Dissertation Type: research

DEPARTMENT OF COMPUTER SCIENCE

Extensible, Scoped Domain Specific Languages

Albie Baker-Smith

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Monday 8th July, 2019

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Albie Baker-Smith, Monday 8th July, 2019

Contents

Executive Summary

Programming languages are one of the main ways people interact with computers. General Purpose Languages, such as Java or Haskell, need to be able to solve a variety of problems. This fact can be seen in their syntax, which is generic enough to solve problems in any domain. A downside of this is that solving a problem in a *specific* domain can lead to verbose code. Domain Specific Languages are languages with syntax tailored to a specific domain, allowing programs to be more concise and readable, such as HTML or SQL.

To avoid needing to create new tools, Domain Specific Languages are often *embedded* in a host language. However, the design of such a language needs to be carefully considered in order to afford two important aspects of programming languages: 1) The ability to compose a DSL from easily extensible pieces allows them to be extended with new features after-the-fact. 2) The ability to represent scoping constructs is also vital, as these are common in the majority of high-level languages.

New methods of represented scoping constructs have been proposed, however, it has not yet been demonstrated how these techniques can be used to create modular and extensible DSLs. This is the problem this project aims to solve. More concretely, the project will explore the following:

- How the new methods to represent scope can also be used to create modular DSLs, which will be done by applying methodologies of Data Types à la Carte and Effect Handlers.

- How the use of modular, scoped DSLs can increase code reuse, which will be done through the creation of various 'low-level' DSLs (e.g. Reader, Writer, State, Exceptions) and used to implement "high-level" DSLs.

- Investigating how the techniques scale to larger, more complex programs in order to expose problems that would otherwise not be encountered, which will be done through the creation of a compiler.

- Investigate the runtime characteristics of the created DSLs, an important factor for wider audience uptake, which will be done by performing various benchmarks.

Supporting Technologies

- I used *Haskell* (GHC 8.0.2, Cabal 1.24.0.2), and a number of libraries:
  - *Megaparsec* for implementing a parser for While.
  - *HSpec* for unittesting.
  - *Criterion* for benchmarking runtimes.
  - *Weigh* for benchmarking memory allocation.
- I used NodeJS to run compiled WebAssembly programs.

Acknowledgements

I wish to thank the following people for their help and support:

- My parents for their help and support.
- Jamie Willis for giving up time to look over this thesis.
- My supervisor, Nick, for helping me find this project and his invaluable guidance and discussion during my work.

Chapter 1

Introduction

General Purpose Programming Languages are a great tool for solving a variety of different problems. However, this ability can also be a hindrance - in trying to express the problem of a specific domain their general syntax may lead to verbose code. They may also afford too much power to the programmer, allowing them to write code which does not represent a legal action in the specific domain. Domain Specific Languages (DSLs) aim to solve this by providing constrained syntax and power. To name a few examples of such languages: HTML is used for representing a formatted document; Gherkin is a language used to test software behaviour and is tailored to be very human readable; Parser Combinator libraries, such as MegaParsec, are used to describe parsers. However, creation of new languages is not an easy task, and so DSLs are often embedding in a host language - the syntax and semantics of the DSL are expressed in terms of the host language, for example, through defining functions to represent each piece of syntax. Use of DSLs is becoming increasingly popular due to allowing more concise code to be written that is being easier to maintain, and easier to reason about when compared to general purpose languages.

The use of modularity has been described as being the key to successful programming [8]. Recursively splitting a problem into smaller pieces allows individual modules to be more easily understood, better encapsulate a single goal, lead to more reuse, and allow for easier testing. Closely associated with this, another important design aspect is extensiblity, or the ability to add features to a program after it has been written in order to solve new problems. By constructing programs in a modular manner, extensibility often follows along. Extensibility is a desirable feature of programming languages, as it allows them to remain current and solve problems in their initial designs. Examples of languages being extended are bountiful, to name a few: Java 8 introduced functional features to the language; Swift has had five releases that the time of writing; even languages such as C, which seem timeless, have been extended.

With the promises use of DSLs makes, and the importance of modularity and extensibility, it is unsurprising others have worked to combine them. However, many existing methods of constructing embedded DSLs using these methods can have problems when trying to represent scoped constructs [15], i.e. syntax which contains other syntax. This is a vital feature in many programming languages and DSLs. For example, modern general purpose programming languages contain scoped constructs such as functions, control-flow structures, and classes. DSLs, such as HTML and SQL, also contain scoped constructs, such as div tags and case statements, respectively. Even unconventional DSLs, such as that which describe exceptions contain scoped syntax used to catch exceptions. Because of this, work has been done to

allow the creation of modular DSLs containing scoped syntax. However, these methods can have problems faithfully representing structure, or hindering development due to complexity [4].

To solve these issues, new a method of representing scoped constructs was been proposed [12], which makes use of a new datatype `Prog` in order to correctly encode scope. However, methods to allow DSLs created using `Prog` to be created in a modular and extensible manner have not yet been explored. This project aims to fill this gap and extend the capabilities Prog, allowing it to be used to create modular, extensible DSLs. A case study is also performed into how well this method works in a larger project - a compiler from While to WebAssembly - with the aim to find other areas `Prog` is applicable and expose problems that otherwise would not be encountered.

To outline, the overarching context of each chapter will be extending the capabilities of the compiler using new techniques and methods investigated during the chapter. To give background, Chapter 2 explores different methods used to embedded a DSL into a host language. The advantages and drawbacks of each method are assessed and used as motivation for the use of `Prog`. Chapter 3 demonstrates how `Prog` can be applied to a real-world application - pretty printing text. Chapter 4 takes a look at applying Data Types à la Carte to `Prog` in order to create extensible DSLs. Use of the method is demonstrated by representing the abstract syntax trees of the source and target languages of the compiler. Chapter 5 discusses some of the issues with Data Types à la Carte and shows how Effect Handlers can be used to solve these. Using Data Types à la Carte and Effect Handlers provides ways to create extensible DSLs, however, specification of their semantics can still be cumbersome. Chapter 6 demonstrates how this can be solved by expressing the semantics of a DSL in terms of another DSL. Using this method, DSLs are created to allow While to be extended with local variables. Chapter 7 discusses some problems encountered using `Prog` when attempting to extend the source language with local variables, as well as some possible solutions. In Chapter 8, the methods of creating extensible, scoped DSLs explored in previous chapters are combined together in order to create more complex DSLs, allowing the compiler to handle compilation of functions. Up until this point, the discussion has been focused on *writing* code, Chapter 9 shifts this and takes a look at the runtime characteristics of DSLs constructed using `Prog`. Chapter 10 explores whether the goal of creating modular, scoped DSLs has been achieved. Finally, Chapter 11 sums up the findings and concludes with a discussion of future work.

Chapter 2

Background

This chapter covers the background of representing Domain Specific Languages in a host language, Haskell in this case. Throughout the project, a compiler will be created as a case study into creating DSLs. This chapter will first give an introduction to the source and target languages of this compiler, While and WebAssembly respectively, and then go on to discuss the different methods of representing these languages as DSLs in Haskell.

2.1 Source and Target Languages

This section introduces the source and target languages of the compiler that will be constructed, giving the BNF and example programs written in each.

2.1.1 While

While is a simple, high-level, imperative language with standard features such as global variables, if-statements, and while-loops [11]. To allow programs output values an `export` statement is also added, which is analogous to returning a value from the main function in a C program. Further extensions can be added such as local variables and procedures which are implemented in Chapters 5 and 8, respectively. For the moment, the core of While can be represented with the BNF below, where quotes are used to delimit terminals:

```
Num  : ('0'-'9')+
Var  : ('a'-'z')+
AExp : Num | Var | AExp '+' AExp | AExp '-' AExp | AExp '*' AExp
BExp : 'true' | 'false' | AExp '=' AExp | AExp '<=' AExp
     | '!' BExp | BExp '&&' BExp
Stm  : Var ':=' AExp | 'skip' | Stm ';' Stm
     | 'if' BExp 'then' Stm 'else' Stm | 'while' BExp 'do' Stm
     | 'export' AExp
```

Even with a small number of features, interesting programs can still be created, such as the program below which computes 6! and returns the result to the user:

```
r := 1;
n := 6;
while !(n = 1) do (
    r := r * x;
    n := n - 1
);
export r
```

The language's simplicity, coupled with extensions to its syntax and semantics, make it a good fit for investigating it can be represented as a modular DSL.

2.1.2 WebAssembly

WebAssembly (WASM) is a portable instruction format for a stack-based virtual machine [7]. JavaScript is the only language to run natively on the Web and so it has been used for tasks it is not ideally suited to, such being a compilation target, or performance hungry applications like 3D visualisation. WebAssembly aims to solve these problems through having a compact representation and little to no runtime overhead.

A subset of the BNF of WebAssembly given below. The instructions revolve around pushing and popping values from the stack, or are scoped, control flow instructions such as if-statements, blocks, and loops (discussed further in Chapter 8):

```
Int  : ('0'-'9')+
WASM : 'i32.const' Int
     | 'i32.load' | 'i32.store'
     | 'get_local' | 'set_local'
     | 'if' WASM 'else' WASM 'end'
     | 'block' WASM 'end'
     | 'loop' WASM 'end'
     | 'return'
```

This can be used to construct programs such as that below, which loads the value at some address. The program contains a function `load_val` which takes two parameters `base` and `offset`. These two parameters are added together by pushing them onto the stack followed by an add instruction. Next, a load instruction is used to pop the address off the top of the stack, and then push the value at the address onto the stack. Finally, the function returns the value on the top of the stack to the caller.

```
(func $load_val (param $base i32) (param $offset i32)
    get_local $base
    get_local $offset
    i32.add
    i32.load
    return
)
```

Instead of jumps, WebAssembly also contains scoped syntax. The snippet below shows how an if-statement can be used to returns `1` or `0` depending on whether the local variable `x` is equal to `1`:

```
get_local $x
if
    i32.const 1
    return
else
    i32.const 0
    return
end
```

For investigating modular, scoped DSLs WebAssembly makes a good choice as, unlike most assembly languages, WebAssembly contains scoped syntax such as if-statements, blocks, and loops. The language is also currently a Minimum Viable Product with more features scheduled for release [5], therefore, also it appropriate for representation using a DSL which can be easily extended.

2.2 Domain Specific Languages

A domain specific language, or DSL, is a language tailored to a specific domain. Through the use of specialised syntax, programs written using a DSL can be made more concise and readable compared to the same program written using a general purpose language.

This section takes a subset of While and WebAssembly and explores how each can be represented as a DSL embedded in Haskell, as well as how each method affects modularity and the ability to define syntax and semantics. To explore how syntax can be represented using different methods, each subsection will implement a DSL representing While and/or WebAssembly. To explore how semantics can be defined, each section also presents compilation from While to WebAssembly. To motivate the introduction of each method, the merits and downsides of each are also discussed.

2.2.1 Shallow Embeddings

A shallow embedding immediately translates the source language into the target language, which in Haskell is represented through the use of functions. Pretty printing WebAssembly as a string could be represented as a function from the number of tabs to indent by, to a string:

```haskell
type Indent = Word
type WASM = Indent -> String
```

Because of the immediate translation, the functions used to define the syntax of the language also define the semantics. The syntax and semantics of pushing a 32-bit integer into the WebAssembly stack can be represented as a function `i32Const` which returns a pretty printed string representing this instruction:

```haskell
i32Const :: Int -> WASM
i32Const n i = indent i ++ "i32.const " ++ show n ++ "\n"

indent :: Indent -> String
indent i = replicate (fromIntegral (i*2)) ' '
```

Using a shallow embedding makes it easy to define the initial semantics of a DSL. Compiling While to WebAssembly is done by simply defining a function from While to WebAssembly. For example, to compile While numbers into pushing a number onto the WebAssembly stack is simply a case of calling `i32Const`:

```
type While = WASM

num :: Int -> While
num = i32Const
```

Shallow embeddings are also very modular and extensible in terms of syntax - new syntax can be added to the DSL by creating new functions. Adding an operator to compose to WebAssembly instructions together requires no modification to existing instructions:

```
thenDo :: WASM -> WASM -> WASM
thenDo w1 w2 i = w1 i ++ w2 i
```

However, this method suffers when trying to *change* the semantics of the language as the type of every function must be changed. For example, for WebAssembly to make use of local variables they must be predefined at the start of a function. To compile this requires collating together the variables used in a While program, therefore, the type of `While` need to be modified to be a tuple of WebAssembly instructions *and* a set of local variables:

```
type Locals = Set String
type While = (WASM, Locals)
```

Because the type representing While was changed, functions representing the While DSL, such as `num`, must also be modified to return a tuple of results:

```
num :: Int -> While
num n = (i32Const n, Set.empty)
```

In order to add new meaning, the tuple of results must be extended further, making for unweildy and bloated code. This also disallows the semantics to be changed if the programmer does not have access to modify the original source. In summary, shallow embeddings allow for the syntax of a language to be extended but make it hard to extend the semantics making method is inappropriate to use in many cases.

2.2.2 Deep Embeddings

Deep embeddings aim to solve a shallow embedding's non-extensible semantics by representing the DSL as a data structure. This allows different semantics over the same structure to be defined by creating different functions over the structure. Representing a subset of statements in While can be done with data type below, where each constructor represents a piece of syntax:

```
data Stm
    = If BExp Stm Stm
    | Export AExp
    | Comp Stm Stm
```

Different semantics can be given to the DSL by using different functions over the data type. Assuming a deep embedding of WebAssembly, and functions to compile arithmetic and boolean expressions, compiling While to WebAssembly can be done by defining a mapping from While syntax to WebAssembly syntax:

```
compile :: Stm -> WASM
compile (If b t e)   = compileBExp b `Seq` PopIf (compile t) (compile e)
compile (Export x)   = compileAExp x `Seq` Ret
compile (Comp s1 s2) = compile s1    `Seq` compile s2
```

With the shallow embedding, it was difficult to change the semantics of the DSL and add local variables. However, this is trivial using a deep embedding and can be done by creating a new function over the DSL:

```
localVars :: Stm -> Set String
localVars (If b t e)   = Set.unions [localVarsBExp b, localVars t, localVars e]
localVars (Export x)   = localVarsAExp x
localVars (Comp s1 s2) = Set.union (localVars s1) (localVars s2)
```

Unfortunately, this makes it hard to extend the *syntax* of the language as doing so requires modification to every function which pattern matches on the datatype representing the DSL's syntax. For example, to support a variable assignment in While requires modification to the original data structure:

```
data Stm =
    ...
    | Assign String AExp
```

and therefore modification to both `compile` and `localVars`. This is infeasible if the programmer does not have access to modify the original code. In summary, deep embeddings allow the semantics of a DSL to be easily extended and hence created in a modular fashion. However, changing the syntax requires changing the original data structure, therefore making the syntax of deep embeddings hard.

2.2.3 Data Types à la Carte

Deep embeddings provide a way to easily give new semantics, however, the method suffers when trying to add new syntax. However, the use of Data Types à la Carte can be used to solve this problem [13]. In order to not concretely set the syntax of a DSL, `Fix` is used. Notice how this appears to create an infinitely telescoping typing:

```
newtype Fix f = In { unIn :: f (Fix f) }
```

The type `Fix` takes a type parameter `f` representing the signature of the syntax allowed in the DSL. For example, the syntax of While statements can be represented as data structure `Stm` with a type parameter `k`, below. By creating the type `Fix Stm`, the type parameter `k` in the subtrees of statements become other statements. Therefore, `k` represents the points of recursion in the DSL. To tie the recursive knot, the `Nop` constructor is used, which contains no subtrees.

```
data Stm k
    = Nop
    | If k k k
    | Export k
    | Comp k k
```

Using this 'programs' can be created using a combination of `In` and the constructors of `Stm`:

```
ex :: Fix Stm
ex = In (Comp (In Nop) (In Nop))
```

The main point to notice is that the type that will be present a subtrees of `Stm` is `k` and so is not fixed - it can be any type. Currently only syntax of statements can be used in the subtrees, however, other syntax can be *composed* in in order to solve this problem: The notion of composing the syntax of two DSLs, `f` and `g`, is captured by the coproduct `(:+:)`, which contains two constructors to 'box' together syntax of `f` and `g` into a single type:

```
data (f :+: g) a = L (f a) | R (g a)
```

The coproduct can be used to compose syntax together, and hence compose the syntax allowed in the DSL:

```
ex :: Fix (AExp :+: BExp :+: Stm)
ex = In (R (R (Export (In (L (Num 1))))))
```

This solves the problem of extending the syntax of a deep embedding. Extra benefits are also afforded: as well as adding syntax, syntax can also be subtracted by omitting it from the composed type signature. This is a useful feature to precisely describe the transformations that occur to the AST during compilation.

Unfortunately, use of the coproduct and `Fix` makes it much harder to write programs using the DSL as it requires use of `L`, `R`, and `In`. Writing these expressions by hand is laborious and creates brittle code, for example, if the types composed together change, the combination of `L` and `R` may also change. Fortunately, choosing the combination of `L` and `R` can be automated through the use of the subtyping relation [13], written as `sub :<: sup` and read as any signature `sup` which supports `sub`. This is defined using the following typeclass which defines a single function `inj`, short for inject, which is used to inject a sub-type into a super-type:

```
class (Functor sub, Functor sup) => sub :<: sup where
    inj :: sub a -> sup a
```

The instances of the typeclass below are used to perform a linear search from left to right through a type composed using coproducts. This allows the correct combination of `L` and `R` to be selected to inject a type into a signature which supports it. The first instance says that `(:<:)` is reflexive:

```
instance f :<: f where
    inj = id
```

The instance `f :<: (f :+: g)` serves as the base case of the linear search, and the final instances serves as the recursive case:

```
instance f :<: (f :+: g) where
    inj = L

instance f :<: g => f :<: (h :+: g) where
    inj = R . inj
```

To aid with injection into `Fix`, the function `inject` can be used, which simply uses inject then wraps the result inside `Fix`:

```
inject :: f :<: g => f (Fix g) -> Fix g
inject = In . inj
```

The use of `inject` precisely solves the problem of writing brittle, hard to read code due to a cloud of `L`, `R`, and `In`. However, the code is still not entirely readable due to use of `inject` before each constructor:

```
ex :: Fix (AExp :+: BExp :+: Stm)
ex = inject (Export (inject (Num 1)))
```

To solve this, smart constructors which automatically insert a type into a signature can be created for each piece of syntax. Through the use of subtyping, the smart constructors can work for any type that supports the required syntax, therefore increasing code reuse. For example, the smart constructor for export statements and constants are given below and work to inject into any type which supports statements and arithmetic, respectively.

```
export :: Stm :<: f => Fix f -> Fix f
export x = inject (Export x)

num :: AExp :<: f => Fix f -> Fix f
num n = inject (Num n)
```

The example can finally be cleaned in order to write readable, extensible DSLs:

```
ex :: Fix (AExp :+: BExp :+: Stm)
ex = export (num 1)
```

The problem now is defining the semantics of a DSL with syntax defined using `Fix` and coproduct. To achieve this, `cata` can be used to fold from the leaves of a Fix tree upwards, folding the tree it into some result type `a`.

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg = alg . fmap (cata alg) . inop
```

This is analogous to folding over a list using `foldr`:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

The first parameter to both `cata` and `foldr` is an *algebra*, which is used while folding in order to transform one level of the recursive structure. One advantage of using these functions is that algebra does not need to deal with any recursion, and so an entire class of bugs can be eliminated.

Only using `cata`, an algebra could be created which used `L` and `R` to choose the specific syntax in a DSL to give semantics to. However, this again leads to brittle code as changes to the composed type would require changes to the exact pattern of `L` and `R`.

Instead, by using Haskell's typeclass system, algebras for the semantics of different pieces of syntax be automatically assembled. The typeclass `Alg` is used to define an algebra given to `cata` and used to give semantics to a Fix tree. Here, `f` represents the signature of the syntax to give semantics to, and `a` represents the type resulting from folding over a tree containing said syntax:

```
class Functor f => Alg f a where
    alg :: f a -> a
```

By defining an instance of `Alg` for coproduct, algebras for individual pieces of syntax can be automatically composed together to create an algebra over a tree containing many different pieces of syntax. This is done by either applying the algebra for `f` or `g`.

```
instance (Alg f a, Alg g a) => Alg (f :+: g) a where
    alg (L x) = alg x
    alg (R x) = alg x
```

This machinery allows the semantics to be defined in a modular manner, therefore, allowing semantics of a tree to be extended without modification to any original code. For example, compilation from While statements to WebAssembly is demonstrated below. To define different semantics would be a case of defining a new typeclass instance.

```
instance Alg Stm (Fix WASM) where
    alg (If cond t e) = cond `thenDo` popIfElse t e
    alg (Export x)    = x `thenDo` ret
    alg (Comp s1 s2)  = s1 `thenDo` s2

compile :: Fix While -> Fix WASM
compile = cata alg
```

This approach works well for representing some DSLs, however, Haskell programs often make use of do-notation in order to cleanly express sequential logic. Currently, this ability is lacking as `Fix` cannot be made into a monad. Therefore making programs written using these DSLs less succinct, such as with the use of `thenDo` in the example above. To solve this will require definition of a new datatype.

In summary, Data Types à la Carte allows modular specification of syntax *and* semantics, helping solve the problems encountered with shallow and deep embeddings. However, `Fix` does not provide an ideal basis for expressing sequential computations.

2.2.4 Free

The datatype `Fix` suffered from not being able to cleanly express sequential logic. Monads capture the essence of sequential computations, however, they are difficult to combine [13]. By restricting the syntax of the language to be represented using the Free monad (instead of Fix), it becomes easier to combine them using the coproduct.

The recursive definition of the Free monad is defined very much like `Fix`, but with an extra constructor `Var` to represent leaves of a tree:

```haskell
data Free f a = Var a | Op (f (Free f a))
```

By making `f` a functor, the Free monad can becomes a Monad for free!

```haskell
instance (Functor f) => Monad (Free f) where
    return = Var
    Var a >>= k = k a
    Op  m >>= k = Op (fmap (>>= k) m)
```

Since a monad can now be used to describe the syntax of a DSL, sequential computations can be more easily expressed. When designing the data types representing DSL syntax, the notion of one instruction followed by another manifests as each constructors containing the remaining program, or *continuation*. Below demonstrates how a subset of WebAssembly can be described in this way. Each constructor contains a parameter `k` representing the remaining program:

```haskell
data Instr k
    = Ret k
    | SetLocal String k
    deriving Functor
```

Like with `Fix`, smart constructors can be used to allow more readable DSLs program to be written.

```haskell
ret :: Instr :<: f => Free f ()
ret = inject (Ret (Var ()))
```

Notice the continuation is `Var ()`, making it appear like the program would incorrectly stop upon reaching this point. However, `x >>= k` essentially acts to substitute the leaves (`Var`) of `x` with the tree created by `k`. In this way, bind captures 'running' one program followed by another.

By defining appropriate smart constructors which inject into a Free tree, it becomes much easier to describe a WebAssembly program in a form that looks like source code. The use of do-notation uses bind to construct a DSL where the continuations contain the remaining program.

```haskell
ex :: Free (ArithInstr :+: Instr) ()
ex = do
    i32Const 1
    getLocal "x"
    i32Add
    setLocal "y"
```

In summary, the Free monad can be used with Data Types à la Carte to create DSLs with modular syntax and semantics, but which also cleanly represent sequential computations. However, there are problems using Free. Notice in the example above that no scoping constructors were used. WebAssembly contains scoped syntax, such as `loop`. The intention here was to use the first `k` to represent the body of the loop, and the second represents the remainder of the program:

```
data ControlInstr k = Loop k k
```

Due to how the functor instance is implemented for `Loop`

```
instance Functor ControlInstr where
    fmap f (Loop body k) = Loop (f body) (f k)
```

and how Monad is implemented for `Free`

```
instance (Functor f) => Monad (Free f) where
    ...
    Op  m >>= k = Op (fmap (>>= k) m)
```

the statement `Loop body r >>= k` becomes `Loop (body >>= k) (r >>= k)`. This is undesirable as it will cause the continuation `k` to be pushed up inside the body of the loop. If executing this program, the remainder of the program would be run every loop iteration. This undesired behaviour is discussed further in Chapter 3. Clearly, Free cannot properly handle scoped constructs when used with continuations - a new datatype is required.

2.2.5 Prog

Using the Free monad with Data Types à la Carte allowed the creation of modular DSLs with the ability to cleanly represent sequential operations. However, the capability to represent scoped constructs was lacking. To solve this problem, a new datatype `Prog` can be created [12]:

```
data Prog f g a
    = Var a
    | Op (f (Prog f g a))
    | Scope (g (Prog f g (Prog f g a)))
```

This is essentially the Free monad, but with an extra constructor for creating scoped syntax. The leaves of the `Op` constructor contain `a`, whereas the leaves of the `Scope` constructor contain entire programs `Prog f g a` representing nested syntax.

Also resembling the Free monad, if both `f` and `g` are functors, `Prog f g` is a Monad, therefore allowing it to cleanly represent DSLs containing sequential computations. Again, bind essentially works like tree substitution - replaces occurences of `Var x` with the tree constructed by `f x`:

```
instance (Functor f, Functor g) => Monad (Prog f g) where
    return = Var
    Var x    >>= f = f x
    Op op    >>= f = Op (fmap (>>=f) op)
    Scope sc >>= f = Scope (fmap (fmap (>>=f)) sc)
```

As an introduction to creating DSLs using `Prog`, the Reader monad will be implemented using `Prog`, with While and WebAssembly being implemented in Chapter 4 once the necessary Data Types la a Carte machinery is in place. The Reader monad represents a computation which can retrieve the value of the environment `r` using *ask*. This can be represented using the following syntax where `Ask` contains a function used to pass the value of the environment to the continuation.

```
data Ask r k = Ask (r -> k)
```

A local scope can also be entered using *local* x in which querying the environment will return x. This is represented using `Local` which contains the new value of the environment `r`, and a parameter `k` which during evaluation is used to represent *both* the scoped continuation and program remaining after.

```
data Local r k = Local r k
```

Using these, Reader can be represented by using `Ask r` as the `f` parameter to `Prog`, therefore making the syntax of `Ask` non-scoped. `Local` is used the `g` parameter, making this syntax scoped:

```
type Reader r = Prog (Ask r) (Local r)
```

Like with `Fix` and `Free`, smart constructors can be used to make creating DSL syntax cleaner. The following example demonstrates how smart constructors and `Prog` makes it possible to create clean DSL programs containing both sequential logic and scoped syntax:

```
ex :: Reader Int Int
ex = do
    x <- ask
    y <- local 10 (do
        y <- ask
        return (y+1))
    return (x+y)
```

Like with `Fix` and `Free`, an algebra is used to give semantics to a `Prog` tree. Folding over a `Prog` tree also begins at the leaves and works upwards. Unlike `Fix` and `Free`, the algebra for `Prog` contains three functions which make use of a a `Nat`-indexed carrier type `a`, with the type-level natural numbers used to keep track of the level of scoping.

```
data Nat = Z | S Nat

data Alg f g a = A {
    a :: forall n. f (a n) -> a n
  , d :: forall n. g (a ('S n)) -> a n
  , p :: forall n. a n -> a ('S n)
}
```

The algebra `a` is like that of the algebra used to fold over `Fix` and `Free` (which had type `f a -> a`), and in this case is used to give semantics to 'normal' non-scoped syntax `f`. The difference here is that `a` is indexed by `Nat` to keep track of scoping levels.

The promotion algebra, `p`, is applied when the *end* of a scope is encountered - the end is encountered before the start since the tree is folded from the leaves upwards. The function `p` is used to 'wrap up' the non-scoped value, indicated by the increase in the level of scoping `'S n`. This can then be given to the demotion algebra.

The demotion algebra, `d`, is used to give semantics to scoped syntax `g` and is applied after the promotion algebra when the start of a scoped section is encountered. Notice that `g` contains `a ('S n)`, i.e. the result of promotion, which represents both the scoped and non-scoped

continuations. These can be individually 'unwrapped' to combine scoped and non-scoped continuations.

Due to the Nat-indexed carrier type, natural numbers also pervade the fold function used to give semantics to the tree.

```
fold :: (Functor f, Functor g) => Alg f g a -> Prog f g (a n) -> a n
fold _   (Var x)    = x
fold alg (Op op)    = a alg (fmap (fold alg) op)
fold alg (Scope sc) = d alg (fmap (fold alg . fmap (p alg . fold alg)) sc)
```

Since the type stored in the tree `a` is unlikely to be indexed, such as with the `ex` example which had a return type of `Int`, it is convenient to use a generator function to turn the unindexed return type `x` into an index carrier `a 'Z`, where `'Z` represents no scoping:

```
run :: (Functor f, Functor g) => (x -> a 'Z) -> Alg f g a -> Prog f g x -> a 'Z
run gen alg prog = fold alg (fmap gen prog)
```

With the machinery in place, the semantics of the Reader can be defined. A `Nat`-indexed carrier *Carrier r a (n :: Nat)* is required such that *Carrier r a 'Z* $\cong r \to a$, and *Carrier r a ('S 'Z)* $\cong r \to (r \to a)$, and so on. Increasing values of `Nat` correspond to increasing levels of scoping, therefore, the carrier describes that every time a level of scoping is entered, a new value of the environment is used. This carrier can be represented using the following types in Haskell:

```
data Carrier r a n = C { runC :: r -> Carrier' r a n }

data Carrier' r a :: Nat -> * where
    CZ :: a -> Carrier' r a 'Z
    CS :: (r -> Carrier' r a n) -> Carrier' r a ('S n)
```

Using the carrier, an algebra over the tree can be defined. At a high-level, `Ask` creates a continuation from `k` by passing it the value of the environment. This is then run continuing program execution.

```
alg :: Alg (Ask r) (Local r) (Carrier r a)
alg = A a d p where
    a :: Ask r (Carrier r a n) -> Carrier r a n
    a (Ask k) = C $ \r -> runC (k r) r

    ...
```

The demotion algebra for `Local` runs the scoped continuation `k` with the local value of the environment `r'`; the non-scoped continuation is unwrapped by pattern matching on `CS`; finally non-scoped execution continues using the original environment `r`.

```
    ...

    d :: Local r (Carrier r a ('S n)) -> Carrier r a n
    d (Local r' k) = C $ \r ->
        case runC k r' of
            CS runK' -> runK' r

    ...
```

The promotion algebra wraps up the carrier in a level of scoping using `CS`, which allows `run` to be unwrapped in demotion `d` in order to continue non-scoped execution.

```
    ...

    p :: Carrier r a n -> Carrier r a ('S n)
    p (C run) = C $ \_ -> CS run
```

To run the reader, a function, like that provided by `Control.Monad.Reader`, can be defined. The function takes the initial reader environment and executes the reader using the `run` function which folds over the tree.

```
runReader :: Reader r a -> r -> a
runReader prog r = case (runC (run gen alg prog) r) of
    (CZ x) -> x

gen :: a -> Carrier r a 'Z
gen x = C $ const (CZ x)
```

Running the example `ex` using `runReader 10 ex` gives the expected result `21`. In summary, `Prog` can be used to solve the scoping problems associated with the Free monad. However, unlike the Free monad, the syntax and semantics of DSLs created using `Prog` are fixed - neither the syntax or semantics of the Reader example can be extended without changing the data type or algebra. This is main problem of the problems tackled in Chapters 4 and 5.

Chapter 3

Pretty Printing

To further motivate `Prog`, and to give another example of its usage, it will be used to create a DSL which could be used in many different programs - a pretty printer. For compilers, in particular, pretty printing plays an important role in generating target code. Indented sections of code help the programmer discern control flow through the program. In this compiler, pretty printing is used to produce more readable WebAssembly, as well as display the output of various transformation stages to the While AST. This ability is also vital when targeting some languages, for example, incorrect indentation in Python will produce a compile-time error.

As a slight digression, implementing pretty printing using the Free monad will be explored in order to provide a visual representation of the problems `Free` has representing scope. After, an implementation using `Prog` is given motivating its use over the Free monad. The pretty printer DSL implemented is not yet extensible, but the machinery to achieve this is presented in Chapter 4.

3.1 Free Monad

Using Free monad, continuation style can be leveraged to make writing an instance of a pretty printing DSL like simply writing in a text file. However, to represent scoped syntax, such as indented sections of document, presents problems which will be discussed, as well as possible solutions.

3.1.1 Problem

A DSL to print text or newlines can be realised with the syntax below, where each constructor contains the rest of the document to be printed `k`.

```
data Text k = Text String k | Newline k
```

For simple lines of text this works well, however, problems are encountered when trying to represent scoped constructs such as indentation. As discussed in Chapter 2, it seems sensible

for the constructor to contain two values of `k` - first `k` is intended to represent the nested section of document, and the second to represent the continuation after the nested section.

```
data Indent k = Indent k k
```

The semantics for this can be implemented with no obvious flaws, and are given in Figure 3.1. However, creating an instance of the DSL, `ex` below, flags up problems. Firstly, the non-scoped continuation to `indent` is given as another argument to the function, instead of simply writing the rest of the continuation after. Secondly, it is valid to further construct the document outside the continuation passed into `indent`.

```
ex :: Free (Text :+: Indent) ()
ex = do
    text "while true do"; nl
    indent (text "skip" >> nl) (text "end" >> nl)
    text "x := 1"
```

Running the example produces the output below, where the extra text after `indent` has been pushed up inside the indented section, as well being appended to the end of the string.

```
while true do
    skip
    x := 1end
x := 1
```

As discussed in Chapter 2, this is due to the implementation of bind for the Free monad:

```
instance (Functor f) => Monad (Free f) where
    ...
    Op op >>= k = Op (fmap (>>= k) op)
```

and how the Functor instance is implemented for `Indent`:

```
instance Functor Indent where
    fmap f (Indent nest k) = Indent (f nest) (f k)
```

The implementation of `fmap` applies `f` over both `nest` and `k`. Therefore, when using `(>>=)` (implicitly through use of do-notation), `Indent nest x >>= k` becomes `Indent (nest >>= k) (x >>= k)`, appending `x := 1` inside and outside the body of the if-statement.

3.1.2 Possible Solution

One solution to this is to delimit where scoping starts and ends [15]. In this way, each constructor only has one continuation and so the use of bind does not append text onto multiple places. For example, to delimit where indented sections of documents start and end, a type with constructors representing indentation and de-indentation could be created:

```
data Indent k = Indent k | Dedent k
```

To represent an indented section of document would be represented by opening the scope with `Indent`, having the indented document, and then closing the scope with `Dedent`. Given appropriate smart constructors, this would look like:

```haskell
type IndentLvl    = Word
type ShouldIndent = Bool

indentBy :: IndentLvl -> ShouldIndent -> String
indentBy _   False = ""
indentBy lvl True  = replicate (fromIntegral lvl*2) ' '

type Carrier = IndentLvl -> ShouldIndent -> String

instance Alg Text Carrier where
    alg (Text str k) lvl shouldIndent = indentBy lvl shouldIndent ++ str ++ k lvl False
    alg (Newline k)  lvl _            = "\n" ++ k lvl True

instance Alg Indent Carrier where
    alg (Indent nest k) lvl shouldIndent = nestedStr ++ kStr where
        nestedStr = nest (lvl+1) shouldIndent
        kStr      = k lvl shouldIndent

pretty :: Alg f Carrier => Free f a -> String
pretty ast = eval (\_ _ _ -> "") ast (0 :: Word) True
```

Figure 3.1: Semantics of Pretty Printing using Free Monad

```haskell
ex :: Free (Text :+: Indent) ()
ex = do
    indent
    ...
    dedent
```

Since the concept of an indented section in split, the possibility for bugs with mismatched tags has been introduced. To help prevent this, a smart constructor can be used which automatically inserts tags around some section of document. By only exporting the smart constructor from the module, not the constructors, bugs are limited to only occurring inside the module.

```haskell
indent :: Indent :<: f => Free f () -> Free f ()
indent inner = begin >> inner >> end where
    begin = inject (Indent (Var ()))
    end   = inject (Dedent (Var ()))
```

One plus of this method is it makes defining the semantics of entering and exiting scope easy. To indent a section of document, simply increase its indentation level, and to dedent decrease the level.

```haskell
instance Alg Indent (IndentLvl -> ShouldIndent -> String) where
    alg (Indent k) lvl shouldIndent = k (lvl+1) shouldIndent
    alg (Dedent k) lvl shouldIndent = k (lvl-1) shouldIndent
```

Now, when writing DSL instances the continuation to `indent` is not given separately. This makes writing instances more natural - like text in a text file:

```haskell
ex :: Free (Text :+: Indent) ()
ex = do
    text "while true do"
    indent (do
        text "skip"; nl)
    text "end"; nl
    text "x := 1"
```

The output is also correctly indented:

```
while true do
    skip
end
x := 1
```

In summary, the Free monad has issues representing scoped constructs. One method to solve this is through the use of delimiters, however, this approach suffers in that the delimiters may be mismatched. This can be somewhat mitigated by only exposing smart constructors for use outside the module, however, this does not prevent bugs inside the module.

Splitting up nested syntax into two constructors also makes it hard to transform an AST. For example, imagine a language containing a for-loop constructor, this could be transformed into a while loop, however, splitting the representation of the for-loop into `BeginFor` and `EndFor` would make this much harder to perform.

3.2 Prog

As discussed, `Prog` introduces another constructor `Scope` to the Free monad in order to solve problems with scoped syntax.

```haskell
data Prog f g a
    = Var a
    | Op (f (Prog f g a))
    | Scope (g (Prog f g (Prog f g a)))
```

Using `Prog`, indentation will become the `g` type parameter and therefore be used to represented scoped syntax. Notice the scoped constructor `Indent` now only contains a single `k` parameter.

```haskell
data Indent k = Indent k
```

While folding over the tree, is used to store both the scoped and non-scoped continuations. This allows the representation of indentation needs to be kept as a single constructor. Therefore, eliminating bugs which may be introduced due to mismatching, as well as making transformations to trees easier.

Using this, representation of a pretty document `Doc` is defined by using `Text` as the `f` type parameter, and `Indent` as `g` type parameter to `Prog`:

```haskell
data Text k = Text String k | Newline k
type Doc    = Prog Text Indent
```

To evaluate the document to create a pretty printed string requires knowledge of the current indentation level and whether the piece of text should be indented (i.e. it proceeds a newline). The ability to append to a string representing the output is also required. This can be captured by the carrier *Carrier a (n :: Nat)* such that *Carrier a 'Z* $\cong$ *Env a*, and *Carrier a ('S Z)* $\cong$ *Env (Env a)*, and so on where *Env a* $=$ *Lvl* $\rightarrow$ *ShouldIndent* $\rightarrow$ *String* $\rightarrow$ *(a, String)*. This is represented using the following types, where `IndentLvl` is a synonym for `Word`, and `ShouldIndent` is a synonym for `Bool`:

```
type Env a = IndentLvl -> ShouldIndent -> String -> (a, String)
data Carrier a n = C { runC :: Env (Carrier' a n) }

data Carrier' a :: Nat -> * where
    CZ :: a -> Carrier' a 'Z
    CS :: Env (Carrier' a n) -> Carrier' a ('S n)
```

With this, the algebra for `Text` can be defined below, which appends either text to the accumulated string, or a newline. This essentially acts like a Reader for the indentation level and whether the line should be indented, and a State for the output string [1]. This suggests the semantics could be represented using DSLs for Reader and State, which in fact can be done using Effect Delegation, and is discussed later in Chapter 6.

```
alg :: Alg Text Indent (Carrier a)
alg = A a d p where
    a :: Text (Carrier a n) -> Carrier a n
    a (Text s  (C k)) = C $ \lvl shouldIndent acc ->
        k lvl False (acc ++ indentBy lvl shouldIndent ++ s)
    a (Newline (C k)) = C $ \lvl shouldIndent acc ->
        k lvl True  (acc ++ "\n")
    ...
```

The demotion algebra runs the nested continuation with an increased level of scope; from this the remaining non-nested continuation can be retrieved by pattern matching using `CS`; finally, the non-nested continuation can be run with the original level of scoping. The important part of this is how indenting and dedent can be defined separately - increasing and decreasing the level of nesting. This gives benefits of using delimiters with the Free monad but has the advantage that the constructor representing indentation is not split up.

```
    ...
    d :: Indent (Carrier a ('S n)) -> Carrier a n
    d (Indent (C k)) = C $ \lvl shouldIndent acc ->
        -- Run scoped continuation indented one level.
        case k (lvl+1) shouldIndent acc of
            -- Run remaining continuation at original level.
            (CS k', acc') -> k' lvl shouldIndent acc'
```

The full implementation is given in Figure 3.2. To create an instance of the DSL, only the type signature needs to be changed compared to the implementation using the Free monad, and running the example produces a correctly indented, pretty string.

```
ex :: Prog Text Indent ()
ex = do
    text "while true do"
    indent (do
        text "skip"; nl)
    text "end"; nl
```

[1] It would also appropriate to treat the accumulation of the string like a does Writer, but using a state transformation makes defining the semantics easier.

```haskell
type IndentLvl = Word
type ShouldIndent = Bool

type Env a = IndentLvl -> ShouldIndent -> String -> (a, String)

data Carrier a n = C { runC :: Env (Carrier' a n) }

data Carrier' a :: Nat -> * where
    CZ :: a -> Carrier' a 'Z
    CS :: Env (Carrier' a n) -> Carrier' a ('S n)

gen :: a -> Carrier a 'Z
gen x = C $ \lvl shouldIndent acc -> (CZ x, acc)

indentBy :: IndentLvl -> ShouldIndent -> String
indentBy _ False = ""
indentBy i True  = replicate (fromIntegral i*2) ' '

alg :: Alg Text Indent (Carrier a)
alg = A a d p where
    a :: Text (Carrier a n) -> Carrier a n
    a (Text s  (C k)) = C $ \lvl shouldIndent acc ->
        k lvl False (acc ++ indentBy lvl shouldIndent ++ s)
    a (Newline (C k)) = C $ \lvl shouldIndent acc ->
        k lvl True  (acc ++ "\n")

    d :: Indent (Carrier a ('S n)) -> Carrier a n
    d (Indent (C k)) = C $ \lvl shouldIndent acc ->
        -- Run scoped continuation indented one level.
        case k (lvl+1) shouldIndent acc of
            -- Run remaining continuation at original level.
            (CS k', acc') -> k' lvl shouldIndent acc'

    p :: Carrier a n -> Carrier a ('S n)
    p (C k) = C $ \_ _ acc -> (CS k, acc)

pretty :: Doc a -> (a, String)
pretty prog = case runC (run gen alg prog) (0 :: IndentLvl) (True :: ShouldIndent) "" of
    (CZ x, str) -> (x, str)
```

Figure 3.2: Semantics of Pretty Printing

```haskell
    text "x := 1"
```

3.3 Summary

This section motivated the use of the Free monad or `Prog` by being able cleanly represent effectful computations using do-notation. However, there are problems when trying to represent scoped constructs using the Free monad. This is somewhat solved through use of delimiters to mark when scope starts and ends, but may introduce bugs through non-matching delimiters, as well as making it hard to transform a tree where syntax is split up. `Prog` solves these issues by allowing scoped syntax to be represented using a single constructor. which was demonstrated through the creation of a pretty printing DSL.

Chapter 4

Prog à la Carte

Chapter 2 and 3 it was demonstrated how `Prog` could be used to create DSLs containing scoped syntax, but that currently the syntax and semantics of a DSL using `Prog` cannot be created in a modular, extensible manner. Currently, to extend a DSL the original code needed to be modified. This chapter aims to fix this by applying Data Types à la Carte to `Prog`. The section first presents how syntax and semantics can be created modularly. Then, to help motivate use of modular, scoped DSLs a simple, yet extensible, compiler from While to WebAssembly is implemented.

4.1 Modularity

In order to create DSLs containing scoped syntax, which can also be created in a modular manner, this section presents use of Data Types à la Carte combined with `Prog`.

4.1.1 Syntax

To compose syntax together, the coproduct can be used to separately compose normal syntax `f` and scoped syntax `g`, which are combined using `Prog f g` to create a program containing normal and scoped syntax. For example, the following data structures represent a subset of non-scoped WebAssembly instructions:

```
data Arith k = I32Const Int k | GetLocal String k | I32LEq k
data Instr k = Ret k
```

A subset of scoped instructions can be represented as:

```
data If   k = PopIf k k
data Loop k = Loop k
```

To create a program containing all these types of instructions, `Arith` and `Instr` are composed together, as well as `If` and `Loop` being composed together:

```
type WASM = Prog (Arith :+: Instr) (If :+: Loop)
```

To avoid the explicit use of `L`, `R`, `Op`, and `Scope`, the function `inject` is used to inject into normal syntax:

```
inject :: h :<: f => h (Prog f g a) -> Prog f g a
inject = Op . inj
```

Injection into scoped syntax can be done in a similar manner, through use of `injectSc`:

```
injectSc :: h :<: g => h (Prog f g (Prog f g a)) -> Prog f g a
injectSc = Scope . inj
```

With this machinery, smart constructors to be defined to allow creation of more readable DSLs, such as those below.

```
i32Const :: ArithInstr :<: f => Int -> Prog f g ()
i32Const x = inject (I32Const x (Var ()))

popIfElse :: (If :<: g, Functor f) => Prog f g a -> Prog f g a -> Prog f g a
popIfElse t e = injectSc (fmap (fmap return) (PopIf t e))
```

Subsequently, WebAssembly programs can be written as if writing source code. The program below evaluates whether the variable x is less than or equal to *0* and returns *1* or *0* depending on the comparison.

```
ex :: WASM ()
ex = do
    i32Const 0
    getLocal "x"
    i32LEq
    popIfElse
        (i32Const 1 >> ret)
        (i32Const 0 >> ret)
```

4.1.2 Semantics

Chapter 3 showed how semantics could be given to a pretty printer through defining an algebra over a `Prog` tree. However, the algebra was fixed and could not be extended, therefore, disallowing the semantics of any new syntax to be given. Like with `Fix` and `Free`, this can be solved through use of typeclasses to define an algebra. Chapter 2 presented the `Alg` datatype given to `fold` in order to give semantics to a `Prog` tree:

```
data Alg f g a = A {
    a :: forall n. f (a n) -> a n
  , d :: forall n. g (a ('S n)) -> a n
  , p :: forall n. a n -> a ('S n)
}
```

Due to different functions in `Alg`, different typeclasses are used to allow semantics of the different functions to be defined separately. The `OpAlg` typeclass is used to define the algebra over normal, non-scoped syntax. Whereas the `ScopeAlg` typeclass is used to define the demotion algebra for some syntax `g`:

```
class Functor f => OpAlg f (a :: Nat -> *) where
    alg :: f (a n) -> a n

class Functor g => ScopeAlg g a where
    dem :: g (a ('S n)) -> a n
```

Promotion has a type signature `a n -> a ('S n)`, and since it contains neither `f` nor `g` it not included in the typeclasses above. Instead, it is given directly to the function folding over the tree (`eval` discussed below), therefore allowing the programmer to avoid implementing promotion each time an instance of `OpAlg` or `ScopeAlg` is created.

Defining the semantics of a coproduct is done by choosing to use the either algebra `f` or `g`. Haskell automatically chooses the correct overloaded version of either `alg` or `dem`:

```
instance (OpAlg f a, OpAlg g a) => OpAlg (f :+: g) a where
    alg (L x) = alg x
    alg (R x) = alg x

instance (ScopeAlg f a, ScopeAlg g a) => ScopeAlg (f :+: g) a where
    dem (L x) = dem x
    dem (R x) = dem x
```

For convenience, a function `eval` can be defined which automatically chooses the instances of `OpAlg` and `ScopeAlg` in order to evaluate the tree. Since instances of `OpAlg` and `ScopeAlg` were defined for the coproduct, `eval` will allow the semantics of trees containing composed syntax to be evaluated.

```
type Pro a = forall (n :: Nat). a n -> a ('S n)

eval :: (OpAlg f a, ScopeAlg g a) => (r -> a 'Z) -> Pro a -> Prog f g r -> a 'Z
eval gen pro = run gen alg' where
    alg' = A alg dem pro
```

In summary, by applying Data Types à la Carte to `Prog`, the ability has been gained to create modular syntax and semantics for DSLs containing scoped syntax.

4.2 Compiler Example

To give a real example of using Data Types à la Carte with `Prog`, a simple, extensible compiler from While to WebAssembly is created. The compiles takes as input an instances of `Prog` representing the While abstract syntax tree (AST); it compiles this to a representation of WebAssembly, and finally generates a pretty printed string from this WebAssembly. This will become the core of the compiler build upon in further chapters, therefore demonstrating how modular, scoped DSLs can be extended.

4.2.1 Pretty Printing WebAssembly

Pretty printing WebAssembly will involve using the definition of WebAssembly from Section 4.1.1, and the pretty printing DSL `Doc` defined in Chapter 3. Recall the pretty printing DSL acts like the Writer monad and describes the layout of a pretty printed document by making

use of `Prog`. At a high-level, the DSL provides the ability to output text `s` using `text s`; output a newline `nl`; or output an indented section of the document `d` using `indent d`.

To transform WebAssembly into a pretty string, a carrier *Carrier a (n :: Nat)* is required such that *Carrier a 'Z* $\cong$ *Doc a*, and *Carrier r a ('S 'Z)* $\cong$ *Doc (Doc a)*, and so on. Increasing the value of `n` corresponding to increasing the levels of indentation of the document. The carrier also describes that every time a level of scoping is entered, a different document will be written to. Once the scope is exited, the pretty string output from 'running' the nested document can be combined with the string outputted from the non-nested continuation.

Notice that this has a similar structure as the carrier of the reader example in Chapter 2, and the pretty printer in Chapter 3, suggesting commonalities can be factored out. `Nest` fulfills this purpose, and is used to avoid redefinition of code and eliminate boilerplate:

```
data Nest f n = Nest { runNest :: f (Nest' f n) }

data Nest' f :: Nat -> * where
    NZ :: a -> Nest' f 'Z
    NS :: f (Nest' f n) -> Nest' f ('S n)
```

To create the carrier required to turn WebAssembly into a pretty string, `Nest` is combined with `Doc`, avoiding needing to define datatypes for `Carrier` and `Carrier'`, as was done for Reader.

```
type Carrier = Nest Doc
```

With this carrier, the semantics of pretty printing non-scoped arithmetic WebAssembly instructions `Arith` are be defined below. The string representing each instruction is written out, followed by a newline, and then any remaining instructions.

```
instance OpAlg Arith Carrier where
    alg (I32Const n          (Nest k)) = Nest $ text "i32.const " >> showable n >> nl >> k
    alg (GetLocal v (Nest x) (Nest k)) = Nest $ text "get_local " >> text v >> nl >> k
    alg (I32LEq              (Nest k)) = Nest $ text "i32.le_s" >> nl >> k
```

Use of Data Types à la Carte means the semantics for `Instr` can be defined separately from `Arith`, therefore, implying that extensions to the language could be easily added without needing to modify any original source code.

```
instance OpAlg Instr Carrier where
    alg (Ret (Nest k)) = Nest $ text "return" >> nl >> k
```

This is also true for the demotion algebra, the full definition of which is given in Figure 4.1.

4.2.2 While to WebAssembly

The syntax of a subset of While can be defined as below. This is slightly different to the Reader and WebAssembly examples as arithmetic and boolean expressions can be terminals which do not contain a continuation `k`, namely the constructors `Num`, `Ident`, and `T`.

```
instance ScopeAlg LoopInstr Carrier where
    dem (LOOP (Nest body)) = Nest $ do
        line "loop"
        (NS k) <- indented body
        line "end"
        k

instance ScopeAlg IfInstr Carrier where
    dem (IF (Nest t) (Nest e)) = Nest $ do
        line "if"
        indented t
        line "else"
        (NS k) <- indented e
        line "end"
        k
```

Figure 4.1: Pretty Printing Scoped WebAssembly

```
data AExp k = Num Int | Ident String | Add k k
data BExp k = T | Equ k k
data Stm  k = Export k k

data Control k = If k k k

type While = Prog (AExp :+: BExp :+ Stm) Control
```

Because of this, when defining the semantics of compilation, the approach used before to construct a carrier with only `CZ` and `CS` constructors will not work when evaluating arithmetic or boolean expressions. For example, given a carrier *Carrier a (n :: Nat)* such that *Carrier a 'Z* $\cong$ *WASM a*, and *Carrier r a ('S 'Z)* $\cong$ *WASM (WASM a)*, represented in Haskell as:

```
data Carrier a n = C { runC :: WASM (Carrier' a n) }

data Carrier' a :: Nat -> * where
    CZ :: a -> Carrier' a 'Z
    CS :: WASM (Carrier' a n) -> Carrier' a ('S n)
```

Attempting to define an algebra for the for the terminal `Num` in a sensible manner, such as:

```
instance OpAlg AExp (Carrier a) where
    alg (Num n) = C $ i32Const n
    ...
```

results in the type error:

```
Couldn't match type '()' with 'Carrier' a n'
```

This is because `i32Const n` has type `()`, but `C` is expecting a `WASM (Carrier' a n)`. There is nothing at hand to create an instance of `Carrier' a n` because the `Num` constructor contains no continuation. The compiler does do not know the level of nesting `n` and so `CZ` cannot be used. This project solves the problem through the introduction of another constructor `CN` to `Carrier'`:

```
data Carrier' a :: Nat -> * where
    CN :: Carrier' a n -- New
    CZ :: a -> Carrier' a 'Z
    CS :: WASM (Carrier' a n) -> Carrier' a ('S n)
```

The constructor performs no change to the level of nesting and is used only to be able to satisfy the type error, and can be used to amend the problem by returning `CN`:

```
instance OpAlg AExp (Carrier a) where
    alg (Num n) = C $ i32Const n >> return CN
    ...
```

It appears as if the expression does nothing since the result of `i32Const n` is discarded by `(>>)`. However, this is not the case because the WebAssembly DSL is effectful - a side effect of `i32Const n` is to write out a string representing the instruction. Therefore, although the result is discarded, the correct WebAssembly is still generated.

The addition of `CN` does not change how the semantics are declared for syntax containing continuations, as demonstrated in the example below, which compiles export statements to return statements in WebAssembly. The full compiler is given in Figure 4.2.

```
instance OpAlg AExp (Carrier a) where
    alg (Export (C x) (C k)) = C $ x >> ret >> k
```

4.3 Summary

Examples creating a Reader and pretty printing DSL showed the potential to use `Prog` to describe DSLs containing scoped syntax, however, there was no method to create the DSLs in a modular manner. This section tackled this problem through applying Data Types à la Carte to `Prog`, and demonstrated how the technique proved effective in creating a simple, yet extensible, compiler from While to WebAssembly. Through implementing the compiler, problems defining the semantics of terminals in a language were unearthed with defining the semantics of terminals. But, by solving this problem `Prog` can be applied to a wider range of DSLs.

```haskell
instance OpAlg AExp (Carrier a) where
    alg (Num n)           = C $ i32Const n >> return CN
    alg (Ident v)         = C $ getLocal v >> return CN
    alg (Add (C x) (C y)) = C $ x >> y >> i32Add >> return CN

instance OpAlg BExp (Carrier a) where
    alg (T)               = C $ i32Const 1 >> return CN
    alg (Equ (C x) (C y)) = C $ x >> y >> i32Equ >> return CN

instance OpAlg Stm (Carrier a) where
    alg (Export (C x) (C k)) = C $ x >> ret >> k

instance ScopeAlg Control (Carrier a) where
    dem (If (C cond) (C t) (C e)) = C $ do
        cond
        CS k <- popIfElse t e
        k

pro :: Carrier a n -> Carrier a ('S n)
pro (C runC) = C (fmap (CS . return) runC)

gen :: a -> Carrier a 'Z
gen x = C (return (CZ x))

compile :: While a -> WASM a
compile ast = case (eval gen pro ast) of
    (C wasm) -> fmap (\(CZ x) -> x) wasm
```

Figure 4.2: A modular compiler from While to WebAssembly making use of Data Types à la Carte and `Prog`.

Chapter 5

Effect Handlers

In Chapter 4, it was demonstrated how Data Types à la Carte could be used with `Prog` to create extensible, scoped DSLs. It is important to notice that extending any DSL was done by extending into the same semantic domain. For example, While's arithmetic and boolean expressions both mapped to WebAssembly. This can also be seen in the original Data Types à la Carte paper, in which the language of arithmetic expressions is extended with a new operator, which has the same semantic domain.

There is often a need to compose DSLs with different semantic domains. For example, imagine a state DSL with some semantic domain $s \to (a, s)$ representing a state transformation; and a DSL for console operations with the semantic domain `IO a`. It may be desirable to combine the two DSLs, for example, to keep track of some user input.

This section demonstrates the problem of trying to combine DSLs, defined using Data Types à la Carte, which have different semantic domains. This is used to motivate Effect Handlers [15], a different method of defining modular semantics for a DSL. This method allows for different syntax to have different semantic domains. This will be applied to `Prog` with the goal of increasing the range of DSLs `Prog` can be applied to, and in fact is a technique that is used extensively throughout the construction of the compiler.

5.1 Representing Effects using Data Types à la Carte

Before introducing Effect Handlers, further motivation is given by exploring the problem of trying to combine A la Carte DSLs with different semantic domains. This is done by trying to combine a DSL for Exceptions with the Reader DSL introduced in Chapter 2.

5.1.1 Exceptions

Exceptions provide the ability to terminate execution of some computation by throwing an error value, as well the ability to catch exceptions thrown inside a scoped block and recover program execution. Throwing an exception `e` is modelled using:

```
data Throw e k = Throw e
```

This has a type variable `k`, allowing it to be composed using `(:+:)`. However, the constructor contains no resumption as control flow jumps upon throwing an exception. To model "catching" a thrown exception, `Catch` is used:

```
data Catch e k = Catch (e -> k) k
```

where `Catch hdl p >>= k` executes `p` and continues with `k` unless an exception is thrown, in which case, `hdl` is run.

The semantic of exceptions uses the approach of encoding errors using `Either e a`, where `Left e` encodes an error, and `Right a` encodes successful termination. Below the stubbed semantics are defined, the important thing to notice here is the type of the carrier, not the implementation.

```
type CarrierExc e = Nest (Either e)

instance OpAlg    (Throw e) (CarrierExc e) where alg = ...
instance ScopeAlg (Catch e) (CarrierExc e) where dem = ...
```

For running programs that *only* contain exception syntax this works well, and can be run with the function `runExc :: Prog (Throw e) (Catch e) a -> Either e a`.

5.1.2 Combining with Reader

To combine the `Reader` with exceptions, it seems sensible to use the approach introduced in Chapter 4 of using coproduct to compose the types. This, in fact, compiles and allows the creation of DSL programs containing both Reader and Exception syntax:

```
ex :: Prog (Throw String :+: Ask Int) (Catch String :+: LocalR Int) Int
ex = do
    x <- ask
    if x > 0
        then return 1
        else throw "Too small!"
```

However, when trying to create a function to run `ex`, such as

```
runExcReader
    :: Prog (Throw String :+: Ask Int) (Catch String :+: LocalR Int) Int
    -> ExcCarrier String Int 'Z
runExcReader = eval _ _
```

results in the error

```
"No instance for (OpAlg Ask CarrierExc) arising from a use of 'eval'"
```

The problem results from how instances of `OpAlg` and `ScopeAlg` are implemented for types composed with the coproduct. In order to evaluate a composite type, all the types must adhere to the `OpAlg`/`ScopeAlg` typeclass with the same type of carrier. However, Reader and Exceptions DSLs use different carriers and so they cannot be evaluated using the same call to `eval`. The non-solution is to reimplement `Ask`'s semantics to use `ExcCarrier`, however, this is undesirable as it leads to code duplication.

5.2 Effect Handlers

Data Types à la Carte runs into problems trying to compose DSLs with different semantic domains. This is because A la Carte attempts to evaluate the *entire* tree at once, giving semantics to all syntax. Effect handlers aim to solve this problem by only giving semantics to *part* of the syntax in a tree. Effect handlers are essentially tree-to-tree transformations, where a *handler* function "peels" off one layer of syntax giving it semantics and leaving the rest of the syntax for another handler. This section explores implementing effect handlers using `Prog`.

As a quick preliminary example, a handler for a state effect would have the type:

```
handleState :: s -> Prog (State s :+: f) (LocalSt s :+: g) a -> Prog f g (a, s)
```

It is made evident by the type that no state syntax, `State` or `LocalSt`, is contained in the resulting tree. The type stored in the leaves of the tree is also modified to contain the resulting state.

The rest of this section will explore how Exceptions and Reader need to be adapted in order to be correctly composed without the need to reimplement any semantics.

5.2.1 Exceptions

In order to use the syntax of exceptions for an effect handler, only minor aesthetic changes are required to the datatypes `Throw` and `Catch`, i.e. renaming the constructors:

```
data Throw e k = Throw' e
data Catch e k = Catch' (e -> k)
```

This is done to avoid name clashes with the patterns:

```
pattern Throw err <- (prj -> Just (Throw' err))
pattern Catch hdl <- (prj -> Just (Catch' hdl))
```

These are used to help define the semantics of exceptions, which is defined by pattern matching on syntax to be handled and leaving uninteresting syntax. The patterns allow for more succinct pattern matching on interesting syntax. They work by *projecting* out exception syntax from a signature containing it. For example, the throw pattern will work for any type `Prog f a` where `Throw e :<: f`. In the `Throw` pattern, `prj` is applied at the site of the pattern, and if the result is `Just (Throw' err)` the match is a success.

The handler also needs to leave uninteresting syntax to be handled by another handler. This is done through use of the `Other` pattern:

```
pattern Other op = R op
```

To start defining the semantics, a carrier is required. Unlike the previous carrier for exceptions, `Nest (Either e)`, the new carrier needs to return a `Prog` tree containing unhandled syntax `f` and `g`, with leaves containing either the an error or computation result. This carrier has the form

Carrier f g e a (n :: Nat) such that *Carrier f g e a 'Z* $\cong$ *Prog f g (Either e a)*, and *Carrier f g e a ('S 'Z)* $\cong$ *Prog f g (Either e (Prog f g (Either e a)))*, and so on. This is captured with the following types:

```
data CarrierExc f g e a n
    = Exc { runExc :: Prog f g (Either e (CarrierExc' f g e a n)) }

data CarrierExc' f g e a :: Nat -> * where
    CZ :: a -> CarrierExc' f g e a 'Z
    CS :: (Prog f g (Either e (CarrierExc' f g e a n))) -> CarrierExc' f g e a ('S n)
```

Defining the algebra of an effect handler does not add too much overhead compared to defining the algebra for normal DSLs, the main difference being that the result type must be lifted into the `Prog` tree containing the remaining syntax. The algebra must also match on uninteresting syntax, i.e. using `Other`. These factors add extra work for the developer, however, it could be argued this is outweighed by the ability to compose DSLs with different semantics and so increase code reuse.

```
algExc :: Alg (Throw e :+: f) (Catch e :+: g) (CarrierExc f g e a)
algExc = A a d p where
    a :: (Throw e :+: f) (CarrierExc f g e a n) -> CarrierExc f g e a n
    a (Throw err) = Exc (return (Left err))
    a (Other op)  = Exc (Op (fmap runExc op))
```

It is a similar story for the demotion algebra, which now operates in the context of `Prog`, leading to slight development time overheads:

```
    ...

    d :: (Catch e :+: g) (CarrierExc f g e a ('S n)) -> CarrierExc f g e a n
    d (Catch hdl k) = Exc $ do
        x <- runExc k
        case x of
            Right (CS k') -> k'
            Left err -> do
                r <- runExc (hdl err)
                case r of
                    Right (CS k') -> k'
                    Left err      -> return (Left err)

    ...
```

The promotion algebra must also operate in the context of the `Prog` tree containing uninteresting syntax:

```
    ...

    p :: CarrierExc f g e a n -> CarrierExc f g e a ('S n)
    p (Exc runExc) = Exc $ do
        r <- runExc
        case r of
            Left err -> return (Left err)
            Right x  -> return (Right (CS (return (Right x))))
```

With the algebra in place, a handler for exceptions can be defined which strips exception syntax from a `Prog` tree, changing the result type of the program to be either an error or the result of a successful computation.

```
handleExc :: Prog (Throw e :+: f) (Catch e :+: g) a -> Prog f g (Either e a)
handleExc prog = do
    r <- runExc (run genExc algExc prog)
    case r of
        Left err     -> return (Left err)
        Right (CZ x) -> return (Right x)
```

5.2.2 Void

As well as exceptions, the original Reader can be transformation in order to create a handler `handleReader :: r -> Prog (Ask r :+: f) (LocalR r :+: g) a -> Prog f g a`, the full code of which is given in Figure 5.1. Notice that after running reader and exception handlers over a tree containing only this syntax, there will remain a tree with no syntax, where the leaves contain the result of the computation. In order to extract the data stored in the leaves, a base case handler is required. This is last to be run and extracts a final value from a program with no more syntax. `Void` provides this representation of "no more syntax":

```
data Void k
```

It has no constructors and so it is impossible to use either the `Op` or `Scope` constructors. Therefore, the `a`, `d`, and `p` of the algebra are can be left undefined:

```
data CarrierV a (n :: Nat) = V a

algV :: Alg Void Void (CarrierV a)
algV = A undefined undefined undefined
```

Using this, a handler that extracts the final result of a computation can be created:

```
handleVoid :: Prog Void Void a -> a
handleVoid prog = case run V algV prog of
    (V x) -> x
```

5.2.3 Combining

With handlers for both Reader syntax, Exception syntax, and a handler to extract the final result of a computation, a composite handler can be created by using function composition `(.)` to compose the handlers together. This handles all the syntax in the tree using `handleReader` and `handleExc` leaving a tree of type `Prog Void Void (Either String a)`. Finally, `handleVoid` is used to extract `Either String a` from the tree:

```
handle
    :: r -> Prog (Ask r :+: Throw String :+: Void) (LocalR r :+: Catch String :+: Void) a
    -> Either String a
handle r = handleVoid . handleExc . handleReader r
```

Using this, the example `ex` can be run with `handle 1 ex` to produce `Right 1`, or `handle (-1) ex` to produce `Left "Too small!"`. Therefore, demonstrating how effect handlers can be used with `Prog` to solve the problem of needing to reimplement semantics if attempting to combine DSLs with different semantic domains.

```haskell
data Ask r k
    = Ask' (r -> k)
    deriving Functor

data LocalR r k
    = LocalR' r k
    deriving Functor

data Carrier f g r a n
    = Re { runR :: r -> Prog f g (Carrier' f g r a n) }

data Carrier' f g r a :: Nat -> * where
    CZ :: a -> Carrier' f g r a 'Z
    CS :: (r -> Prog f g (Carrier' f g r a n)) -> Carrier' f g r a ('S n)

genR :: (Functor f, Functor g) => a -> Carrier f g r a 'Z
genR x = Re (const (return (CZ x)))

algR :: (Functor f, Functor g) => Alg (Ask r :+: f) (LocalR r :+: g) (Carrier f g r a)
algR = A a d p where
    a :: (Functor f, Functor g) => (Ask s :+: f) (Carrier f g s a n) -> Carrier f g s a n
    a (Ask fk)   = Re $ \r -> runR (fk r) r
    a (Other op) = Re $ \r -> Op (fmap (\(Re run) -> run r) op)

    d :: (Functor f, Functor g)
      => (LocalR s :+: g) (Carrier f g s a ('S n)) -> Carrier f g s a n
    d (LocalR r' k) = Re $ \r -> do
        CS run' <- runR k r'
        run' r

    d (Other op) = Re $ \r -> Scope (fmap (\(Re run) -> fmap (f r) (run r)) op) where
        f :: r -> Carrier' f g r a ('S n) -> Prog f g (Carrier' f g r a n)
        f r (CS run') = run' r

    p :: (Functor f, Functor g)
      => Carrier f g r a n -> Carrier f g r a ('S n)
    p (Re run) = Re $ \_ -> return (CS run)

handleReader :: (Functor f, Functor g)
    => r -> Prog (Ask r :+: f) (LocalR r :+: g) a -> Prog f g a
handleReader r prog = do
    CZ prog' <- runR (run genR algR prog) r
    return prog'
```

Figure 5.1: Full implementation of the Reader Effect Handler.

5.2.4 Semantics for Free

Notice that when defining a composite handler of both Reader and with Exceptions in Section 5.2.3, there is a choice regarding the ordering of handlers. Different orderings can change the semantics, the result of this and possible applications are briefly discussed in this section.

To demonstrate, a State DSL is used, which is more formally introduced in Chapter 6. At a high-level, State acts on some underlying state *s*, which can be updated using *put s* or retrieved using *get*. This is captured by the syntax:

```
data State s k = Get (s -> k) | Put s k
```

A local, scoped state can also be created using *local s k* - updates to the state inside the scope of local state are not propagated to the outer scope. This is captured by the following syntax:

```
data LocalSt s k = LocalSt s k
```

A handler for this syntax can be created which removes State syntax, and changes the leaves of the resulting `Prog` tree to contain the result of the computation *and* value of the state:

```
handleState :: s -> Prog (State s :+: f) (LocalSt s :+: g) a -> Prog f g (a, s)
```

A non-determinism DSL, which non-deterministically chooses which branch to execute can be defined using a constructor containing two branches representing the two choices [12], one of which will be picked non-deterministically:

```
data Choice k = Or k k
```

A handler for this can be created such that every possible combination of choices is explored, therefore, the handler removes Non-Determinism syntax and changes the result type of the program to represent all possible values of `a`, i.e. using a list of `a`:

```
handleNonDet :: Prog (Choice :+: f) g a -> Prog f g [a]
```

Like when composing the handlers of Reader and Exceptions, composing non-determinism and state gives a choice about which handler to apply first. The State handler can be applied first, followed by the Non-Determinism handler, in which case each branch block shares the same global state. This is reflected in the result type, where there is a single value of state.

```
handleGlobal ::
    s -> Prog (Choice :+: State s :+: f) (Once :+: k) a -> Prog f g ([a], s)
handleGlobal s = handleState s . handleNonDet
```

Alternatively, the Non-Determinism handler can be run followed by the State Handler. In this case, each branch of the computation, created using the `Or` constructor, has its own copy of the state. Again, this is reflected in the type, where there is a value of state `s` for each value of `a`:

```
handleLocal ::
    s -> Prog (State s :+: Choice :+: f) (LocalSt s :+: Once :+: k) a -> Prog f g [(a, s)]
handleLocal s = handleNonDet . handleState s
```

The ability to change the semantics by reordering handlers was not explicitly used in this project, however, there are applications for its use. For example, if implementing a Java runtime, Exception and State handlers could be composed to use the global representation of state. Whereas reordering Exception and State handlers could be used if implementing a language with transactions, where the state should be rolled back if a transaction fails.

5.3 Summary

In summary, use of Effect Handlers solves the problem of needing to reimplement semantics if attempting to combine DSLs with different semantic domains using Data Types à la Carte. This greatly increases the use cases for `Prog` and decreases potential development costs through code reuse. These factors lead to effect handlers being used extensively through the creation of the compiler. Some of these areas included: during renaming of variables the DSLs for state, fresh values, and exceptions were combined. When computing variables which are modified in two different procedures, a writer was combined with a reader. And, during code generation a writer was combined with a reader to provide safety. In the future, a concurrency effect could be used with State to simulate execution of threaded programs.

Chapter 6

Effect Delegation

Chapters 4 and 5 demonstrated how modular DSLs can be created using `Prog`. These DSLs have been used to represent ASTs and effectful programs, however, an unexplored avenue is Effect Delegation [14]: defining a high-level DSL in terms of a lower-level DSL. This has the potential to allow easier specification of a DSL's semantics, as well as decreasing development times due to reuse of lower-level DSLs.

This chapter explores the feasibility of using effect delegation to implement the semantics of DSLs defined using `Prog`. To provide context, and to further expand the compiler, this exploration is be done through the lens of adding scoped variables to While. To introduce scoped variables, While will be extended using Data Types à la Carte. To motivate the creation of new DSLs to assist with compilation, the section also explores how local variables can be represented in WebAssembly, before going on to define the syntax and semantics of said DSLs.

6.1 Extending While

Modern languages provide the ability to create locally scoped variables, which help to avoid mutation of global state. In While, local variables are defined inside a block to which they are locally scoped. Once the block is exiting any local variables defined in its header cease to exist. The BNF of blocks is defined as [11]:

```
VarDecl : 'var' Var ':=' AExp ';'
Stm     : 'begin' VarDecls* Stm 'end'
        | ...
```

This allows programs such as the following to be created, which creates a local scope containing the variables `x` and `y`, with `x` shadowing the outer declaration. At the export statement, however, the global `x` is used since the inner `x` is no longer in scope:

```
x := 1;
begin
    var x := 2;
    var y := 3;
    x := 4
end;
export x
```

Using Data Types à la Carte, the syntax of While can be extended to include scoped local variables using the scoped `Block` constructor. This contains a list of local variables and the value assigned to them upon entry to the block, as well as the body of the block.

```
type VarDecls v k = [(v, k)]
data Block v k = Block (VarDecls v k) k
```

To use `Block` as scoped syntax in the While AST, it can simply be composed with other scoped syntax:

```
type While v = Prog (AExp v :+: BExp :+: Stm) (Control :+: Block v)
```

With an appropriate smart constructor, `Block` can be used to create programs containing local variables. For example, the following program is how the compiler would internally represent the parsed AST presented above.

```
ex :: While ()
ex = do
    assign "x" (num 1)
    block [("x", num 2), ("y", num 3)] (do
        assign "x" (num 4))
    export (var "x")
```

This example further demonstrates how Data Types à la Carte can be used to create extensible DSLs. Note that Data Types à la Carte is an appropriate still choice for representing an AST, instead of using Effect Handlers. This is because the compiler needs to compile to the *entire* tree at once, as opposed to only compiling to arithmetic expressions, for example.

6.2 Representing Local Variables in WebAssembly

In WebAssembly, variables are global to a function and must be declared at the start of the function. However, in While, local variables may shadow variables in an the outer scope. This poses a problem in that While cannot be directly compiled to WebAssembly because if a variable at an inner scope shadows a variable at the outer scope, the compiled WebAssembly will modify the outer variable, therefore changing the semantics of the program.

A simple solution to this is to simply disallow shadowing of outer variables. However, this solution is undesirable as local variables therefore pose no real benefit over global variables. This also leaves the programmer responsible for checking there are no name clashes. A better solution is to have the compiler modify the variable names in the AST to ensure no clashes occur, this is the approach taken an involves renaming each variable in the While AST to be unique, ensuring inner variables do not shadow outer variables - essentially Alpha-Conversion [1].

For example, the While program:

```
x := 1;
begin
    var x := 2;
    x := 3
end;
export x
```

could be transformed into:

```
$0 := 1;
begin
    var $1 := 2;
    $1 := 3
end;
export $0
```

Notice that in the original program the `x` variable inside the block shadowed the outer `x`. But in the transformed program, the `x` in the inner scope has been given a distinct name.

6.3 Renaming DSL

Because While containing local variables cannot directly to WebAssembly, the AST must be transformed by renaming variables in the tree so they are unique. To assist, a DSL is created to encapsulates the logic of alpha-conversion. This makes it easy to reuse renaming, for example, in Chapter 8 the DSL is used to rename procedures.

To rename variables in the AST, the compiler requires the ability to look up the mapping from a variable to its newly assigned name, and the ability to give a variable a fresh, unseen name. Effect delegation will be used to express these complex semantics in terms of more primitive effects encapsulating these abilities respectively. This helps reduce the complexity of the semantics of the Alpha-Conversion (Renaming) DSL.

A DSL *specifically* for keeping track of a mapping from variable names to new names could be created, as well as a DSL *specifically* to generate fresh names. However, these are not particularly generic and so may not be able to be used to define the semantics of other DSLs in the future. Instead, generic State and New DSLs will be created to handle each job respectively. Before implementing the Renaming DSL, the DSLs for State and New are given.

6.3.1 State

The State DSL is not implemented using effect delegation and so its implementation is not given in detail. However, the New and Renaming DSLs to delegate to the State DSL. As introduced in Chapter 5, the State DSL gives the ability to retrieve some underlying state s using *get*, and update the state using *put* s:

```
data State s k = Get' (s -> k) | Put' s k
```

It is also useful to provide the ability to create a local scope in which updating the value of the state with *put* s does not affect the outer value of state. This is captured using the following scoped constructor:

```
data LocalSt s k = LocalSt' s k
```

To aid with creating programs containing State, smart constructors can be defined:

```
data Carrier f g s a n
    = St { runSt :: s -> Prog f g (Carrier' f g s a n, s) }

data Carrier' f g s a :: Nat -> * where
    CZ :: a -> Carrier' f g s a 'Z
    CS :: (s -> Prog f g (Carrier' f g s a n, s)) -> Carrier' f g s a ('S n)

genSt :: (Functor f, Functor g) => a -> Carrier f g s a 'Z
genSt x = St (\s -> return (CZ x, s))

algSt :: (Functor f, Functor g) => Alg (State s :+: f) (LocalSt s :+: g) (Carrier f g s a)
algSt = A a d p where
    a :: (Functor f, Functor g) => (State s :+: f) (Carrier f g s a n) -> Carrier f g s a n
    a (Get fk)   = St $ \s -> runSt (fk s) s
    a (Put s' k) = St $ \_ -> runSt k s'
    a (Other op) = St $ \s -> Op (fmap (\(St run) -> run s) op)

    d :: (Functor f, Functor g)
      => (LocalSt s :+: g) (Carrier f g s a ('S n)) -> Carrier f g s a n
    d (LocalSt s' k) = St $ \s -> do
        (CS run', s'') <- runSt k s'
        run' s

    d (Other op) = St $ \s -> Scope (fmap (\(St run) -> fmap f (run s)) op) where
        f :: (Carrier' f g s a ('S n), s) -> Prog f g (Carrier' f g s a n, s)
        f (CS run, s) = run s

    p :: (Functor f, Functor g) => Carrier f g s a n -> Carrier f g s a ('S n)
    p (St run) = St $ \s -> return (CS run, s)

handleState :: (Functor f, Functor g)
    => s -> Prog (State s :+: f) (LocalSt s :+: g) a -> Prog f g (a, s)
handleState s prog = do
    (CZ prog', s') <- runSt (run genSt algSt prog) s
    return (prog', s')
```

Figure 6.1: Full Implementation of the State DSL

```
get     :: State   s :<: f => Prog f g s
put     :: State   s :<: f => s -> Prog f g ()
localSt :: LocalSt s :<: g => s -> Prog f g a -> Prog f g a
```

An effect handler is used to give the semantics to State syntax:

```
handleState :: s -> Prog (State s :+: f) (LocalSt s :+: g) a -> Prog f g (a, s)
```

The implementation of the handler does not use effect delegation and so is not discussed here, however, for reference the full semantics of the DSL is given in Figure 6.1.

6.3.2 New

The New DSL provides a way to generate an unseen `Enum` value. It operates on some underlying state e, with *new* used to generate an unseen value by incrementing the value of e. This acts much like State, but provides safety in that e cannot be accidentally modified which could result in non-unique values being produced. This section demonstrates how effect delegation

can be used with `Prog` to implement New in terms of State in order to simplify defining its semantics.

The syntax of New is captured with the following datatype `New`, which passes a fresh value of `e` to the continuation.

```
data New e k = New' (e -> k)
```

Because there are no scoping constructors, the semantics of `New` can be defined using a new carrier, `CarrierId`, which is polymorphic with the level of nesting `n` but does not use it in its type constructor thus keeping its structure flattened. A Nat-indexed carrier still has to be used in this case, since `Alg` expected one. However, the carrier is defined such that *CarrierId a 'Z* $\cong$ *Id a*, *CarrierId r a ('S 'Z)* $\cong$ *Id a*, and so on without ever increasing the level of nesting. This is captured with the following type:

```
newtype CarrierId a (n :: Nat) = Id { unId :: a }
```

The `run` function, used when evalauting `Prog`, and can be specialised for use with `CarrierId` for added convenience:

```
runId :: (r -> CarrierId a 'Z) -> Alg f g (CarrierId a) -> Prog f g r -> a
runId gen alg prog =
    case run gen alg prog of
        (Id x) -> x
```

Back to the New DSL, to implement New in terms of the State effect, the compiler must translate the syntax of New to that of State. This is performed using a handler for New with the signature:

```
delegate :: Prog (New e :+: f) g a -> Prog (State e :+: f) (LocalSt e :+: g) a
```

As is made evident by the type, any New syntax is removed and *replaced* with the syntax for State, this is the key to effect delegation. The remaining syntax `f` and `g` is left unchanged in the tree, therefore allowing any other syntax to be handled by another carrier.

To define the semantics of `delegate`, the following carrier is used, which contains the tree containing State syntax:

```
type Carrier f g e a = CarrierId (Prog (State e :+: f) (LocalSt e :+: g) a)
```

The algebra below translates `New` into retrieving the current value of `e` using `get`; incrementing the stored value using `put`; and then passing the current value to the continuation. Expressing this logic in terms of the State DSL makes the code very readable, as well as being easier to write since there is no need to explicitly keep track of state.

```
alg :: Enum e => Alg (New e :+: f) g (Carrier f g e a)
alg = A a d p where
    a :: Enum e => (New e :+: f) (Carrier f g e a n) -> Carrier f g e a n
    a (New fk) = Id $ do
        next <- get
        put (succ next)
        let (Id r) = fk next
        r
```

Since New contains no scoped syntax, promotion and demotion simply ignore any scoped syntax.

```
    ...

    d :: g (Carrier f g e a ('S n) -> Carrier f g e a n
    d op = Id (Scope (fmap (return . unId) (R op)))

    p :: Carrier f g e a n -> Carrier f g e a ('S n)
    p (Id prog) = Id prog
```

To handle New syntax in a program, it is first converted to State syntax using `delegate`. The state syntax can be handled using `handleState`,

```
delegate :: Enum e => Prog (New e :+: f) g a -> Prog (State e :+: f) (LocalSt e :+: g) a
delegate = runId (Id . return) alg

handleNew :: Enum e => e -> Prog (New e:+: f) g a -> Prog f g (a, e)
handleNew n = handleState n . delegate
```

In summary, defining the semantics of New gives a small example of how effect delegation can be used to simplify defining the semantics of a DSL through expression using another, typically lower-level, DSL.

6.3.3 Renaming

The goal of creating New and State DSLs was to aid in defining the semantics of the Renaming DSL with the ability to lookup or generate a fresh name of a variable. Both these operations are captured using the single `Fresh` syntax below. The semantics of which passes the new name of a variable to the continuation, if one exists, or creates a new name if not.

```
type FreshName = Word
data Fresh v k = Fresh' v (FreshName -> k)
```

It is worth mentioning that `Fresh` is polymorphic with the type of variable it renames `v`. This will allow it to be used to rename both variables and procedures in Chapter 8.

The compiler also requires the ability to create a scope in which certain variables are assigned fresh names, and upon exiting the scope the original names are restored. This is used to rename variables inside a block, and is captured by the syntax:

```
data Rename v k = Rename' [v] k
```

An example instance of the DSL is given below, where the type `String` is the type of variables which will be renamed. The intended semantics will assign the outer `x1` to the unsigned integer `0`. The inner `x2` will be assigned to `1` because `"x"` is given to `rename` in the list of variables to generate fresh names for. Once the outer scope is returned to, scoped names will be discarded, thus assigning `x3` to `0`.

```
testRename :: Prog (Fresh String :+: Void) (Rename String :+: Void) ()
testRename = do
    x1 <- fresh "x"
    rename ["x"] (do
        x2 <- fresh "x"
        return ())
    x3 <- fresh "x"
    return ()
```

Using effect delegation, the syntax of renaming will be translated into the syntax of the State DSL *composed* with New DSL. Like with effect delegation used for New, the carrier [1] describes nested trees containing the lower-level syntax.

```
type Names v = Map v FreshName

type Ctx     f g v = Prog (State (Names v) :+: New Word :+: f) (LocalSt (Names v) :+: g)
type Carrier f g v = Nest1 (Ctx f g v)
```

The algebra for `Fresh` is defined in an imperative style by first checking whether a mapping from the variable with name `v` already exists, if it does, the existing fresh name is passed to the continuation. If not, `insFresh` (Figure 6.2) is used to generate a unique name *unq* and then store a mapping from `v` to *unq* mapping. Use of effect delegation makes it easy to express the complex logic of renaming in a simple, concise fashion, thereby increasing code readability and decreasing development time.

```
alg :: Alg (Fresh v :+: f) (Rename v :+: g) (Carrier f g v a)
alg = A a d p where
    a :: (Fresh v :+: f) (Carrier f g v a n) -> Carrier f g v a n
    a (Fresh v fk) = Nest1 $ do
        env <- get
        case Map.lookup v env of
           Just fresh -> runNest1 (fk fresh)
           Nothing -> do
               f <- insFresh v
               runNest1 (fk f)
    ...
```

Defining demotion uses a similarly imperative style: the continuation `k` is run in a local scope created by giving fresh names to the variables in `vs`, using `insManyFresh` (the full definition which is found in Figure 6.2) . After running the continuation, the original state is restored using `union` to ensure that mappings created inside the scoped continuation are persisted outside the scope. Use of effect delegation again makes it easy to understand the algorithm, even to someone unfamiliar with the codebase.

```
    ...
    d :: (Rename v :+: g) (Carrier f g  v a ('S n)) -> Carrier f g v a n
    d (Rename vs k) = Nest1 $ do
        names <- getNames
        insManyFresh vs

        (NS1 runK') <- runNest1 k
        names' <- getNames

        put (Map.union names names')
        runK'
```

[1] `Nest1` is essentially the same as `Nest`, but with an extra parameter for the return type of the computation.

```haskell
insFresh :: (Functor f, Functor g, Ord v)
    => v -> Prog (State (Names v) :+: New Word :+: f) (LocalSt (Names v) :+: g) FreshName
insFresh v = do
    env <- get
    next <- new
    put (Map.insert v next env)
    return next

insManyFresh :: (Functor f, Functor g, Ord v)
    => [v] -> Prog (State (Names v) :+: New Word :+: f) (LocalSt (Names v) :+: g) ()
insManyFresh vs = mapM_ insFresh vs
```

Figure 6.2: Definitions of `insFresh` and `insManyFresh`.

6.4 Summary

This section demonstrates how effect delegation can be used to simplify the semantics of scoped DSLs created using `Prog`, as well as allowing readable, imperative style code to be written. Through creating generic 'low-level' DSLs, effect delegation can also allow reuse of DSLs, such as was done for State, therefore potentially decreasing development times by increasing code reuse.

Chapter 7

Renaming AST

Chapter 6 explored the creation of a DSL to assist rename variables in a While AST in order to compile locally scoped variables into WebAssembly. This chapter attempts to put this DSL to work, but unfortunately, in doing so roadblocks are encountered which prevent further use of `Prog` to represent the While AST.

In order to rename the While AST, the intention was to use the Renaming DSL to as a monadic context to provide the syntax for renaming, i.e. `fresh` and `rename`. Unfortunately, however, a carrier containing the DSL is defined results in either compile time or runtime problems. This section provides a rundown of the different methods tried and their problems. As a reference, the program in Listing 7.1 *should* have x renamed such that program is equivalent to that in Listing 7.2.

```
x := 0;
begin
    var x := 1;
    x := 2;
end
x := 3;
```

Listing 7.1: Program Before Renaming

```
$0 := 0;
begin
    var $1 := 1;
    $1 := 2;
end
$0 := 3;
```

Listing 7.2: Program After Renaming

7.1 Context Per Level

Given the DSL for renaming:

```
type Ctx = Prog (Rename String :+: Void) (LocalName String :+: Void)
```

it seems reasonable to try to construct a carrier *Carrier f g a (n :: Nat)* for renaming variables in the While AST (i.e. *Prog f g a*) such that *Carrier f g a 'Z* $\cong$ *Ctx (Prog f g a)*, and *Carrier f g a ('S 'Z)* $\cong$ *Ctx (Prog f g (Ctx (Prog f g a))*, and so on. This can be captured by the following type:

```
data Carrier f g a n
    = Rn { runRn :: Ctx (Prog f g (Carrier' f g a n)) }

data Carrier' f g a :: Nat -> * where
    CZ :: a -> Carrier' f g a 'Z
    CS :: Ctx (Prog f g (Carrier' f g a n)) -> Carrier' f g a ('S n)
```

Notice that this carrier differs from previous carriers, which have had the form *Carrier a (n :: Nat)* such that *Carrier a 'Z* $\cong$ *H a*, and *Carrier a ('S Z)* $\cong$ *H (H a)*, etc. The difference being in that for each increment of *n*, `Carrier` wraps `a` inside *two* data types [1].

This presents problems when attempting to define demotion. Given the stub below

```
instance BlockStm FreshName Ident :<: g
    => ScopeAlg (BlockStm Ident Ident) (Carrier f g a) where

    dem (Block varDecls (Rn body)) = Rn _
```

the constructor `Rn` expects a type `Ctx (Prog f g (Carrier f g a n))`. To construct something of this type, `body` seems a promising place to start, which has type `Ctx (Prog f g (Carrier' f g a ('S n))`. To get to the carrier `k` contained inside the body, the following can be done:

```
instance BlockStm FreshName Ident :<: g
    => ScopeAlg (BlockStm Ident Ident) (Carrier f g a) where

    dem (Block varDecls (Rn body)) = Rn  $ do
        rnBody <- body
        return (do
            CS k <- rnBody
            ...)
```

However, since `k` has type `Ctx (Prog f g (Carrier' f g a n))` there is no way to 'run' the continuation as the result of the innermost do-block must return something of type `Prog (Carrier' f g a n)`, therefore leaving no way to compile demotion using a carrier with this form.

7.2 Global Context

Another approach is to only use a global `Ctx`, with the idea to use `CS` and `CZ` to only keep track of nesting of the renamed tree to output.

```
data Carrier' f g a :: Nat -> * where
    CZ :: a -> Carrier' f g a 'Z
    CS :: Prog f g (Carrier' f g a n) -> Carrier' f g a ('S n)
```

However, this again leads to problems when implementing demotion - the the non-scoped continuation `k` can only be unwrapped inside the `rename` block:

[1] This is not strictly true, since the carrier for Exceptions in Chapter 5 wrapped the result type inside two datatypes, i.e. `Prog` and `Either`. However, pattern matching could be performed on `Either` to remove this context.

```
instance BlockStm FreshName Ident :<: g
    => ScopeAlg (BlockStm Ident Ident) (Carrier f g a) where

    dem (Block varDecls (Rn body)) = Rn $
        rename (map fst varDecls) (do
            decls' <- mapM fv varDecls
            body' <- body
            return (do
                (CS k) <- block decls' b'
                k))
```

While this compiles, the semantics are incorrect as variables in the continuation `k` are not restored to their original names. Taking the program in Listing 7.1 as input, this implementation incorrectly renames the last occurrence of `x` to `$1`, instead of `$0`.

```
$0 := 0;
begin
    var $1 := 1;
    $1 := 2;
end
$1 := 3;
```

7.3 Data Types à la Carte

In order to avoid nesting `Prog` inside some context, it is possible to use the coproduct to compose the Renaming DSL into the syntax of While. The intention would be to run some function `rename` which took a tree representing the While AST (`Prog f g a`), and transformed it to contain Renaming DSL syntax:

```
rename :: Prog f g a -> Prog (Fresh String :+: f) (Rename String :+: g) a
```

The handler function for the Rename DSL could then be run in order to remove Renaming DSL syntax while renaming variables in the AST.

```
handleRename :: Prog (Fresh v :+: f) (Rename v :+: g) a -> Prog f g a
```

This sounds promising, but due to orderings of effect handlers giving different semantics (Chapter 5), this approach creates a local renaming scope inside branches of While constructs, such as if-statements. Using if-statements as an example: if a variable is renamed in the "then" branch, the update to the next available unique name is not communicated to "else" branch. Therefore, there is the potential clashing of variable names.

To illustrate this problem, the program below contains an if-statement inside which two variables are seen for the first time:

```
if true do
    x := 1
else
    y := 1
end
```

Running this through the algorithm described results in the program below, where both `x` and `y` have been renamed to the same name. This is a problem as later analysis of the program

relies on each variable having a unique name, for example, to create a mapping from variable names to some metadata.

```
if true do
    $0 := 1
else
    $0 := 1
end
```

7.4 Fix

Due to the problems faced, the rest of the implementation falls back on using `Fix` to represent the While AST in order to further investigate other uses of `Prog` when compiling. `Fix` does not suffer the same problems because a recursively defined carrier is not required to give semantics to the tree.

The WebAssembly AST retains its implementation using `Prog` as no transformations are applied to it, however, it would also require refactoring if, for example, optimisations were performed to it.

In order to rename the While AST, typeclass instances to rename each piece of syntax are defined. The algebra translates the AST into being in the context of the Renaming DSL, below, which can then be handled using `handleRename` in order to generate a renamed AST.

```
type Ctx = Prog (Rename String :+: Void) (LocalName String :+: Void)
```

Notable parts of the algebra include the semantics of `GetVar` and `SetVar` which make use of syntax `fresh` to lookup the fresh name of a variable.

```
instance VarExp FreshName :<: f => FixAlg (VarExp Ident) (Ctx (Fix f)) where
    alg (GetVar v) = do
        v' <- fresh v
        return (getVar v')

instance VarStm FreshName :<: f => FixAlg (VarStm Ident) (Ctx (Fix f)) where
    alg (SetVar v x) = do
        v' <- fresh v
        rnX <- x
        return (setVar v' rnX)
```

The algebra for `Block` uses `rename` to assign fresh names to any variables declared in the head of the block.

```
instance BlockStm FreshName :<: f => FixAlg (BlockStm Ident) (Ctx (Fix f)) where
    alg (Block varDecls body) =
        rename (fsts varDecls) $ do
            rnVarDecls  <- map2M fresh id varDecls
            rnBody      <- body
            return (block rnVarDecls rnBody)
```

For completeness, the algebras for the rest of the While syntax can be found in Figure 7.1.

```haskell
instance AExp :<: f => FreeAlg AExp (Ctx (Fix f)) where
    alg (Num n)   = return (num n)
    alg (Add x y) = add <$> x <*> y
    alg (Sub x y) = sub <$> x <*> y
    alg (Mul x y) = mul <$> x <*> y

instance BExp :<: f => FreeAlg BExp (Ctx (Fix f)) where
    alg (T)       = return true
    alg (F)       = return false
    alg (Equ x y) = equ  <$> x <*> y
    alg (LEq x y) = leq  <$> x <*> y
    alg (And x y) = andB <$> x <*> y
    alg (Not x)   = notB <$> x

instance Stm :<: f => FreeAlg Stm (Ctx (Fix f)) where
    alg (Skip)            = return skip
    alg (Export x)        = export <$> x
    alg (If cond t e)     = ifElse <$> cond <*> t <*> e
    alg (While cond body) = while  <$> cond <*> body
    alg (Comp s1 s2)      = comp   <$> s1   <*> s2

map2M :: Monad m => (a -> m c) -> (b -> m d) -> [(a, b)] -> m [(c, d)]
map2M f g = mapM $ \(x, y) -> do
    x' <- f x
    y' <- g y
    return (x', y')
```

Figure 7.1: Renaming While AST represented using Fix

7.5 Summary

In summary, this section presents a problem faced when attempting to transform a `Prog` tree inside a stateful context. It explores possible solutions, but demonstrates their problems, and finally falls back on a `Fix` implementation. This is an important area to devote time into finding a solution in order to be able to fully adopt the use of `Prog` in compilers.

Chapter 8

Procedures

Chapter 4 introduced Data Types à la Carte to create extensible DSLs; Chapter 5 shows how Effect Handlers can also be used to give semantics to composed syntax; and Chapter 5 shows how defining semantics can be simplified through use of Effect Delegation. This gives all the tools required to create modular, scoped DSLs meaning more complex problems can be tackled. With the goal of further extending the compiler, one such problem is compiling While procedures into WebAssembly. Exploring this problem will allow further demonstration of how `Prog` can be used with the techniques mentioned.

To introduce procedures, the syntax While will be extended. To motivate the introduction of new DSLs to aid with compilation, this chapter also delves quite deeply into how procedures can be represented in WebAssembly. Using `Prog`, these DSLs are then implemented in order to compile procedures.

8.1 Extending While

Functions increase the reusability of code as well as encapsulate program logic, and hence have become vital parts of programming languages. While, has a construct similar to functions - procedures. These are blocks of code which can be invoked but do not take arguments or return values. Instead, procedures *capture* variables from their outer scope, and mutate them in order to relay results back to their caller.

An extension to While [11] gives the BNF of procedures as below. This defines procedures as being named pieces of code which are defined in the header of a block, along with local variable declarations:

```
VarDecl  : 'var' Var ':=' AExp ';'
ProcDecl : 'proc' Var 'is' Stm ';'
Stm      : 'begin' VarDecls* ProcDecl* Stm 'end'
         | 'call' Var
         | ...
```

Below is an example of such a program, which creates a block starting with `begin` and ending with `end`. Inside the block, a procedure `f` is defined starting with the keyword `proc`. The procedure has the body `export 1`, which is run when the procedure is invoked with `call f`.

```
begin
    proc f is export 1;
    call f
end
```

More complex procedures may also *capture* variables from their outer scope. For example, `f` below modifies the variable `x`, such that, when the export occurs the value of `x` is `2` instead of `1`.

```
x := 1;
begin
    proc f is x := 1;
    call f;
    export x
end
```

This is how procedures affect the state of the program, instead of taking arguments and returning a value, and is one of the main problems behind compiling procedures. The other main problem with compiling procedures is that they may be nested within one another, for example, g is nested within f in the program below, discussed further in Section 8.2.

```
x := 1;
begin
    proc f is (
        begin
            proc g is x := 1;
            call g
        end
    );
    call f;
    export x
end
```

However, before procedures can be compiled, they must be represented in the While AST. It seems sensible to use Data Types à la Carte to extend the language to contain procedures, like was done for local variables in Chapter 6. However, the syntax for blocks containing local variables is fixed and cannot be changed to include a list of procedures in order to model the BNF more closely:

```
type VarDecls v k = [(v, k)]
data Block v k = Block (VarDecls v k) k
```

This demonstrates a problem with Data Types à la Carte - only the type can be extended, not the *constructors* of a type. There are solutions to this, such as not exactly mirroring the BNF in the implementation, and creating a procedure datatype which can be composed into the language:

```
data Proc name k = Proc name k
```

This will work but will make analysis of the tree slightly harder. For example, a list of names of procedures inside a block are not immediately at hand if given an instance of `Block`. This information is required, for example, to rename the procedures inside a block. Having claimed that Data Types à la Carte can be used, some slight cheating will occur to make analysis easier, and the `Block` constructor is modified to include a list of procedures:

```
type ProcDecls p k = [(p, k)]
data Block v k = Block (VarDecls v k) (ProcDecls v k) k
```

8.2 Representing Procedures in WebAssembly

While allows procedures to be nested inside each other, meaning procedures cannot be directly translated into WebAssembly function as functions *cannot* be nested within each other. Therefore, procedures need to be flattened out into global functions - closure conversion needs to be performed [2]. This presents problems as procedures may capture variables from their outer scope. In order to convert procedures into WebAssembly functions, the compiler needs a way to compile code such that global WebAssembly functions can modify values of variables in other functions. Different methods to implement this are briefly discussed below.

8.2.1 Non-solution - Disallowing Captures

A non-solution is to simply disallow variables to be captured and mutated. However, since While procedures cannot return values, this method destroys any way to pass values back out to the caller of a procedure.

8.2.2 Functions as Objects

Another option is to store the state of a function in an object on a heap. When a procedure finishes execution, the caller could lookup the new values of variables by looking up the member variables of the callee object. This method may be suited to languages which can return procedures from other procedures, but is overly complex in this situation. Also, since WebAssembly is in MVP it does not yet support garbage collection, making this method difficult to implement.

8.2.3 Variables as Pointers

In C, pointers to variables can be created which allow modification of a variable in one function to affect the value of a variable stored in another function. Using pointers to represent variables in WebAssembly is a viable solution, but involves some extra work. So far, variables have been stored as local variables (implemented in Chapter 4), however, WebAssembly does not support pointers to local variables, and so the pointer mechanism must be implemented manually.

This can be done by maintaining a stack of variables using WebAssembly's ability to define contiguous blocks of memory. A stack pointer (SP) is also used to point to the next available space to store a variable. Unfortunately, this turns every variable access from a single `get_local` instruction into pushing the address of the variable onto the stack, followed by a load instruction. For example, the While statement `x := y` is currently compiled to:

```
get_local "y"
set_local "x"
```

By storing variables in memory, this would become something the instructions below (but with different offsets from the stack pointer):

```
get_global "sp"
i32.const 4
i32.sub
i32.load

get_global "sp"
i32.const 8
i32.sub
i32.store
```

Over a large program, executing these extra instructions may have performance implications. However, the number of variables stored in memory can be minimised through the realisation that only variables which are modified by multiple *different* procedures need to be stored in memory. In the program below, variable `x` requires being stored in memory as it is modified in the 'main' procedure, i.e. top-level scope, and also inside procedure *f*. However, variables `y` and `z` can be stored as local variables because neither are mutated in multiple different procedures.

```
x := 0; y := 0; z := 0;
begin
    proc f is x := y;
    call f;
    z := 1
end
```

With this solution, a method of passing variables to invoked functions is also required. This can be done by passing variables as parameters to WebAssembly functions. If a variable is not mutated by the callee, its *value* can simply be passed to the callee. However, if a variable *is* mutated by the callee, the *address* of the variable is given to the callee. This allows the callee to use WebAssembly's `store` instruction to modify the value at that address, hence modifying the value in the caller.

Using this solution, each function is required to keep track of a number of variables. These variables can either be stored as local variables, or in memory in the stack frame of the function. For example, a WebAssembly function f with local variables x, y; a parameter z; and storing two 32-bit integers in memory would look like:

```
(func $f (local $x i32) (local $y i32) (param $z i32)
    get_global "sp"
    i32.const 8
    i32.add
    set_global "sp"

    ...

    get_global "sp"
    i32.const 8
    i32.sub
    set_global "sp"
)
```

Here, the local variables and parameter are defined in the function header and so space for these variables is allocated implicitly by the WebAssembly runtime. However, space for the two 32-bit integers must be explicitly allocated and deallocated. Therefore, each function

body starts with allocating any required space by incrementing the stack pointer, and end by deallocating space through decrementing the stack pointer.

In summary, compiling procedures to efficient WebAssembly involves a lot of work. However, DSLs can be used to ease this an aid compilation.

8.3 Flattening Procedures

As mentioned in Section 8.2, WebAssembly does not support nested functions and so nested procedures need to be flattened into global WebAssembly functions. However, the name shadowing problem with local variables, discussed in Chapter 6, also applies to procedures, which may shadow procedures in their outer scope. By flattening procedures into global functions there would be the possibility for name clashes, and therefore an inability to correctly resolve which function should be called. Fortunately, this can be solved by assigning each procedure a unique name, making use renaming logic supplied by the Renaming DSL from Chapter 6, and demonstrating how the syntax for a single DSL can be composed together multiple times.

Recall from Chapter 7 that the carrier used when folding over the While AST to rename variables was defined as a `Prog` tree with normal syntax `Fresh` and scoped syntax `Rename`. The type `String`, given to `Fresh` and `Rename`, denotes the type of variables in the original AST which are replaced with fresh names.

```
type Carrier = Prog (Fresh String :+: Void) (Rename String :+: Void)
```

To reuse `Fresh` and `Rename` syntax to rename procedures, this syntax needs to be composed into the carrier. Assuming that procedure names are represented using `String`s in the original AST, it may seen sensible to simply compose more `Fresh` and `Rename` syntax:

```
type Carrier
    = Prog (Fresh  String :+: Fresh  String :+: Void)
           (Rename String :+: Rename String :+: Void)
```

The carrier compiles, but GHC cannot disambiguate between the two sets of syntax when attempting to use `fresh` or `rename`. To fix this, a method is required to distinguish between names of variables and names of procedures. To solve this, newtype wrappers can be created to represent variable and procedure names:

```
newtype VarName  = VarName  String
newtype ProcName = ProcName String
```

The carrier is then created using the newtypes, instead of just `String`:

```
type Carrier
    = Prog (Fresh  VarName :+: Fresh  ProcName :+: Void)
           (Rename VarName :+: Rename ProcName :+: Void)
```

The algebra for renaming the While AST requires very little change to use the newtypes instead of just `String`s. For example, when renaming retrieving a variable, use of `fresh` requires wrapping the name of the variable in `VarName`:

```
instance VarExp FreshName :<: f => FixAlg (VarExp Ident) (Carrier (Fix f)) where
    alg (GetVar v) = do
        v' <- fresh (VarName v)
        return (getVar v')
```

The same is done to rename syntax to call a procedure, except the name of the procedure is wrapped in `ProcName` instead of `VarName`. In summary, this section demonstrates how the same DSL can be reused multiple times with aid from newtype wrappers, therefore, expanding the cases `Prog` can be used in.

8.4 Dirty Variables

As discussed in Section 8.2, variables which are modified in two different procedures need to be stored in memory in order for functions to update the values of variables in other functions. For the rest of the project, these variables will be referred to as *dirty variables*. In order to compile procedures, the compiler requires a set of all the dirty variables in a program. To generate this, a scoped DSL is used with the goal of allowing the concise expression of which variables are dirty.

This "Dirty DSL" acts like the Writer monad with the ability to tell some environment that a variable `v` was modified in the current scope. This is captured with the syntax `Modified v k`:

```
data Modified v k = Modified' v k
```

The DSL also has scoped syntax `ModScope` to tell the environment a procedure was entered. If a variable is marked as modified, using `Modified`, inside and outside the scoped continuation of `ModScope`, the variable is marked as dirty. This is realised with the following syntax:

```
data ModScope k = ModScope' k
```

Using this syntax, trees such as those below can be created. The variables y and z are seen to only be modified in a single scope, however, x is seen to be modified at the top-level scope as well as in the nested scope. Therefore, running the DSL will produce the set $\{x\}$ of dirty variables.

```
ex :: Prog (Modified String :+: Void) (ModScope :+: Void)
ex = do
    modified "x"
    modScope (do
        modified "y"
        modified "x")
    modified "z"
```

At a high-level, to calculate the set of dirty variables each scope is assigned an index. A mapping is also maintained from variable names to the index of the scope a variable was first modified in. Using this, the algebra for the DSL can check whether a variable v was modified in two different procedures by comparing the index of the current scope to the index of the scope v was first modified in. To ensure each scope has a unique index, a fresh index can be generated each time a scope is entered.

This algorithm lends itself to being implemented using effect delegation: a Reader can be used to keep track of the current scope's index; a State can be used to maintain the mapping from variable names to the index of the scope a variable was first modified in; the New DSL can be used to create a unique index for each scope; and a Writer DSL can be used to write out the set of dirty variables.

For the implementation of the algorithm, type synonyms are used to make the code more readable. `FirstScope` represents the mapping from variables to the index of the first scope a variable was modified in; `ScopeIdx` represents the index of a scope; and `DirtyVars` represents the set of dirty variables.

```
type FirstScope v = Map v ScopeIdx
type ScopeIdx     = Word
type DirtyVars v  = Set v
```

As in Chapter 6, to implement the semantics of a high-level DSL in terms of lower-level DSLs, a function must transform the high-level syntax into low-level syntax. In this case, the function `delegate` transforms a tree containing "Dirty DSL" syntax, into the syntax of the State, Reader, New, and Writer DSLs:

```
type Op v f
    = State   (FirstScope v) :+: Ask    ScopeIdx :+: New Word :+: Tell (DirtyVars v) :+: f
type Sc v g
    = LocalSt (FirstScope v) :+: LocalR ScopeIdx :+: g

delegate
    :: Prog (Modified v :+: f) (ModScope :+: g) a
    -> Prog (Op v f) (Sc v g)
```

The carrier for the algebra contains the tree of new syntax being built up:

```
type Carrier f g v a = Nest1 (Prog (LowLvlOp v f) (LowLvlSc v g)) a
```

The algebra for `Modified` can now be defined, notably in a very imperative manner: if the variable `v` has not been seen before then a mapping from it to the index of the current scope is created, and program execution continued.

```
algD :: Ord v => Alg (Modified v :+: f) (ModScope :+: g) (Carrier f g v a)
algD = A a d p where
    a :: Ord v => (Modified v :+: f) (Carrier f g v a n) -> Carrier f g v a n
    a (Modified v k) = Nest1 $ do
        varScopes <- get
        case v `Map.lookup` varScopes of
            Nothing -> do
                currScopeIdx <- ask
                put (Map.insert v currScopeIdx varScopes)
                runNest1 k

            ...
```

If `v` has been seen before, then compare the index of the scope it was first modified in, `scIdx`, to the index of the current scope `currScopeIdx`. Only if they are different is `v` added to the set of dirty variables.

```
            ...
            Just scIdx -> do
                currScopeIdx <- ask
                if scIdx == currScopeIdx
                    then runNest1 k
                    else do
                        tell (Set.singleton v)
                        runNest1 k
```

The demotion algebra is defined in an equally imperative manner. The semantics of `ModScope` executes the continuation `k` in an environment with a fresh scope index `newScIdx`. Therefore, any usages of `Modified` will be given `newScIdx` when the scope index is queried.

```
d :: (ModScope :+: g) (CarrierD f g v a ('S n)) -> CarrierD f g v a n
d (ModScope k) = Nest1 $ do
    newScIdx <- new
    (NS1 run') <- localR newScIdx (runNest1 k)
    run'
```

Finally, a handler can be defined which strips away `Modified` and `ModScope` syntax, returning a tree whose result type contains a set of dirty variables.

```
handleDirtyVars :: Prog (Modified v :+: f) (ModScope :+: g) a -> Prog f g (a, DirtyVars v)
```

The implementation of this DSL is intended to demonstrate how the logic of a fairly complex algorithm can make use of effect delegation to allow its semantics to be expressed in an imperative manner making the code easier to follow and understand.

8.5 Code Generation

Compiling procedures requires a lot of extra information to be known during compilation. For example, just to properly compile variables requires knowledge of whether a variable is stored in memory or not, the address of a variable in memory, and whether the variable is passed into the function as a parameter.

While this information can be calculated, making it available during compilation would be cumbersome if it needed to be explicitly passed into the functions representing the compilation algebra. To solve this, another *Code Generation*, or CodeGen, DSL can be created to provide a monadic context in which compilation can occur. The syntax of the DSL will allow information such as the offset of a variable from the stack pointer to be easily retrieved:

```
data Emit k = VarSPOffset' SrcVar (Word -> k) | ...
```

Section 8.2 claimed that While procedures needed to be flattened in order to be compiled into global WebAssembly functions. The CodeGen DSL will also provide the syntax to emit a WebAssembly function, represented using `Func`, to be appended to some global list of functions:

```
data Emit k = ... | EmitFunc' Func k
```

It is important to notice that the DSL has a sense of the *current* function - for example, the offset of variables from the stack pointer will be different depending on the function. This

implies a need to change the current function during compilation, which is done using the following scoped syntax. Inside the scope, the mapping from variable names to their offset from the stack pointer is changed.

```
data Block k = FuncScope' (SrcVar -> SPOffset) k
```

In an initial implementation of the semantics, it is tempting to have some stateful environment to which the algebra of the DSL appends functions to `funcs`, and reads parameters such as `globalSPName`.

```
data Env = Env {
    funcs        :: [Func]
  , globalSPName :: GlobalName
  , spOffset     :: SrcVar -> SPOffset
    ...
}
```

The carrier for the semantics would like that of the State DSL: *Carrier a (n :: Nat)* such that *Carrier a 'Z* $\cong$ *Env* $\rightarrow$ *(a, Env)*, and *Carrier a ('S 'Z)* $\cong$ *Env* $\rightarrow$ *(Env* $\rightarrow$ *(a, Env), Env)*, and so on.

This approach works, however, it lacks safety as the read-only-ness of parameters, such as `globalSPName`, is not enforced. It also does not enforce that the list of functions is only appended to; a bug could accidentally replace the accumulated list with the empty list.

To solve this problem, the compiler makes use of effect delegation to segregate the environment into stateful and read-only sections. Notice below that `funcs` has been removed from `Env` and instead a Writer (`Tell`) is used to aggregate a list of functions. A Reader is also used to ensure the environment `Env` of read-only parameters is not accidentally modified.

```
data Env = Env {
  , globalSPName :: GlobalName
  , spOffset     :: SrcVar -> SPOffset
    ...
}

type Ctx     f g = Prog (Tell [Func] :+: Ask Env :+: f) (LocalR Env :+: g)
type Carrier f g = Nest1 (Ctx f g)
```

Now when defining the algebra it is impossible to accidentally reset the list of global functions, as `tell` is the only syntax allowed to be used with a Writer. Read-only variables also cannot also not be modified:

```
alg :: Alg (Emit :+: f) (FuncScope :+: g) (Carrier f g a)
alg = A a d p where
    a :: (Emit :+: f) (Carrier f g a n) -> Carrier f g a n
    a (EmitFunc func k)   = Nest1 $ tell [func] >> runNest1 k
    a (VarSPOffset v fk)  = Nest1 $ ask >>= \env -> runNest1 (fk (spOffset env v))
    ...
```

However, it would technically be possible to create a local reader scope using `localR`, therefore changing the environment returned by `ask`. This could be prevented through the creation of a Reader handler which only handled non-scoped syntax, therefore allowing `LocalR` to be omitted from `Ctx` above. This would allow attempted usage of `localR` to generate a compile-time error.

In summary, the creation of the CodeGen DSL demonstrates how effect delegation can be used to create safer, less error-prone code by limiting the power available to the programmer.

8.6 Compilation

As well as demonstrating how procedures can be compiled, this section demonstrates how natural the translation from While to WebAssembly can be made by using `Prog` by presenting and discussing other notable parts of compilation.

8.6.1 Procedures

Use of the CodeGen DSL presented in section 8.5 provides all the syntax required to compile procedures. In order to use this syntax, the following context is used during compilation:

```
type CodeGen = Prog (Emit :+: Void) (FuncScope :+: Void)
```

Using this, procedures can be compiled using the `compProc` function below. This takes the name of the procedure to compile, and the body of the procedure, and uses the CodeGen DSL to write out the compiled function to the list of global functions.

For illustrative purposes, the function has been simplified. At a high-level, the function calculates the offset from the stack pointer (SP) of variables used in the function, i.e. `spOffsets`. This is supplied to `funcScope` to create a scope in which SP offsets are specialised for the function being compiled. Inside this scope, the body of the function is compiled, and the WebAssembly function appended to the list of global functions using `emitFunc`.

```
compProc :: String -> CodeGen WASM -> CodeGen ()
compProc pname body = do
    let spOffsets = ...
    funcScope spOffsets (do
        bodyWasm <- body
        let func = Func pname bodyWasm
        emitFunc func)
```

8.6.2 Retrieving Variable Values

The WebAssembly instructions to retrieve the value of a variable depends on how the variable is stored. If the *value* of a variable is stored in a local variable or parameter then the `get_local` function can be used to retrieve its value. However, if the variable is stored in memory, to access its value the address needs to be looked up and a load performed. Representing this sequential logic using `Prog` is supremely clean.

Firstly, the syntax supplied by the CodeGen DSL is used to retrieve the name of the global variable representing the stack pointer, as well as the number of bytes the variable `v` is offset from the stack pointer:

```
localPtrAddr :: SrcVar -> CodeGen WASM
localPtrAddr v = do
    sp     <- spName
    offset <- varSPOffset v

    ...
```

The value of the stack pointer is then pushed onto the stack using `getGlobal sp`; the offset of the variable from the stack pointer is pushed into the stack; finally, the address of the variable is calculated by subtracting.

```
    ...

    return $ case offset of
        0 -> getGlobal sp
        x -> do
            getGlobal sp
            constNum (fromIntegral offset)
            binOp SUB
```

Since the DSL is embedded in Haskell, Haskell's syntax can also be mixed in with the DSL, in this case optimising the outputted WebAssembly.

8.6.3 Arithmetic and Boolean Expressions

One of the downsides of using the CodeGen DSL is that WebAssembly contained within it must be extracted before it can be combined with other WebAssembly. For example, to compile addition requires unwrapping both operands in order to combine them with WebAssembly's `add` instruction:

```
instance FixAlg AExp (CodeGen WASM) where
    alg (Add x y) = do
        wx <- x
        wy <- y
        return (do wx; wy; binOp ADD)
```

Since this pattern is repeated often throughout the semantics of compilation, it can be factored out into an operator which can be interspersed between operands of type `CodeGen`:

```
infixr 0 >>>

(>>>) :: CodeGen WASM -> CodeGen WASM -> CodeGen WASM
(>>>) x y = do x' <- x; y' <- y; return (x' >> y')
```

Making the algebra for addition much more readable:

```
instance FixAlg AExp (CodeGen WASM) where
    alg (Add x y) = x >>> y >>> return (binOp ADD)
```

This example shows how Haskell functions can be used to extend the syntax of a language.

8.6.4 Skip Statements

Skip statements in While perform no action and so could be compiled to their WebAssembly equivalent - `nop`.

```
instance FixAlg Stm (CodeGen WASM) where
    alg (Skip) = return nop
```

However, to decrease code size no instruction needs to be emitted at all! `Prog` is used to represent WebAssembly, with the result type of unit. Therefore, no instruction can be easily represented using `return ()`:

```
instance FixAlg Stm (CodeGen WASM) where
    alg (Skip) = return (return ())
```

This demonstrates another advantage of `Prog`. If a deep embedding was used to represent WebAssembly, the datatype would need to be edited to contain a "no instruction" constructor. By using `Prog` this is provided for free.

8.6.5 While Loops

While loops are also interesting and are good demonstrators of how naturally returned WebAssembly can be created using `Prog`.

WebAssembly uses scoped constructs, as opposed to jumps, in order to create loops. Scoped constructs come in a several flavours including `block` and `loop`. Each scoped construct introduces an implicit label, which is referenced by branch instructions to be used as a branch target. The indexing of labels is relative to nesting depth, for example, label 0 refers to the innermost scoped instruction; label 1 refers to the scoped instruction encasing the innermost scoped instruction; and so on.

Branches also come in several flavours including unconditional `br` and conditional `br_if` branches. The label supplied to the branch instruction dictates the semantics of the branch. If the label supplied to a branch is that of a `block` then control flow jumps to the end of the block. Whereas, if branching to a `loop` control flow jumps back to the start of the loop.

In the program below, from the perspective of instructions in the body of the loop the `block` has an implicit label `1`, and `loop` has the implicit label `0`. `br_if` causes control flow to jump to the end of the program if the top of the stack contains a non-zero number. Whereas `br` causes control flow to jump unconditionally to the start of the loop, i.e. `i32.const 0`.

```
block
  loop
    i32.const 0
    br_if 1
    br 0
  end
end
```

With this knowledge, the While loop can be compiled using the algebra below, which very closely resembles the indented WebAssembly:

```
instance FixAlg Stm (CodeGen WASM) where
    ...
    alg (While cond body) = do
        condWasm <- cond
        bodyWasm <- body
        return (
            block (
                loop (do
                    condWasm; uniOp NOT; brIf 1
                    bodyWasm
                    br 0)))
```

8.7 Summary

This section gives further examples of `Prog`, effect handlers, and effect delegation in the concrete setting of compiling While with procedures into WebAssembly. More examples of the niceties of using `Prog` are also discusses by examining how While is compiled, with the goal of promoting use of `Prog`.

Chapter 9

Left vs Right Associativity

The work done so far has been to allow for the easy creation of modular DSLs, however, there potential pitfalls into which unaware developers may fall: left associative `Prog` trees have catastrophically worse performance than right associative trees. The problem is explored using a simple example with binary trees. Once the reason why the performance is so much worse has been established, Smart Views [9] can be applied to the implementation of `Prog` in order to fix this issue.

9.1 Binary Trees

Given a binary tree structure:

```
data Tree = B Tree Tree | L
```

The following function can be defined to substitute the leaves of the first tree with that of the second tree:

```
substitute :: Tree -> Tree -> Tree
substitute (L)     t = t
substitute (B x y) t = B (substitute x t) (substitute y t)
```

If substituting together two trees, there is a choice of how to bracket the expression to make it either left or right associative:

```
left  x y z = (x `substitute` y) `substitute` z
right x y z = x `substitute` (y `substitute` z)
```

In the right associative case, each tree is only traversed once by `substitute`. However, in the left associative, case `x` is traversed *twice*. Once to replace the leaves of `x` with `y`, and again when replacing the leaves of `x` `` `substitute` `` `y` with `z`. This increases the runtime cost of the left associative expression and gives clues as to why left associative `Prog` trees are slow.

9.2 Prog Trees

The structure of a binary tree can be recreated using `Prog` by supplying an appropriate `f` and `g`. The `Branch` data type represents the `B` constructor from the binary tree. The tree has no scoped constructors, and leaves are represented through `Prog`'s `Var` constructor.

```
data Branch k = B k k
data Void k

type Tree = Prog Branch Void ()
```

The substitution function for binary trees is essentially monadic bind for `Prog`. A `Var` (represented as leaf `L` in a binary tree) is replaced using the function `f`. The cases for `Op` and `Scope` map over subtrees, recursively applying bind to them. This has the same structure as the case for branches in `substitution`, which recursively substituted the subtrees of a branch.

```
Var x    >>= f = f x
Op op    >>= f = Op (fmap (>>=f) op)
Scope sc >>= f = Scope (fmap (fmap (>>=f)) sc)
```

Therefore, the same associativity problem is also present in `Prog`. For example, in the expression below, the tree `x` must be traversed twice using `(>>=)`.

```
left :: Tree -> Tree -> Tree -> Tree
left x y z = (x >>= const y) >>= const z
```

This problem does not only occur when trying to represent binary trees with `Prog`; the same problem occurs in all `Prog` trees. For example, the State DSL from Chapter 6 can be benchmarked to compare how associativity affects runtime performance. The benchmarks performed some number of *get* and *put* operations in right associative

```
benchRight :: Int -> State Int ()
benchRight n = forM_ [1..n] $ \_ -> do
    s <- get
    put (s+1)
```

and left associative trees:

```
benchLeft :: Int -> State Int ()
benchLeft 0 = return ()
benchLeft n = do
    benchLeft (n-1)
    s <- get
    put (s+1)
```

The benchmarks were run in the using Criterion with GHC 8.0.2 on a Dell XPS with a 2.6 GHz GHz Intel Core i7, with 16 GB Ram running Ubuntu 18.04.2, and showed left-associative trees to be 83 times slower than right-associative trees!

9.3 Smart Views

Smart views [9] provide a way to solve the slow runtime of left associative trees by transforming them into right associative trees. To use this method, a new constructor `(:>>=)` is added to

`Prog`:

```
data Prog f g a
    = Var' a
    | Op' (f (Prog f g a))
    | Scope' (g (Prog f g (Prog f g a)))
    | forall x. (Prog f g x) :>>= (x -> Prog f g a) -- New
```

This is used to encode the bind operation, and so the monad instance for `Prog` is modified to make `(>>=)` equal to `(:>>=)`:

```
instance (Functor f, Functor g) => Monad (Prog f g) where
    return = Var'
    (>>=) = (:>>=)
```

The purpose of the constructor is to allow pattern matching on in order to distinguish between left and right associative trees, and hence transform one into the other (discussed later). However, working with `Prog` in this state would be cumbersome as every function pattern matching on `Prog` would have to be extended with a case for `(:>>=)`.

To solve this, `Prog` will be translated to a new data type `ProgView` which has constructors to represent `Var`, `Op`, and `Scope`, but *not* `(:>>=)`. By translating to `ProgView`, functions over `Prog` can retain their original structure and not need to be adapted with an extra case for `(:>>=)`.

```
data ProgView f g a
    = VarV a
    | OpV (f (Prog f g a))
    | ScopeV (g (Prog f g (Prog f g a)))
```

To translate from `Prog`'s `Var`, `Op`, and `Scope` constructors into `ProgView`, the equivalent constructor is used:

```
viewP :: (Functor f, Functor g) => Prog f g a -> ProgView f g a
viewP (Var' x)            = VarV x
viewP (Op' op)            = OpV op
viewP (Scope' sc)         = ScopeV sc
...
```

To translate `Prog's` new `(:>>=)` constructor, the original logic of bind is substituted into `viewP`, since the bind operator for `Prog` was modified to simply construct using `(:>>=)`:

```
...
viewP (Var'   a  :>>= f)  = viewP (f a)
viewP (Op'    op :>>= f)  = OpV (fmap (:>>= f) op)
viewP (Scope' sc :>>= f)  = ScopeV (fmap (fmap (:>>= f)) sc)
...
```

Finally, the *key* to transforming left associative trees into right associative trees is to pattern match on left association, transforming it using the associativity law for monads [3]:

```
...
viewP ((m :>>= f) :>>= g) = viewP (m :>>= \x -> f x :>>= g)
```

This solution works, but is not currently ideal as `viewP` will need to be applied to `Prog` before pattern matching on the "original" constructors can be performed. For example, the `fold` over `Prog` would need to be redefined to use a case statement:

```
fold :: (Functor f, Functor g) => Alg f g a -> Prog f g (a n) -> a n
fold alg prog =
    case viewP prog of
        VarV   x  -> x
        OpV    op -> a alg (fmap (fold alg) op)
        ScopeV sc -> d alg (fmap (fold alg . fmap (p alg . fold alg)) sc)
```

To solve this, *ViewPatterns* and *PatternSynonyms* can be used to apply `viewP` while pattern matching, therefore removing the need for a case statement and making the code more readable.

```
pattern Var   a  <- (viewP -> VarV a)
pattern Op    op <- (viewP -> OpV op)
pattern Scope sc <- (viewP -> ScopeV sc)
```

Using this, the implementation of `fold` looks exactly the same as the original before using Smart Views, however, `Var`, `Op`, and `Scope` are actually patterns. Therefore this method provides a way to solve issues with left associative trees, with no cost to the programmer using the 'interface' to `Prog`.

```
fold :: (Functor f, Functor g) => Alg f g a -> Prog f g (a n) -> a n
fold alg (Var x)    = x
fold alg (Op op)    = a alg (fmap (fold alg) op)
fold alg (Scope sc) = d alg (fmap (fold alg . fmap (p alg . fold alg)) sc)
```

9.4 Benchmarks

Running the same benchmarks using the State DSL confirms that Smart Views improves the runtime performance of left associative trees, giving them the same performance as right associative trees. No performance is sacrificed by using Smart Views to represent right associative trees as well, with benchmarks giving no significant difference in speed.

9.5 Summary

Analysis of the performance of left vs right associative trees showed a performance discrepancy between the two. Exploring the problem in the context of binary trees allowed insight into the problem to be gained which pointed towards Smart Views being a possible solution. Benchmarking this demonstrated how the technique successfully gave left and right associative trees the same runtime performance, therefore, meaning developers using `Prog` would be no longer be required to consider this while implementing DSLs.

Chapter 10

Evaluation

Now the implementation of modular DSLs using Prog has been implemented and their use in a larger scale project completed, this section aims to evaluate how well `Prog` worked and could be adapted to be used to create modular DSLs, as well as considering other factors to give a more rounded view.

10.1 Modularity and Extensibility

Data Types à la Carte proved an effective method to allow modular DSLs to be created which had semantics in a single domain. This proved a useful technique for compiling While to WebAssembly, as well as pretty printing WebAssembly. This was due to A la Carte's ability to compose together algebras, created in individual typeclass instances, in order to give semantics to an entire tree `Prog` tree at once. Adapting this technique to be used with `Prog` was also relatively pain-free, as the original paper discussed the implementation with the Free monad, upon which `Prog` builds.

Giving semantics to the entire tree at once was not always desirable as this limited DSLs to all having the same semantic domain. Therefore, Effect Handlers were also adapted to be used with `Prog`, allowing handler functions to give semantics only to interesting syntax in a tree. This technique was found to drastically speed up the time taken to implement programs in a DSL as it made it easy to simply compose in new syntax as required. An example of this was the semantics for renaming variables and procedures in the While AST; initially only variables were renamed, however, extending renaming to handle to procedures was a case of simply composing another instance of the Rename DSL.

10.2 Ease of Use

One of the areas use of `Prog` shone was providing the ability to write readable, concise, imperative code. For example, this made it easy to define the semantics of generating dirty variables in Chapter 6. The algorithm for generating the set of dirty variables mapped cleanly into writing the DSL through using scoped syntax, and syntax composed from multiple different DSLs.

The ability to create readable code was also true of other areas, such as defining the transformation from While to WebAssembly. In particular, compilation of While loops made use of nested constructs to cleanly represent outputted WebAssembly. It is clear the transformation that is occurring, even to someone unfamiliar with the codebase.

```
alg (While cond body) = do
    condWasm <- cond
    bodyWasm <- body
    return (
      block (
        loop (do
           condWasm; uniOp NOT; brIf 1
           bodyWasm
           br 0)))
```

Since the creation of instances of DSLs are easier to write, it is easier to adhere to good programming practices such as Test Driven Development. For example, use of `Prog` to represent While and WebAssembly made it easier to increase code coverage of the compiler since do-notation allowed ASTs to be easily created for unit tests.

By creating small, modular DSLs (composed into larger DSLs) it is further easier to perform unit testing, as the purpose of each module better encapsulated. A good example of this is the Renaming DSL, which was not constrained to work only with While ASTs, This made unit testing easier as entire ASTs did not need to be constructed.

10.3 Safety

The use of coproduct allows AST transformations to be precisely described by specifying the syntax in the tree before and after the transformation, hence acting like pre- and post-conditions. This is a valuable ability in a compiler as it can be used instead of simply assuming certain syntax does not appear in an AST, or instead of defining an entirely new data structure to represent the transformed AST.

This modularity was further used to increase safety, such as when defining the semantics of Code Generation. By composing together Writer and Reader, instead of using a single State, the type system disallowed modification to read-only variables or resetting of the list of global functions. This principle is applicable to many other cases, for example, the `IO` monad is capable of performing many different actions including file modification, console operations, and exceptions. Therefore, if `IO` is used in a program, it is not possible to tell exactly what it will do without examining the source code. By defining DSLs for each type of action, hence making it explicit the types commands that will occur in a program, this can be solved. For example, the following DSL could be defined for file actions:

```
data File     k = ReadLine (String -> k) | WriteLine String k
data OpenFile k = Open FilePath k
```

Through composition, this could be composed with DSLs for exceptions and console operations giving the DSL the normal capabilities of `IO`:

```
type IO' = Prog (File :+: Console :+: Throw) (OpenFile :+: Catch)
```

The difference between this and standard `IO`, however, is that the operations are made explicit in the types, and so programmers can more easily understand what actions a section of code can perform.

10.4 Defining Semantics

Creation of `OpAlg` and `ScopeAlg` typeclasses allowed independent specification of semantics, therefore increasing modularity by allowing extensions to the semantics of normal and scoped syntax without modification to previous semantics. Use of effect handlers also allowed semantics to be given to trees in a modular manner. While these techniques improved defining the semantics of `Prog` trees, there are still some issues. In order to fold over a `Prog` tree requires the use of a nested carrier thus complicating definition of an algebra. This impacts code readability by requiring unwrapping and wrapping of the carrier and use of dependent typing; factors which potentially have negative impacts on development times by being objectively harder to work with and understand compared to algebras over Free trees, for example.

However, these implementation details do not affect the interface provided to use DSLs, for example, the `handleState` function from Chapter 6 does not expose a nested carrier nor dependent typing used in the algebra. Therefore, the price of added complexity is only paid by the library developer, and not necessarily users of it, provided they can compose the effects desired from those provided.

10.5 Reusability

One of the aims was to investigate code reuse. Making use of Effect Delegation allowed high-level DSLs to be implemented in terms of lower-level DSLs. Because of this, generic DSLs can be created which are applicable to a wide range of tasks and hence can be reused. To name a few instances of reuse: generating a set of dirty variables and renaming both made use of a common State effect; generating the set of dirty variables, and generating a list of compiled functions both used a Writer; and the algebra for renaming the While AST made use of two instances of the renaming effect handler, for variables and procedures. The ability to allow easy reuse of code helped contribute to faster development.

10.6 Scalability

One of the aims was to investigate how well the use of modular, scoped DSLs scaled to larger projects. One of the notable points found through creation of the compiler was how use of the effect delegation allowed for abstraction, building up layers of DSLs each with increasingly complex semantics but retaining simple syntax. For example, the syntax of Reader is simple with only *ask* and *local* commands. The semantics are also fairly simple, either returning the environment, or entering a scope with a new environment. The syntax of the Renaming DSL is also simple, with only *fresh* and *rename* syntax. However, the underlying semantics are much more complicated. This use of abstraction allows for creation of concise programs which are easy to reason about, even when composed into a large, complex program.

10.7 Coproduct Problems

There are some pitfalls with the current implementation. Since the coproduct acts to box different types, it makes it impossible to restrict which types are stored inside constructors. For example, a subset of statements and boolean expressions in While are represented as:

```
data BExp k = T | And k k | ...
data Stm  k = While k k | ...
```

Clearly, the body of a loop should contain a statement, but there is nothing in the types to prevent a boolean expression from being placed inside instead, for example. This can be mitigated through use of index types, which are used to tag constructors with the type they 'produce':

```
data Type = Boolean | Void | ...
```

Using this, statements can be represented using a GADT to tag constructors with the correct result type [6]:

```
data Stm ty k where
    While :: k Boolean -> k Void -> Stm Void k
    ...
```

However, indexed types must be used everywhere now - indexed functors, monads, continuations. This makes working with, and defining algebras over, the trees more difficult therefore potentially increasing development times.

10.8 Transformation Problems

As discussed in Chapter 7, there are currently issues transforming a `Prog` tree inside some context. Yhis hindered representing While using `Prog`, but, all other DSLs used in the compiler continued to use `Prog` because they did not need to be transformed. This issue is an important one to solve before `Prog` can be used to represent ASTs, however, for representing most DSLs `Prog` is still applicable in its current state.

10.9 Performance

Expanding on the performance the evaluation performed in Chapter 9, it is important to measure the penalty paid for composition of effect handlers - how does the runtime vary with the number of handlers composed.

To evaluate this, benchmarks were run where increasing numbers of State DSLs were composed together, and a number of *get* and *put* operations performed. For example, the benchmark to compose four DSLs together is given below, where `I1`, `I2`, `I3`, and `I4` are newtype wrappers around `Int`:

```
bench n = forM_ [1..n] $ \_ -> do
    I1 s <- get
    put (I1 (s+1))
    I2 s <- get
    put (I2 (s+1))
    I3 s <- get
    put (I3 (s+1))
    I4 s <- get
    put (I4 (s+1))
```

The benchmarks were run in the using Criterion using GHC 8.0.2 on a Dell XPS with a 2.6 GHz GHz Intel Core i7, with 16 GB Ram running Ubuntu 18.04.2. The results are given in Figure 10.1 and show that as the number of DSLs composed together increased, so did the runtime. As a baseline, the conventional definition of the State monad transformer from MTL was also evaluated, and had an average runtime around 1000 times faster than the `Prog` implementation.

Examining the memory allocation characteristics of `Prog` and MTL running the benchmarks helps to shed light on why the performance of MTL is so much better. Using Weigh, details about garbage collection and memory allocation were collected running each of the benchmarks. This showed that `Prog` allocated around 200 times more memory than MTL, which meant much more time was also spent garbage collecting. The extra memory allocation and garbage collecting is why `Prog` is so much slower.

This extra allocation can be explained by examining how handlers for DSLs are composed. To handle a DSL composed of a Reader, Writer, and Exceptions requires the handlers for each to be run:

```
handle = handleWriter . handleReader 0 . handleExc
```

The composition of handlers produces an intermediate tree, requiring memory to be allocated, and hence slowing execution. However, there is hope - it has been demonstrated how handlers acting on Free trees can be fused together in order to eliminate the creation of intermediate trees [14]. This fusion acts like stream fusion, which GHC performs, for example, to eliminate the intermediate creation of lists in the expression:

```
map show $ map (+1) $ map (*2) [1, 2, 3]
```

The results of fusing handlers of Free monads gave a speedup of up to 300 times on the performed benchmarks, giving performance equal to or better than monad transformers [14]. Since the structure of `Prog` closely resembles that of the Free monad, it is likely this research could be applied to `Prog` without too much modification to code given in the paper. Applying this to `Prog` would be an interesting, and potentially very beneficial, research avenue.

10.10 Summary

Applying Data Types à la Carte, Effect Handlers, and use of Effect Delegation to `Prog` allowed extensible DSLs to be created. These methods also afforded many other benefits including easy to define imperative programs; widely reusable code; and type safety. There are still areas which require work, but `Prog` has proved itself to be very capable.

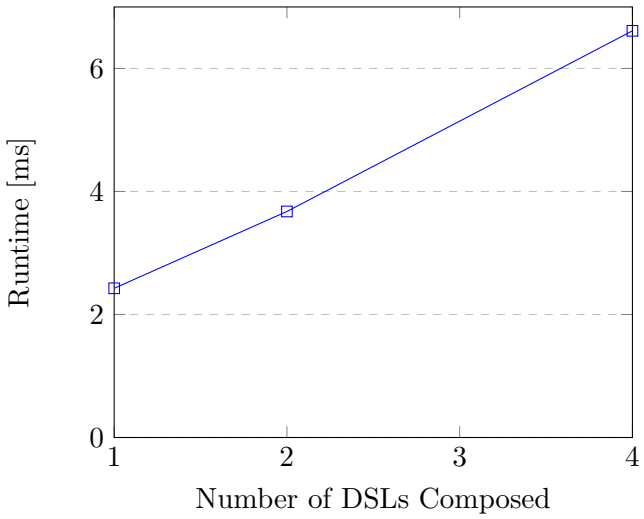

Figure 10.1: Runtime vs Number of DSLs Composed using `Prog`

Chapter 11

Conclusion

DSLs provide a way to encapsulate the semantics of a domain in a concise and readable manner. Representing a DSL in a host language gives many design choices, exploration of these choices demonstrated how `Prog` builds on the Free monad to solve issues representing scoped syntax, a vital feature of many DSLs. However, since the Free monad is more mature, work has been put into allowing DSLs created using it to be extensible, allowing their syntax and semantics to be changed after-the-fact. Before `Prog` can be used to represent real DSLs, it needs this ability, and exploring this is one the contributions of this project.

Data Types à la Carte is a technique used to create extensible DSLs using `Fix` and `Free`. It was demonstrated how this technique could also be applied to `Prog`, making use of the coproduct to separately compose syntax for normal and scoped operations. By defining typeclass instances for normal and scoped syntax, the semantics of these trees could be evaluated. This showed how Data Types à la Carte allowed for extensible DSLs containing scoped syntax.

Because Data Types à la Carte evaluates the *entire* tree at once, it is not well suited to applications where different syntax requires different semantic domains. To fix this problem, effect handlers peel away specific syntax, translating it into semantics and leaving uninteresting syntax. It was demonstrated how this technique could be used with `Prog`, providing another tool with which to create extensible DSLs. The flexibility of this meant it was used extensively throughout the construction of the compiler, and likely has many other applications outside of compiler construction.

Defining the semantics of a DSL often resulting in duplication of code, for example, the semantics of both renaming and dirty variables involved a notion of state. This suggests commonalities can be factored out. This factoring out manifests itself as a DSL which encapsulates the common syntax and semantics. Effect Delegation can then be used to make use of factored out syntax, implementing a high-level DSL in terms of a lower-level DSL, in a sense compiling from one DSL to the another. It was demonstrated how this technique could be applied to `Prog`, and how it helps reuse code and make the semantics of the high-level DSLs more readable and concise.

With these contributions, it is possible to use `Prog` to create scoped DSLs representing complex logic, as was demonstrated with DSLs for renaming variables and procedures, computing dirty variables, and code generation. However, another consideration is how well Prog performs at runtime - a factor which could affect wider audience adoption. Through benchmarking it was found that the runtime performance of left associative trees was vastly slower than that

of right associative trees. To solve this issue, Smart Views were applied to `Prog` in order to transform left associative trees into right associative trees. Benchmarking this again showed tree associativity no longer played a role in determining runtime.

11.1 Future Work

Analysis of the runtime of composed effect handlers in Chapter 10 concluded with a discussion on future work to improve the runtime characteristics of `Prog` through fusing handlers together. This section discusses some other avenues which could be interesting to explore.

11.1.1 Transforming Prog

The main problem encountered when using Prog was an inability to transform its structure in some monadic context, as discussed in Chapter 7. This problem was due to the continuation, required to continue folding over the AST, being nested within the monadic context *and* the context of the new `Prog` tree being constructed. Therefore, it was not possible to access this continuation from the outer monadic context.

This problem has already been solved in another context - monad transformers. The example below demonstrates that, given a monad transformer stack containing a State and a Reader, bind can be used to get the value nested inside both:

```
ex :: StateT String (Reader String) Int -> StateT String (Reader String) Int
ex prog = do
    x <- prog
    return (x*2)
```

Therefore, a possible solution to this problem would be to combine monad transformers and `Prog`. Solving this problem would allow `Prog` to be used to represent trees which can be transformed in some context. This is a vital ability to have for `Prog` to be used in future compilers, for example, GHC performs compilation by transformation in order to compile Haskell [10].

11.1.2 Dependent Typing

The most complex part of the compilation process is arguably the transformation from While to WebAssembly. However, GHC currently gives no feedback on whether the emitted WebAssembly is valid or not. For example, it would not be valid to push only *one* value onto the stack before executing an add instruction, which pops two values from the top of the stack.

Dependent typing could be used to solve this problem, which allows *types* to be predicated on *values*. With this ability, the number of items on the WebAssembly stack could be kept track of at compile time using a Nat-indexed version of `Prog`. This would allow GHC to ensure the stack is always in a valid state, e.g. instructions are not emitted which consume more values than are on the stack.

This idea could be applied to other areas, like the DSL for file operations presented in Chapter 10. Currently, a program could be constructed that attempts to read or write a file before any have been opened. Through dependent typing, GHC could check that only after a file has been opened can it be read from or written to. Clearly, pursuing this research has the potential to eliminate an entire class of bugs.

11.2 Reflection

The goal of this thesis was to allow scoped, extensible DSLs to be created. By exploring existing methods of achieving this, and combining them with new research I believe this goal was achieved. In the future, I hope others may benefit from this contribution.

Bibliography

[1] Alpha conversion. https://wiki.haskell.org/Alpha_conversion. Accessed: 2010-05-09.

[2] The essence of closure conversion. http://siek.blogspot.com/2012/07/essence-of-closure-conversion.html. Accessed: 2010-05-09.

[3] Monad laws. https://wiki.haskell.org/Monad_laws. Accessed: 2010-05-09.

[4] Polysemy: Chasing performance in free monads. https://www.youtube.com/watch?time_continue=983&v=-dHFOjcK6pA. Accessed: 2010-05-09.

[5] Webassembly roadmap. https://webassembly.org/roadmap/. Accessed: 2010-05-09.

[6] Laurence E Day and Graham Hutton. Compilation à la carte. In *Proceedings of the 25th symposium on Implementation and Application of Functional Languages*, page 13. ACM, 2013.

[7] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *ACM SIGPLAN Notices*, volume 52, pages 185–200. ACM, 2017.

[8] Paul Hudak. Domain-specific languages. *Handbook of programming languages*, 3(39-60):21, 1997.

[9] Mauro Jaskelioff and Exequiel Rivas. Functional pearl: a smart view on datatypes. *ACM SIGPLAN Notices*, 50(9):355–361, 2015.

[10] SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993.

[11] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications*, volume 104. Springer, 1992.

[12] Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. Syntax and semantics for operations with scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 809–818. ACM, 2018.

[13] Wouter Swierstra. Data types à la carte. *Journal of functional programming*, 18(4):423–436, 2008.

[14] Nicolas Wu and Tom Schrijvers. Fusion for free. In *International Conference on Mathe-*

matics of Program Construction, pages 302–322. Springer, 2015.

[15] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. 2014.