of `Inl` and `Inr`. Instead, it might be possible to transform the function to use projection from `compdata`, to then split on the cases `Just (Const i)` and `Nothing`.

To do so, we need to modify our intermediary class `PartOf` to make the projection function available. We can do this by introducing `project'` as our version of `project` in `compdata`, just as `inject'` is our version of `inject`. This gives the following type class and instance:

```
class PartOf f e where
    inject' :: f e -> e
    project' :: e -> Maybe (f e)
instance f :<: g => PartOf f (Term g) where
    inject' = inject
    project' = project
```

To then use this projection function when transforming the `evalIntClosed` example above, we could transform the function as follows, using an auxiliary function that pattern matches over the result of `project'`:

```
evalIntClosed :: (PartOf Const a) => a -> Int
evalIntClosed a = evalIntClosed' (project' a)

evalIntClosed' :: (PartOf Const a) => Maybe (Const a) -> Int
evalIntClosed' (Just (Const i)) = i
evalIntClosed' Nothing          = ...
```

This can be further simplified through the use of the `ViewPatterns` extension in Haskell. With this, we could instead use view patterns to match against the result of `project'` directly, simplifying the transformation and giving the following:

```
evalIntClosed :: (PartOf Const a) => a -> Int
evalIntClosed (project' -> Just (Const i)) = i
evalIntClosed _ = ...
```

5.5.2 Derivations on Pieces

For composable data types to be more versatile, it is desirable to be able to use type class derivations on them. For example, one could want to derive a `Show` instance. As the pieces are what correspond to data types in the transformed code, we will primarily consider putting derivations on those. This would be written as follows:

```
data piece ExprCat ==> Const = Const Int
    deriving Show
```

Note that this could be done without derivations in our syntax, through defining an extensible function for a `show` function, and then use that in an instance of the `Show` class. To make it simpler to use, however, it is desirable to be able to write a derivation statement as above.

In `compdata`, such a derivation corresponds to a Template Haskell splice like the following:

```
$(derive [makeShowF] [''Const])
```

This derives an instance `ShowF f`, which is defined in `compdata` to give rise to an instance `Show (Term f)`. [29] Derivations of several other instances, such as `EqF` and `OrdF` are defined in a similar manner in `compdata`, while others may require an extension to be able to derive them in this way.

Note that to be able to print an expression, one would need to have a `Show` instance for all the included pieces. It could be possible to allow a syntax where one could put a derivation statement on a category or a fixed composed type, which would be transformed by putting that derivation on every included piece.

However, putting it on a category would restrict the types of pieces that could be defined in that category. For instance, if we put a derivation of `Show` on a category, it would not be possible to define pieces in that category which have a function type as a field to any of its constructors, for which a `Show` instance cannot be derived. Therefore, it is more suitable to require the derivations to be written on each piece, and not on a category.

Putting it on a composed type is not sensible either, since we can have different compositions containing the same piece, and a derivation for a piece would automatically derive the instance in all uses of the piece, including other compositions. Thus, one has to make sure that there is only one derivation splice, even if multiple compositions containing the same piece have such a derivation written for them.

5.5.3 Decomposing Multiple Composed Types

The extensible function examples `eval` and `desug` given in Sections 3.4 and 3.6.2 each decompose a single composed type, which is what is possible with the syntax and implementation so far. It could be possible to have extensible functions that decompose not just one, but multiple composed types. An example could be a function that in some way combines two expressions. For instance, it could be a function with a declaration like the following:

```
combine -: ExprCat -> ExprCat -> Int
```

This is a destructive function, but one could possibly write a transformative function in a similar manner. Moreover, this combines values of the same category, `ExprCat`, but one could also have other examples that combine values of two different categories, such as `ExprCat` and `OtherCat`.

It is possible to achieve this to some extent in the syntax we have so far by chaining extensible functions, so that each extension to the first extensible function will have its own extensible function that handles the second composed type. However, this is verbose, as seen in the following example:

```
combine -: ExprCat -> c -> Int

ext (c with combineConst) => combine @c for Const where
    combine (Const i) c = combineConst c i

combineConst -: ExprCat -> Int -> Int

ext combineConst for Op where
    combineConst (Add e1 e2) i = ...
    combineConst (Mul e1 e2) i = ...
```

The function `combine` here has an extension for each piece of the first argument to be decomposed. From this extension, a second extensible function is called, which handles all the pieces of the second argument to be decomposed. In that way, all combinations of pieces are covered through multiple extensible functions. Note that this is only possible when we can dispatch on the first type first, and then the second, as opposed to on both of them together.

For a less verbose syntax, we could consider writing an extensible function that can decompose two arguments in the same extension. This means that we could instead dispatch on the combination of the two arguments. The declaration for the `combine` function could then look like the following instead:

```
combine -: ExprCat -> ExprCat -> Int
```

Note the two categories in the type, denoting that those arguments are to be decomposed through this function. Then, we could write extensions like the following, combining in this case constants and operations:

```
ext combine for (Const, Op) where
    combine (Const i) (Add e1 e2) = ...
    combine (Const i) (Mul e1 e2) = ...
```

Note here the two pieces in the extension, written as `(Const, Op)`. We would need a similar such extension for each combination of pieces from the two argument types. This would thus still require much code for large sets of pieces. The amount of code could be lessened through an ability to cover more than one piece in a single extension, that is, to allow polymorphic extensions. We will discuss this possibility in Section 5.6.2.

This would then be transformed into a class with one variable for each of the arguments to be decomposed. In particular, the function `combine` above with type `ExprCat -> ExprCat -> Int` would be transformed into a class `Combine f g`, like the following:

```
class Combine f g where
    combine' :: Combine h i => f (Term h) -> g (Term i) -> Int
```

From this class, we can point out a significant problem regarding recursive calls. With a type class written in this manner, it is impossible for an extension to impose further constraints on `h` and `i`. While this difficulty is also present for extensible functions with a single decomposing argument, it is a more notable problem here. We see this in the consequence that `Term h` can only be combined with `Term i`. If, in an extension, we were to combine a value of `Term h` with the result of the constructor `Const`, a compiler would be unable to ensure that this combination is possible, as the extension cannot add the constraint `Combine h Const` or `Const :<: i`.

Regardless of this unsolved problem, we move on to look at a different problem, which involves the coproduct instances for this type class. These instances are normally generated using `liftSum` from `compdata`. However, this only works for classes operating on a single type, and not those decomposing more than one type. As a result, an important step to achieve this feature would be to implement an altered version of `liftSum`.

For the function combine above, which decomposes two types, we need to have type class instances for when either of the arguments is a coproduct. In particular, we need three instances: one where both are coproducts, and two where one of them is a coproduct but the other is not. This would probably look like the following:

```haskell
instance
    (Combine f h, Combine f i, Combine g h, Combine g i)
    => Combine (f :+: g) (h :+: i) where
        combine' (Inl a) (Inl c) = combine' a c
        ...

instance {-# OVERLAPPABLE #-}
    (Combine f h, Combine g h) => Combine (f :+: g) h where
        ...

instance {-# OVERLAPPABLE #-}
    (Combine f g, Combine f h) => Combine f (g :+: h) where
        ...
```

The implementations not shown are analogous to the one for `combine'` in the first instance. Note the use of the pragma `{-# OVERLAPPABLE #-}`, indicating that the instances are allowed to overlap. In this case, we can see that the instance where both arguments are coproducts is also covered by the other two instances. Using the pragma means that this can be resolved without errors, so that the first, more specific, instance would be used if both arguments are coproducts. For a function that decomposes even more arguments, the number of instances for the coproduct would grow exponentially.

In summary, it is possible to some degree to implement functions that decompose multiple arguments. As mentioned, however, it has a limitation on the kinds of recursive calls that can be made.

5.5.4 Composition Conversion

Recall the example shown in Sections 3.3 and 3.5.2. We begin with an expression with a single constant that has been given the concrete type `ExprCat ==> (Const)`, as follows:

```haskell
constTwo :: ExprCat ==> (Const)
constTwo = Const 2
```

We want to reuse this in a larger expression, which also contains operations. This larger example can then be defined to have the type `ExprCat ==> (Const | Op)`, and be implemented as the following:

```haskell
twoMulThreePlusFive :: ExprCat ==> (Const | Op)
twoMulThreePlusFive = constTwo `Mul` (Const 3 `Add` Const 5)
```

This is not possible, since the use of the constructor `Mul` requires both its operands, as well as the result, to be of the same type. This restriction could be relaxed, which we will come back to shortly. In this example, the restriction means that it would require `constTwo` to have the type `ExprCat ==> (Const | Op)` as well. Although this simple example can be resolved using polymorphic types via piece constraints, as seen in Section 3.5.2, it can be useful to be able to reuse examples with a fixed composed type in larger examples. To be able to do this, one needs a conversion from one composed type to another. In particular, it should be possible to convert a composed type into a composition of which it is a subtype, that is, a type containing, at least, all the pieces included in the original composed type. In this section, we discuss how such a conversion could be implemented in our solution, while we discuss how subtyping could be expressed as a type constraint in Section 5.6.4.

Conversion from a composition to a larger composition could be done either implicitly when trying to use a value of one composed type as another type, or explicitly using some function. Note that there is no implicit type conversion in standard Haskell, so that approach is not as Haskell-like. However, one kind of implicit conversion can be implemented within Haskell itself. It could be achieved by implementing smart constructors with more general types, such as the following, written using constraints from `compdata`:

```haskell
iAdd :: (Add :<: f3, f1 :<: f3, f2 :<: f3) =>
    Term f1 -> Term f2 -> Term f3
```

To write such smart constructors with our constraints and intermediary classes, we need a constraint for stating a composed type as a subtype of another; see Section 5.6.4. This is to have a constraint matching the subsumption constraint in the example above where the left-hand side is a coproduct, while our syntax currently only has a constraint corresponding to a subsumption constraint with a single data type on the left-hand side. In addition, the additional type variables would give problems with ambiguous type variables, whose possible solutions are discussed in Section 5.3.3. With these solved, one could relax the restriction that the arguments and the result of a constructor must be of the same type. This means that `Mul` in `twoMulThreePlusFive` above would no longer require both operands to have the same type as the result, so we could write ``constTwo `Mul` (...)`` directly without the typing problem we had above.

Next, we consider the possibility of explicit conversions. An explicit conversion via a function that converts one composition to a larger one can be written in our syntax. This would be written as an extensible function, and would require one extension for each piece. For expressions, it could look like the following:

```
convert -: ExprCat -> e

ext (Const partof e) => convert @e for Const where
    convert (Const i) = Const i

ext (Op partof e) => convert @e for Op where
    convert (Add e1 e2) = Add (convert e1) (convert e2)
    convert (Mul e1 e2) = Mul (convert e1) (convert e2)
```

This function can then be used in `twoMulThreePlusFive` as follows:

```
twoMulThreePlusFive :: ExprCat ==> (Const | Op)
twoMulThreePlusFive = (convert constTwo) `Mul`
                      (Const 3 `Add` Const 5)
```

We can consider simplifying this for a user, so that they would not need to write such a conversion function with all its extensions for every category and piece themselves. A design choice that has to be made here is if the user should still be required to implement the function themselves, even if we make it simpler to do so, or if the function should be available to a user directly. The function could be made available to a user either through generating it in the transformation phase or through having it in a library module, similar to `PartOf`. Making a function available in this way may be unitutive for a user, since understanding how the function works requires understanding of the backend where it is implemented.

If we choose to generate the function through the transformation, we need to generate a type class corresponding to the `convert` function above, like the following:

```
class Convert f a where
    convert' :: (Convert g a) => f (Term g) -> Term a

convert :: (Convert f a) => Term f -> Term a
convert = convert' . unTerm

instance (Const :<: a) => Convert Const a where
    convert' (Const c) = iConst c

instance (Op :<: a) => Convert Op a where
    convert' (Add e1 e2) = iAdd (convert e1) (convert e2)
    convert' (Mul e1 e2) = iAdd (convert e1) (convert e2)

$(derive [liftSum] [''Convert])
```

Note that this is independent of the category used for the input argument, as long as we do not insert any category constraints[1]. As a result, we could write extensions

[1]How to insert category constraints in the transformed code is not yet implemented. See Section 5.1 for a discussion on how this could be done.

to this function regardless of the category. However, we would like the categories to restrict which pieces are to be used together, so we would want a different conversion function for each category. To do so, we could generate the class `Convert` along with the function `convert` and the deriving for the coproduct instance when transforming a category declaration, and then generate the extensions when transforming the corresponding piece declarations. This means that, with this approach, we can restrict conversions to be possible only within a category using the different `Convert` classes.

In order to have the function in a library module, we need to make it more general, to remove the uses of concrete pieces. This means that we want to write an instance of type class `Convert f a` that uses a polymorphic piece as the first variable, `f`, instead of specifying it to be a concrete piece. This instance should then be used for all conversions. To do so, we can take inspiration from the implementation of `desug` in the `compdata` library [16]. There, we have a general instance of the type class `Desug`, used for every piece that should not be desugared, such as `Const` and `Op`. This is written as follows:

```
instance {-# OVERLAPPABLE #-} (f :<: g) => Desug f g where
    desugAlg = inject
```

Note the variable `f` in place of the input argument instead of a concrete piece. This instance uses `inject` as a way to return the input with the appropriate composed type. However, `desug` in `compdata` is written using an algebra, meaning that the instance above only specifies how the function handles that specific step of the recursion, and the recursion is then carried out through a folding function, as follows:

```
desug :: (Functor f, Desug f g) => Term f -> Term g
desug = cata desugAlg
```

The function `cata` is a function in `compdata` which is equivalent to the function `foldTerm` from Data Types à la Carte. In order to use `cata` ourselves, all pieces must have functor instances to be able to fold over the result of each recursion step. We could then want to leave it to the user to implement or derive functor instances for all pieces in order to use the `convert` function on them. However, that is not possible, since the type variable over which a functor instance can be defined is only introduced in the transformation phase. Instead, since that variable is always introduced by the transformation and in the correct place to define a functor instance, we could define functor instances for the pieces as part of the transformation. This can be done by deriving them using the `DeriveFunctor` extension [17], as follows:

```
data Op a = Add a a | Mul a a
    deriving Functor
```

With the pieces declared as functors, we can use `cata`. If we were to follow the example of `desug`, we would declare a class with an algebra function, and an instance just like the one shown above. Unlike `desug`, however, we expect all pieces to be injected. In fact, the subsumption criterion can be used with a coproduct on the left-hand side as well, thanks to improvements done with `compdata` [12]. As a consequence of these two observations, the conversion class does not need a coproduct instance, as any such coproduct could be injected directly through the `inject` instance. In fact, for the same reason, we do not even need a class at all. We can use `inject` as the algebra function to be passed into `cata` and get the `convert` function simply as:

```
convert :: (Functor f, f :<: g) => Term f -> Term g
convert = cata inject
```

If this function were available directly for a user of our syntax, it would be possible to write an explicit conversion just as above:

```
twoMulThreePlusFive :: ExprCat ==> (Const | Op)
twoMulThreePlusFive = (convert constTwo) `Mul`
                      (Const 3 `Add` Const 5)
```

To write similar conversions without stating concrete types for both functions, one would want some kind of constraint indicating that one composed type is a subtype of, and thus can be converted into, another composed type. To do so, we need to introduce an intermediary class for the new constraint for the same reason that we needed to introduce `PartOf` in Section 4.4. We return to this in Section 5.6.4.

Instead of making this library function immediately available to a user, we could define syntax for writing a function `convert` in our syntax similar to the library function. We have no suggestion of this, but it would result in a usage similar to the explicit conversion in `twoMulThreePlusFive`. Recall that we also have the possibility of making the conversion implicit, which would require an insertion of `convert` at the necessary places in the transformation phase.

5.6 Polymorphism in New Ways

Polymorphism is a key concept in Haskell for expressing limitations through types. While our syntax already supports some polymorphism, there are additional situations when polymorphism might be used. We present some of these ideas in this section.

5.6.1 Categories Depending on Categories

When representing an abstract syntax tree of a full language, one can expect it to consist of different data types that in some way are dependent on each other. Let us consider a Haskell-like language with declarations and expressions. We begin with the expression data type used in the running example of this report, and add declarations that bind an expression to a name:

```
data Expr = Const Int | Add Expr Expr | Mul Expr Expr
data Decl = Bind String Expr
```

To this with composable types in our syntax, we could do it as follows:

```
piececategory ExprCat
data piece ExprCat ==> Const = Const Int
data piece ExprCat ==> Op = Add ExprCat ExprCat
                          | Mul ExprCat ExprCat

piececategory DeclCat
data piece DeclCat ==> Bind = Bind String ExprComp

type ExprComp = ExprCat ==> (Const | Op)
type DeclComp = DeclCat ==> (Bind)
```

We write the expression type in the same way as always, but when writing the declaration type, we refer to a fixed composed type for expressions, namely `ExprComp`. In order to let us use different expression compositions, we could try to parameterize the `Bind` piece, like the following:

```
piececategory DeclCat
data piece DeclCat ==> (ExprCat ==> e) => Bind e = Bind String e

type DeclComp = DeclCat ==> (Bind ExprComp)
```

This would be possible in our current syntax.[2]

If we were to continue to add declaration variants that use expressions, we might consider having a way of declaring the expression composition for the declaration composition as a whole, instead of attaching it to every individual piece as we just did with `Bind`. However, this is not necessary in order to express one-way dependencies between types, as we saw with these expressions and declarations.

Instead, let us consider the case of two extensible data types that are codependent. We use the expression data type again and consider a new variant: `do` blocks, which we define similar to `do` blocks in Haskell. Each `do` block is a list of statements, where each statement is either an assignment of an expression to a name, or just an expression. With regular data types, we define this example as follows:

```
data Expr = Const Int | Add Expr Expr | Mul Expr Expr | Do [Stmt]
data Stmt = Assign String Expr | SExpr Expr
```

If we were to write this with composable data types in our syntax in the same way as in the declaration example, we would get the following:

[2]Note the use of a category constraint. While the category constraint has not been fully implemented and is not technically needed here, it shows what sort of type `e` is expected to be.

```
piececategory ExprCat
data piece ExprCat ==> Const = Const Int
data piece ExprCat ==> Op = Add ExprCat ExprCat
                          | Mul ExprCat ExprCat
data piece ExprCat ==> (StmtCat ==> s) => Do s = Do [s]

piececategory StmtCat
data piece StmtCat ==> (ExprCat ==> e)
        => Assign e = Assign String e
data piece StmtCat ==> (ExprCat ==> e) => SExpr e = SExpr e

type ExprComp = ExprCat ==> (Const | Op | Do StmtComp)
type StmtComp = StmtCat ==> (Assign Exprcomp | SExpr ExprComp)
```

The two last rows would then be transformed into:

```
type ExprComp = Term (Const :+: Op :+: Do StmtComp)
type StmtComp = Term (Assign ExprComp :+: SExpr ExprComp)
```

This leads to a compilation error, as Haskell does not permit cycles in type synonyms. Hence, we would need some other way of achieving this effect. A way of solving this may be by closing the recursion with a new data type similar to `Term`. Recall the definition of `Term`, given in Section 2.2:

```
data Term f = In (f (Term f))
```

Here, `f` is an open composition, which has the kind `f :: * -> *`. This is then used to fix the recursive type through applying `f` to the closed composition `Term f`. If we instead try to express two codependently recursive types, we will instead have two open compositions, `f` and `g`, that each needs to be applied to two closed compositions. This means that `f` and `g` would have the kind `f, g :: * -> * -> *`. If we let the last type that gets applied to `f` be the closed composition corresponding to `f`, we get the following modification of `Term`:

```
data Term2 f g = In2 (f (Term2 g f) (Term2 f g))
```

With this, we need to adapt the signature of a number of different parts used in the above example. For example, the transformation of pieces now needs a type variable for each extensible data type among the codependent types. This means that any dependencies between extensible types must be known from the start when defining pieces in order for this approach to work. We will return to this limitation later in this section.

Adapting the piece definitions, the transformation of the pieces above would now be:

```
data Const s e = Const Int
data Op s e = Add e e | Mul e e
data Do s e = Do [s]

data Assign e s = Assign String e
data SExpr e s  = SExpr e
```

All transformed pieces now have type variables to match the two closed compositions, with `e` being the type for expressions and `s` being the type for statements. We ensure that the last type variable refers to the composition to which the piece is expected to belong.

To then combine these pieces, we also need a new coproduct operator that accepts one more type variable than the original coproduct. We can define it as follows:

```
data (f :++: g) d e = Inl2 (f d e)
                    | Inr2 (g d e)
```

With `Term2`, `:++:` and the pieces with extra type variables, the two closed compositions can now be represented in the transformed code as:

```
type ExprComp = Term2 (Const :++: Op :++: Do)
                      (Assign :++: SExpr)
type StmtComp = Term2 (Assign :++: SExpr)
                      (Const :++: Op :++: Do)
```

The number of type variables and their placements will vary depending on how many different types are involved and how they depend on each other. We can therefore not define a single new version of `Term` that is suited for every possibility. What we can do is to let the dependency between types be declared in some form, from which the transformation generates suitable data types and constraints. The closest thing we have to representing an extensible data type or a family of composed types is the category. Thus, we may consider declaring relations between different categories as part of the category declaration, which can then be transformed into data types and constraints adapted for the specific category, such as what we saw with `Term2` and `:++:`.

With a syntax and system for this, we may also extend it for one-directional dependencies described earlier, where the dependency on expressions for the declaration data type now is defined for the composition, rather than an individual piece. Through defining a version of `Term` for declarations that depend on expressions, we would get something like this:

```
data DeclTerm f g = DeclIn (f (Term g) (DeclTerm f g))

type ExprComp = Term (Const :+: Op)
type DeclComp = DeclTerm (Bind) (Const :+: Op)
```

As noted earlier, this approach only works as long as it is known from the start which dependencies an extensible type has to other types. We see this with the type variables in transformed pieces, as well as in the need for different versions of `Term` and `:+:`. The consequence is that any new piece cannot introduce new dependencies of this kind, and any new dependency has to be introduced by changing the definition of the category. This would require recompilation of any pieces and functions, and is a noteworthy limitation with this approach. A different approach that solves this might be possible using closed type families instead of additional type variables, but this is not a possibility we have explored further.

5.6.2 Piece Polymorphism in Function Extensions

When writing function extensions, we must write an extension for each piece. It would be desirable to instead be able to cover several pieces in the same extension. In particular, this could be used for a default case that would be used if there is no specific extension for a certain piece. This is possible with, for instance, `compdata`, as seen in the following default instance of the `desug` function:

```
-- | Default instance of Desug
instance {-# OVERLAPPABLE #-} (f :<: g) => Desug f g where
    desugAlg = inject
```

This simply calls the function `inject`, which returns the input with the appropriate composed type. Note the variable `f`, where we would usually have a concrete type such as `Const`. This instance would then be used for all variants that should not be desugared, while the variants to be desugared would have their own specific instances, like one for `Neg`. The full example of `desug` written in `compdata` is given in Appendix C.2.

We want to achieve a similar thing in our syntax, to allow type variables instead of explicit pieces in functions extensions, and thus have the ability for default cases. In Section 5.5.4, we saw another example where this would be useful, namely to be able to write a composition conversion function in our syntax. In this section, we take a closer look at the `desug` function. In our syntax, it would have the following declaration:

```
desug -: (ExprCat ==> e) => ExprCat -> e
```

To write a default extension, like the instance given with `compdata` above, one would write a type variable in the place of a piece as follows:

```
ext (p partof a) desug @a for p where ...
```

Note the type variable `p` in place of a piece. This would mean a function extension that can be used for any piece that is part of the composition `a`. As before, we also use a type application for the variable `a` representing the composition, so that it corresponds to the variable `e` in the declaration.

To be able to transform this, we look again at the example in `compdata`. The default instance and the instance for `Neg` are written as follows:

```
-- | Default instance of Desug
instance {-# OVERLAPPABLE #-} (f :<: g) => Desug f g where
    desugAlg = inject

-- | Desug instance for negation
instance (Const :<: g, Op :<: g)
        => Desug Neg g where
    desugAlg (Neg e) = iConst (-1) `iMul` e
```

One can easily see that negation can also match the default case. Thus, we have overlapping instances, made possible by the use of the pragma `{-# OVERLAPPABLE #-}`. The compiler is then able to deduce which instance to use when several overlaps, since one is more specific than the other. As a result, the second, more specific, instance is used when handling negation.

Another thing to note here is that the use of `inject` as the default instance for multiple pieces is made possible since the function is written using an algebra. This means that the recursion is not specified in each step, but rather handled through folding over the algebra. Thus, `inject` can be used both for `desug (Const i) = iConst i` and `desug (Add e1 e2) = iAdd e1 e2`. Furthermore, such folding is possible as long as all pieces are functors. In `compdata`, the functor instances are derived through the extension `DeriveFunctor`. For us to have a similar default case using `inject` or similar, we need to make sure that the pieces are functors. We could do this similarly and include such derivation statements in the transformation.

On the other hand, it is possible to achieve the same effect without an algebra as well, by handling the recursion manually through `fmap`. This means that we can write a default instance as the following:

```
-- Default instance
instance {-# OVERLAPPABLE #-} (Functor f, f :<: v)
    => Desug f v where
        desug' = inject . fmap desug
```

Piece polymorphism in our syntax would involve inserting `{-# OVERLAPPABLE #-}` in the transformation where suitable, as well as handling the variable for pieces in extensions and piece constraints. Still, much of what is done with the piece argument in these examples cannot be done directly within our syntax, and would require introducing counterparts in our syntax or other solutions. For instance, it would require some part that would be transformed into using the `inject` function. It may be possible to not introduce any counterpart function in the syntax, and instead, in the transformation, infer where an injection is needed and insert it. Furthermore, the `Functor` class and `fmap` function cannot be used directly, as the type variable `f` in the above example is not present in our syntax, nor does it have an exact equivalent. In conclusion, the transformed code of a polymorphic extension is straightforward, while how to implement it in our syntax is not.

5.6.3 Remainder Constraints

In our syntax, `desug` has the following type signature:

```
desug -: (ExprCat ==> e) => ExprCat -> e
```

We could want to be able to express the composed types here more precisely, to express that the function removes some parts of the type. For instance, we could write a specific `desug` function that is specified to transform expressions containing negations into expressions without negations. Then, we want to express that the result type is the same as the ingoing type, except that it does not contain `Neg`.

In Variations on Variants, this is expressed with a type operator `:-:`, giving `desug` the constraint `f :-: Neg ~ g`, where `Term f` is the ingoing type and `Term g` is the result type (see Appendix C.3). More precisely, this denotes that the ingoing type and the result type have exactly the same pieces, except the `Neg` in the ingoing type. Allowing a similar constraint in our syntax could then mean that we can write a `desug` function with a more specific type signature, as in the following:

```
desugNeg :: (e1 with desug @e2, e1 :-: Neg ~ e2) => e1 -> e2
desugNeg = desug
```

This would restrict the function `desug` so that it cannot contain extensions returning something that includes `Neg` when used with `desugNeg`. In this example, we have chosen to keep the syntax of the remainder operator `:-:`, while it could be changed to another symbol in our syntax. In fact, it is important to note that this constraint in our syntax should be on closed composed types, rather than open coproducts as the constraints in `compdata` and Variations on Variants.

While this may be useful for specific pieces in the case of `desugNeg`, it cannot automatically treat all desugared pieces in this manner. If we introduced a new piece that should be desugared, we would need to modify the type signature of `desugNeg` or write a new version in order to treat it in the same way as `Neg`.

Moving on to how this constraint can be transformed in our syntax, we note that this constraint is not readily available in `compdata`. However, it could be expressed with an analogous constraint using an isomorphism constraint and a coproduct. With those, one can express the type of a desugaring function that explicitly removes `Neg` as in the following, written using `compdata`:

```
desugNeg :: (Desug f g, f :=: g :+: Neg) => Term f -> Term g
desugNeg = desug
```

Note that this, just as with the `:-:` constraint, restricts the argument and result types to contain the same pieces, except for `Neg`. However, since the extension for negation requires `Const` and `Op` to be part of the result type `Term g`, they are required to be included in the argument type `Term f` as well. Thus, the function `desugNeg` above cannot, for instance, be used as a function of type `Term (Const :+: Neg) -> Term (Const :+: Op)`. This can be relaxed through exchanging the isomorphism constraint for a subsumption constraint, as in the following:

```
desugNeg :: (Desug f g, f :<: g :+: Neg) => Term f -> Term g
desugNeg = desug
```

Now, it is possible to use pieces in `g` that do not appear in `f`.

As a final note, this approach does not work directly in our syntax, due to the way our composed types are parameterized, as described in Section 4.4. As the constraint `f :<: g :+: Neg` specifically uses a coproduct, we would want to introduce a new intermediary class and use that to replace the constraint in the transformed code. Doing so might be enough to create a remainder constraint with our parameterization of composed types.

5.6.4 Subtype Constraints

While a remainder constraint discussed in the previous section indicates pieces to be removed, it could be useful to have a more general way of expressing a composed type as being a subset of another composed type. This would be a generalization of the piece constraint to express a composed type as contained in another composed type, rather than a piece as part of a composed type.

For instance, consider the `mulTwo` example from Section 3.5.2:

```
mulTwo :: (Const partof e, Op partof e) => e -> e
mulTwo expr = expr `Mul` Const 2
```

Instead of writing the type as above, forcing the ingoing and outgoing types to be equal, it could be possible to express the ingoing type as being a subtype of the result. The ingoing and outgoing types are restricted to be the same because of how the constructors are defined. In the example above, `Mul` restricts the result to be of the same type as the two operands. This means that the result must have the same type as the argument `expr`. This restriction could possibly be alleviated by composition conversion, described in Section 5.5.4, which could be used to convert one composed type to another. In this section, we assume that one can perform composition conversion in some way, either explicitly or implicitly, and instead focus on how to rewrite the type signature to indicate a subtype relation between input and output. This means that we can, assuming explicit conversion, implement the function as:

```
mulTwo expr = (convert expr) `Mul` Const 2
```

This would then simply be translated into:

```
mulTwo expr = (convert expr) `iMul` iConst 2
```

The function `convert` here is described in Section 5.5.4 as a function converting a composed type to another.

Returning to the type signature, we want to write it like the following:

```
mulTwo :: (e1 subtypeof e2, Const partof e2, Op partof e2)
       => e1 -> e2
```

This would indicate that all the pieces in the argument type `e1` are also included in the result type `e2`. In particular, this constraint could be used as a constraint for when a conversion between types is possible.

In the transformation, a subtype constraint corresponds to subsumption, just as the piece constraint. This is because `compdata` allows coproducts on the left-hand side of a subsumption constraint as well as on the right-hand side [12]. This will be fulfilled if all summands on the left-hand side are part of the coproduct on the right-hand side.

With this, one could want to generalize our piece constraint to also handle composed types on the left-hand side. This is, however, not directly possible, since we hide the open coproduct from the user, and instead write constraints on the closed composed type. This means that there is a difference in kind between a piece and a composed type, and thus also between a piece constraint and a subtype constraint. Therefore, we need a second constraint for subtyping in our syntax and transformation, while it can be implemented using subsumption just as the piece constraint.

In the transformed code, this constraint could be written using a type class similar to `PartOf`. This needs a new version of `inject`, converting a composed type to another composed type, rather than the types being open coproducts. Hence, the new `inject` function would behave as the `convert` function in Section 5.5.4, so we can write the class as follows:

```
class SubtypeOf e1 e2 where
    inject'' :: e1 -> e2

instance (Functor f, f :<: g)
    => SubtypeOf (Term f) (Term g) where
        inject'' = cata inject
```

This would be implemented in a library module just as `PartOf`. The function convert used above would then simply be this inject''. With this, we can transform the type signature of mulTwo above into the following:

```
mulTwo :: (SubtypeOf e1 e2, PartOf Const e2, PartOf Op e2)
          => e1 -> e2
```

6

Discussion

In this chapter, we look back at aspects of our syntax from three main perspectives. First, we refer back to the conditions given in Section 1.2 to confirm that our syntax fulfills all of them. Second, we will see how our solution compares to the related work with regard to the two additional properties covered in Table 2.1 in Section 2.9 that give an overarching picture of how the syntax is structured. Third, we discuss some important aspects of our syntax in further detail and see how they relate to the goals stated for our syntax in Section 1.2.

6.1 Fulfillment of General Conditions

In this section, we cover the conditions given in Section 1.2 and see that our syntax fulfills all of them.

- **A solution to the expression problem:**
  - **Extensibility in both directions:** A data type can be extended by defining a new piece and form a new composition that includes it. It is also possible to add a new case for it to an extensible function. The set of functions is extended simply by defining a new function.
  - **Strong static type safety:** A function in our syntax can only be applied to data type variants that it can handle, since that is the case in the corresponding type classes in the transformation, just as it is fulfilled by our backend, `compdata`.
  - **No modification or duplication:** All extensions are possible without modification or duplication. The introduction of categories may give some minor duplication from the restrictions of pieces belonging to different categories, or not being able to reuse composed types in new compositions. This is to be expected, and the ability to separate pieces and compositions in different categories is one of the reasons to introduce categories in the first place.
  - **Separate compilation:** A piece declaration is only dependent on the category it uses, so defining a new data type variant through a piece declaration does not require any recompilation of existing code. Likewise,

extensions to a function depend only on the declaration and are independent from each other, and can thus be compiled separately without recompilation of previous extensions. Last, functions over an extensible data type are independent of each other and can be compiled separately.

  - **Independent extensibility:** Piece declarations can be written independently in different modules and combined in yet another module, with the only restriction that the piece declaration modules must depend on a module containing the category declaration. Likewise, with a function declaration in a base module, extensions to the function can be written in different modules and combined later.

- **Extension to Haskell:** Our syntax is usable as an extension to standard Haskell through our transformation implementation, which transforms code written in our syntax into compilable code written with standard Haskell and `compdata`.

- **Self-sufficiency:** A user of our syntax does not need to know about or use any components from our backend, `compdata`. At the same time, error messages arising from the transformed code can be difficult to understand without knowledge of the transformation or `compdata`. This is, however, not a result from the syntax itself, but from the proof-of-concept implementation of the transformation.

- **Independent design:** The syntax is designed independently from `compdata`. This means that, while we have taken inspiration from concepts of `compdata` alongside other solutions, the syntax design is not dependent on the fact that we use `compdata` as our backend.

6.2 Properties Regarding Automatic Extension

We also check how our solution compares to the related work regarding the additional two properties presented in Table 2.1 in Section 2.9:

- **Automatic extension of data types:** Our syntax does not have automatic extension of data types. Instead, we have composable data types; that is, one can form many different composed types from a set of pieces.

- **Automatic extension of functions:** Functions are extended automatically in our syntax. This means that an extensible function is usable directly for all variants over which it has an extension.

6.2.1 Lack of Automatic Extension of Data Types

As mentioned, our syntax does not have automatic extension of data types, but instead composable data types. Thus, we can form several compositions from the same set of pieces. For instance, we can form different compositions from the pieces `Const`, `Op` and `Neg` like the following:

```
type ExprComp = ExprCat ==> (Const | Op)
type ExprCompWithNeg = ExprCat ==> (Const | Op | Neg)
```

This means that we can, for instance, write functions specifying exactly the pieces for which they are intended to work. We can do so either via using fixed composed types, or parameterized types using type constraints, as seen in the following examples:

```
desugSimple :: ExprCompWithNeg -> ExprComp
desugSimple = desug
```

```
mulTwo :: (Const partof e, Op partof e) => e -> e
mulTwo expr = expr `Mul` Const 2
```

Having automatic extension of data types, as do Open Data Types and Open Abstract Types (Sections 2.7–2.8), would lead to a simpler and less verbose code, since one does not need to form different compositions. On the other hand, having automatic extension of data types causes an important limitation in expressiveness. The expressive power of being able to restrict functions to only handle a specific set of pieces is only possible through having different compositions of a data type, that is, having composable types.

Another complexity with composable types is that it requires the use of smart constructors to form the correct composition in different situations. In our syntax, we hide this complexity, and let the user use the piece constructors as smart constructors, rather than constructors of the pieces themselves. Similarly, we propose simplifying pattern matching in regular functions over pieces in a similar manner in Section 5.5.1. Therefore, the internal structure of the composition is not to be considered a limitation of our syntax.

As opposed to Data Types à la Carte and the other solutions presented in Chapter 2 also having composable types, we have categories as the first step. This means that all pieces depend on a category, and thus we must have a module containing the category declaration as the base for piece declarations, either written in the same module or in modules importing the base module. In Figure 6.1, we present the structure for constructing composable data types in our syntax as a flowchart. Data Types à la Carte and similar could then be seen as a similar graph but without the category step. On the other hand, Open Data Types and Open Abstract Types can be seen as a similar graph but without the compositions, as our categories serve as their declarations of open data types. Compare to the graphs in Figure 1.1, which show these two approaches: extensible and composable data types.

In total, our structure for constructing composable data types is the same as in Trees that Grow (Section 2.5). In that solution, there is also a declaration of the data type in the definition of the ground type, and then a composition is formed through defining type instances for that specific composition. There are, however, drawbacks to Trees that Grow compared to our solution, especially in user-friendliness. As mentioned in Section 2.5, Trees that Grow handles the extensibility of data types

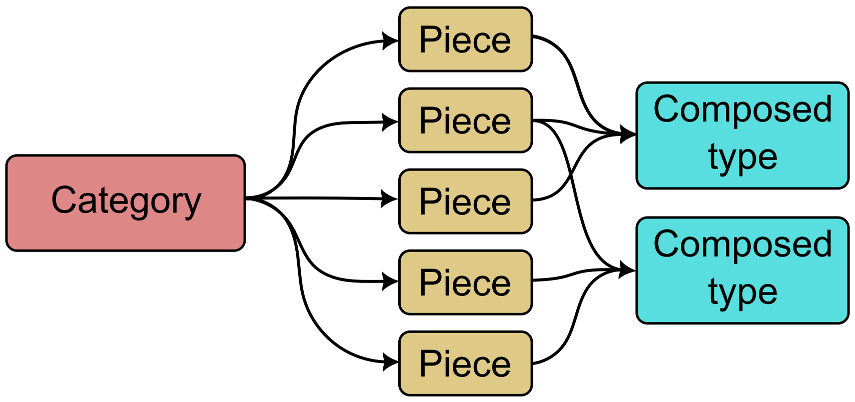

Figure 6.1: *A flowchart showing the parts of our syntax required to construct an extensible data type. A category is declared, which can be used for many different pieces, which can then be used together in different combinations in composed types.*

through a constructor inside the data type, while Data Types à la Carte handles it outside the data type, using coproducts. The conclusion was that this means that Trees that Grow is more complex to use than Data Types à la Carte in this regard. Our solution works like Data Types à la Carte, and handles extensibility outside the data type using the composed type. Hence, our syntax is simpler to use than Trees that Grow in this regard.

6.2.2 Automatic Extension of Functions

In contrast to the extension of data types, our syntax provides automatic extension of functions. This means that a function in our syntax consists of two parts: the declaration and the extensions. The extensions are then directly usable in all future uses of the function, without having to specify which extensions to include. We illustrate this in a flowchart in Figure 6.2. Note the lack of compositions from the extensions.

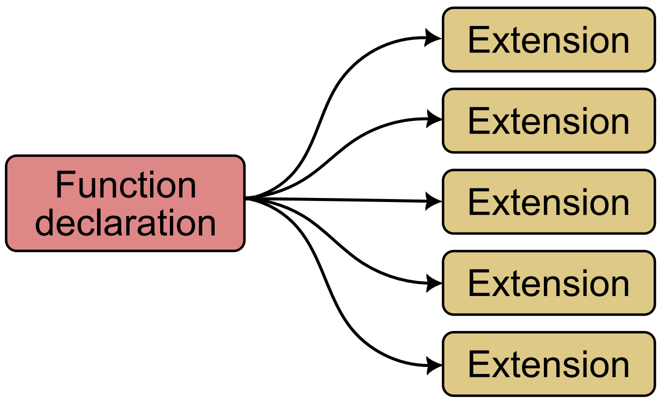

Figure 6.2: *A flowchart of the construction of extensible functions in our syntax. First, a function is declared. This can then be extended through writing extensions for different pieces. Note that there is no need for any composition of the extensions; they are automatically extending the function for all future uses.*

This is the same as with Data Types à la Carte and `compdata`, which provide a similar structure based on type classes and instances. On the other hand, one can have composable functions, as do Variations on Variants (Section 2.4) and Trees that Grow (Section 2.5). The flowchart for those would then first have the function cases and then different compositions, similar to composable data types.

Automatic extension in this case means a generally simpler syntax, since we do not need to differentiate combinations of function extensions, but can use them all together. Moreover, this is not a significant limitation of expressiveness either, since we specify a fixed composed type when calling the function, which then indicates the extensions that will be used. Thus, it is more the case that all extensions are available, and then a call to the function will use the extensions specified by the composed type. Furthermore, static type safety ensures that the function is only applied to variants which it can handle. The other situation where the expressiveness is limited is if one were to define multiple different function cases for the same variant. This is possible through composable functions, while with extensible functions, it can instead be done through defining a new function instead that has the new case. As a result, this does not limit the expressiveness to a significant extent either.

6.2.3 Choice of Backend

Our choice of having composable data types and automatic extension of functions is the same as for Data Types à la Carte and `compdata`, Sections 2.2–2.3. This means that the flowcharts given in Figures 6.1 and 6.2 would look mostly the same for those solutions, with one main difference: the category. While we require a category as the base for the pieces, data types for the variants can be written independently in those solutions.

With these similarities, the choice of `compdata` as our backend is sensible, since the similarities simplify the transformation. With the syntax we have designed, the correspondence is direct except for the categories, so there would be no reason to have a different backend. Still, we must consider whether there are possible improvements to our syntax that would make another backend more suitable.

One aspect discussed in Section 5.6.3 is to have a remainder constraint in our syntax, that is, express that a composition contains the same pieces as another composition minus a specified piece. Such a constraint is readily available in Variants on Variants. However, since it is possible to achieve a similar result with an isomorphism constraint and a coproduct, we do not view this as a sufficient argument for any change of backend.

6.3 Goals for the Syntax

In the following subsections, we discuss different aspects of our syntax and how they relate to the goals we presented in Section 1.2: simplicity, familiarity, and expressiveness.

6.3.1 Categories

Our main difference from Data Types à la Carte and `compdata` is the concept of categories. We first introduced categories with the goal of making the syntax more simple and familiar by making data type pieces more similar to regular data types. The idea is to use the category to represent the regular data type, and then use it in place of the recursive parameter. Categories can also be used for grouping pieces under one purpose or functionality. Furthermore, we have noted that categories correspond to the automatically extended data types seen in Open Data Types and Open Abstract Types (Sections 2.7– 2.8), where new variants can be added to the so called open data type without modifying the original definition. However, categories are still not actual types, and cannot be used as such outside of piece constructors.

In this section, we want to discuss to what extent categories fulfill the goals stated in Section 1.2. The syntax of having categories as recursive parameters is similar to the intuitive syntax of regular data types, which has the data type itself as the recursive parameter. This is more intuitive than the parameterization of data types where a type variable is used as the recursive parameter, as in Data Types à la Carte and similar. As a result, the syntax for piece constructors is more familiar since it is more similar to that of regular data types. On the other hand, the left-hand side of piece declarations introduces the new concept of categories in an unfamiliar way. Compare our syntax to that of regular data types and the parameterized syntax of Data Types à la Carte in the following:

```haskell
-- Data type piece in our syntax
data piece ExprCat ==> Add = Add ExprCat ExprCat

-- Regular Haskell data type
data Expr = ... | Add Expr Expr

-- Data type in Data Types à la Carte
data Add e = Add e e
```

The first thing to note is that the syntax that uses categories is more verbose than the backend counterpart, in particular in the non-familiar left-hand side. However, the exact syntax for piece declarations could be adjusted to make this problem less apparent. We can also note that while we hide the type variable for the recursive parameter present in the backend, we do so by introducing a new concept. Hence, we cannot clearly state that this is a simplification.

A problem with using categories in place of type variables is that categories can easily be mistaken for regular concrete types, which are different from the recursive parameter that the category represents. It is then possible that a user needs to understand the concepts of parameterization or polymorphism to understand how the recursion in a piece works in practice, with or without the category replacement. We can, however, say that our syntax with categories helps to separate data type pieces from regular data types. This makes pieces more distinct while also limiting their use as they cannot be used directly as regular data types.

The other feature to discuss about categories is the possibility of grouping pieces. Just as with the distinction of pieces from regular data types, this is a limitation in use, since pieces from different categories cannot be used in the same composition. Organizing pieces into categories serves to limit what one can do with a piece, so that composed types and functions are limited to only accept pieces of a specific category. Thus, this is only an organizational feature that introduces limitations in use. This does not mean that the feature has no value. It can be compared to a type system, which limits the expressiveness by distinguishing terms of different types, a limitation that is useful in practice. What this means for categories is that we cannot fully judge if it is useful to group pieces in this way, without observing more practical examples that use our syntax.

Still, there is one potential use case for categories where they may be particularly useful. When expressing more complicated recursion schemes between several different data types, which we discuss in Section 5.6.1, categories serve as the ideal container for expressing how these recursions are structured. This possibility exists because of how categories symbolise the extensible data types.

In conclusion, changes to or exclusion of the design of categories may be relevant if this work is continued. We have not fully explored the concept of categories. While there exist some problems with their current use, there is also some potential for refining how they are used in piece declarations, and we also described a potential new use case.

6.3.2 Hiding the Concept of Coproducts

Another of our design choices that differentiates our syntax from that of `compdata` is how we leave out the concept of the coproduct. Whereas `compdata` has individual parameterized data types, open coproducts of data types and closed composed types, our syntax has data type pieces that are composed directly into composed types. When we consider coproducts as a stepping stone between the individual types and the complete composed type, avoiding it in our syntax serves as a simplification. It does so not only by simplifying composed types by combining all pieces in one step, but also by limiting parameterization to be done on the entire composed type, rather than on the coproduct that is frequently seen in `compdata`. In our syntax, one can thus use the composed type directly in constraints such as a piece constraint.

However, using the composed type for parameterization comes at the cost of some expressiveness. Although various type constraints in `compdata` that use coproducts were easily reused in our syntax using intermediate classes, combining compositions is more difficult. With `compdata`, two coproducts can be combined without a challenge, although with the caveat that any duplicate types are not removed. This cannot be done as easily with closed composed types due to how they must be collected in the transformation phase. We have a suggestion on how composed types can be combined in Section 5.4.2, but it still has some limitations in expressiveness compared to coproducts. Nevertheless, hiding the concept of coproducts seems to be a generally good choice.

6.3.3 Using the Piece Constructors

The piece constructors in our syntax behave as smart constructors from Data Types à la Carte and similar in that they can construct values of different composed types in a polymorphic manner. It is thus important for a user to remember that the constructors do not construct values of the pieces themselves, but of composed types containing the pieces. This concept may be counterintuitive if one sees the pieces as data types. However, if one considers them to be variants of the data types, as we want, they work similarly to constructors for variants of a regular data type.

Another comparison that can be made is with the constructors of Trees that Grow. Constructors in that solution must be used with the same structure as how the data type is extended, for instance as `Expr_Ext (Neg (...)))`. Thus, our syntax is simpler in this regard, but it could be possible to simplify the constructors in Trees that Grow via smart constructors or a transformation from simplified syntax into that solution.

Similarly, we want the piece constructors to be used as deconstructors in pattern matching in regular functions, a goal discussed in Section 5.5.1. This would then mean as intuitive a syntax for pattern matching in regular functions as for regular data types. Without that addition, our syntax has no intuitive syntax for deconstruction in regular functions, while it can be performed in extensible functions.

6.3.4 Single- or Multi-variant Pieces

While our syntax divides an extensible data type into pieces, each individual piece can define several variants as piece constructors. There are benefits and drawbacks to grouping constructors under fewer pieces, rather than defining one constructor per piece.

The clear benefit of writing pieces that have multiple constructors is that one ends up with fewer pieces in total. Consequences to this are that compositions will be smaller, fewer piece constraints might be needed, and one overall gets fewer extensions and piece declarations. Grouping of piece constructors also serves an organizational purpose, where groups of constructors are made inseparable by belonging to the same piece. In particular, this is useful in examples like having lambda abstractions and applications in the same piece, where it should not be possible to use one without the other.

On the other hand, with function extensions, our discussion in Section 5.3.2 becomes relevant. There, we discussed how the syntax for extensible functions could be simplified. For an extension to `eval` for `Const`, we proposed the following simplified syntax:

```
ext eval (Const i) = i
```

This syntax makes extensions very similar to cases for regular functions. Although this syntax could be allowed for pieces with multiple constructors, we have the problem of how the syntax is perceived. An extension for such a piece should be transformed into a single instance, with all cases inside. We would then need to require that all one-line extensions to the same piece lie in the same module, so that the transformation can move these under the same instance. This is not expressed properly by the syntax, unlike our original extension syntax, where these cases are grouped under the piece in question.

This consequence is related to another observation. With pieces that have multiple constructors, there is a clear distinction between the pieces themselves and their constructors. On the other hand, if all pieces had only one constructor each, then pieces and piece constructors could be seen as the same concept and symbol, which may open up different approaches to expressing this syntax. A great example of this is to exclude the piece name from the piece declaration. Then we could have a syntax like the following:

```
piececategory ExprCat

data piece ExprCat = Const Int
data piece ExprCat = Add ExprCat ExprCat
data piece ExprCat = Mul ExprCat ExprCat

type ExprComp = ExprCat ==> (Const | Add | Mul)
```

This syntax would be possible if we did not have a distinction between pieces and piece constructors, and would essentially bring our syntax closer to regular data types and to the syntax in Open Data Types and Open Abstract Types.

6.3.5 Distinction in Function Syntax

Extensible functions in our syntax are similar to the extensible functions written as type classes in Data Types à la Carte and `compdata`. However, our syntax has an advantage in familiarity, since it is closer to the syntax of regular functions than it would be with the type class approach. Especially our declaration of an extensible function is closer in syntax to a regular type signature. Still, our function extensions are more similar to instance declarations than to regular function equations. However, the proposed simplification mentioned in the previous section, and in Section 5.3.2, would make the syntax more similar to regular functions. The following example would then be possible if one kept `Add` and `Mul` in separate pieces:

```
eval -: ExprCat -> Int
ext eval (Const i)   = i
ext eval (Add e1 e2) = eval e1 + eval e2
ext eval (Mul e1 e2) = eval e2 * eval e2
```

This is very similar to the syntax of Open Data Types (Section 2.7). They have the keyword `open` to distinguish the function declaration from a type signature for a regular function, while we use the symbol `-:`. They keep the syntax for the extensions identical to regular functions, which would be possible for our implementation as well, provided that the transformation keeps track of which functions are declared extensible and transforms only those.

We can further compare this syntax to that of Open Abstract Types (Section 2.8), a solution that omits the need for any specific keywords in the functions completely. Again, this could be possible with our syntax via a more rigorous transformation. However, by having the syntax identical to that for non-extensible functions, there may arise some confusion for a user when the distinction is not clear. If functions for regular data types and composable data types worked in the same way, with the same possibilities and limitations, this would not be a problem. However, they do not. Regular functions must be written together, with pattern matching working from top to bottom. Extensible functions can have their cases separate, meaning that there is no top-to-bottom order. Those functions must then either have some other disambiguation between extensions such as best-fit pattern matching, or disallow wildcards and overlapping extensions altogether. Therefore, it is useful to provide a distinction in the syntax for the functions as well, not just the data types. If it were possible to provide the same expressiveness for extensible data types and functions as for regular ones, one could use the same syntax and let all data types be extensible, or have the possibility to be extensible.

Another aspect to consider in the syntax of extensible functions is that we restrict the first argument to being a category in destructive and transformative functions. This is to signal that the first argument should be handled in a special way, namely, to be deconstructed through the function. This gives a more intuitive syntax for destructive functions, as it is similar to regular functions. It is, on the other hand, less intuitive in transformative functions, where it means that we do not have as intuitive a difference between input and output. It is useful to provide a distinction between the input and output types as they are not necessarily the same, but it is unintuitive to represent one of them with a category and the other with a type variable and constraint, as in the following example:

```
desug -: (ExprCat ==> e) => ExprCat -> e
```

In Sections 5.6.3 and 5.6.4, we discussed additions to our syntax which could make it easier to distinguish input and output types, while still connecting them through for instance a subtype constraint.

6.3.6 Explicit Recursion Rather than Algebras

We choose to represent extensible functions with explicit recursion rather than with algebras. Hence, an extension to `eval` for addition looks like the following in our syntax:

```
ext eval for Op where
    eval (Add e1 e2) = eval e1 + eval e2
```

On the other hand, it is written as the following in Data Types à la Carte, using an algebra that is folded over:

```
instance Eval Op where
    evalAlg (Add e1 e2) = e1 + e2
```

The choice to not represent extensible functions with algebras is based on several aspects. First, explicitly stating the recursion is closer to the syntax of regular functions. In our syntax, the user is also expected to write a function simply as a declaration and extensions, while we insert the parts relating to the type class and the type class function in the transformation. In Data Types à la Carte, we have a function that folds over the algebra function given in the type class, as follows:

```
eval :: Eval f => Term f -> Int
eval = foldTerm evalAlg
```

This, along with the type class and its function, would be generated in our transformation, rather than written by a user. As a result, the place where a fold would naturally be written in the transformed code would not be present in the code written in our syntax. Hence, it would be less suitable, and less flexible, to implicitly use algebras in our syntax.

Since we hide the parameterization of pieces, we also hide the fact that they are functors over which we can perform a fold. Thus, it would be counterintuitive for a user to be expected to write function extensions using algebras.

6.3.7 Type Applications

The final part of our syntax for extensible functions to discuss is the type applications. Since we do not represent the functions as type classes in our syntax, we need another way to represent type variables that is not simply as variables to the type class. We recall here the example of `evalNum` given in Section 3.6.1, an evaluation function with a polymorphic result type. In the type class approach, which is also used in our backend, an instance to `evalNum` is represented as follows:

```
instance EvalNum Const Int where
```

Here, `Const` is the argument over which the function is extended, and `Int` is the result type. The first argument is distinguished in our syntax as being the argument over which the function is extended, and specifically the argument to be decomposed through the function. This is done in the function declaration by being written as a category, and here in the extension as the piece written after the keyword `for`. On the other hand, the instantiations of type variables from the declarations are represented with type applications. This is done with inspiration from type applications used in different contexts of Haskell, because there are similarities in how they refer to a type from a type signature (in our case the declaration of an extensible function) in order of appearance. With type applications, the extension is written as follows in our syntax:

```
ext evalNum @Int for Const where
```

This row alone may feel less intuitive and more complex, but one must consider that we cannot write it as the type class instance above without representing functions as type classes. A syntax closer to the type class instance above is the following:

```
ext evalNum for Const Int where
```

This is less intuitive, since it may seem like `Int` is an argument to `Const`. One could alleviate this by having our syntax even more similar to type classes by removing the keyword `for`, as follows:

```
ext evalNum Const Int where
```

In this case, `Const Int` does not look as much as an application, but more as two type parameters to a type class. However, we have chosen to have a syntax that is not as similar to type classes, because we do not represent extensible functions as type classes directly. Instead, we want a syntax more close to that of regular functions, something that is even more present in the syntax for the function declaration. As a result, we argue that taking inspiration from explicit type applications is reasonable in our syntax.

6.4 Conclusion

To conclude this thesis report, we cover our main contributions with this project.

We looked at a number of approaches to solving the expression problem, and, in an attempt to compare them, formulated the property of automatically extended. We used this term to describe the sufficiency of declaring individual parts of a whole, as opposed to composition, where individual parts need to be explicitly combined.

The main contribution of our syntax is the concept of categories. We want to combine the benefits of the two main alternatives of solving the expression problem: composable data types and automatically extended data types. With categories, we have a way of doing so. This means that we are able to construct a syntax with, to some extent, both the expressive power of composable data types, and the familiarity and simplicity of automatically extended data types. Another solution offering this combination is Trees that Grow. That solution is, however, fundamentally different in how composed types are formed, along with in how extensibility of functions is handled. Introducing categories comes with some limitations as well, and we have not developed the concept completely. We have also discussed categories as a mean to express and support more complex recursive data types.

Another main difference between our syntax and that of Data Types à la Carte is that we hide the concept of the coproduct, and thus the difference between open and closed composed types. This means that composed types are always closed in our syntax, and we are able to write constraints on such closed compositions.

We have also introduced a declaration for extensible functions, which is simpler and closer to regular functions than what is needed for introducing such a function as a type class. Furthermore, some adaptions would make the syntax for function extensions closer to regular functions in that we could allow single-line extensions for single-variant pieces.

Through these main contributions, we have achieved a syntax design with several improvements compared to existing solutions to the expression problem in Haskell.

Bibliography

[1] Philip Wadler et al. The Expression Problem. `https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`, 1998. Posted on the Java Genericity mailing list.

[2] Wouter Swierstra. Data types à la carte. *Journal of functional programming*, 18(4):423–436, 2008.

[3] Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. Technical report, École Polytechnique Fédérale de Lausanne, 2004.

[4] Andres Löh and Ralf Hinze. Open Data Types and Open Functions. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 133–144, 2006.

[5] J Garrett Morris. Variations on Variants. *ACM SIGPLAN Notices*, 50(12):71–81, 2015.

[6] Sean Seefried and Manuel MT Chakravarty. Solving the expression problem with true separate compilation. Technical report, The University of New South Wales, 2007.

[7] Benedict R Gaster and Mark P Jones. A polymorphic type system for extensible records and variants. Technical report, Citeseer, 1996.

[8] Racket. 3.9 C Union Types. `https://docs.racket-lang.org/foreign/C_Union_Types.html`.

[9] HaskellWiki. Fold – HaskellWiki. `https://wiki.haskell.org/Fold`, 2019.

[10] Orphan instance - HaskellWiki.

[11] Patrick Bahr and Tom Hvitved. Compositional Data Types. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming*, pages 83–94, 2011.

[12] Patrick Bahr. Composing and Decomposing Data types: A Closed Type Families Implementation of Data Types à la Carte. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming*, pages 71–82, 2014.

[13] Patrick Bahr and Tom Hvitved. compdata: Compositional Data Types. `https://hackage.haskell.org/package/compdata`, 2021.

[14] Patrick Bahr and Tom Hvitved. compdata: Example.Common. `https://hackage.haskell.org/package/compdata-0.12.1/src/examples/Examples/Common.hs`, 2011.

[15] Patrick Bahr and Tom Hvitved. compdata: Example.Eval. `https://hackage.haskell.org/package/compdata-0.12.1/src/examples/Examples/Eval.hs`, 2011.

[16] Patrick Bahr and Tom Hvitved. compdata: Example.Desugar. `https://hackage.haskell.org/package/compdata-0.12.1/src/examples/Examples/Desugar.hs`, 2011.

[17] GHC Team. 6.6.4.1. Deriving Functor instances – Glasgow Haskell Compiler 9.4.3 User's Guide. `https://downloads.haskell.org/ghc/9.4.3/docs/users_guide/exts/deriving_extra.html#deriving-functor-instances`, 2020.

[18] GHC Team. 6.10.2. Equality constraints and Coercible constraint – Glasgow Haskell Compiler 9.4.1 User's Guide. `https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/equality_constraints.html?#equality-constraints`, 2020.

[19] Shayan Najd and Simon Peyton Jones. Trees that Grow. *Journal of Universal Computer Science*, 23(1):42–62, 2017.

[20] Edward Kmett. Data.Void. `https://hackage.haskell.org/package/base-4.17.0.0/docs/Data-Void.html`, 2014.

[21] Bruno C d S Oliveira and William R Cook. Extensibility for the Masses – Practical Extensibility with Object Algebras. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2012.

[22] Sean Seefried. *Language extension via dynamically extensible compilers.* PhD thesis, UNSW Sydney, 2006.

[23] Konstantin Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–518, 1996.

[24] Simon Marlow. Chapter 10 Syntax Reference. `https://www.haskell.org/onlinereport/haskell2010/haskellch10.html`, Jul 2010.

[25] GHC Team. 6.4.14. Visible type application – Glasgow Haskell Compiler 9.3.20220306 User's Guide. `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/type_applications.html`, 2020.

[26] Niklas Broberg. haskell-src-exts: Manipulating Haskell source: abstract syntax, lexer, parser, and pretty-printer. `https://hackage.haskell.org/package/haskell-src-exts`, 2020.

[27] Philipp Schuster, Roman Cheplyaka, and Lennart Augustsson. haskell-names: Name resolution library for Haskell. `https://hackage.haskell.org/package/haskell-names`, 2020.

[28] Python Software Foundation. 5. Data Structures – Python 3.10.6 documentation. `https://docs.python.org/3/tutorial/datastructures.html`, 2022.

[29] Patrick Bahr. compdata: Data.Comp.Derive.Show. `https://hackage.haskell.org/package/compdata-0.12.1/src/src/Data/Comp/Derive/Show.hs`, 2011.

A
Resulting Syntax

In Table A.1, we present the syntax presented in this report in a more formal way than in Chapter 3. This means that we remove most of the simplifications mentioned in Section 3.1. However, we still show only one possible way to construct many syntactic components. For example, piece constructors can be written in a piece declaration in more ways than as in `Add Expr Expr`, for instance through infix or record syntax, while we only show the first alternative. Furthermore, these syntax rules are written from a user perspective, meaning that we show what would be accepted by the parser rather than actual parsing rules.[1]

The table is separated into five categories. First, we have *Declaration*, which contains the parts of our syntax that are defined in a syntactic component for top-level declarations, a syntactic component also containing other declarations from regular Haskell. Second, we have *Type*, which is correspondingly the part of our syntax which is defined as a *Type*. Third, we have *Constraint*, which includes our constraints on composed types. Fourth, we have *Miscellaneous* specifying parts of our syntax components further. This includes syntactic components we have added ourselves, and some that are standard Haskell to which we have added more alternatives covering our syntax. In particular, *Asst* used in *Context* has more standard Haskell components, which we do not present here. Last, *Explained symbols* explains informally the standard Haskell components that we have used, but not modified.

[1]The interested reader can see how the parsing rules are actually implemented in our Github project at `https://github.com/kirderf1/composable-types`.

Declaration	
PieceCatDecl	`piececategory` *Con*
PieceDecl	`data piece` *QCon* `==>` *Con* `=` *PieceConDecls*
CompFunDecl	*Var* `-:` [*Context* `=>`] *QCon* `->` *Type*
CompFunExt	`ext` [*Context* `=>`] *QVar* {*TypeApp*} `for` *PieceRef* `where` *InstanceDecls*
Type	
ComposedType	*QCon* `==>` `(` *CompList* `)`
Constraint	
Constraint	*Var* `with` *QVar* {*TypeApp*}
	PieceRef `partof` *Var*
	QCon `==>` *Var*
Miscellaneous	
PieceConDecls	*PieceConDecl* {`|` *PieceConDecl*}
PieceConDecl	*Con* {*PieceConArg*}
PieceConArg	*Type*
	QCon
Context	*Asst*
	`(` *Asst* { `,` *Asst* } `)`
	`( )`
Asst	*Constraint*
	`(` *Asst* `)`
	...
TypeApp	`@` *Type*
PieceRef	*QCon* {*Type*}
CompList	*PieceRef* {`|` *PieceRef*}
Explained symbols	
Type	A type, such as `Int`, a type variable or a composed type
Con	A constructor name (a name starting with an uppercase letter)
QCon	A constructor name optionally qualified with a module
Var	A variable name (a name starting with a lowercase letter)
QVar	A variable name optionally qualified with a module
InstanceDecls	A list of binding declarations as seen in Haskell instances

Table A.1: *The complete syntax we have designed in this project.*

B
Example of our Syntax and Transformation

This chapter shows a complete example program written using our syntax. It also contains the transformation of the example program, which is usable with `compdata`. This program is based on the running example presented in Section 2.1. It contains the data type for expressions, initially with constants, addition and multiplication, and then extended with negation. Furthermore, it contains two different destructive functions: `eval` for evaluation to an integer value, and `asString` for writing an expression as a string. The program also contains an example of a transformative function, namely the desugaring function, `desug`, that removes negation from expressions. Finally, it contains a main module for combining the previous parts, which includes writing examples and performing operations on those examples.

B.1 Syntax

This section covers the different modules for expressions, evaluation, writing as string, negation, desugaring, and finally a testing module. In all of these, note the language pragma `ComposableTypes`, which we use to distinguish modules that are written in our syntax and should be transformed. Recall also, as mentioned in Section 1.1, that the emphasis is not on the exact syntax regarding, for instance, choices of keywords.

B.1.1 Expressions

For expressions, we define a category `ExprCat`, which is used to define the pieces `Const` and `Op`, and the composed type `ExprComp`. Note that, similar to the example written with Data Types à la Carte in Section 2.2, we have the variants `Add` and `Mul` kept together in the same piece, `Op`.

```
{-# LANGUAGE ComposableTypes #-}

module Expr where

-- | Data type for expression language
-- | Category ExprCat
piececategory ExprCat

-- | Data type variants for Const, Add and Mul, contained in the
-- two pieces Const and Op
data piece ExprCat ==> Const = Const Int

data piece ExprCat ==> Op = Add ExprCat ExprCat
                          | Mul ExprCat ExprCat

-- | Composed type ExprComp
type ExprComp = ExprCat ==> (Const | Op)
```

B.1.2 Evaluation

Evaluation, `eval`, is a simple destructive function. Thus, the declaration is written using the category `ExprCat`, and we have an extension for each piece.

```
{-# LANGUAGE ComposableTypes #-}

module Eval where

import Expr

-- | Evaluation function
eval -: ExprCat -> Int

-- | Eval extension for constants
ext eval for Const where
    eval (Const i) = i

-- | Eval extension for operations
ext eval for Op where
    eval (Add e1 e2) = eval e1 + eval e2
    eval (Mul e1 e2) = eval e1 * eval e2
```

B.1.3 Expressions as Strings

The `asString` function is written analogous to `eval`.

```
{-# LANGUAGE ComposableTypes #-}

module AsString where

import Expr
```

```
-- | AsString function
asString -: ExprCat -> String

-- | AsString extension for constants
ext asString for Const where
    asString (Const i) = show i

-- | AsString extension for operations
ext asString for Op where
    asString (Add e1 e2) = "(" ++ asString e1 ++ " + "
                               ++ asString e2 ++ ")"
    asString (Mul e1 e2) = "(" ++ asString e1 ++ " * "
                               ++ asString e2 ++ ")"
```

B.1.4 Negation

The module for negation contains the piece `Neg`, along with function extensions for `eval` and `asString`. Note that this module is dependent on the module `Expr`, since we are using the category `ExprCat`. On the other hand, it is independent from the pieces defined in the module `Expr`, so it would be easy to write the category in a separate module, to have the two modules, one with `Const` and `Op`, and one with `Neg`, written independently from each other.

```
{-# LANGUAGE ComposableTypes #-}

module Negation where

import Expr
import Eval
import AsString

-- | Data type piece for negation
data piece ExprCat ==> Neg = Neg ExprCat

-- | Eval extension for negation
ext eval for Neg where
    eval (Neg e) = (-1) * eval e

-- | AsString extension for negation
ext asString for Neg where
    asString (Neg e) = "(-" ++ asString e ++ ")"
```

B.1.5 Desugaring

The next module contains a desugaring function, `desug`, which is a transformative function. As such, it decomposes a value of a composed type, in this case some expression, and constructs a value, also of a composed type, in this case an expression without negation. We express this return type with a type variable and a category constraint in the declaration. Piece constraints are then used in the function extensions to indicate which pieces are included in the result. The case for negation is included directly in this module, as opposed to in a separate module as for the previous functions, since that is the relevant part of the `desug` function.

```
{-# LANGUAGE ComposableTypes #-}

module Desug where

import Expr
import Negation

-- | Transformative function desug
desug -: (ExprCat ==> e) => ExprCat -> e

-- | Desug extension for constants
ext (Const partof a) => desug @a for Const where
    desug (Const i) = Const i

-- | Desug extension for operations
ext (Op partof a) => desug @a for Op where
    desug (Add e1 e2) = Add (desug e1) (desug e2)
    desug (Mul e1 e2) = Mul (desug e1) (desug e2)

-- | Desug extension for negation
ext (Const partof a, Op partof a) => desug @a for Neg where
    desug (Neg e) = Const (-1) `Mul` (desug e)
```

B.1.6 Main Module

Lastly, we have a main module, containing examples written using the constructors of the pieces. These are also used with the defined functions. Note that the type annotations of the examples are necessary to call the functions on them, for the same reason as in Data Types à la Carte and `compdata`; see Section 2.2. Likewise, the result type of `desug threePlusNegFive` must be specified in order to call `asString` with it.

```
{-# LANGUAGE ComposableTypes #-}

module Main where

import Expr
import Eval
import AsString
import Negation
import Desug
```

```haskell
-- | Examples of type ExprComp, containing constants, addition and
-- multiplication
threePlusFive :: (Const partof e, Op partof e) => e
threePlusFive = Add (Const 3) (Const 5)

twoMulThreePlusFive :: ExprComp
twoMulThreePlusFive = Mul (Const 2) threePlusFive

-- | Composed type ExprCompWithNeg, also containing negation
type ExprCompWithNeg = ExprCat ==> (Const | Op | Neg)

-- | Example with negation
threePlusNegFive :: ExprCompWithNeg
threePlusNegFive = Const 3 `Add` (Neg (Const 5))

-- | Evaluation examples
evalAddMul = eval twoMulThreePlusFive

evalAddNeg = eval threePlusNegFive

-- | AsString example
asStringAddNeg = asString threePlusNegFive

-- | Desugar example
desugAddNeg = asString (desug threePlusNegFive :: ExprComp)

-- | Main, printing results of above examples
main :: IO ()
main = do
    putStrLn "Evaluation examples:"
    print evalAddMul
    print evalAddNeg
    putStrLn "AsString example:"
    putStrLn asStringAddNeg
    putStrLn "Desugar example:"
    putStrLn desugAddNeg
```

B.2 Transformation

In this section, we present the transformation of the example in the previous section. In all of these, we include some language pragmas which may be needed for code using `compdata`.

B.2.1 Expressions

The following module contains the transformation of the module `Expr`. As mentioned in Section 4.2, a category declaration does not correspond to anything in the transformed code. Each piece is transformed into a regular data type with a type variable `a` instead of the category, which is used for recursion. For each piece, a smart constructor is also derived, which uses our library class `PartOf` for the constraint in the type signature, corresponding to subsumption. Finally, the composition of pieces is transformed into a coproduct that is closed by `Term`.

```haskell
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE UndecidableInstances #-}

module Expr where

import Data.Comp
import Data.Comp.Derive
import ComposableTypes

data Const a = Const Int

iConst :: PartOf Const e => Int -> e
iConst e1 = inject' (Const e1)

data Op a = Add a a | Mul a a

iAdd :: PartOf Op e => e -> e -> e
iAdd e1 e2 = inject' (Add e1 e2)

iMul :: PartOf Op e => e -> e -> e
iMul e1 e2 = inject' (Mul e1 e2)

type ExprComp = Term (Const :+: Op)
```

B.2.2 Evaluation

The evaluation function `eval` is transformed into a type class `Eval`, with instances for all extensions to the function. An instance of this class for the coproduct is derived using Template Haskell, via `liftSum` in `compdata`. The actual function `eval` is wrapped in another type class, `Eval_outer`, to facilitate the use of constraints on composed types, as described in Section 4.4 and 4.6.

```haskell
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE UndecidableInstances #-}

module Eval where

import Data.Comp
import Data.Comp.Derive
import ComposableTypes
import Expr

class Eval f where
    eval' :: (Eval g) => f (Term g) -> Int

$(derive [liftSum]['Eval])

class Eval_outer t where
    eval :: t -> Int

instance Eval g => Eval_outer (Term g) where
    eval = eval' . unTerm

instance Eval Const where
    eval' (Const i) = i

instance Eval Op where
    eval' (Add e1 e2) = eval e1 + eval e2
    eval' (Mul e1 e2) = eval e1 * eval e2
```

B.2.3 Expressions as Strings

The module for converting an expression to a string, `AsString`, is transformed in a similar manner to `Eval` in the previous section.

```haskell
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE UndecidableInstances #-}

module AsString where
```

```
import Data.Comp
import Data.Comp.Derive
import ComposableTypes
import Expr

class AsString f where
    asString' :: (AsString g) => f (Term g) -> String

$(derive [liftSum] [''AsString])

class AsString_outer t where
    asString :: t -> String

instance AsString g => AsString_outer (Term g) where
    asString = asString' . unTerm

instance AsString Const where
    asString' (Const i) = show i

instance AsString Op where
    asString' (Add e1 e2) = "(" ++ asString e1 ++ " + "
                                ++ asString e2 ++ ")"
    asString' (Mul e1 e2) = "(" ++ asString e1 ++ " * "
                                ++ asString e2 ++ ")"
```

B.2.4 Negation

The module for negation of an expressions contains nothing new regarding transforming its code.

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE UndecidableInstances #-}

module Negation where

import Data.Comp
import Data.Comp.Derive
import ComposableTypes
import Expr
import Eval
import AsString

data Neg a = Neg a
```

```haskell
iNeg :: PartOf Neg e => e -> e
iNeg e1 = inject' (Neg e1)

instance Eval Neg where
    eval' (Neg e) = (-1) * eval e

instance AsString Neg where
    asString' (Neg e) = "(-" ++ asString e ++ ")"
```

B.2.5 Desugaring

The function `desug` is a transformative function, and therefore, the module for it contains some new aspects compared to previous modules. First, the category constraint in the function declaration is transformed into an empty context. Second, the piece constraints in the extensions are transformed into constraints using `PartOf`. Finally, the use of constructors in the function equations are transformed into using the smart constructors that were generated for each piece.

```haskell
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE UndecidableInstances #-}

module Desug where

import Data.Comp
import Data.Comp.Derive
import ComposableTypes
import Expr
import Negation

class () => Desug f e where
    desug' :: (Desug g e) => f (Term g) -> e

$(derive [liftSum] [''Desug])

class Desug_outer t e where
    desug :: t -> e

instance Desug g e => Desug_outer (Term g) e where
    desug = desug' . unTerm

instance (PartOf Const a) => Desug Const a where
    desug' (Const i) = iConst i
```

```
instance (PartOf Op a) => Desug Op a where
    desug' (Add e1 e2) = iAdd (desug e1) (desug e2)
    desug' (Mul e1 e2) = iMul (desug e1) (desug e2)

instance (PartOf Const a, PartOf Op a) => Desug Neg a where
    desug' (Neg e) = iConst (-1) `iMul` (desug e)
```

B.2.6 Main Module

The content of the main module is transformed as described in the previous sections in this chapter.

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE UndecidableInstances #-}

module Main where

import Data.Comp
import Data.Comp.Derive
import ComposableTypes
import Expr
import Eval
import AsString
import Negation
import Desug

threePlusFive :: (PartOf Const e, PartOf Op e) => e
threePlusFive = iAdd (iConst 3) (iConst 5)

twoMulThreePlusFive :: ExprComp
twoMulThreePlusFive = iMul (iConst 2) threePlusFive

type ExprCompWithNeg = Term (Const :+: Op :+: Neg)

threePlusNegFive :: ExprCompWithNeg
threePlusNegFive = iConst 3 `iAdd` (iNeg (iConst 5))

evalAddMul = eval twoMulThreePlusFive

evalAddNeg = eval threePlusNegFive

asStringAddNeg = asString threePlusNegFive
```

```haskell
desugAddNeg = asString (desug threePlusNegFive :: ExprComp)

main :: IO ()
main = do
    putStrLn "Evaluation examples:"
    print evalAddMul
    print evalAddNeg
    putStrLn "AsString example:"
    putStrLn asStringAddNeg
    putStrLn "Desugar example:"
    putStrLn desugAddNeg
```

C

Full Programs in Different Approaches

In this chapter, we show full programs written using some of the approaches presented in Chapter 2. The approaches showcased here are all except Open Data Types and Open Abstract Types; each of those includes new syntax that needs to be transformed in some way, which means that a transformation program is needed to be able to compile and run an example with that syntax. As a result, we are not able to present fully working programs for those here.

The programs we have written each have a library module when necessary, to provide necessary definitions such as the coproduct. These are given in Appendix D, so that we can focus on the syntax of using the solutions in this chapter. The programs here include the data type for expressions with the extension of negation, two destructive functions (`eval` and `asString`), and one transformative function (`desug`). They also contain a main module that combines the previous parts through writing examples of the data type and performing operations on those examples. This example program is the same as the program given in Appendix B, where it is written using the syntax that we have designed.

C.1 Data Types à la Carte

This section covers Data Types à la Carte [2] that was introduced in Section 2.2. To be able to use this, one needs the library module `DTalC` given in Section D.1.

C.1.1 Expressions

Using the library module in Section D.1, we can define the data type for expressions. This is done by defining a data type for each of the variants: one for constants and another one for operations, containing both addition and multiplication. For each of these, functor instances and smart constructors are also defined.

```
{-# LANGUAGE TypeOperators, FlexibleContexts #-}

module ExprDTalC where
```

```haskell
import DTalC

-- | Data type for expression language

-- | Data type variants for Const, Add and Mul
data Const a = Const Int

data Op a = Add a a | Mul a a

-- | Composed type Expr
type Expr = Term (Const :+: Op)

-- | Functors
instance Functor Const where
    fmap f (Const i) = Const i

instance Functor Op where
    fmap f (Add e1 e2) = Add (f e1) (f e2)
    fmap f (Mul e1 e2) = Mul (f e1) (f e2)

-- | Smart constructors
iConst :: (Const :<: f) => Int -> Term f
iConst i = inject (Const i)

iAdd :: (Op :<: f) => Term f -> Term f -> Term f
iAdd e1 e2 = inject (Add e1 e2)

iMul :: (Op :<: f) => Term f -> Term f -> Term f
iMul e1 e2 = inject (Mul e1 e2)
```

C.1.2 Evaluation

Next, we have the implementation of the evaluation function, which is done through a type class `Eval`. The function is written using an algebra, which means that evaluation is carried out via the function `foldTerm` defined in the module `DTalC`.

```haskell
{-# LANGUAGE TypeOperators #-}

module EvalDTalC where

import DTalC
import ExprDTalC

-- | Eval function using an algebra
class Functor f => Eval f where
    evalAlg :: f Int -> Int
```

```haskell
-- | Fold using evalAlg
eval :: Eval f => Term f -> Int
eval = foldTerm evalAlg

-- | Eval instance for constants
instance Eval Const where
    evalAlg (Const i) = i

-- | Eval instance for operations
instance Eval Op where
    evalAlg (Add e1 e2) = e1 + e2
    evalAlg (Mul e1 e2) = e1 * e2

-- | Eval instance for coproduct
instance (Eval f, Eval g) => Eval (f :+: g) where
    evalAlg (Inl a) = evalAlg a
    evalAlg (Inr b) = evalAlg b
```

C.1.3 Expressions as Strings

It is also possible to write an extensible function with Data Types à la Carte without an algebra. This is how the function `asString` is written in the following module.

```haskell
{-# LANGUAGE TypeOperators #-}

module AsStringDTalC where

import DTalC
import ExprDTalC

-- | AsString function, written without an algebra
class AsString f where
    asString' :: AsString g => f (Term g) -> String

-- | Unpack a Term and call asString'
asString :: AsString f => Term f -> String
asString (In t) = asString' t

-- | AsString instance for constants
instance AsString Const where
    asString' (Const i) = show i

-- | AsString instance for operations
instance AsString Op where
    asString' (Add e1 e2) = "(" ++ asString e1 ++ " + "
                                ++ asString e2 ++ ")"
    asString' (Mul e1 e2) = "(" ++ asString e1 ++ " * "
                                ++ asString e2 ++ ")"

-- | AsString instance for coproduct
instance (AsString f, AsString g) => AsString (f :+: g) where
    asString' (Inl a) = asString' a
    asString' (Inr b) = asString' b
```

C.1.4 Negation

Next, we want to extend the data type with a variant for negation. In the following module, we define the data type for negation, as well as instances for `eval` and `asString`. Note that this module can be written independently of the module `ExprDTalC`. The data types can later be combined in a composition in a different module, which depends on both `ExprDTalC` and `NegationDTalC`.

```
{-# LANGUAGE TypeOperators, FlexibleContexts #-}

module NegationDTalC where

import DTalC
import EvalDTalC
import AsStringDTalC

-- | Data type for negation
data Neg a = Neg a

-- | Functor
instance Functor Neg where
    fmap f (Neg e) = Neg (f e)

-- | Smart constructor
iNeg :: (Neg :<: f) => Term f -> Term f
iNeg e = inject (Neg e)

-- | Eval instance for negation
instance Eval Neg where
    evalAlg (Neg e) = (-1) * e

-- | AsString instance for negation
instance AsString Neg where
    asString' (Neg e) = "(-" ++ asString e ++ ")"
```

C.1.5 Desugaring

The following module contains the definition of the transformative desugaring function, again written using an algebra. This contains the case for negation directly, since that is the relevant part of this function. The cases for the original data type defined in `Expr` are covered by a default case that just calls the function `inject`.[1]

```haskell
{-# LANGUAGE TypeOperators, UndecidableInstances, FlexibleInstances,
    MultiParamTypeClasses #-}

module DesugDTalC where

import DTalC
import ExprDTalC
import NegationDTalC

-- | Transformative function desug using algebra
class (Functor f, Functor g) => Desug f g where
    desugAlg :: f (Term g) -> (Term g)

-- | Fold using desugAlg
desug :: (Desug f g, Functor f) => Term f -> Term g
desug = foldTerm desugAlg

-- | Default instance of Desug
instance {-# OVERLAPPABLE #-}
        (Functor f, Functor g, f :<: g)
        => Desug f g where
    desugAlg = inject

-- | Desug instance for negation
instance (Functor g, Const :<: g, Op :<: g) => Desug Neg g where
    desugAlg (Neg e) = iConst (-1) `iMul` e

instance (Desug f h, Desug g h) => Desug (f :+: g) h where
    desugAlg (Inl a) = desugAlg a
    desugAlg (Inr b) = desugAlg b
```

C.1.6 Main Module

Finally, we have a module to combine the previous modules, in which we write examples and perform operations on those examples. First, we have two examples containing the parts in the original data type `Expr`, namely constants, addition and multiplication. The first of these is written with constraints and type variables, and the second with the fixed composed type `Expr`. Then, we define a new composed type, `ExprWithNeg`, that also contains negation, which is used in a third example. These examples are used in different functions, to either evaluate, convert to a string, or desugar the examples. Lastly, a main function is provided to print the results of running the functions.

[1]Note that having a default case like this is not possible in our syntax, although we discuss the possibility of adding a way to do so in Section 5.6.2.

```haskell
{-# LANGUAGE TypeOperators, FlexibleContexts #-}

module Main where

import DTalC
import ExprDTalC
import EvalDTalC
import AsStringDTalC
import NegationDTalC
import DesugDTalC

-- | Examples of type Expr, containing constants, addition and
-- multiplication
threePlusFive :: (Const :<: f, Op :<: f) => Term f
threePlusFive = iAdd (iConst 3) (iConst 5)

twoMulThreePlusFive :: Expr
twoMulThreePlusFive = iMul (iConst 2) threePlusFive

-- | Composed type ExprWithNeg, also containing negation
type ExprWithNeg = Term (Const :+: Op :+: Neg)

-- | Example with negation
threePlusNegFive :: ExprWithNeg
threePlusNegFive = iConst 3 `iAdd` (iNeg (iConst 5))

-- | Evaluation examples
evalAddMul = eval twoMulThreePlusFive
evalAddNeg = eval threePlusNegFive

-- | AsString example
asStringAddNeg = asString threePlusNegFive

-- | Desugar example
desugAddNeg = asString (desug threePlusNegFive :: Expr)

-- | Main, printing results of above examples
main :: IO ()
main = do
    putStrLn "Evaluation examples:"
    print evalAddMul
    print evalAddNeg
    putStrLn "AsString example:"
    putStrLn asStringAddNeg
    putStrLn "Desugar example:"
    putStrLn desugAddNeg
```

C.2 compdata

This section covers a program written using `compdata` [13, 11, 12] that was introduced in Section 2.3. Since it is a Haskell library [13], we use that directly. Still, we present the crucial parts of the library in Section D.2, so that it is possible to understand the program here without referring to the complete Haskell library.

C.2.1 Expressions

First, we begin with a module containing the data type for expressions, with variants for constants, addition and multiplication. The notable differences from Data Types à la Carte, presented in the previous section, is that here, functor instances are derived using `derive Functor`, and smart constructors (and more) are derived through Template Haskell. The derivation of functor instances is of course something that could be done already in the Data Types à la Carte example, using the same language extension as here.

```haskell
{-# LANGUAGE TemplateHaskell, TypeOperators, FlexibleContexts,
    DeriveFunctor #-}

module ExprCompdata where

import Data.Comp hiding (Const)
import Data.Comp.Derive

-- | Data type for expression language

-- | Data type variants for Const, Add and Mul
data Const a = Const Int
    deriving Functor

data Op a = Add a a | Mul a a
    deriving Functor

-- | Composed type Expr
type Expr = Term (Const :+: Op)

-- | Derivation of smart constructors etc using Template Haskell
$(derive [makeEqF, makeShowF, smartConstructors]
         [''Const, ''Op])
```

C.2.2 Evaluation

Next, we have the evaluation function for all the data type variants that were defined in the module `ExprCompdata`, which is defined in a separate module. This function is implemented using an algebra, just like in the previous section. The notable difference is again the use of Template Haskell, which is used here to derive an instance for the coproduct. Some other small differences are the uses of `Alg` and `cata`, which are defined in the `compdata` library. `Alg f Int` is simply a way to represent an algebra `f Int -> Int`, and `cata` is `compdata`'s version of Data Types à la Carte's `foldTerm`.

```haskell
{-# LANGUAGE TemplateHaskell #-}

module EvalCompdata where

import ExprCompdata

import Data.Comp hiding (Const)
import Data.Comp.Derive

-- | Term evaluation algebra
class Eval f where
  evalAlg :: Alg f Int

-- | Lift the evaluation algebra to a catamorphism
eval :: (Functor f, Eval f) => Term f -> Int
eval = cata evalAlg

-- | Eval instance for constants
instance Eval Const where
  evalAlg (Const i) = i

-- | Eval instance for operations
instance Eval Op where
  evalAlg (Add e1 e2) = e1 + e2
  evalAlg (Mul e1 e2) = e1 * e2

-- | Derive Eval instance for coproduct using Template Haskell
$(derive [liftSum] [''Eval])
```

C.2.3 Expressions as Strings

Then, we have the function `asString`, again implemented without an algebra. This is almost identical to the Data Types à la Carte implementation, except once again for the derivation of the coproduct instance using Template Haskell. Furthermore, the `compdata` function `unTerm` is used in `asString` instead of matching on the structure of a `Term`, since that is hidden in `compdata`.

```haskell
{-# LANGUAGE TemplateHaskell #-}

module AsStringCompdata where

import ExprCompdata
import Data.Comp hiding (Const)
import Data.Comp.Derive

-- | Function asString, written without an algebra
class AsString f where
  asString' :: AsString g => f (Term g) -> String

-- | Unpack the Term and call asString'
asString :: (AsString f) => Term f -> String
asString = asString' . unTerm

-- | AsString instance for constants
instance AsString Const where
  asString' (Const i) = show i

-- | AsString instance for operations
instance AsString Op where
  asString' (Add e1 e2) = "(" ++ asString e1 ++ " + "
                              ++ asString e2 ++ ")"
  asString' (Mul e1 e2) = "(" ++ asString e1 ++ " * "
                              ++ asString e2 ++ ")"

-- | Derive AsString instance for coproduct using Template Haskell
$(derive [liftSum] [''AsString])
```

C.2.4 Negation

The module for negation in straightforward. The data type variant is written in the same way as the previous variants in the module `Expr`, and the instances for `Eval` and `AsString` are the same as before.

```haskell
{-# LANGUAGE TemplateHaskell, FlexibleContexts, DeriveFunctor #-}

module NegationCompdata where

import EvalCompdata
import AsStringCompdata

import Data.Comp
import Data.Comp.Derive

-- | Data type for negation
data Neg a = Neg a
    deriving Functor

-- | Derivation of smart constructor etc using Template Haskell
```

```haskell
$(derive [makeEqF, makeShowF, smartConstructors]
         [''Neg])

-- | Eval instance for negation
instance Eval Neg where
    evalAlg (Neg e) = (-1) * e

-- | AsString instance for negation
instance AsString Neg where
    asString' (Neg e) = "(-" ++ asString e ++ ")"
```

C.2.5 Desugaring

The desugaring function is again written using an algebra and is very similar to how it is written using Data Types à la Carte. The differences are the same as discussed for the Eval example.

```haskell
{-# LANGUAGE TemplateHaskell, TypeOperators, MultiParamTypeClasses,
    FlexibleInstances, FlexibleContexts, UndecidableInstances #-}

module DesugCompdata where

import ExprCompdata
import NegationCompdata

import Data.Comp hiding (Const)
import Data.Comp.Derive

-- | Transformative function desug using algebra
class Desug f g where
  desugAlg :: Alg f (Term g)

-- | Lift the desugaring algebra to a catamorphism
desug :: (Functor f, Desug f g) => Term f -> Term g
desug = cata desugAlg

-- | Default instance of Desug
instance {-# OVERLAPPABLE #-} (f :<: g) => Desug f g where
    desugAlg = inject

-- | Desug instance for negation
instance (Const :<: g, Op :<: g)
        => Desug Neg g where
    desugAlg (Neg e) = iConst (-1) `iMul` e

-- | Derive Desug instance for coproduct using Template Haskell
$(derive [liftSum] [''Desug])
```

C.2.6 Main Module

The main module for the example in `compdata` is identical to the one in the Data Types à la Carte example.

```
{-# LANGUAGE TypeOperators, FlexibleContexts #-}

module Main where

import ExprCompdata
import EvalCompdata
import AsStringCompdata
import NegationCompdata
import DesugCompdata

import Data.Comp hiding (Const)

-- | Examples of type Expr, containing constants, addition and
-- multiplication
threePlusFive :: (Const :<: f, Op :<: f) => Term f
threePlusFive = iAdd (iConst 3) (iConst 5)

twoMulThreePlusFive :: Expr
twoMulThreePlusFive = iMul (iConst 2) threePlusFive

-- | Composed type ExprWithNeg, also containing negation
type ExprWithNeg = Term (Const :+: Op :+: Neg)

-- | Example with negation
threePlusNegFive :: ExprWithNeg
threePlusNegFive = iConst 3 `iAdd` (iNeg (iConst 5))

-- | Evaluation examples
evalAddMul = eval twoMulThreePlusFive

evalAddNeg = eval threePlusNegFive

-- | AsString example
asStringAddNeg = asString threePlusNegFive

-- | Desugar example
desugAddNeg = asString (desug threePlusNegFive :: Expr)

-- | Main, printing results of above examples
main :: IO ()
main = do
    putStrLn "Evaluation examples:"
    print evalAddMul
    print evalAddNeg
    putStrLn "AsString example:"
    putStrLn asStringAddNeg
    putStrLn "Desugar example:"
    putStrLn desugAddNeg
```

C.3 Variations on Variants

Variations on Variants [5], presented in Section 2.4, describes two solutions to the expression problem. One is based on instance chains, a mechanism not implemented in a Haskell compiler, and the other is based on closed type families. The second one is possible to implement in existing Haskell, albeit with the need for many language extensions, so that is what this program will use. The implementation of this is given in the library module `VariationsOnVariants` in Section D.3. In this section, we show how to use the library module for the example program.

C.3.1 Expressions

This module shows the definition of the data type for expressions, containing constants, addition and multiplication, along with functor instances and smart constructors. This is done in the same way as in Data Types à la Carte.

```
{-# LANGUAGE TypeOperators, FlexibleContexts #-}

module ExprVariations where

import VariationsOnVariants

-- | Data type for expression language

-- | Data type variations for Const, Add and Mul
data Const e = Const Int
data Op e = Add e e | Mul e e

-- | Composed type Expr
type Expr = Term (Const :+: Op)

-- | Functors
instance Functor Const
    where fmap _ (Const i) = Const i

instance Functor Op
    where fmap f (Add e1 e2) = Add (f e1) (f e2)
          fmap f (Mul e1 e2) = Mul (f e1) (f e2)
```

```
-- | Smart constructors
iConst :: Const :<: f => Int -> Term f
iConst i = inject (Const i)

iAdd :: Op :<: f => Term f -> Term f -> Term f
iAdd e1 e2 = inject (Add e1 e2)

iMul :: Op :<: f => Term f -> Term f -> Term f
iMul e1 e2 = inject (Mul e1 e2)
```

C.3.2 Evaluation

Next, we have the module for the evaluation function. Instead of defining this through type classes as in previously showcased approaches, it is done by defining a case for each data type variant, and then combine them with a function called `cases`. Each of these has an extra argument for handling the recursive case. For instance, `evalOp` has an argument of type `e -> Int`, which is a function describing how to evaluate the inner part `e` of `Op e`. In the case for constants, there is no recursion, so the type of that argument is irrelevant.

Note that the combination using `cases` means that the function `eval` only works for the cases it is defined for, and a new version must be defined to handle more cases. This is similar to how the closed type `Term (Const :+: Op)` only can contain those data type variants, while a new closed type must be defined to handle other ones. It is, on the other hand, different from the type class approach, which can handle new variants directly, with only a new instance declaration for the variant.

```
module EvalVariations where

import VariationsOnVariants
import ExprVariations

-- | Evaluation of constants
evalConst :: Const e -> r -> Int
evalConst (Const i) r = i

-- | Evaluation of operations Add and Mul
evalOp :: Op e -> (e -> Int) -> Int
evalOp (Add e1 e2) r = r e1 + r e2
evalOp (Mul e1 e2) r = r e1 * r e2

-- | Evaluation function that can evaluate expressions containing
-- constants and operations
eval :: Expr -> Int
eval = cases (evalConst ? evalOp)
```

C.3.3 Expressions as Strings

The module for the `asString` function is analogous to the one for evaluation, since none of these are defined through algebras, which was the difference shown in the previous examples.

```haskell
module AsStringVariations where

import VariationsOnVariants
import ExprVariations

-- | asString of constants
asStringConst :: Const e -> r -> String
asStringConst (Const i) r = show i

-- | asString of operations
asStringOp :: Op e -> (e -> String) -> String
asStringOp (Add e1 e2) r = "(" ++ r e1 ++ " + " ++ r e2 ++ ")"
asStringOp (Mul e1 e2) r = "(" ++ r e1 ++ " * " ++ r e2 ++ ")"

-- | asString function that can handle expressions containing
-- constants and operations
asString :: Expr -> String
asString = cases (asStringConst ? asStringOp)
```

C.3.4 Negation

The module for negation is straightforward to implement in a similar manner to how the other data type variants and function cases were defined. Note that this is independent from the module `ExprVariations`, so it does not contain any composed type nor any version of the functions handling more cases.

```haskell
{-# LANGUAGE TypeOperators, FlexibleContexts #-}

module NegationVariations where

import VariationsOnVariants
import EvalVariations
import AsStringVariations

-- | Data type for negation
data Neg e = Neg e

-- | Functor
instance Functor Neg where
    fmap f (Neg e) = Neg (f e)
```

```haskell
-- | Smart constructor
iNeg :: Neg :<: f => Term f -> Term f
iNeg x = inject (Neg x)

-- | Evaluation of negation
evalNeg :: Neg e -> (e -> Int) -> Int
evalNeg (Neg e) r = (-1) * r e

-- | asString of negation
asStringNeg :: Neg e -> (e -> String) -> String
asStringNeg (Neg e) r = "(-" ++ r e ++ ")"
```

C.3.5 Desugaring

The module for desugaring contains two variants of the `desug` function. The first describes how to handle negation, and the second is the default case, where nothing should be desugared. The type signature for the case of negation, `desugNeg`, is similar to that in Data Types à la Carte, with the exception of the argument `e -> Term f`, for handling recursive expressions. Note the constraints that `Const` and `Op` are part of the result type. When it comes to the type signature for the default case, `desugDef`, the only necessary constraint is that the type in the resulting `Term` is a functor, and otherwise the type signature is simple with a general type `g e` for the variant to be handled.

On the other hand, the type signature for the combining function `desug` is more complicated. This is because, apart from the constraints in `desugNeg` and `desugDef`, it has constraints specifying the subtraction of types that is made. This means that this can actually express that the result type is the result of removing some part (that is, the part that is to be desugared away, in this case `Neg`) of the ingoing type.

```haskell
{-# LANGUAGE TypeOperators, TypeFamilies, FlexibleContexts #-}

module DesugVariations where

import VariationsOnVariants
import ExprVariations
import NegationVariations

-- | Desugaring of negation
desugNeg :: (Const :<: g, Op :<: g)
    => Neg e -> (e -> Term g) -> Term g
desugNeg (Neg e) r = iMul (iConst (-1)) (r e)

-- | Default case for desugaring where there is nothing to desugar
desugDef :: Functor g => g e -> (e -> Term g) -> Term g
desugDef e r = In (fmap r e)
```

```haskell
-- | Desugaring function that removes negation and leaves
-- the rest as it is
desug :: (f :-: Neg ~ g, Without f Neg (Minus f Neg)
          , Op :<: g, Const :<: g, Functor g)
          => Term f -> Term g
desug = cases (desugNeg ? desugDef)
```

C.3.6 Main Module

Last, we have the main module. Apart from the examples seen in previous such modules, we also have to define the versions of the functions that also handle negation, that is, the `evalWithNeg` and `asStringWithNeg`. Then, it is also necessary to make sure to use the correct version of the function when using them, based on the types included in the argument.

```haskell
{-# LANGUAGE TypeOperators, FlexibleContexts #-}

module Main where

import VariationsOnVariants
import ExprVariations
import NegationVariations
import EvalVariations
import AsStringVariations
import DesugVariations

-- | Examples of type Expr, containing constants, addition and
-- multiplication
threePlusFive :: (Const :<: f, Op :<: f) => Term f
threePlusFive = iAdd (iConst 3) (iConst 5)

twoMulThreePlusFive :: Expr
twoMulThreePlusFive = iMul (iConst 2) threePlusFive

-- | Composed type ExprWithNeg, also containing negation
type ExprWithNeg = Term (Const :+: Op :+: Neg)

-- | Evaluation function also considering negation
evalWithNeg :: ExprWithNeg -> Int
evalWithNeg = cases (evalNeg ? (evalConst ? evalOp))

-- | Function asString also considering negation
asStringWithNeg :: ExprWithNeg -> String
asStringWithNeg = cases (asStringNeg ? (asStringConst ? asStringOp))
```

```haskell
-- | Example with negation
threePlusNegFive :: ExprWithNeg
threePlusNegFive = iConst 3 `iAdd` (iNeg (iConst 5))

-- | Evaluation examples
evalAddMul = eval twoMulThreePlusFive

evalAddNeg = evalWithNeg threePlusNegFive

-- | AsString example
asStringAddNeg = asStringWithNeg threePlusNegFive

-- | Desugar example
desugAddNeg = asString (desug threePlusNegFive :: Expr)

-- | Main, printing results of above examples
main :: IO ()
main = do
    putStrLn "Evaluation examples:"
    print evalAddMul
    print evalAddNeg
    putStrLn "AsString example:"
    putStrLn asStringAddNeg
    putStrLn "Desugar example:"
    putStrLn desugAddNeg
```

C.4 Trees that Grow

Trees that Grow [19], as introduced in Section 2.5, is possible directly in Haskell and does not require an extra library or implementation. Therefore, we do not need a module for an implementation of Trees that Grow with the necessary definitions as in the previous examples. Instead, we can directly write the example program. In the following modules, we will show how to extend a data type with a new variant, but we will not consider extensions in the fields of the existing variants.

C.4.1 Expressions

The following module contains the expression data type `Expr`, with variants for `Const`, `Add` and `Mul`, as well as an open constructor for extending the data type, `ExprExt`. The open constructor has an extension field `X_ExprExt`, which is defined through a type family.

The module also contains an undecorated composition of `Expr` that has no extensions, represented by `Expr UD`, where `UD` is simply an empty data type. For the extension field of the open constructor, `X_ExprExt`, we define a type instance for `UD` as `Void`, since there is no extension to the data type.

```haskell
{-# LANGUAGE TypeFamilies #-}

module ExprTrees where

import Data.Void

-- | Data type for expression language

-- | The data type, with extensible fields and
-- one extensible variant
data Expr e = Const Int
            | Add (Expr e) (Expr e)
            | Mul (Expr e) (Expr e)
            | ExprExt (X_ExprExt e)

-- | The extension fields is defined through a type family
type family X_ExprExt e

-- | Undecorated composition of Expr, i.e. with no extension

data UD

-- | The type instance is Void since there is no extension
type instance X_ExprExt UD = Void
```

C.4.2 Evaluation

The general evaluation function is written with an extra argument that tells how to handle an extension. The argument for this is passed along in the recursive calls, but otherwise the function is very similar to a standard, non-extensible function. To evaluate an undecorated expression, we have the function `evalUD`, which just passes `absurd :: Void -> a` as the way to handle the extension, since the type of the undecorated expression has no extensions.

```haskell
module EvalTrees where

import ExprTrees
import Data.Void

-- | Evaluation function for Expr
eval :: (X_ExprExt e -> Int) -> Expr e -> Int
eval _ (Const i)    = i
eval f (Add e1 e2) = eval f e1 + eval f e2
eval f (Mul e1 e2) = eval f e1 * eval f e2
eval f (ExprExt e) = f e
```

```
-- | Evaluation of the undecorated composition of Expr
evalUD :: Expr UD -> Int
evalUD = eval absurd
```

C.4.3 Expressions as Strings

The `asString` function is written analogously to the evaluation function.

```
module AsStringTrees where

import ExprTrees
import Data.Void

-- | Function asString for Expr
asString :: (X_ExprExt e -> String) -> Expr e -> String
asString _ (Const i)    = show i
asString f (Add e1 e2) = "(" ++ asString f e1 ++ " + "
                             ++ asString f e2 ++ ")"
asString f (Mul e1 e2) = "(" ++ asString f e1 ++ " * "
                             ++ asString f e2 ++ ")"
asString f (ExprExt e) = f e

-- | asString of the undecorated composition of Expr
asStringUD :: Expr UD -> String
asStringUD = asString absurd
```

C.4.4 Negation

To add negation, we want to add it in a way such that it can later be extended with more variants. This gives the definition of the extensible data type `Neg e`, which contains variants for negation as `Neg` and more possible extensions as `NegExt`. As usual, the extension field to the open constructor is a type family.

We also define a pattern synonym that can be used instead of having to write the full structure of the composition in which `Neg` is an extension to `Expr`. Hence, we can write `NegP e` instead of `ExprExt (Neg e)`. Note that this is specific for compositions where `Neg` is a direct extension to `Expr`, and we cannot use it for compositions where we have another extension in between the two.

Lastly in this module, we define the `eval` and `asString` functions for `Neg`. This is done through an extensible function similar to `eval` and `asString` in previous modules, but this function has an additional argument for how to handle the recursive call. The recursive call should use a function that describes how to handle the complete composition, not just the case of `Neg`. Therefore, the type for this is `Expr e -> Int` in the evaluation function, since that covers `Expr` along with its extensions.

```haskell
{-# LANGUAGE TypeFamilies, PatternSynonyms #-}

module NegationTrees where

import ExprTrees
import EvalTrees
import AsStringTrees

-- | Data type for Neg, containing Neg for negation, and is also
-- extensible in variants.
data Neg e = Neg (Expr e) | NegExt (X_NegExt e)

-- | The extension field of Neg is defined through a type family
type family X_NegExt e

 -- | Useful pattern synonym
pattern NegP :: (X_ExprExt e ~ Neg e) => Expr e -> Expr e
pattern NegP e <- ExprExt (Neg e)
    where NegP e = ExprExt (Neg e)

-- | Evaluation of Neg
evalNeg :: (Expr e -> Int) -> (X_NegExt e -> Int) -> Neg e -> Int
evalNeg g _ (Neg e)     = (-1) * g e
evalNeg _ f (NegExt e) = f e

-- | asString of Neg
asStringNeg :: (Expr e -> String) -> (X_NegExt e -> String)
    -> Neg e -> String
asStringNeg g _ (Neg e)     = "(-" ++ g e ++ ")"
asStringNeg _ f (NegExt e) = f e
```

C.4.5 Desugaring

The transformative desugaring function is defined similarly to `eval` and `asString`. For the general case, no change is to be made to `Const`, `Add` and `Mul`, other than passing on the argument for how to desugar an extension. The result type is, however, given as a polymorphic type `Expr e2`, that is, some composition with `Expr` as its base.

The desugaring function for `Neg`, `desugNeg`, is similar to the corresponding functions `evalNeg` and `asStringNeg`, but with the return type `Expr e2`. In this case, we construct the result using constructors `Mul` and `Const` from `Expr`. Note that this works since the constructors are in the base type `Expr`, while another structure would be needed for constructors in extensions to specify the chain of extensions. This means that we would need some kind of smart constructors to have the function be fully composable in the return type.

```
{-# LANGUAGE TypeFamilies #-}

module DesugTrees where

import ExprTrees
import NegationTrees
import Data.Void

-- | Desugaring function for Expr
desug :: (X_ExprExt e1 -> Expr e2) -> Expr e1 -> Expr e2
desug _ (Const i)    = Const i
desug f (Add e1 e2) = Add (desug f e1) (desug f e2)
desug f (Mul e1 e2) = Mul (desug f e1) (desug f e2)
desug f (ExprExt e) = f e

-- | Desugaring of the undecorated composition of Expr
desugUD :: Expr UD -> Expr UD
desugUD = desug absurd

-- | Desugaring of Neg
desugNeg :: (Expr e1 -> Expr e2)
    -> (X_NegExt e1 -> Expr e2) -> Neg e1 -> Expr e2
desugNeg g _ (Neg e)     = Mul (Const (-1)) (g e)
desugNeg _ f (NegExt e) = f e
```

C.4.6 Main Module

The main module here contains, as before, examples of constructing values of the data type, as well as functions for evaluating, writing as a string and desugaring such examples.

It also contains the composition of `Expr` and `Neg`. This is given with the extension descriptor `WithNeg`, by defining the type instances for the extension fields of `Epxr` and `Neg`. `Neg` is defined as the extension to `Expr`, with no further extensions. Along with this, we also have the functions for this concrete composition, `evalWithNeg`, `asStringWithNeg` and `desugWithNeg`. Note that `desugWithNeg` also specifies the result type to a concrete composition, `Expr UD`, although this is not strictly necessary. These, or, where suitable, the corresponding functions for the undecorated composition, `evalUD` and `asStringUD`, are the functions we use on the example expressions. The use of these composed functions is similar to in Variations on Variants.

```
{-# LANGUAGE TypeFamilies #-}

module Main where

import ExprTrees
```

```haskell
import EvalTrees
import AsStringTrees
import NegationTrees
import DesugTrees
import Data.Void

-- | Examples of type Expr, containing constants, addition and
-- multiplication
threePlusFive :: Expr e
threePlusFive = Add (Const 3) (Const 5)

twoMulThreePlusFive :: Expr UD
twoMulThreePlusFive = Mul (Const 2) threePlusFive

-- | Composition of Expr and Neg

data WithNeg

-- | Type instance for the original Expr type, now containing
-- Neg as its extension variant
type instance X_ExprExt WithNeg = Neg WithNeg

-- | Type instance for the composition of Neg that is not extended
type instance X_NegExt WithNeg = Void

-- | Evaluation of the composition of Expr where it is extended
-- with Neg, where Neg has no extensions
evalWithNeg :: Expr WithNeg -> Int
evalWithNeg e = eval (evalNeg evalWithNeg absurd) e

-- | asString of the composition of Expr where it is extended
-- with Neg, where Neg has no extensions
asStringWithNeg :: Expr WithNeg -> String
asStringWithNeg e = asString (asStringNeg asStringWithNeg absurd) e

-- | Desugaring of the composition of Expr where it is extended
-- with Neg, where Neg has no extensions
desugWithNeg :: Expr WithNeg -> Expr UD
desugWithNeg e = desug (desugNeg desugWithNeg absurd) e

-- | Example with negation
threePlusNegFive :: Expr WithNeg
threePlusNegFive = Const 3 `Add` (NegP (Const 5))

-- | Evaluation examples
evalAddMul = evalUD twoMulThreePlusFive
```

```
evalAddNeg = evalWithNeg threePlusNegFive

-- | AsString example
asStringAddNeg = asStringWithNeg threePlusNegFive

-- | Desugar example
desugAddNeg = asStringUD (desugWithNeg threePlusNegFive)

-- | Main, printing results of above examples
main :: IO ()
main = do
    putStrLn "Evaluation examples:"
    print evalAddMul
    print evalAddNeg
    putStrLn "AsString example:"
    putStrLn asStringAddNeg
    putStrLn "Desugar example:"
    putStrLn desugAddNeg
```

C.5 Object Algebras

This section will cover how the example program is written with object algebras [21], introduced in Section 2.6. In contrast to the other related work, this is a solution for object-oriented languages, and the example program here is written in Java. It is possible to use object algebras directly in Java, so we do not need a ground implementation, but can go directly into the example.

C.5.1 Expressions

The data type for expressions with constants, addition and multiplication corresponds to an object algebra interface in this solution. This has a generic type `T`, which represents the result type of applying a function to an expression. This is the result type for the interface functions, which each corresponds to a variant of the data type for expressions.

```
// Object algebra interface for expressions
interface ExprAlg<T> {
    T cons(int i);
    T add(T e1, T e2);
    T mul(T e1, T e2);
}
```

C.5.2 Evaluation

In this approach, functions are represented as object algebras. Here, we show an object algebra for evaluation of expressions. This implements the object algebra interface with a concrete type in place of `T`. In this case, it is `Eval`, defined as an interface with a function `int eval()`.

```
// Interface for the evaluation function
interface Eval {
    int eval();
}

// Object algebra for evaluation of expressions
class ExprEval implements ExprAlg<Eval> {
    public Eval cons(int i) { return () -> i; }
    public Eval add(Eval e1, Eval e2) {
        return () -> first.eval() + second.eval();
    }
    public Eval mul(Eval e1, Eval e2) {
        return () -> first.eval() * second.eval();
    }
}
```

C.5.3 Expressions as Strings

As mentioned in Section 2.6, it is possible to write an object algebra in a simpler way, without the use of an interface such as `Eval`, when the interface has a single function taking no arguments and not using mutable state. We present the function `asString` in this way. This means that we use the return type `String` in place of the generic type `T` in the object algebra.

```
// Object algebra for asString of expressions
class ExprAsString implements ExprAlg<String> {
    public String cons(int i) { return String.valueOf(i); }
    public String add(String e1, String e2) {
        return "(" + e1 + " + " + e2 + ")";
    }
    public String mul(String first, String second) {
        return "(" + e1 + " * " + e2 + ")";
    }
}
```

C.5.4 Negation

Now, we want to extend the data type with a variant for negation. This is done by writing an object algebra interface corresponding to the variant. This could either extend the previous interface `ExprAlg`, or be used together with `ExprAlg` in a second interface that extends both. Here, we present it as extending `ExprAlg`. We also need to define how to use the functions `eval` and `asString` on negations. This is done by defining an object algebra for each, `NegEval` and `NegAsString`. These extend the previous object algebras for expressions, `ExprEval` or `ExprAsString`.

```
// Object algebra interface for negation, extending interface for
// expressions
interface NegAlg<T> extends ExprAlg<T> {
    T neg (T e);
}

// Object algebra for evaluation of negation, extending the OA
// for eval of expressions
class NegEval extends ExprEval implements NegAlg<Eval> {
    public Eval neg(Eval e) {return () -> -e.eval();}
}

// Object algebra for asString of negation, extending the OA
// for asString of expressions
class NegAsString extends ExprAsString implements NegAlg<String> {
    public String neg(String e) {return "(-" + e + ")";}
}
```

C.5.5 Desugaring

A transformative function, for instance a desugaring function, is written as an object algebra with a generic type `T` in the object algebra interface, like `ExprAlg<T>`, instead of a concrete type like `Eval` or `String` in the above destructive function examples. Furthermore, the object algebra of the transformative function has a constructor with an object algebra as a parameter, which corresponds to the result type. This parameter has the type `ExprAlg<T>`. For instance, the constructor can be called with an object algebra `new ExprAsString()`, which has the type `ExprAlg<String>`, indicating that the result of desugaring should be written as a `String`.

The desugaring function is here implemented as one object algebra for the original object algebra interface `ExprAlg`, and one for the extension for negation `NegAlg`, where the second extends the first one.

```
// Object algebra for desug of expressions
class ExprDesug<T> implements ExprAlg<T> {
    ExprAlg<T> alg;
    public ExprDesug(ExprAlg<T> alg) {
        this.alg = alg;
    }
    public T cons(int i) { return alg.cons(i); }
    public T add(T e1, T e2) {
        return alg.add(e1, e2);
    }
    public T mul(T e1, T e2) {
        return alg.mul(e1, e2);
    }
}

// Object algebra for desug of negation
class NegDesug<T> extends ExprDesug<T> implements NegAlg<T> {
    public NegDesug(ExprAlg<T> alg) {
        super(alg);
    }
    public T neg(T e) {
        return alg.mul(alg.cons(-1), e);
    }
}
```

C.5.6 Main Module

Constructing examples of an object algebra is written as a function, which takes an object algebra, for instance of type `ExprAlg<T>`, and returns an object of type `T`. The result object is the result of calling the interface functions (such as `cons` and `add` corresponding to data type variants) on the object algebra.

To then, for instance, evaluate such an example, we create an instance of an object algebra, like `new ExprEval()` and gives that as a parameter to the function corresponding to the example to be evaluated, like `twoMulThreePlusFive`. Since the evaluation object algebra here is written using the interface `Eval`, we must also apply its function `eval()` to the resulting object to complete the evaluation. In the case of `AsString`, where we have defined the object algebra with the return type `String` directly, we can simply call, for instance, `threePlusNegFive(New NegAsString())`.

Finally, for desugaring an example, note that the constructor `NegDesug` needs both a generic type and a parameter for the result object algebra, meaning that we construct it as `new NegDesug<String>(new ExprAsString()))`. However, since the generic type can be inferred, we do not need to specify it and can simplify the construction to `new NegDesug<>(new ExprAsString()))`.

```
class Test {

    // Examples of expressions
    <T> T threePlusFive(ExprAlg<T> alg) {
        return alg.add(alg.cons(3), alg.cons(5));
    }

    <T> T twoMulThreePlusFive(ExprAlg<T> alg) {
        return alg.mul(alg.cons(2), threePlusFive(alg));
    }
    <T> T threePlusNegFive(NegAlg<T> alg) {
        return alg.add(alg.cons(3), alg.neg(alg.cons(5)));
    }

    // Main, printing results of examples
    public static void main(String args[]) {
        Test t = new Test();
        t.print();
    }

    // Helper printer function
    void print() {
        System.out.println("Evaluation examples:");
        System.out.println(twoMulThreePlusFive(
                new ExprEval()).eval());
        System.out.println(threePlusNegFive(new NegEval()).eval());
        System.out.println("AsString example:");
        System.out.println(threePlusNegFive(new NegAsString()));
        System.out.println("Desug example:");
        System.out.println(threePlusNegFive(
                new NegDesug<>(new ExprAsString())));
    }
}
```

D

Library Modules for Related Work Examples

In this chapter, we present implementations of some of the solutions to the expression problem not possible to use directly. This means that these form library modules for some of the example programs given in Appendix C, to make those compilable. The exception to compilability is `compdata`, where we present only a portion of the implementation available in the Haskell library.

D.1 Data Types à la Carte

This is a module with the implementation of Data Types à la Carte, containing definitions of the coproduct, injection and other symbols. Most of it should be familiar from Section 2.2.

```
{-# LANGUAGE TypeOperators, UndecidableInstances, FlexibleInstances,
    MultiParamTypeClasses #-}

module DTalC where

-- |  Parameterization of a data type using Term
data Term f = In (f (Term f))

-- | Coproduct
data (f :+: g) e = Inl (f e) | Inr (g e)
infixr :+:

instance (Functor f , Functor g) => Functor (f :+: g) where
    fmap f (Inl e1) = Inl (fmap f e1)
    fmap f (Inr e2) = Inr (fmap f e2)

-- | Fold for Terms
foldTerm :: Functor f => (f a -> a) -> Term f -> a
foldTerm f (In t) = f (fmap (foldTerm f) t)
```

```haskell
-- | Subsumption and injection
class (Functor sub, Functor sup) => sub :<: sup where
    inj :: sub a -> sup a

instance Functor f => f :<: f where
    inj = id

instance (Functor f , Functor g) => f :<: (f :+: g) where
    inj = Inl

instance {-# OVERLAPPABLE #-}
        (Functor f , Functor g, Functor h, f :<: g)
        => f :<: (h :+: g) where
    inj = Inr . inj

inject :: (g :<: f) => g (Term f) -> Term f
inject = In . inj
```

D.2 compdata

Here we include important snippets of code from the `compdata` library. While it contains the same symbols as those in Section 2.2 and 2.3, some have been generalized beyond the simpler definitions that we used when describing Data Types à la Carte. We have still chosen to exclude portions of code. For example, we do not present code for any of the functions used with Template Haskell that were mentioned in this thesis. Parts of the implementation of the subsumption constraint have also been excluded. To see the complete source code of `compdata`, it is recommended to look at the source code repository for that library [13].

```haskell
-- | A form of generalization of Term by compdata
data Cxt :: * -> (* -> *) -> * -> * where
            Term :: f (Cxt h f a) -> Cxt h f a
            Hole :: a -> Cxt Hole f a
data Hole
data NoHole

-- | A Term as corresponding to Data types à la Carte
type Term f = Cxt NoHole f ()

-- | A helper function for unwrapping a Term
unTerm :: Cxt NoHole f a -> f (Cxt NoHole f a)
{-# INLINE unTerm #-}
unTerm (Term t) = t


infixr 6 :+:
```

```haskell
-- | The coproduct definition
data (f :+: g) e = Inl (f e)
                 | Inr (g e)

instance (Functor f, Functor g) => Functor (f :+: g) where
    fmap f (Inl e) = Inl (fmap f e)
    fmap f (Inr e) = Inr (fmap f e)


infixl 5 :<:
infixl 5 :=:

-- | Subsumption criterion, which uses
--   closed type families to search for f in g
type f :<: g = (Subsume (ComprEmb (Elem f g)) f g)

type f :=: g = (f :<: g, g :<: f)

data Emb = Found Pos | NotFound | Ambiguous
data Pos = Here | Le Pos | Ri Pos | Sum Pos Pos
data Proxy a = P

-- | Class that uses the result e
--   for a search of f in g to implement inject and project
class Subsume (e :: Emb) (f :: * -> *) (g :: * -> *) where
  inj' :: Proxy e -> f a -> g a
  prj' :: Proxy e -> g a -> Maybe (f a)

-- Instances to Subsume that unpack the result
-- will not be shown here, nor will we show
-- the type families ComprEmb or Elem that produce this result.
-- If you want to know how these work,
-- you should look at the source code repository for compdata.

inj :: forall f g a . (f :<: g) => f a -> g a
inj = inj' (P :: Proxy (ComprEmb (Elem f g)))

proj :: forall f g a . (f :<: g) => g a -> Maybe (f a)
proj = prj' (P :: Proxy (ComprEmb (Elem f g)))

-- | This definition of injection is slightly simplified
--   from what can be found in compdata source code
inject :: (g :<: f) => g (Cxt h f a) -> Cxt h f a
inject = Term . inj
```

```
-- | This definition of projection is slightly simplified
--   from what can be found in compdata source code
project :: (g :<: f) => Cxt h f a -> Maybe (g (Cxt h f a))
project (Hole _) = Nothing
project (Term t) = proj t



-- | A type representing an algebra function
type Alg f a = f a -> a

-- | Catamorphism for applying an algebraic function onto a Term
cata :: forall f a . (Functor f) => Alg f a -> Term f -> a
{-# NOINLINE [1] cata #-}
cata f = run
    where run :: Term f -> a
          run   = f . fmap run . unTerm
```

D.3 Variations on Variants

This section contains a module of the closed type families implementation of Variations on Variants. The definitions of `Term` and the coproduct are familiar from previously mentioned solutions. Then, for membership test, injection and remainder, we have implementations of proof-searches that are used to define the operations. Note that remainder (`:-:`), meaning that it gives the part remaining after removing a type from a composed type, is introduced in this implementation, while it is not present in Data Types à la Carte or `compdata`.

What is more important than the details of this implementation is how it is used. This is why the last definition in this module, `cases`, is important to note, because that is what is used when defining functions.

```
{-# LANGUAGE TypeFamilies, UndecidableInstances, ExplicitForAll,
    TypeOperators, ScopedTypeVariables, ConstraintKinds,
    FlexibleContexts, FlexibleInstances, MultiParamTypeClasses #-}

module VariationsOnVariants where

-- | Term, the fixed point recursive knot
data Term e = In (e (Term e))


-- | Coproduct
data (f :+: g) e = Inl (f e) | Inr (g e)
```

```
-- | Functor instance for the coproduct
instance (Functor f , Functor g) => Functor (f :+: g) where
    fmap f (Inl e1) = Inl (fmap f e1)
    fmap f (Inr e2) = Inr (fmap f e2)


-- | Branching operator
-- Given functions for f and g respectively,
-- return function for their coproduct
(<?>) :: (f e -> a) -> (g e -> a) -> (f :+: g) e -> a
(f <?> g) (Inl x) = f x
(f <?> g) (Inr x) = g x


-- | Auxillary empty data types
data Yep
data Nope

-- | Membership test
type family IsIn f g where
    IsIn f f         = Yep
    IsIn (f :+: f') (g :+: g') = Or (Or (IsIn (f :+: f') g)
                                        (IsIn (f :+: f') g'))
                                    (And (IsIn f (g :+: g'))
                                         (IsIn f' (g :+: g')))
    IsIn f (g :+: h) = Or (IsIn f g) (IsIn f h)
    IsIn f g         = Nope

-- | Auxillary type family for disjunction
type family Or b c where
    Or Nope Nope = Nope
    Or b c       = Yep

-- | Auxillary type family for conjunction
type family And b c  where
    And Yep Yep = Yep
    And b c     = Nope


-- | More auxillary empty data types
data Refl
data L x
data R x

-- | Proof search for type-level witness for subsumption
type family Into f g where
```

```
    Into f f          = Refl
    Into f (g :+: h) = Ifi (Into f g) (IsIn f h)
                           (Into f h) (IsIn f g)
    Into f g          = Nope

-- | Auxillary witnesess
type family Ifi lp inr rp inl where
    Ifi Nope inr Nope inl = Nope
    Ifi Nope inr rp Nope  = R rp
    Ifi lp Nope rp inl    = L lp
    Ifi lp inr rp inl     = Nope

-- | Injection, using witness p of the proof that f is summand of g
class Inj f g p where
    injp :: p -> f e -> g e

instance Inj f f Refl where
    injp _ = id

instance Inj f g p => Inj f (g :+: h) (L p) where
    injp (_ :: L p) = Inl . injp (undefined :: p)

instance Inj f h p => Inj f (g :+: h) (R p) where
    injp (_ :: R p) = Inr . injp (undefined :: p)

-- | Injection function, hiding the type-level witness Into
inj :: forall f g e. (Inj f g (Into f g)) => f e -> g e
inj = injp (undefined :: Into f g)

-- | Auxillary function to simplify injection
inject :: (Inj f e (Into f e)) => f (Term e) -> Term e
inject = In . inj


-- | More auxillary data types
data Onl (h :: * -> *)
data Onr (h :: * -> *)
data Le (g :: * -> *) p
data Ri (f :: * -> *) p
data Found (f :: * -> *)

-- | Proof search for witness for subtracting types in coproducts
type family Minus f g where
    Minus f f          = Nope
    Minus (f :+: g) f = Onl g
    Minus (f :+: g) g = Onr f
```

```haskell
    Minus (f :+: g) h = Ifm g (Minus f h) (IsIn f g)
                            f (Minus g h) (IsIn f h)
    Minus f g         = Found f

-- | Auxillary witnesses
type family Ifm g lp inr f rp inl where
    Ifm g Nope inr f Nope inl = Nope
    Ifm g Nope inr f rp Nope  = Ri f rp
    Ifm g lp Nope f rp inl    = Le g lp

-- | Extract the type from a proof search done with Minus
type family OutOf p where
    OutOf (Onl x)  = x
    OutOf (Onr x)  = x
    OutOf (Le f p) = OutOf p :+: f
    OutOf (Ri f p) = f :+: OutOf p

-- | Branching operator, using witness p of the proof that g
-- can be removed from f
class Without f g p where
    (??) :: (g e -> r) -> (OutOf p e -> r) -> p -> f e -> r

instance Without (f :+: g) f (Onl g) where
    (m ?? n) _ = m <?> n

instance Without (f :+: g) g (Onr f) where
    (m ?? n) _ = n <?> m

instance Without f h p => Without (f :+: g) h (Le g p) where
    (m ?? n) (_ :: Le g p) = (m ?? (n . Inl)) (undefined :: p)
        <?> (n . Inr)

instance Without g h p => Without (f :+: g) h (Ri f p) where
    (m ?? n) (_ :: Ri f p) = (n . Inl)
        <?> (m ?? (n . Inr)) (undefined :: p)

-- | Wrapper branching operator, hiding the Minus witness
(?) :: forall f g e r. Without f g (Minus f g) => (g e -> r)
    -> (OutOf (Minus f g) e -> r) -> f e -> r
m ? n = (m ?? n) (undefined :: Minus f g)


-- | Shorthands for subtraction and subsumption
type f :-: g = OutOf (Minus f g)
type f :<: g = Inj f g (Into f g)
```

```haskell
-- | cases, used to evaluate using different function variants
-- e.g. cases (evalConst ? evalOp)
cases :: (e (Term e) -> (Term e -> t) -> t) -> Term e -> t
cases cs = f where f (In e) = cs e f
```