Rewrite rules for the Disciplined Disciple Compiler

By
Amos Robinson
Supervisor: Ben Lippmeier
Assessor: Manuel Chakravarty

A thesis submitted for the degree of
Bachelor of Computer Science (Hons)

THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY · AUSTRALIA

Computer Science and Engineering,
The University of New South Wales.

December 2012

Abstract

Optimising compilers can only use shallow reasoning by examining the literal meaning of program text, and are morally bound not to change the meaning of a program. Extra information about the intent of programs can be used for further optimisations, but this is usually only available to the programmer. A method allowing programmers to express intent as rewrite rules in the presence of effect typing is developed.

List fusion combines list producers with consumers, removing intermediate lists. This is only applicable if the producer and the consumer's effects do not interfere. A version of foldr/build fusion that only fuses non-interfering effects is shown.

The language used is Disciple, a strict-by-default Haskell dialect with effect typing.

Contents

Thanks

Put the tape on erase, rearrange a face. We always liked Picasso anyway.

(DEVO – Through Being Cool)

Praise “Bob”, hail “Connie”.
Thanks Kathleen.

Chapter 1

Introduction

Consider a function that takes a list and creates an intermediate list by adding one to each element. It then constructs the result list by squaring each element in the intermediate list. In Haskell, this can be expressed as `map square (map (+1) the_list)`.

It is possible to remove the intermediate list and perform both operations in one iteration: `map (\x -> square (x+1)) the_list`. This new version is more efficient as it performs fewer allocations and requires less memory.

The programmer has knowledge of the program's meaning and semantic properties of the program. The property used above is `map f (map g xs) = map (f.g) xs`. That is, two maps can be simplified to one map that applies both functions.

While a human can use this knowledge to justify transformations, a compiler is generally unable to perform such reasoning. Optimising compilers can transform inefficient programs into more efficient ones by repeatedly applying a set of program transformations such as inlining, case-of-constructor and let-floating [de M. Santos, 1995]. At some point, however, the optimiser will reach a boundary it cannot pass. Without knowing the intent of the code, the optimiser is doomed to fail in cases where a person could use these semantic properties to simplify the program [Peyton Jones et al., 2001]. As a result, many optimisation opportunities are left untouched and valuable processor time is spent on work that could have been avoided.

The aim of this thesis is to allow the programmer to express intent and semantic properties of code as rewrite rules. With these properties available, the compiler can use them during optimisation. This can have a cascading effect, exposing more opportunities for optimisation. We want to encourage library authors to include these properties as much as possible, so the rules should be easy to formulate, using syntax that the programmer already knows.

1.1 Disciple

Disciple [Lippmeier, 2010] is a strict dialect of Haskell with effect and closure typing, which allows destructive update. Function types are annotated with the effects they can perform: such as reading or writing heap objects, or printing to the console. Heap objects are partioned into regions which may be mutable or constant.

This allows programmers to reason about side effects as easily as in a pure language such as Haskell, while solving some problems more elegantly. For example, a random number generator can use and update a global seed instead of needing to thread the seed through each call. The compiler is often able to perform more optimisations than in impure languages such as C or ML, since it can be certain that reordering or removing expressions will not have undesired consequences. For example, dead code removal must not remove any function calls with side effects. If the function's effects are known to be benign they may be safely removed.

The Disciplined Disciple Compiler (DDC) parses a source program and performs type inference and type checking, then converts the program into an intermediate representation known as the core language. As in the Glasgow Haskell Compiler (GHC) [Peyton Jones and Santos, 1998], the core language is based on System F. System F is an extension of the simply typed lambda calculus with type polymorphism [Reynolds, 1974] (similar to Java's generics).

After the core program has been optimised it is converted to either C or LLVM, and their respective compilers are used to produce object code. This allows us to benefit from their optimised code-generation techniques with little effort.

The core language is used for the majority of optimisations because it has fewer constructs than the source language, but retains the Disciple-specific types that are lost after conversion to C or LLVM. Overlapping constructs in the source language such as pattern matching and case statements, and function definitions and lambdas are converted to single constructs in the core language.

1.2 Side effects

Optimisers have to be careful to only apply transformations that will not change the real meaning of the program. In impure languages such as C or ML, hidden side effects such as those inside function calls can make innocent-looking transforms invalid. For example, constant propagation allows a variable's value to be inlined at each use of the variable, provided the variable has not been updated.

```
var x  = 1
func() = return rand()

x := 0
return func() + x;
```

Constant propagation in this case could inline the last use of `x` to `0`, and further optimisation might remove the addition altogether. However this relies on the definition of `func` not updating `x`. If the definition of `func` is changed to update `x`, applying constant propagation to `x` would produce a different result.

```
var x  = 1
func() = x := rand(); return rand()

x := 0
return func() + x;
```

This means that if a function's definition is unavailable (for example as a function argument) then the compiler must assume the worst.

1.3 Related work

There are many existing rewrite rule frameworks for other languages:

- the Glasgow Haskell Compiler supports simple rewrite rules, and is widely used in practice [Peyton Jones et al., 2001];
- there are more complex rule systems for Haskell that haven't gathered as much attention [de Moor and Sittampalam, 1998];
- one exists for ML and allows many transforms, but only some primitives are considered pure; user functions are assumed to have side effects [Visser et al., 1998];
- they exist for various imperative languages, generally requiring complex conditions about variable assignments to deal with side effects [Lacey and de Moor, 2001].

However, none of these are quite ideal for porting to Disciple. Whereas Haskell constrains side effects to the `IO` monad, in Disciple arbitrary expressions may perform side effects and mutate variables.

Unlike an ordinary imperative language, each function's effects are tracked in Disciple's type system; this allows constraints such as purity to be expressed more succinctly than otherwise. For example, instead of traversing a control-flow graph to check that a variable is not updated, we can just require its region to be constant.

1.4 Correctness

Just as a compiler cannot check whether a program is correct, it also cannot check whether a rewrite rule is correct. Furthermore, some rules may even change the overall result of a program but still preserve the programmer's intent. If we have a rule `int32ToInt64 (int64ToInt32 x) = x`, then we have changed the meaning for 2^{60} or any other number larger than 2^{31} (for signed integers). We try to focus on programmer intent, rather than functional equivalence. While it is unlikely that a programmer would ever write such a conversion directly, it may occur after inlining and other optimisations.

Confluence and termination are decidable for first-order rewrite system [Mayr, 1998, Jouannaud and Rubio, 1998], which indeed our rewrite rules are. The Glasgow Haskell Compiler is also first-order, but does not check for termination and confluence. This is because other optimisations may conflict: a rule that is terminating on its own may loop indefinitely when interleaved with other optimisations [Peyton Jones et al., 2001]. Experience with GHC's rewrite rules has shown this not to be a problem, so our rules will not check for termination or confluence either.

Chapter 2

Background

Most rewrite rules can be thought of as just adding an extra clause to a function definition, but with the added power of inspecting the expressions at the call site instead of just values. This allows the programmer to distinguish between expressions that would evaluate to the same value.

Suppose we have a simple mutable array type and a function to copy arrays:

```
copy :: Array -> Array
```

We would like to remove unnecessary copies. A copy of a copy will contain the same values as the original, and because the first copy is no longer referenced, it will not be mutated.

```
RULE copyCopy (a : Array).
    copy (copy a) = copy a
```

This rule says that, for any variable of type `Array`, if `copy` is applied to it twice then reduce it to just one. There are also cases where we can apply this rule multiple times: since `copy Array` is itself an `Array`, reapplying this rule will reduce any number of copies.

```
copy (copy (copy empty))
==>
copy (copy empty)
==>
copy empty
```

2.1 Inlining

Inlining is a fundamental optimisation in functional programs [Peyton Jones and Marlow, 1999]. Firstly, it increases cache locality in the object program by reducing jumps to far-off code. More importantly though, it can also expose more optimisations such as case elimination [de M. Santos, 1995] by moving definitions closer to where they are used.

For example, suppose we were calling a simple function with a constant value:

```
ifp p t f
 = case p of
    True  -> t
    False -> f

ifp True x y
```

If we inline the definition of `ifp` into here, and replace the parameters with their values, we end up with a simple case statement. Then a *case of constructor* transform is able to eliminate the case expression altogether:

```
case True of
  True  -> x
  False -> y
==>
x
```

With rewrite rules, however, the order that inlining is performed becomes quite important [Peyton Jones et al., 2001]. We want to inline function definitions mentioning functions used in rewrite rules first, because that will give the rules more opportunities to fire. However we must wait before inlining any functions mentioned in a rule: otherwise the rule would not be able to fire. We may then want to inline those functions after the rules have had ample opportunity to fire.

The Glasgow Haskell Compiler [GHC, 1991] solves this by allowing the user to annotate each inline pragma with a phase number indicating when definitions should be inlined.

```
sum      = foldr (+) 0
{-# INLINE [2] sum #-}

double x = x * 2
{-# INLINE [2] double #-}

{-# RULES
"sumMapDouble"
  forall xs
  . sum (map double xs)
  = (sum xs) * 2
  #-}

sum (map double [1,2])
```

In this case, we would like the rule to fire before either of the functions are inlined. But we do still want the functions to be inlined afterwards, because that may allow other rules and optimisations to occur.

2.2 Fold/build fusion

Where an imperative programmer would write a single loop, functional programmers tend to use pipelines of list combinators to achieve the same result in a more concise and reusable way.

```
result = sum         -- sum of all elements
       $ filter even -- list containing only even elements
       $ map (*2)    -- double all elements
       $ from 0 100  -- enumerate from 0 to 100 [0,1,2,3...]
```

However this clarity comes at a price: a naive[1] compiler would have to allocate a list for each stage of that function. Fold/build fusion, as explained by Gill et al [Gill et al., 1993], is a method of removing such intermediate lists. It uses a conventional `foldr` implementation, but adds a `build` function to abstract list generators over the particular `cons` and `nil` constructors they use.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr k z []     = z
foldr k z (x:xs) = k x (foldr k z xs)
```

[1] "Naive: no accent" – Guardian style guide

Note that the `build` function uses a universal quantifier in its generator function, which means `g` must work for any type when given `cons` and `nil` constructors. Here it is simply instantiated to lists.

```
build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

Because `foldr` replaces a list's `cons` cells with some operation `k` and its `nil` with some operation `z`, we can fuse those together:

```
foldr k z   (1  :  2  :  3  : [])
=           (1 `k` 2 `k` 3 `k` z)
```

To do this, we add a rule that removes the intermediate list when folding directly on a `build`-list, by using the `foldr`'s arguments instead of the `(:)` and `[]` constructors.

```
RULE foldrBuild k z g.
    foldr k z (build g) = g k z
```

We can write most list-generators using `build` and consumers using `foldr`.

```
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs

map :: (a -> b) -> [a] -> [b]
map f xs = build (\c n -> foldr (\a b -> c (f a) b) n xs)

from :: Int -> Int -> [Int]
from a b = build (from' a b)

from' :: Int -> Int -> (forall b. Int -> b -> b) -> b -> b
from' a b = \c n -> if a > b
                    then n
                    else c a (from' (a+1) b c n)
```

Then whenever we fold a list being generated by `build`, instead of actually constructing the list using `(:)` and `[]` and then destructing it, we can cut out the middle step:

```
sum (from 1 3)
==>                                                 (inline)
foldr (+) 0 (build (from' 1 3))
==>                                                 (foldr/build)
from' 1 3 (+) 0
```

Even if we add a `map` in-between here we need to apply some extra transforms, but we still eventually end up with no intermediate lists:

```
sum (map (+1) (from 1 3))
==>                                                    (inline)
foldr (+) 0 (build (\c n -> foldr (\a b -> c (a+1) b) n (build (from' 1 3))))
==>                                                    (foldr/build)
(\c n -> foldr (\a b -> c (a+1) b) n (build (from' 1 3))) (+) 0
==>                                                    (beta reduction)
foldr (\a b -> (a+1) + b) 0 (build (from' 1 3))
==>                                                    (foldr/build)
from' 1 3 (\a b -> (a+1) + b) 0
```

2.3 Stream fusion

A similar, but more complicated, method of removing intermediate lists is stream fusion. Stream fusion [Coutts et al., 2007] converts lists into pairs of state and nonrecursive stepper functions. Higher-order list functions such as `map` and `filter` are then written to operate on the stream representation, and generous inlining and optimisation can merge consecutive stream functions. Rewrite rules are used to remove unnecessary conversion between streams and lists.

Stream fusion is able to remove more intermediate lists than fold/build, such as when fusing `concatMap`s. However it requires existential types, which are not yet implemented in DDC. We will instead focus on fold/build fusion.

2.4 Higher order rules

All the rules we have seen so far have been first-order: the left of the rules have only contained function applications. If we relax this requirement and allow variable binders on the left, we are able to express many more interesting transforms that otherwise have to be baked-in to the compiler itself. For example, higher-order rules have been used to implement let-hoisting, dead-code removal, inlining and other transforms that cannot be expressed using first-order rules [Visser et al., 1998].

Let-flattening is another example of a higher-order rule. Let-flattening on its own does not improve anything, but it simplifies the structure to make further optimisations more likely.

```
RULE letFlatten.
let x = (let y = e_1 in e_2)
in      e_3
=
let y = e_1 in
let x = e_2 in
in      e_3
```

Note that in this example `y`'s binder would need to be renamed if it conflicted with any variables in `e_3`.

While higher-order rules certainly have potential, experience with GHC's [Peyton Jones et al., 2001] first-order rules has shown them to be sufficient for many applications. For this reason, Disciple also uses first-order rules. Higher-order rules will not be discussed further.

Chapter 3

Disciple core primer

Disciple is a purely functional language similar to Haskell. Effect and region typing are used to ensure purity, so operations can only be made lazy if they do not read from or write to any mutable regions. When values are allocated, they are given a specific region to live in. The regions may be marked as constant or mutable, and objects in mutable regions may be destructively updated.

Allocating, reading and writing values are tracked as effects, and the absence of these effects may be used to justify more optimisations. For example, we can lift an expensive operation out of a lambda where it may be computed multiple times if it is pure: it must not read from or write to any mutable regions.

The Disciplined Disciple Compiler (DDC) uses a small System-F style core language as its intermediate representation [Lippmeier, 2009]. After performing type inference on the source program, it is transformed into this core language.

After optimisations are performed on the core language it is converted to C or LLVM, whose respective compilers are used to output executable object code.

3.1 Syntax

The grammar for the core language is given in Section 3.6 (page 22). Unlike the source language, all applications including type applications are explicit in the core. Region handles such as `R0#` are passed as type arguments to functions.

Type applications are distinguished from value applications by surrounding them with square brackets: `addInt [R0#] [R0#] [R0#]` As a shorthand, repeated type applications can be written with colon-brackets: `addInt [:R0# R0# R0#:]`.

Universal quantifiers in types are introduced with `[BINDER : KIND]`, while in values `/\(BINDER : KIND)` is used (envision an uppercase lambda). Different kinds are used for regions (`%`), types (`*`), effects (`!`) and closure-types (`$`).

Function types take two extra arguments: effects that may be performed, and the regions that may be shared via closures. They are written as `a -(e|c)> b`, or simply `a -> b` when the effect and closure are empty. Empty effects are written `!0` and empty closures as `$0`.

Witness applications are used to provide proof of a fact, for example that a particular region is mutable, and are written inside angle brackets: `f <w>`.

Since integer objects can live in any region, literals are actually functions that allocate a value. An integer in the global region referenced by `R0#` is constructed with `23 [R0#] ()`.

```
23 :: [r : %]. Unit -(Alloc r | $0)> Int r
```

Here we see that the integer constructor `23` takes a region handle, and when applied to a `Unit` (the only value of which is `()`) allocates a value in region `r` and returns an integer in that region.

3.2 Regions

The core interpreter has an infinite supply of global region handles, denoted by `Rn#` where `n` is some nonnegative number.

Most functions which read or allocate values take region handles as parameters, so they can act on any region. The operator `addInt` takes two integers and allocates a new object for the result. Because these can all be in different regions, it also takes three region handle parameters:

```
addInt :: [r0 r1 r2 : %].
    Int r0 ->
    Int r1 -(Read r0 + Read r1 + Alloc r2 | Use r0)>
    Int r2
```

Note that the second function arrow here has a closure type of `Use r0`. This is because if `addInt` were partially applied then the resulting closure would hold a reference to `r0`.

```
addInt [:R0# R1# R2#:] (5 [R0#] ())
    :: Int R1#
    -(Read R0# + Read R1# + Alloc R2# | Use R0#)>
       Int R2#
```

Local regions are defined with `letregion`. Mutable regions may be defined by including a witness at the same time. This witness must be passed to any update function as proof of mutability:

```
updateInt :: [r0 r1 : %].
    Mutable r0 =>
    Int r0 ->
    Int r1 -(Read r1 + Write r0 | Use r0)>
    Unit

letregion r1
with      { w : Mutable r1 }
in  let x = 5 [r1] ()
    in  updateInt [:r1 R0#:] <w> x (23 [R0#] ())
```

A value in a local region cannot escape its `letregion`, because the region may be deallocated afterwards. The typechecker uses effect and closure types to make sure no values escape. To avoid escape, copying local values into an outer region may be required:

```
copyInt :: [r0 r1 : %].
    Int r0 -(Read r0 + Alloc r1 | $0)>
    Int r1

letregion r1
in  let v = 5 [r1] ()
    in  copyInt [:r1 R0#:] v
```

3.2.1 Distinct witnesses

A fellow honours student has implemented distinct witnesses on regions to show when variables will not alias [Ma, 2012]. Aliasing information can be used to remove unnecessary stores and loads:

```
letrec update2 [r1 r2 : %]  (w : Mutable r1)
               (x : Int r1) (y : Int r2) : Unit
 =  updateInt [:r1 r2:] <w> x (addInt [:r2 r2 r2:] y (1 [r2] ()));
    updateInt [:r1 r2:] <w> x (addInt [:r2 r2 r2:] y (2 [r2] ()));
```

Here we update `x` twice, without reading `x` after the first update. It would appear that we could remove the first update with no change to the meaning of the program, however this is not always the case. When `x` and `y` point to the same heap object, the result of the first update *is* read, and is thus necessary. In order to perform these optimisations, we need knowledge of whether variables may alias. Distinct witnesses are used as evidence that region handles point to entirely separate regions, and thus any objects in those regions must also be distinct.

If update2 is modified to require its regions to be distinct, the code generator will be able to make the optimisation, removing the first update.

```
letrec update2 [r1 r2 : %]  (w : Mutable r1) (w2 : Distinct r1 r2)
               (x : Int r1) (y : Int r2) : Unit
 =  updateInt [:r1 r2:] <w> x (addInt [:r2 r2 r2:] y (1 [r2] ()));
    updateInt [:r1 r2:] <w> x (addInt [:r2 r2 r2:] y (2 [r2] ()));
```

Now any calls to update2 must provide the distinctness witness as evidence. Distinct witnesses are constructed similarly to mutability witnesses:

```
letregion r1 with { wM : Mutable r1 }     in
letregion r2 with { wD : Distinct r1 r2 } in
    update2 [:r1 r2:] <wM> <wD> ...
```

Only witnesses that concern the current region and some other region in the environment are allowed. Simply having a different name does not mean that regions are distinct: if a function takes multiple region parameters, they may be bound to the same region at the callsite.

3.3 Effects and closures

Operations like allocating, reading and writing values all have effects and closures. Here, :*: means an expression's data type, :!: its effect, and :$: its closure.

```
5 [R0#] ()
:*: Int R0#
:!: Alloc R0#
:$: Use R0#
```

The scrutinee of a case expression must be read, for the case to inspect the value. If i is an Int R0#, then case i will read and use R0#.

```
case i of { _ -> i }
:*: Int R0#
:!: Read R0#
:$: Use R0#
```

Updating a heap object requires writing to its region. If i is some mutable Int r1 and w is a witness of Mutable r1, then:

```
updateInt [:r1 R0#:] <w>
    i   (23 [R0#] ())
:*: ()
:!: Alloc R0# + Read R0# + Write r1
:$: Use R0# + Use r1
```

Because the effects are so fine-grained, higher-order functions such as `map`, `filter` and `compose` must be polymorphic in their arguments' effects and closures:

```
let compose = /\(a b c : *) (ef eg : !) (cf cg : $).
    \(f : b -(ef|cf)> c).
    \(g : a -(eg|cg)> b).
    \(x : a). f (g x)
in
    compose [Int R0#] [Int R1#] [Int R2#]
            [Read R0# + Read R1# + Alloc R1#]
            [Read R1# + Read R2# + Alloc R2#]
            [Use R2#]
            [Use R1#]
        (\(b : Int R1#). mulInt [:R1# R2# R2#:] (2 [R2#] ()))
        (\(a : Int R0#). addInt [:R0# R1# R1#:] (3 [R1#] ()))
```

3.3.1 Weaken casts

The core language requires that all branches of `case` expressions have the same type - including effect and closure. Suppose one branch writes to a region, but the others do not. In this case `weakeff` casts must be used to add a `Write` effect to the other branches.

Similarly, if the closure types are not the same, `weakclo`s must be used to enlarge the closure.

```
case setFive of {
    True ->
        updateInt [:r1 r2:] <w>
                i (5 [r2] ());
    False ->
        weakclo { [r2] } in
        weakeff [Write r1 + Alloc r2 + Read r2] in
                ()
}
```

3.3.2 Deep effects and closures

For type variables in polymorphic functions, the regions of the type are not actually known. This makes it hard to express an effect that allocates or reads values of that type.

Effect constructors such as `DeepRead`, `DeepAlloc` and `DeepWrite` take a type as an argument. This is equivalent to an effect that reads, allocates or writes to any of the regions mentioned in the type. For closures types `DeepUse` is provided, specifying that all regions in the type may be `Use`d.

```
DeepUse (List r1 (Int r2))
==>
Use r1 + Use r2
```

3.4 Unboxed integers

Boxed integers are allocated on the heap, as pointers to unboxed integer values. Whenever any arithmetic is performed the heap objects must be unboxed, the result is computed, and a new boxed integer is allocated for the result.

Unboxed integers are raw integer values, and are not parameterised by region. They also cannot be mutated. Addition on unboxed integers is a primitive operation, denoted by `add# [Int#]`. Arithmetic for boxed integers is implemented in terms of unboxed integers so that the definitions can be inlined. After inlining, case elimination may be able to remove some of the boxed integer unboxing and allocations.

```
addInt [r1 r2 r3 : %]
       (a : Int r1)
       (b : Int r2)
          : Int r3
 =  case a of {
      I# a' ->
        case b of {
          I# b' -> I# [r3] (add# [Int#] a' b')
        }
    }
```

3.5 Naming problem

It can become quite tedious to plumb a supply of fresh variable names through a compiler implemented in a pure language [Augustsson et al., 1994]: any transformation that needs to create new bindings must be able to use a name that does not conflict with any existing bindings.

We sidestep this issue by using a mixture of named and anonymous (de Bruijn) variables. Anonymous variables are bound using a sole `^`, and referenced by `^n` where `n` is the number of anonymous bindings between the binder and the usage. Named variables can easily be converted to their anonymous representation:

```
let x = 5       in
let y = 2       in
    x  +  y

==>

let ^ = 5       in
let ^ = 2       in
    ^1 + ^0
```

Using anonymous variables means that in order to wrap a new binder around an expression, we do not have to generate a new name. We do, however, need to increment any anonymous variables that refer to the outer environment. All other variables stay the same.

```
let ^ = ()
...
add (5 [R0#] ^0)
    (let ^ = 3 [R0#] ()
     in  ^0)

==>

let ^ = ()
...
let ^ = add     in
^0  (5 [R0#] ^1)
    (let ^ = 3 [R0#] ()
     in  ^0)
```

Here we have introduced a new anonymous binding for `add`. The reference to `()` must be increased by one to get past the new `add` binder, but the reference to `3 [R0#] ()` can stay the same since it is bound inside the `add`.

Type variables and value variables use a different naming environment, so anonymous type variable binders are not counted for value variables, and *vice versa*.

3.6 Grammar

var	::=	**namedvar** | **indexvar**
namedvar	::=	lower **varchar***
indexvar	::=	hat digit$^+$
varchar	::=	lower | upper | digit | underscore | prime
con	::=	**namedcon** | **symbolcon**
namedcon	::=	upper **varchar*** #$^?$
symbolcon	::=	`@@` | `**` | `*` | `%` | `!` | `$` | `@` | `(~>)` | `(=>)` | `->`
type	::=	**var**
	|	**con**
	|	**type type**
	|	`[` **binder** : **type** `]` . **type**
	|	**type** + **type**
	|	`!0`
	|	`$0`
binder	::=	**namedvar** | **hat** | **underscore**
wit	::=	**var**
	::=	**wcon**
	::=	**wit warg**
	::=	**wit** & **wit**
warg	::=	**wit** | `[` **type** `]`

exp ::= **var**
| **con**
| **literal**
(applications)
| **exp xarg**
(lambda abstractions)
| `/\(` **binder** : **type** `)` . **exp**
| `\(` **binder** : **type** `)` . **exp**
(non-recursive and recursive let binding)
| `let` **letbind** `in` **exp**
| `let` **letbind** `lazy <` **witness** `> in` **exp**
| `letrec {` **letbindtyped**$;^{+}$ `} in` **exp**
(region introduction and context)
| `letregion` **binder** `with {` **typesig**$;^{*}$ `} in` **exp**
| `withregion` **var** `in` **exp**
(case expression)
| `case` **exp** `of {` **alt**$;^{+}$ `}`
(type casts)
| `weakeff [` **type** `] in` **exp**
| `weakclo {` **exp**$;^{*}$ `} in` **exp**
| `purify <` **witness** `> in` **exp**
| `forget <` **witness** `> in` **exp**

xargs ::= **exp** | `[` **type** `]` | `<` **witness** `>`

letbind ::= **binder** `=` **exp**
| **binder** : **type** `=` **exp**

letbind ::= **binder** : **type** `=` **exp**

typesig ::= **var** : **type**

alt ::= **pat** `->` **exp**
pat ::= underscore | **con patbind**$^{+}$
patbind ::= **binder** | `(` **binder** : **type** `)`

Chapter 4

Rewrite rules

In this section, we will discuss the complications and intricacies that arise from implementing rewrite rules in Disciple. As discussed in Section 3 (page 14), all core programs require explicit region information and type applications. For ease of exposition and clarity, region and witness information is omitted where it is secondary.

4.1 Unconditional rules

The simplest form of rule is always applicable, regardless of its context. Since we do not have to deal with higher-order rules, it is just a matter of taking some sequence of function applications and matching variables against it. Then once we have a full set of variable assignments, we substitute those into the right-hand side.

Here is an example of a simple rule: if we are creating a copy of a copy, we can safely remove the inner copy.

```
RULE copyCopy [r1 r2 r3 : %] (x : Int r3).
    copyInt [:r2 r1:] (copyInt [:r3 r2:] x)
    = copyInt [:r3 r1:] x

copyInt [:R0# R0#:] (copyInt [:R0# R0#:] (5 [R0#] ())
==>
copyInt [:R0# R0#:] (5 [R0#] ())
```

4.1.1 Implementation

The main function `rewrite` takes a list of rewrite rules and a target expression. It recurses through the expression, building up the necessary context and repeatedly applying rewrite rules at each step.

```
rewrite :: [RewriteRule] -> Exp -> TransformResult Exp
```

We return a `TransformResult`, which wraps the resulting expression and also includes a list of which rules were fired. This list of rules is helpful when debugging a set of rewrite rules.

At each stage, `rewrites` is called and the expression is checked against all the rules: the first that matches is used and the expression is replaced with the rule's right hand side.

```
rewrites :: [RewriteRule] -> Exp -> [Exp] -> RewriteEnv -> Exp

rewrites (r:rules)  exp args env
  = case rewriteX r exp args env of
         Nothing   -> rewrites rules exp args env
         Just exp' -> exp'

rewrites []         exp args env
  = mkApps exp args
```

There is potential for optimisation here, by indexing the rules by the first argument when it is a variable function name such as `map` or `foldr`. This would mean that instead of searching through all the rules, only relevant ones would be tried. GHC [GHC, 1991] does this, but it has not been implemented in Disciple yet.

Each rule is tried, and the first that successfully matches is returned. If no rules match, the original expression is returned. The function `mkApps` creates function application for each of the arguments.

`rewriteX` takes a single rule, a target expression and its arguments, a rewrite environment containing the in-scope witnesses and bound regions. It then returns the rule's replacement expression if matching succeeds, or nothing on failure.

```
rewriteX :: RewriteRule -> Exp -> [Exp] -> RewriteEnv -> Maybe Exp
rewriteX rule f args env
 = do   (s,rest)        <- getSubstitution rule f args env
        checkConstraints   rule env s
        f'              <- substitute (ruleRight rule) s
        return             (mkApps f' rest)
```

We take the rule's left-hand side and attempt to find a substitution of the rule's binders, to match the target expression and its arguments. If a substitution is found, we return the rule's right-hand side with the substitution applied. We must also add

back any 'left-over' arguments that were not used by the rule. For example, if a rule `f x = g x` is matched against `f x y`, the `y` must be added back to the end: `g x y`.

Constraint checking is not relevant for unconditional rules, and the `checkConstraints` call will be explained in the next section.

4.2 Mutability constraints

Certain rules are only applicable if some values are known to be constant, because actually applying the rule with a mutable variable could change the meaning of the program. Consider a rule that uses the fact that adding zero is the identity:

```
RULE add0r [r : %] (x : Int r).
addInt [:r r r:] x (0 [r] ()) = x


letregion r with {w : Mutable r} in
...
let y = (5 [r] ())                                  in
let z = addInt    [:r r r:]     y (0 [r] ())        in
let _ = updateInt [:r r:] <w>   y (23 [r] ())       in
    z
```

This will evaluate to `5`, because `z` is being allocated a new object with value `y + 0` before `y`'s value is updated. However if we perform the rewrite we will not be calling the `addInt`, and hence not allocating a new object:

```
letregion r with {w : Mutable r} in
...
let y = (5 [r] ())                                  in
let z = y                                           in
let _ = updateInt [:r r:] <w>   y (23 [r] ())       in
    z
```

Here we are setting `z` to point to the same object as `y`, then updating both `y` and `z`'s value. This is no good: `z` is now `23`, and the meaning of the program has changed.

To solve this, note that `r` is mutable. If we change the rule to require that the region `r` is constant, the rule will not fire in this case:

```
RULE add0r [r : %] (x : Int r).
    Const r =>
addInt [:r r r:] x (0 [r] ()) = x
```

4.2.1 Implementation

Mutability constraints are implemented by keeping a set of all the in-scope witnesses. After finding a substitution, we check if the witness map contains the constraints required by the rule, and if so allow the rule to fire.

Extending the set of in-scope witnesses is made slightly harder because of the anonymous binders. If we just used a set of witness types, then at every anonymous binder we would have to raise all existing elements in the set by one. Since we expect to see more binders than opportunities for rule firing, constant-time insertion is important.

Instead, then, we can use a list of sets. When we see a new witness we insert it into the head of the list, and at anonymous binders instead of raising the elements, we prepend an empty set to the list.

Then when checking if a type `t` has been seen before, we check the set at the head of the list, and if it's not there we lower the indices in `t` by `1`, then check the next set in the list.

This way we're only raising and lowering de Bruijn indices of small types when it's actually necessary, instead of at every anonymous binder.

This is currently implemented, but for simplicity we use a list of lists:

```
type WitnessMap = [[Type]]
```

To add a new witness, for example `Mutable ^0`, we just add it to the first 'set' (actually a list) in the list. If the list is empty, we just add a new element.

```
extend :: Bind -> WitnessMap -> WitnessMap
extend b [] = [[typeOfBind b]]
extend b (w:ws) = (typeOfBind b : w) : ws
```

Then, when we see a type binder such as `letregion ^ in ...` we must raise all the indices by one. We do this by adding a new empty 'set' to the start of the list, instead of having to visit each member of the set and raise its indices.

```
lift :: Bind -> WitnessMap -> WitnessMap
lift (BAnon _) ws = [] : ws
lift _         ws = ws
```

To check if a witness such as `Mutable ^3` is in the map, we see if the first list contains `Mutable ^3`, or the second list contains `Mutable ^2` and so on, until we find a match or reach the end.

```
contains :: Type -> WitnessMap -> Bool
contains t [] = False
contains t (w:ws) = t `elem` w || contains (liftT t (-1)) ws
```

After finding a substitution for a rule in `rewriteX`, the values are substituted into the rule's constraints. It is checked whether these constraints are contained in the environment.

```
checkConstraints :: RewriteRule -> RewriteEnv -> Substitution -> Maybe ()
```

We return `Maybe ()` here, and use the `Maybe` monad inside `rewriteX` to propagate the failure upward.

4.3 Effect interference

We need to be particularly careful when performing rewrites, because evaluating arbitrary expressions might cause side effects. Reordering computations could then change the overall meaning of the program.

4.3.1 Double computation

Suppose we wanted to replace a multiplication by two with an addition:

```
RULE mul2r [...] (x : Int ...).
    x * 2 = x + x

let put = \((_ : Unit).
    print "Ok";
    3)
in
put () * 2
==>
put () + put ()
```

But alas, our old version only printed "Ok" once, and the new will print it twice. In this case, the solution is as simple as wrapping any expression used multiple times in `let`s. The de Bruijn indices are useful here to introduce a fresh binding:

```
put () * 2
==>
let ^ = put ()
in  ^0 + ^0
```

Similarly, if we perform a rewrite and the result is not used, it could remove some desired side effects. But all our optimisation efforts would be for naught if we could not

remove extraneous expressions! The plan, then, is to introduce `let`s regardless, and have a separate optimisation pass that removes unused bindings if their effects are benign. This is known as *dead code elimination*.

```
RULE mul0r [...] (x : Int ...).
    x * 0 = 0


put () * 0
==>
let ^ = put ()
in  0
```

4.3.2 Disjoint

One of the ubiquitous rewrite rule examples is mapping over a list twice: if we are mapping two pure functions over a list, why bother allocating the intermediate list?

```
RULE mapMap [ef eg : !] [cf cg : $]
    (f : Int -(ef|cf)> Int)
    (g : Int -(eg|cg)> Int)
    (xs : List Int).
  map f (map g xs) = map (f.g) xs
```

This is fine if both functions are pure, but in Disciple the functions are allowed to have side effects. Indeed, even if only one is pure and the other prints to console, it is still a valid transformation.

However if both functions printed to the console or if they both read and write from the same region, this kind of `map`-fusion is not applicable. In this example we are using the first `map g` to calculate the sum of the list, and then in `map f` we increment each element by the accumulated sum:

```
let acc = 0
let g   = (\x. updateInt [...] acc (addInt acc x);
               x)                                  in
let f   = (\x. addInt [...] acc x)                 in
    map f (map g [1,2,3])
```

The unfused version evaluates to `[7, 8, 9]`, but the fused version is `[2, 5, 9]`. Disaster! Looking at the functions' effects, the issue becomes obvious:

```
acc : Int r1
g   : Int -(Read r1 + Write r1)> Int
f   : Int -(Read r1)> Int
```

The `map`-fusion must not occur if the two functions read and write to the same variable; since the type information is not that fine-grained, we require distinct regions instead.

We define two effects to be disjoint if they cannot interfere with each other. Two `Read`s will not interfere with each other, whether to the same region or not:

```
    Disjoint (Read r1) (Read r2)
```

If one effect is a read and the other is a write, they must be to totally distinct regions (see Section 3.2.1, page 16).

```
Distinct r1 r2 =>
    Disjoint (Read r1) (Write r2)
```

Similarly, for two writes not to interfere, they must also be to distinct regions.

```
Distinct r1 r2 =>
    Disjoint (Write r1) (Write r2)
```

Allocation is pure since, done in isolation, nothing else can access the allocated value.

```
    Disjoint (Alloc r1) e
```

Effect sums do not interfere with each other if none of their elements interfere:

```
Disjoint e11 e2 =>
Disjoint e12 e2 =>
    Disjoint (e11 + e12) e2
```

And effect sums are also commutative, so the order does not matter:

```
Disjoint e1 e2 =>
    Disjoint e2 e1
```

Armed with this new knowledge, we can fix our original rule!

```
RULE mapMap [ef eg : !] [cf cg : $]
    (f : Int -(ef|cf)> Int)
    (g : Int -(eg|cg)> Int)
    (xs : List Int).
    Disjoint ef eg =>
  map f (map g xs) = map (f.g) xs
```

This is implemented with a function `checkDisjoint`:

```
checkDisjoint :: Type -> RewriteEnv -> Bool
```

Given a `Disjoint fs gs`, it checks whether every element in `fs` will not interfere with every element in `gs`. If any of the effects are variables we do not know what value they will be instantiated with, so only allocations can be on the other side. To check whether region names are distinct, we record the names of all regions bound by `letregion`s. We consider these regions to be 'concrete', since each `letregion` introduces a fresh, new region. If two regions are bound by `letregion`s and have different names, they must be distinct. Otherwise, we check the witness map for a `Distinct` witness.

4.4 Shadowing

If a rule mentions some bound variable (say, `map`) and that variable gets shadowed, we must not fire the rule for the new definition.

```
RULE mapMap ...
    map f (map g xs) = map (f.g) xs
...
let map = somethingElse
in  map f (map g xs)
```

Because the new `map` does not necessarily have the same meaning, the rule is no longer relevant.

To implement this, we keep track of the set of bound variables in the rewrite environment. We then check if any of a rule's free variables (such as `map` in the rule above) have been re-bound, before matching against the rule. If any variables have been re-bound, the rule match fails instantly.

4.5 Weakening

The effect on the right of a rule may be smaller than the left, for example when eliminating an addition. Similarly, terms mentioned in the left but not mentioned in the right may make the result's closure smaller. So whenever the rewrite rule fires it may need to add a cast on the effect or closure to preserve the original type.

```
RULE addOr [...] (x : Int r)
    Const r =>
    addInt [...] x 0 = x

addInt [...] (5 [r] ()) (0 [r] ())
==>
weakeff [Read r + Alloc r] in
weakclo {0 [r] ()} in
(5 [r] ())
```

When first adding the rule, we find the left-hand side's effect type, and any free variables in the left-hand side that aren't mentioned in the right-hand side.

It would be nice if later optimisations could remove these added casts, because if the type contains fewer effects it may make further optimisations possible. Removing these unnecessary casts applies to more optimisations than just rewrite rules, and will not be dealt with here.

4.6 Inlining

There is a fragile connection between rewrite rules and inlining: inlining a function definition may expose more rules, but inlining in the wrong order means some rules will no longer fire.

Instead of littering each function with inline and phase pragmas, we allow order of inlining, along with other transforms, to be expressed explicitly.

```
Inline Int -copyInt <> Rewrite <> Inline Int
```

First all functions from the `Int` module except `copyInt` are inlined. Then rewrite rules are applied, and the rest of `Int` module is inlined.

4.6.1 Let-holes

Inlining and let-forwarding alone cannot be trusted to expose all opportunities for rewriting. Let-forwarding must be careful not to duplicate any work, so if a locally bound variable is used twice it will not be inlined.

In this example, we are constructing a boxed integer from an unboxed one, but unboxing it immediately in two places:

```
box   :: [r : %]. Int#  -> Int r
unbox :: [r : %]. Int r -> Int#

RULE unbox_box [r : %] (i : Int#).
    unbox [r] (box [r] i) = i

letregion r with {w : Const r}          in
let x = box   [r] (add# [Int#] 5i# 2i#) in
let y = unbox [r] x                     in
       (unbox [r] x, y)
```

In order for `unbox_box` to fire, the definition of `x` would need to be inlined into two places. This rule is a special case because the right-hand side performs no work. In cases like this, where firing the rule will not duplicate any work, we use curly braces to signify that the context of a sub-expression is irrelevant. This allows the sub-expression to be bound elsewhere.

```
RULE unbox_box2 [r : %] (i : Int#). Const r =>
    unbox [r] {box [r] i} = i

letregion r with {w : Const r}          in
let x = box   [r] (add# [Int#] 5i# 2i#) in
let y = unbox [r] x                     in
       (unbox [r] x, y)
==>
letregion r with {w : Const r}          in
let ^ = add# [Int#] 5i# 2i#             in
let x = box [r] ^0                      in
let y = ^0
       (^0,            y)
```

The `Const` constraint is required on this rule. If `x` were updated before being unboxed, the meaning would not be the same.

```
let x = box   [r] (add# [Int#] 5i# 2i#) in
let _ = updateInt [r r] <w>
        x (5 [r] ())                    in
       (unbox [r] x, y)
```

For this reason, both rules should be included. Since the first rule requires the arguments to be adjacent, the boxed heap object cannot be mutated regardless of whether the region is `Mutable` or `Const`.

The Glasgow Haskell Compiler [GHC, 1991] handles this problem slightly differently. The programmer may add `CONLIKE` pragmas to functions to indicate the function is as cheap and simple as a constructor. Then when performing rewrite rules, any occurrences of `CONLIKE` functions will be inlined if it allows more rules to be applied.

4.7 Fold/build

As described in Section 2.2 (page 10), Gill et al [Gill et al., 1993] introduced a relatively simple method of list deforestation. All primitive list producers are written using a `build` combinator, and consumers are written using `foldr`. This allows the primitives to be inlined without worrying about recursive functions. In the original a single rewrite rule is introduced that removes matching `foldr/build`s, but it turns out we will need two rules due to effect interference.

The `foldr` implementation is fairly standard, except it is polymorphic in effects, closures and regions as well as types. The zero constructor must also be a function because it may allocate heap objects and perform arbitrary effects:

```
foldr  [a b : *] [r : %]
    [ek1 ek2 ez : !] [ck1 ck2 cz : $]
    (k : a -(ek1|ck1)> b -(ek2|ck2)> b)
    (z : Unit -(ez|cz)> b)
    (xs: List r a)
    { Read r + ek1 + ek2 + ez | ck1 + cz + Use r }
    : b
 = case xs of {
    Nil -> z ();
    Cons x xs' ->
         k x (foldr [:a b r ek1 ek2 ez ck1 ck2 cz:] k z xs')
    }
```

We then define the `build` combinator that list producers should use. The type of `build`'s argument `g` is very similar to `foldr`'s type, except it doesn't take a `List` argument.

```
build [a : *] [r : %] [eg1 eg2 : !] [cg1 cg2 : $]
    (g : [b : *]. [ec1 ec2 en : !]. [cc1 cc2 cn : $].
         (a -(ec1|cc1)> b -(ec2|cc2)> b)
        -(eg1|cg1)>
         (Unit -(en|cn)> b)
        -(eg2+ec1+ec2+en|cg2+cc1)>
         b)
    { eg1+eg2 + Alloc r | Use r }
    : List r a
 = g [List r a] [!0] [Alloc r] [Alloc r] [:$0 (DeepUse a) $0:]
     (Cons [:r a:]) (Nil [:r a:])
```

We could then define a rewrite rule that merges a `foldr` applied to a `build` to remove the intermediate list. This is only applicable if the producer and consumer functions do not have interfering effects:

```
RULE foldrBuild
    [a b : *] [r : %]
    [ek1 ek2 ez eg1 eg2 : !]
    [ck1 ck2 cz cg1 cg2 : $]
    (k : a -(ek1|ck1)> b -(ek2|ck2)> b)
    (z : Unit -(ez|cz)> b)
    (g : [gb : *]. [ec1 ec2 en : !]. [cc1 cc2 cn : $].
        (a -(ec1|cc1)> gb -(ec2|cc2)> gb)
           -(eg1|cg1)>
        (Unit -(en|cn)> gb)
           -(eg2+ec1+ec2+en|cg2+cc1)>
        gb).
    Disjoint (ek1+ek2+ez) (eg1+eg2) =>
 foldr [:a b r ek1 ek2 ez ck1 ck2 cz:]
    k z
    (build [:eg1 eg2 cg1 cg2:] g)
= g [:b ek1 ek2 ez ck1 ck2 cz:] k z
```

This rule alone works in Haskell [Peyton Jones et al., 2001], but in the presence of arbitrary effects we have had to devise a slightly more phased approach. If we have a stream of list operators with a `build` at the outermost, the inner foldr/builds all have variable effects, and thus cannot be considered disjoint.

```
result = map  double
       $ from 0 100

==>                                                (Inline map, double, from)

result = build [...]
          (/\(b : *). /\(ec1 ec2 en : !). /\(cc1 cc2 cn : $).
            \(c : Int R0# -(ec1|cc1)> b -(ec2|cc2)> b).
            \(n : Unit -(en|cn)> b).
             foldr [...]
                 (\(k : Int R0#).  \(z : b).
                    c (double k) z)
                 n
                 (build [...]
                   (from' [R0#] (0 [R0#] ()) (100 [R0#] ())))
```

There is a foldr/build pair inside the outermost `build` here, but it cannot be fused because the `foldr` calls `c`, which has a variable effect. The `from'` parameter to `build` reads from a region, so if the variable effect in `c` were to write to that region, the effects could interfere. The solution is to inline the outer `build`, which will make the effect concrete. But if we were to inline the inner `build`, there would no longer be a foldr/build! We need a way to selectively inline only `build`s that will not fuse anyway.

To do this, we define a new function `foldr_build` that performs a `foldr` and a `build`, still sequentially. We add a rule that unconditionally changes any foldr/builds into a call to `foldr_build`.

```
foldr_build [...] k z g
 =  foldr [...] k z (build [...] g)
```

This change has no effect at all on the computation, but it allows us to safely inline only the `build`s that have no chance of fusion.

```
result = build [...]
          (/\(b : *). /\(ec1 ec2 en : !). /\(cc1 cc2 cn : $).
            \(c : Int R0# -(ec1|cc1)> b -(ec2|cc2)> b).
            \(n : Unit -(en|cn)> b).
             foldr_build [...]
                 (\(k : Int R0#).  \(z : b).
                    c (double k) z)
                 n
                 (from' [R0#] (0 [R0#] ()) (100 [R0#] ()))

==>                                                          (Inline build)

result = foldr_build [...]
                 (\(k : Int R0#).  \(z : List R0# (Int R0#)).
                    Cons [...] (double k) z)
                 (Nil [...] ())
                 (from' [R0#] (0 [R0#] ()) (100 [R0#] ()))
```

With `build`'s definition inlined, the effects are made concrete and are therefore known not to interfere. We can now perform the fusion on `foldr_build` if its effects are disjoint.

```
RULE foldrBuild_prime
    [a b : *] [r : %]
    [ek1 ek2 ez eg1 eg2 : !]
    [ck1 ck2 cz cg1 cg2 : $]
    (k : a -(ek1|ck1)> b -(ek2|ck2)> b)
    (z : Unit -(ez|cz)> b)
    (g : [gb : *]. [ec1 ec2 en : !]. [cc1 cc2 cn : $].
        (a -(ec1|cc1)> gb -(ec2|cc2)> gb)
           -(eg1|cg1)>
        (Unit -(en|cn)> gb)
           -(eg2+ec1+ec2+en|cg2+cc1)>
        gb).
foldr [:a b r ek1 ek2 ez ck1 ck2 cz:]
    k z
    (build [:a r eg1 eg2 cg1 cg2:] g)
= foldr_build [:a b r ek1 ek2 ez ck1 ck2 cz:]
              [:eg1 eg2 cg1 cg2:]
    k z g
```

```
RULE foldrBuild_fuse
    [a b : *] [r : %]
    [ek1 ek2 ez eg1 eg2 : !]
    [ck1 ck2 cz cg1 cg2 : $]
    (k : a -(ek1|ck1)> b -(ek2|ck2)> b)
    (z : Unit -(ez|cz)> b)
    (g : [gb : *]. [ec1 ec2 en : !]. [cc1 cc2 cn : $].
        (a -(ec1|cc1)> gb -(ec2|cc2)> gb)
           -(eg1|cg1)>
        (Unit -(en|cn)> gb)
           -(eg2+ec1+ec2+en|cg2+cc1)>
        gb).
    Disjoint (ek1+ek2+ez) (eg1+eg2) =>
 foldr_build [:a b r ek1 ek2 ez ck1 ck2 cz:]
             [:eg1 eg2 cg1 cg2:]
    k z g
= g [:b ek1 ek2 ez ck1 ck2 cz:] k z
```

Chapter 5

Other transforms

In order to test the rewrite rules on real programs, several other transforms are required. Inlining is required to bring the definitions of `map`, `filter` and others. This exposes the `foldr`s and `build`s that may be fused. We then need to perform beta reduction, let-forwarding and so on, to simplify the expression and expose the definitions of variables at their use-sites.

5.1 A-normalisation

Before converting to C and LLVM, we convert to a simplified form where the only function arguments are variables. This is known as A-normal form (administrative normal form). This makes it easier to output C code.

```
addInt [:R0# R1# R2#:]
    (1 [R0#] ())
    (addInt [:R1# R1# R1#:]
        (2 [R1#] ())
        (addInt [:R3# R3# R1#:]
            (3 [R3#] ())
            (4 [R3#] ())))
==>
let ^ = 1 [R0#] () in
let ^ = 2 [R1#] () in
let ^ = 3 [R3#] () in
let ^ = 4 [R3#] () in
let ^ = addInt [R3#] [R3#] [R1#] ^1 ^0 in
let ^ = addInt [R1#] [R1#] [R1#] ^3 ^0 in
addInt [R0#] [R1#] [R2#] ^5 ^0
```

This is done in two stages: first any non-trivial value arguments are floated out as let bindings (type applications are left as-is since they are omitted in conversion to C anyway):

```
let ^ = 1 [R0#] () in
let ^ =
    let ^ = 2 [R1#] () in
    let ^ =
        let ^ = 3 [R3#] () in
        let ^ = 4 [R3#] () in
            addInt [R3#] [R3#] [R1#] ^1 ^0 in
        addInt [R1#] [R1#] [R1#] ^1 ^0
addInt [R0#] [R1#] [R2#] ^1 ^0
```

The let-within-lets are then flattened to produce A-normal form. Let-flattening can in general cause naming problems when the inner-let's name is used elsewhere, but since the first stage will only produce anonymous bindings we only need to flatten anonymous ones and lift any binders in the body.

```
let ^ = 1 [R0#] () in
let ^ = 2 [R1#] () in
let ^ = 3 [R3#] () in
let ^ = 4 [R3#] () in
let ^ = addInt [R3#] [R3#] [R1#] ^1 ^0 in
let ^ = addInt [R1#] [R1#] [R1#] ^3 ^0 in
addInt [R0#] [R1#] [R2#] ^5 ^0
```

5.2 Dead-code removal

Dead-code removal is required not just to remove superfluous computations, but it also means let-forwarding is more likely to inline definitions, if they are used fewer times.

To begin with, we type-check the entire expression, attaching effect and usage information to each `let`. We then recurse bottom-up through the expression, checking each `let` binding: if it is not used, or only used in casts, we check the effect. If the expression only has benign effects such as reading or allocating, we can remove the binding. We must still substitute its value into any casts it is used in. Since removing the bindings may make the effect and closure smaller, we must add `weakeff` casts with the original binding's effect type and `weakclos` with any free variables in the term.

```
let x = 5                          in
let y = x + 2                      in
let z = y * 3                      in
    () -- x, y and z unused
==>
weakclo {5; 2; 3}                  in
weakeff [Read R0# + Alloc R0#]     in
    ()
```

5.3 Interactive transform

Inspired by HERMIT [Farmer et al., 2012], I have added an interactive transformation mode to the `ddci-core` interpreter. The user specifies an expression to transform, and then each transform is applied to that expression.

```
> :trans-interact..
let x = 5#                         in
let y = boxInt x                   in
let z = unboxInt y                 in
    z
;;
trans> Rewrite
let x = 5#                         in
let ^ = x                          in
let y = boxInt ^0                  in
let z = ^0                         in
    z
trans> DeadCode
let x = 5#                         in
let ^ = x                          in
weakeff [Alloc R0#]                in
weakclo {boxInt; ^0}               in
let z = ^0                         in
    z
trans> :done
Rewrite <> DeadCode
```

A history of the transformations is also kept, allowing the user to undo transforms using `:back`.

5.4 Bubbling casts

Another useful transform, although not implemented by me, involves floating casts outwards and merging adjacent casts together. This cleans up some of the casts created by rewrite rules and dead code removal, and also simplifies the inner code, which means more rules may be able to apply.

Bubbling the casts outward must not move them through any function barriers, as this could change the type of the function, and also must not move them past the definitions of any variables used in a cast.

```
\((u : Unit).
    let x = 5 [R0#] ()              in
    weakeff [Alloc R0#]             in
    weakclo {boxInt; ^0}            in
    let y = 3 [R0#] ()              in
    weakeff [Read  R0#]             in
    weakclo {y}                     in
        ()
)

==>

\((u : Unit).
    weakeff [Alloc R0# + Read R0#]  in
    weakclo {boxInt; ^0}            in
    let x = 5 [R0#] ()              in
    let y = 3 [R0#] ()              in
    weakclo {y}                     in
        ()
)
```

5.5 Elaborate Distinct and Const

My fellow honours student Tran [Ma, 2012] has implemented a transform to elaborate `letregion`s with extra witnesses available from the context.

Since each `letregion` introduces a completely new region, it is distinct from all existing regions in scope. Adding `Distinct` witnesses for all these regions gives the code

generator more knowledge about which variables alias. It also means more effects will be `Disjoint`, potentially allowing more fusion.

Similarly, if a `letregion` has no `Mutable` witness it may be assumed to be `Const`. Since there is no `Mutable` witness to pass as proof to update functions, no updates can be made. Adding `Const` witnesses can also improve code generation: even if regions are not known to be distinct, constant heap objects needn't be re-read after objects in other regions are updated. More rules with `Const` constraints may also be able to be applied.

```
/\(r1 : %).
  letregion r2 with { }                              in
    ()

==>

/\(r1 : %).
  letregion r2 with { Const r2; Distinct r2 r1; } in
    ()
```

5.6 Simplifier specification

Instead of hardcoding a particular sequence of transforms to perform when optimising, Disciple allows these sequences to be specified interactively or when compiling. This allows the user to gain a better understanding of what the optimiser is actually doing.

A simple specification is just a single transform, such as `Rewrite` or a sequence of transforms one after another: `Rewrite <> Bubble <> DeadCode`.

Fixpoints are also supported, where a simplifier is repeatedly applied until it no longer does any useful work, or it reaches a maximum limit:

```
fix 10 (Rewrite <> Bubble <> DeadCode <> BetaLets <> Forward)
```

Here `BetaLets` performs beta reduction, substituting simple forms into lambdas, and `let`-binding complex applications. `Forward` is `let`-forwarding, that moves single-use `let` definitions into their use-sites, if their effect will not clash.

This works well with our approach to phased inlining. It is possible to inline a small set of definitions, then perform as much general optimisation and rewriting as possible, then to inline the rest of the definitions and optimise again.

```
Inline List -foldr -build -foldr_build <>
fix 10 (Rewrite <> Bubble <> DeadCode <> BetaLets <> Forward) <>
Inline List <>
fix 10 (Bubble <> DeadCode <> BetaLets <> Forward)
```

Chapter 6

Related work

6.1 Glasgow Haskell Compiler

The Glasgow Haskell Compiler (GHC) is an industrial-strength Haskell compiler [Peyton Jones et al., 1992]. A 2001 paper by Simon Peyton Jones et al describes the initial implementation of the rewrite rules currently in use [Peyton Jones et al., 2001]. Features have since been added to the implementation, but the general idea is the same.

GHC restricts the rules to only first-order sequences of function applications. The restriction is partially for simplicity of use and also because it makes it easy to implement matching efficiently. This means that there are many kinds of rules that cannot be written using it, but the simplified rules have been shown to be useful anyway.

Despite using only first-order rules, GHC still does not check termination because of interference with other optimisations. A set of rules on their own may terminate, but when interleaved with other optimisations may introduce a loop. For example, the following rule on its own is terminating, but applying that rule introduces an opportunity for simplification, which allows the rule to fire again.

```
{-# RULES
    "map/nil nonsense"
    forall f.
    map f [] = map f ([] ++ [])
-#}

map f []
==>                                          (rewrite)
map f ([] ++ [])
==>                                          (definition (++))
map f []
==>                                          (rewrite)
map f ([] ++ [])
...
```

In practice, because the right-hand side of rules is generally simpler than the left-hand side, non-termination does not appear to be an issue.

The paper describes the use of phased inlining to give rules a chance to be applied before any definitions mentioned in the rule are inlined. In current GHC, they also keep a map of `let`-bound variables and their values while rewriting [GHC, 1991]. This is used to direct inlining of particularly cheap expressions in cases where the inliner would not apply, such as if an expression is used multiple times or is bound outside but only used once inside a lambda. They have also added ranges for the phase numbers, after discovering that a single phase number was insufficient.

Other optimisations may create rules, too. Specialised versions of polymorphic functions, such as `sum` which works for any `Num` instance, may be created for `Int`. Then a rule is added that changes any `sum` of `Int`s to the specialised `sum_Int` function.

There is also the worker-wrapper transform, which transforms recursive functions to use more efficient unboxed worker functions. It adds rewrite rules to change calls from the boxed version to the unboxed version.

GHC's rewrite rules are shown to work for foldr/build fusion and efficient fusion of rose trees.

```
{-# RULES
 "foldr/build"
  forall k z (g :: forall b. (a -> b -> b) -> b -> b) .
  foldr k z (build g) = g k z
#-}
```

6.2 De Moor and Sittampalam: Haskell

In 1998, de Moor and Sittampalam implemented a rewrite system with higher-order matching, on a subset of Haskell [de Moor and Sittampalam, 1998]. Higher-order matching is generally undecidable because it involves synthesising equivalent function definitions, so they use a restricted form that tries to find a match after at most one beta reduction, or gives up.

The examples they use are small functions, which are obviously correct simple implementations, and they devise specific rules so that the result has better absolute complexity. Some examples use rules such as the associativity of addition from left to right. For a particular implementation of a function that would help, but if the function were implemented slightly differently, perhaps the other direction of associativity would be required.

This means the rules are very tightly tied to the original implementation, whereas our goal is to have active libraries: only the library author should have to worry about rewrite rules, but all users should benefit.

$$
\begin{aligned}
&\mathit{mindepth}\ (\text{Leaf}\ a) &&= 0\\
&\mathit{mindepth}\ (\text{Bin}\ s\ t) &&= \mathit{min}\ (\mathit{mindepth}\ s)\ (\mathit{mindepth}\ t) + 1
\end{aligned}
$$

$$
\begin{aligned}
&\text{transform} && \mathit{mindepth}\ t = \mathit{md}\ t\ 0\ \infty\\
&\text{where} && \mathit{md}\ t\ d\ m = \mathit{min}\ (\mathit{mindepth}\ t + d)\ m\\
&\text{with} && 0 + a = a\\
& && (a + b) + c = a + (b + c)\\
& && \mathit{min}\ (\mathit{min}\ a\ b) + c = \mathit{min}\ (a + c)\ (b + c)\\
& && \mathit{min}\ (a + b)\ c = \text{if}\ b \geq c\ \text{then}\ c\ \text{else}\ \mathit{min}\ (a + b)\ c
\end{aligned}
$$

This example starts with a simple definition of *mindepth* that traverses the entire tree. This is inefficient: it should track the current minimum, and give up immediately after going deeper than the minimum. The rules above can be used to derive an efficient definition.

Other examples include rewriting recursive definitions to use folds, which makes them easier to manipulate and fuse.

6.3 Visser et al: ML

Visser et al [Visser et al., 1998] implemented a rewrite system for a dialect of ML called RML. It is separated into rules and strategies. The strategies determine how and when to apply the rules in order to assure termination and confluence.

The rules use a pattern matching syntax on AST nodes. Constructor names begin with uppercase, variables with lower. Here we find any `let` constructor whose right-hand side contains another `let`, and flatten it out. Renaming of `let` binders is performed automatically, if necessary.

```
Hoist1 : Let(Vdec(t, x, Let(vd, e1)), e2)
      -> Let(vd, Let(Vdec(t, x, e1), e2))
```

Here is a strategy named `opt1` that repeatedly applies one of two rules, bottom-up, on the whole program.

```
opt1 = innermost'(Hoist1 + Hoist2);
```

Applying this strategy to a program will flatten all `let`s:

```
let x = (let y = 5 in y+2) in
    x+3
==>
let y = 5   in
let x = y+2 in
    x+3
```

Since RML is not a pure language, the rules must deal with side effects. Constraints are used to determine when rules are applicable. A constraint such as `<safe> e` checks the expression `e` for any calls to side-effectful primitives such as updating a variable or user-defined functions. Another constraint `<in> (b, e)` checks if `b` occurs anywhere inside `e`.

A rule for dead-code removal is implemented by removing a `let` if the variable is not mentioned in the body, and the definition is safe.

```
Dead1 : Let(Vdec(t, x, e1), e2) -> e2
    where not(<in> (Var(x), e2)); <safe> e1
```

6.4 Lacey and de Moor: imperative

In 2001 David Lacey and Oege de Moor used a rewrite system to optimise an imperative language they describe in the paper [Lacey and de Moor, 2001]. Specifying optimisations

for imperative languages is generally much harder than purely functional ones, because so many transformations are only valid if certain variables are not updated. Sophisticated conditions are required to check that a variable hasn't been assigned a different value, or that side-effects will not interfere.

Lacey and de Moor use rewrites over control flow graph (CFG), unlike most functional systems that use abstract syntax trees (AST). This is because the conditions of transforms are best expressed using the CFG of imperative program, for example looking at all predecessor nodes to make sure a variable is unassigned. They define a powerful language for specifying rewrite conditions.

This is able to express sophisticated optimisations, but perhaps at the cost of transparency. Here is constant propagation, which substitutes $x := v$ with $x := c$ if all input nodes assign v to c:

$$
\begin{aligned}
& n : (x := v) \Longrightarrow x := c \\
& \text{if} \\
& \quad n \vdash A^{\triangle}(\neg def(v)\ \mathrm{U}\ def(v) \wedge stmt(v := c)) \\
& \quad conlit(c)
\end{aligned}
$$

Here we rewrite some statement n from $x := v$ to $x := c$. But we only do the rewrite if c is a constructor literal ($conlit$), and we also check along all of n's predecessor paths ($A^{\triangle}$) that v is not assigned ($\neg def(v)$), until (U) we find an assignment of v that also sets its value to c.

Chapter 7

Conclusion

The aim of this thesis was to implement a system of rewrite rules that fit with the philosophy of effect typing.

I have implemented rewrite rules with mutability constraints and non-interfering effects. A form of phased inlining has been shown that allows certain definitions to be inlined before others, such as those mentioned in rules. We also allow the entire optimisation process to be scripted and transforms rearranged, to make searching for the best sequence possible. A simple version of dead code removal was also implemented, that only removes bindings if their effects are considered benign.

Fold/build fusion has been implemented using rewrite rules, so that large pipes of list functions may be fused together into a single non-allocating loop.

7.1 Contributions

Following is a list of the contributions I have implemented.

- Unconditional rules (Section 4.1)
- Mutability constraints in rules (Section 4.2)
- Disjoint constraints in rules (Section 4.3.2)
- Let-holes in rules (Section 4.6.1)
- Foldr/build fusion in the presence of effects (Section 4.7)
- A-normalisation transform (Section 5.1)
- Dead-code removal (Section 5.2)
- Interactive transformation mode (Section 5.3)

- Fixpoint transforms (Section 5.6)
- Module-specific inlining (Section 4.6)

7.2 Future work

More is possible: rules that require their right-hand side to reference constraints cannot be written, but is conceivable. Higher-order rules have been ignored for simplicity of matching, but could certainly be added.

7.2.1 Witness names

Sometimes, a rewrite rule needs to pass a witness as proof of a constraint to some other function. In this case, we need a different way to mention the constraints:

```
RULE memmove_memcopy
    [a : *] [r1 r2 : %]
    (len: Int)
    (x1 : Ptr r1 a)
    (x2 : Ptr r2 a).
  (w : Distinct r1 r2) =>
    memmove [:a r1 r2:]     len x1 x2
  = memcopy [:a r1 r2:] <w> len x1 x2
```

Here, `memmove` will copy the contents of one array to another, but must account for cases when the two arrays overlap. On the other hand, `memcopy` requires that the two pointers are in distinct regions, and can run faster since it does not need to worry about overlap.

Such a rule is not currently possible because all variables need to be matched by the left-hand side. To implement this, we would have to extend the witness map to contain the binding names as well as their type. We would then look up the name in the witness map, instead of just checking if it contains a given witness type.

Particular care would need to be given to shadowing: if another binding were made with the same name as a witness, the original binding must not be used.

```
letregion r1 with {w : Distinct r1 R0#} in
letregion r2 with {w : Distinct r2 R0#} in
memmove [:a r1 R0#:] ...
```

If the rule were applied in this case and the witness `<w>` passed to `memcopy`, it would be a type error.

7.2.2 Modules

Currently rewrite rules can only be specified in the interactive mode. The module parser will need to be modified to include rewrite rules, either as a GHC-style pragma or as a separate section.

The module system should export all defined and imported rules, similar to how typeclasses work [Peyton Jones et al., 2001]. This allows imported rules to fire after inlining is performed. Issues with shadowing of definitions can be resolved by modifying rules to use fully qualified names.

7.2.3 Disciple source language

In the source language effects, closures and regions are usually omitted. Type inference is also supported, so for many rules types should be able to be left out. Since map fusion requires its effects to be non-interfering, the effect types must still be explicit. The closures, regions, and type applications can be left out.

```
RULE mapMap
    [a b c : *]      [ef eg : !]
    (f : b -(ef)> c) (g : a -(eg)> b)
     xs.
  Disjoint ef eg =>
    map f (map g xs) = map (f.g) xs
```

To parse this rewrite rule, we need to parse the expression on each side of the equals and then perform type inference. We will also need to add extra type binders so the rewrite rule is polymorphic in closures and regions.

The foldr/build rules will still require explicit types. This means Disciple source requires support for at least rank two types.

References

[GHC, 1991] (1991). Glasgow Haskell Compiler. `https://github.com/ghc/ghc`.

[Augustsson et al., 1994] Augustsson, L., Rittri, M., and Synek, D. (1994). Functional pearl: On generating unique names. *Journal of Functional Programming*, 4(01):117–123.

[Coutts et al., 2007] Coutts, D., Leshchinskiy, R., and Stewart, D. (2007). Stream fusion: From lists to streams to nothing at all. In *Proceedings of the International Conference of Functional Programming 2007*.

[de M. Santos, 1995] de M. Santos, A. L. (1995). Compilation by transformation in non-strict functional languages.

[de Moor and Sittampalam, 1998] de Moor, O. and Sittampalam, G. (1998). Generic program transformation. In *Proceedings of the 3rd International Summer School on Advanced Functional Programming, LNCS 1608*, pages 116–149. Springer-Verlag.

[Farmer et al., 2012] Farmer, A., Gill, A., Komp, E., and Sculthorpe, N. (2012). The HERMIT in the machine: a plugin for the interactive transformation of GHC core language programs. In *Proceedings of the Haskell symposium 2012*, pages 1–12. ACM.

[Gill et al., 1993] Gill, A., Launchbury, J., and Peyton Jones, S. L. (1993). A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 223–232, New York, NY, USA. ACM.

[Jouannaud and Rubio, 1998] Jouannaud, J.-P. and Rubio, A. (1998). Rewrite orderings for higher-order terms in η-long β-normal form and the recursive path ordering.

[Lacey and de Moor, 2001] Lacey, D. and de Moor, O. (2001). Imperative program transformation by rewriting. In *Proceedings of Compiler Construction 2001, LNCS 2027, 2001*, pages 52–68. Springer Verlag.

[Lippmeier, 2009] Lippmeier, B. (2009). Witnessing purity, constancy and mutability. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 95–110, Berlin, Heidelberg. Springer-Verlag.

[Lippmeier, 2010] Lippmeier, B. (2010). Type inference and optimisation for an impure world.

[Ma, 2012] Ma, T. (2012). Type-based aliasing control for the Disciplined Disciple Compiler. Undergraduate honours thesis, University of New South Wales.

[Mayr, 1998] Mayr, R. (1998). Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192.

[Peyton Jones et al., 1992] Peyton Jones, S. L., Hall, C., Hammond, K., Cordy, J., Kevin, H., Partain, W., and Wadler, P. (1992). The Glasgow Haskell Compiler: a technical overview.

[Peyton Jones and Marlow, 1999] Peyton Jones, S. L. and Marlow, S. (1999). Secrets of the Glasgow Haskell Compiler inliner. In *Journal of Functional Programming*, page 2002.

[Peyton Jones and Santos, 1998] Peyton Jones, S. L. and Santos, A. L. (1998). A transformation-based optimiser for Haskell. In *Science of Computer Programming*, pages 3–47. Elsevier North-Holland, Inc.

[Peyton Jones et al., 2001] Peyton Jones, S. L., Tolmach, A., and Hoare, T. (2001). Playing by the rules: rewriting as a practical optimisation technique in GHC.

[Reynolds, 1974] Reynolds, J. (1974). Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer.

[Visser et al., 1998] Visser, E., el Abidine, Z., and Tolmach, A. (1998). Building program optimizers with rewriting strategies. In *Proceedings of the International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press.