Heap Recycling Analysis in Helium

Mias van Klei, 4228170

Supervisors
Ivo Gabe de Wolff
dr J.Hage

August 21, 2020

Abstract

Pure functional programming languages, such has Haskell, preclude destructive updates of heap-allocated data. To update a newly computed algebraic value, fresh heap space is allocated to store the result. This causes idiomatic programs to be notoriously inefficient when compared to their imperative and impure counterparts. J. Hage and S. Holdermans describe in 'Heap Recycling for Lazy Languages' a way to partly overcome this shortcoming. We implement the described analysis in the Haskell compiler Helium and analyze the memory usage of various programs.

Contents

Contents **2**

1 Introduction **4**
1.1 Evaluation Strategies . . . 4
1.2 Laziness . . . 4
1.3 Memory management . . . 5
1.4 Purity . . . 6
1.5 Side effects and Purity . . . 7
1.6 Heap Recycling Analysis . . . 8
1.7 Research Questions . . . 9
1.8 Contributions . . . 10

2 Preliminaries **11**
2.1 Substructural Type Systems . . . 11
2.2 Uniqueness Analysis . . . 14
2.3 Uniqueness Analysis: Type System . . . 16

3 Related Work **19**
3.1 Absence Analysis . . . 19
3.2 Sharing Analysis . . . 20
3.3 Uniqueness Analysis . . . 20
3.4 Strictness Analysis . . . 21
3.5 Combined Analyses . . . 21
3.6 Heap Recycling . . . 22
3.7 Reference Counting . . . 22

4 Helium **23**
4.1 Haskell . . . 23
4.2 Core . . . 24
4.3 Iridium . . . 28
4.4 LLVM & Helium runtime . . . 31

5 Heap recycling analysis in Helium **32**
5.1 Syntax & Parser . . . 32
5.2 Static Checks . . . 33
5.3 Core . . . 33
5.4 Iridium . . . 35
5.5 LLVM & Helium runtime . . . 35

6 Uniqueness Analysis in Helium **36**

6.1 Constraint Generation 36
6.2 Primitive operations 43
6.3 Constraint Solving 43
6.4 Constraint Violation 46

7 Results **49**
7.1 Memory pressure 49
7.2 Uniqueness Analysis 51

8 Conclusion **54**
8.1 Future Work 54

Bibliography **56**

1

Introduction

Functional programming languages are based on the concept of evaluating mathematical functions. The following two ways can be used to classify these functional languages: evaluation strategy and purity of the language.

1.1 Evaluation Strategies

In strict or eager evaluation an expression is evaluated as soon as it is bound to a variable, while in lazy evaluation or call-by-need [33], evaluation is delayed until its value is needed. The following example shows what this means:

```
print (length [2+1, 3*2, 1/0, 5-4])
```

Under strict evaluation this example will fail because of the division by zero in the third element of the list. Under lazy evaluation, the list won't be evaluated fully, since the length function doesn't need the elements of the list to calculate the length of the list. In short, strict evaluation always fully evaluates the arguments of a function before invoking the function, while in lazy evaluation arguments are only evaluated when strictly needed.

1.2 Laziness

Programs in functional languages using lazy evaluation, like Haskell [24], tend to use a lot of memory. The reason for this is the way laziness is implemented. A thunk, a piece of code that has yet to be evaluated, is generated for each expression. This thunk is passed around until another piece of code needs a part of the result of this thunk. When this is the case, the thunk is evaluated, possibly resulting in other thunks that have to be evaluated as well. When the result has been evaluated it is written back to the thunk, so that other pieces of code can reuse the result. This prevents re-evaluating the same pieces of code.
The consequence is that even for small programs a lot of thunks are generated. This can result in the high memory pressure mentioned earlier. Memory pressure can be reduced by analyzing the thunks in the program. Four scenarios can be distinguished:

- A thunk is never used, or a part is never used. An example is using only the first element of tuple.
- A thunk is used only once. This can be the case when an intermediate result is computed, to make code more readable, but is used only once.

- A thunk is used at least once. A call to a function will always use the argument it is given.
- A thunk may be used. This is for example the case when branching occurs.

We can use these observations, to describe three different analyses:

- If a thunk is never used, then why create a thunk at all? Absence analysis [28] determines which expressions are never used. It is similar to dead-code elimination in other languages
- If a thunk is used only once, then why update the thunk, by writing the result back? It will never be accessed again. Sharing analysis [18, 29] determines which expressions are used only once. If an expression is used only once, then no update of a thunk has to be performed, after evaluating the expression.
- If a thunk will definitely be used, thus evaluated, then why create a thunk at all? Strictness analysis [12, 25], determines which arguments to a function call will definitely be used. Then the caller can evaluate the arguments instead of the callee. This avoids creating unnecessary thunks.

All these three analyses have one thing in common: they determine how often an expression is used in the program. Based on that knowledge, various optimizations can be performed. Since these analyses are very similar, they can be combined in one analysis. GHC, the de facto Haskell compiler, combines these analyses into one, and calls it a Cardinality Analysis [9].

1.3 Memory management

Almost all programming languages use a heap to hold objects. In the case of Haskell these objects are thunks, or representations of data types. Before an object can store information it will need to have memory allocated. The amount of memory an object needs equals the amount of information it needs to store. For example, if we have an object that contains two integers, then we need to allocate memory which equals the size of the two integers combined. Now when an object is not used anymore, for example a thunk has been fully evaluated and the results is discarded, we can deallocate the memory associated with that object. The process of when and how much to allocate and when to deallocate memory is called memory management.
Memory management can be done in various ways:

- Manually: The language provides functions to manage memory. It is up to the programmer to allocate and free heap memory in the program and do this in a correct way.
- Garbage collection: This is one of the most common ways of memory management. At periodic intervals, the garbage collector is run and thus might cause a minor overhead. For latency critical applications, this can get problematic.
  Three major kinds of garbage collectors exist:
  - Mark & Sweep: Also called the Tracing GC and is part of GHC's collector since 8.10 [2]. Its a two-phase process, where first the live objects are marked and in a second phase un-marked objects are freed.
  - Moving collector: Heap reclamation involves copying all live data out of its current location (from-space) to a new segment of memory (to-space)[35]. The collector can then free from-space and consequently reclaim the memory occupied by the now-dead allocations. The heap is organized in a block-structured way [16]. This kind of GC was used as the main GC in GHC until 8.10, and is now part of the hybrid collection strategy of GHC.
  - Reference counting: Every object gets a reference count, which is the number of references to that object. If the reference count drops to zero, it can be garbage collected. An disadvantage of this kind of GC, is that cyclic references cannot be handled, since the reference count never drops to zero. So measures/workarounds have to be taken.

- Automatic Reference Counting (ARC): Instead of incrementing/decrementing reference counts at run-time, 'retain' and 'release' instructions are inserted at compile-time. Now when a reference count drops to zero, it is freed automatically. ARC can also not handle cyclic reference counts, so programming languages using ARC have specific keywords to deal with this problem. ARC is used in the programming languages Swift and Objective-C.

- Resource Acquisition Is Initialization (RAII): An object's memory allocation is tied to its lifetime, which is from construction until destruction. This programming idiom was developed primarily by Stroustrup and Koenig [32] for exception-safe resource management in C++. Besides used in C++ for implementation of smart pointers it is also used in Ada and Rust.

- Ownership & Borrowing: This kind of memory management combines RAII with an ownership model and is used in the programming language Rust [8].
  The ownership principle states that a value has one owner at any given time. Deallocation happens when the owner falls out of scope. Values can be borrowed, but only one borrow may exist at any given time. All other borrows are invalidated for the lifetime of a mutable borrow.
  Ownership and borrowing is enforced through a sophisticated type system to reason about the lifetime of data. The type system used in Rust is a form of linear type system, which is in turn a substructural type system. In §2.1 we will go into more detail about substructural type systems.

In Haskell, the approach used for memory management is a garbage collector. This reason is twofold: thunks are used for laziness and memory allocation/de-allocation is invisible to the user. The garbage collector of GHC is combination of the Mark & Sweep allocator and Moving allocator.

1.4 Purity

In a pure functional language, a function is pure if it (1) is given the same arguments and (2) it does not show any side-effects. Take for example the following C snippet:

```
double pi = 3.14;

double area(double radius) { return radius * radius * pi }

int main() {
    printf("%f\n", area(2.0)); // prints 12.56
    pi = 3; // pi is changed
    printf("%f\n", area(2.0)); // should print 12.56, but prints 12
}
```

The method, `area`, returns the area of a circle given the radius. This method is impure, since `pi` was not passed as a parameter to the function. This means that unless `pi` is immutable, this method will return a different solution if `pi` is modified. So to make this method pure, pass `pi` explicitly:

```
// pi is passed explicitly
double area(double radius, double pi) {
    return radius * radius * pi
}
```

Now his method will always return the same result given the same parameters for `pi` and `radius`.
The second (2) property can best be explained by giving another example:

```c
void counter(int *value) {
    *value += 1;
}

int main() {
    int count = 1;
    counter(&count); // increase by 1
    printf("%d", count); // prints 2
}
```

Here we initialize `count` with 1 and use `counter` to increase it by 1. The method `counter` is not pure, since the variable is mutated in place. Instead, we should return the updated counter:

```c
// a new variable is constructed containing the value increased by 1.
int counter(int value) {
    int updated = value + 1;
    return updated;
}
```

Now `counter` doesn't mutate variables. Note that when calculating the area in previous examples, `main` also mutated a variable: `pi`.
So in pure functional languages, operations a function performs are reflected in the return type of the function. Since pure functions can't modify state, all data is essentially immutable. So if you want to change an object, you instead return a new object with a new value, as shown in the examples.

1.5 Side effects and Purity

Purity has some clear advantages when it comes to compiler optimization techniques such as common subexpression elimination, memoization, and parallel evaluation strategies. These optimizations are possible because of *referential transparency*: Each of a program's terms can be replaced by its value without changing the meaning of a program. Since we have pure functions and immutable data this is possible.
Unfortunately programming languages that only support pure functions are useless in practice, besides using it for mathematical purposes. Working with randomness, files etc. becomes impossible, since they are impure by nature. So to add these features we need to be able to guarantee referential transparency. These side-effects can either be excluded from the language or we can deal with it in a special way.
Say for example we first want to write "1" and then "2" to a file. Our first attempt is the following:

```haskell
\f -> (writeFile f "1", writeFile f "2")
```

Here we use the hypothetical function `writeFile :: File -> String -> File` to achieve our goal. Unfortunately, laziness throws a spanner in the works: the evaluation is nondeterministic. If we are lucky the result is "12" or "21". However, if we are not, we could partially evaluate the first element of the tuple and then the second or vice versa. That is why in safe Haskell every operation that can potentially perform side effects, has to be captured inside a Monad [24]. For IO operations this is the IO Monad. The actual definition of `writeFile` is therefore `writeFile :: FilePath -> String -> IO ()`. The use of these monads, however, comes with a downside: the loss of the functional programming paradigm. To be able to correctly perform the operations in above example, we'll have to transform it to the following:

```haskell
do writeFile f "1"; writeFile f "2"
```

The operations are sequenced and written down in do-notation. Using this do-notation is similar to how a C programmer would write the program. This style is of course not very functional. This is fine if the operations

are side-effecting by nature, such as the given example. However, if side-effects are introduced with a goal of increasing the performance of the program, then the encapsulation of these side-effects should not affect the style of the whole program.
Take for example, the in-place updating of data structures, which is necessary to implement some algorithms efficiently. Quicksort and union-find come to mind. To do this in a safe way, it makes sense to encapsulate it in a controlled, monadic setting. However, if the whole program is functional, but would benefit from in-place updates, it shouldn't be necessary to rewrite the whole program in a monadic style.

1.6 Heap Recycling Analysis

Since thunk allocation is expensive we would like to find ways to reduce the number of thunks that are created in a program. In section §1.2 we already mentioned what kind of analyses we can use to reduce that. However, these analyses only reduce the first kind of thunks, those of functions and expressions. There is a second category of thunks, that also take a big part in the memory of consumption of a program: allocations of data. In the context of Haskell: "constructor application".
In the previous section we mentioned that in-place updates of data structures are a form of a solution. However, this requires a monadic programming style. Hage and Holdermans [15] describe an analysis that allows performing in-place updates while keeping a functional programming style. We will go now into more detail on what they have achieved.

1.6.1 Syntactic Extension

Take for example the following program for the reversal of an array:

```
function reverse(a[0..n - 1])
    allocate b[0..n - 1]
    for i from 0 to n - 1
        b[n - 1 - i] := a[i]
    return b
```

This program is not very efficient, since $O(n)$ extra space is required to store b. Furthermore, allocation is a relatively expensive operation.
The Haskell equivalent is the following:

```
reverse l = rev l []
  where rev []     acc = acc
        rev (x:xs) acc = rev xs (x:acc)
```

Two observations can be made: The C-like version uses arrays, while the Haskell version uses linked lists and because of lazy evaluation, the Haskell version causes the allocation to occur in the body of `rev`.
The first example can be made more efficient, by reusing the array `a`: overwrite `a` with its own reversal. This results in the following program.

```
function reverse_in_place(a[0..n - 1])
    for i from 0 to floor((n - 2) / 2)
        tmp := a[i]
        a[i] := a[n - 1 - i]
        a[n - 1 - i] := tmp
```

We can of course only do this if we do not need the original `a` elsewhere in the program.
Unfortunately, for the Haskell version, no such transformation can be made. However, since the Haskell version uses linked lists and each cons-node is stored in a thunk, we know that an already allocated thunk becomes

available for reuse: the thunk in which the cons-node is stored that was pattern matched against. Creating an in-place version of `reverse` is thus easy: we mark which thunk should be reused to store the result of the current computation. This marking comes in the form of a small extension that allows programmers to explicitly recycle available thunks.
Using this extension, the following in-place version of `reverse` can be given:

```
reverse l = rev l []
 where rev []       acc = acc
       rev l@(x:xs) acc = rev xs l@(x:acc)
```

In this example the conjunctive pattern `l@(x:xs)` is used on the cons-case of the helper function `rev`. In Haskell this pattern provides the user with a means to associate a name with a value that is successfully matched against a specific pattern. The extension now amounts to allowing the @-construct to appear not only on the left-hand side, but on the right-hand size of a function as well. If on the right-hand side a term of the form $x@e$ is used, it indicates that a thunk represented by x should be reused to store the result of operation e, where x is a variable and e a constructor application.

1.6.2 Correctness

To perform these in-place updates safely we must have the guarantee that the variables involved are used uniquely, i.e. at most once, in the program.
In section §1.2 we briefly mentioned that sharing analysis determines which expressions are used once. However, 'once', in the context of sharing, means demanded once. It is thus perfectly fine for an expression to be used more than once. In other words, there is no enforcement that expressions must be used once. Uniqueness analysis, which we will describe in §2.2, does give this guarantee.
Is this the only requirement for safe in-place updates? No, and this has to do with the way constructors are represented.
A common in-memory representation of constructors is a structure containing information about itself and a pointer for each constructor argument. But then, for example overwriting a cons-cell with a nil-cell is problematic. A cons-cell requires two pointers, one for the element and one for the tail. A nil-cell, however, requires no pointers at all. The other way around is troublesome as well, since special arrangements will have to be made to reclaim unused space.
To solve this problem, extra restrictions will be placed on what is a valid constructor update: Updates involving not only unequally sized constructors but also different constructors are flagged as invalid.
To be able to enforce this restriction, the binding for `l` in `reverse` is annotated with information about the constructor. Now, at the update site, `l@(x:acc)`, all information is available to check if this update marker is applied correctly.

1.7 Research Questions

The goal of this research is to see if it is worthwhile to implement heap recycling analysis which we described briefly in §1.6, in a real compiler targeting the programming language Haskell. The compiler in which we will implement this analysis is called Helium. The context of the research questions will thus be based on the Helium compiler.
We thus would like to answer the following research question:

Is heap recycling analysis effective in a functional language?

Heap recycling analysis is essentially made of two parts: uniqueness analysis, to verify if it is safe to recycle heap cells and heap recycling i.e. in-place updates. So to answer the main research question we need to answer the following questions first:

Q1: **Can we implement uniqueness analysis in Helium?**
Uniqueness analysis can be implemented in various degrees of complexity. Since we implement the analysis in a real compiler, we are in some form bounded by what the compiler provides. Another important factor, is that we are time bounded. We thus cannot implement all aspects of uniqueness analysis.

Q2: **What is a suitable source language for Heap Recycling Analysis?**
Helium has two intermediate representations. So depending on where uniqueness and heap recycling analysis are run, determines what sort of language we can or should use.
If we run uniqueness analysis separately from heap recycling analysis we may even have to devise syntax for multiple representations.

Q3: **What kind of performance increase can we expect when doing in-place updates?**
The goal of heap recycling analysis is to reduce the amount of memory allocations and thus reduce the pressure on the garbage collector. So we have to measure CPU and memory pressure. Of course, if there are no programs to test with, implementing the analyses does not make much sense. So we will write programs such that we can measure the performance.

1.8 Contributions

The type-based constructor and uniqueness analysis we explained, has not been incorporated in a real compiler. So we implement both analyses in the Haskell compiler Helium, a compiler mainly focused on generating high quality type errors.
While the typing rules combine constructor and uniqueness analysis, we separate the two. Correct constructor usage becomes a static check and uniqueness analysis becomes an analysis on the intermediate representation called Core. Since Core is already typed, we do not have to infer the type of the expressions, only the annotations. Uniqueness analysis thus becomes an annotation inference analysis.
The Helium compiler also supports user defined data types. Since the typing rules only have derivations for lists, we add support for arbitrary data types in the analysis.
All of our work will be implemented on the new LLVM based back-end introduced by Ivo Gabe de Wolff.
To get familiar with the compiler we have made `deriving` typed, since in the new backend this was not the case yet. In the process we discovered some bugs related to types and thunk evaluation.

2

Preliminaries

2.1 Substructural Type Systems

In type systems, such as Haskell, unrestricted use of variables is allowed in the type checking context. A variable may be used once, twice, thrice, etc. or not at all. A more precise way in allowing the use of variables, can be reflected in the type system. Such a type system is called a substructural type system [6, 23]. In such systems one or both of the substructural rules, weakening and contraction, are discarded.
Take for example the following two Haskell functions:

```
wasteful :: a -> b -> a
wasteful x y = x

frugal :: a -> (a, a)
frugal x = (x, x)
```

The weakening rule, as shown in `wasteful`, states that functions don't have to use all their arguments. The contraction rule states that functions can reuse their arguments, as can be seen in `frugal`.
Several type systems have evolved that discard weakening or contraction or both.

- Linear type systems discard weakening and contraction. This means that variables must be used exactly once. Thus both `wasteful` and `frugal` are ill-typed.
- Affine type systems discard contraction. This means variables must be used at most once. Thus `frugal` will not type check.
- Relevant type systems discard weakening. This means variables must be used at least once. Thus `wasteful` will not type check.

To show why substructural type systems are useful, we'll look at two examples: writing to a file and violation of a protocol.
Take again `\f -> (writeFile "1" f, writeFile "2" f)` as example. Since the evaluation order is undefined, the result is undefined as well. However, if the contraction rule does not apply, this example would be ill-typed , since `f` is used twice. Another example is given by using the following Haskell function:

```
openFile :: FilePath -> IOMode -> IO Handle
```

This function definition has two problems, as is shown by the following two functions:

```
dangling :: FilePath -> IO ()
dangling fp = do
  handle <- openFile fp ReadMode
  return ()

dfree :: FilePath -> IO ()
dfree fp = do
  handle <- openFile fp ReadMode
  hClose handle
  hClose handle
```

In `dangling` we forget to close the handle, leaving a dangling pointer. Since Haskell uses a garbage collector, dangling pointers could be cleaned up by the garbage collector. However, `dfree` is more problematic, since we close the same handle twice. A possible solution here, is to maintain some per-handle state.
Both solutions for dealing with `dangling` and `dfree` are not preferred, especially since we are working in a functional language. A simple observation, however, shows that `dangling` is a bit like `wasteful` and that `dfree` is similar to `frugal`. In other words, substructural type systems allows expressing protocols. In the case of `dangling` and `dfree`: A opened file must be used once and must be closed exactly once.
For example, lets say we typecheck `dangling` and `dfree` in a linear type system. Now, if `openFile` enforces in the type that the return value must be used exactly once, thus `openFile` return a handle that must be used exactly once and be closed exactly once, then both functions will not typecheck

2.1.1 Applying of structural rules

Discarding weakening and contraction for all arguments is a bit limiting. So we want to discard it only for some arguments. We will describe three ways in which this can be done. For each way the type of `writeFile :: File -> String -> File` is modified. To be consistent across all three ways, we will use the same syntax. This syntax may not represent the actual syntax in the target language.

Types: This is the most obvious way. This approach is taken in the functional programming language Clean[11]. `writeFile` could have the following type:

$\texttt{writeFile} :: \texttt{File}^1 \to \texttt{String}^\omega \to \texttt{File}^1$

Here a value of type a^ω means that the value can be reused or not used at all, while a value of type a^1 means that it must be used exactly once. Consider again the the `writeFile` example:

```
\f -> (writeFile "1" f, writeFile "2" f)
```

This will not typecheck, since `f` is used twice. The same applies to the `frugal` example:

$\texttt{frugal} :: \texttt{a}^1 \to (\texttt{a}^1, \texttt{a}^1)$
`frugal x = (x, x)`

Kinds: Instead of annotating types, we can separate types into two kinds: a unique kind and an unrestricted kind. Values with a type whose kind is unique are unique, and the others are unrestricted. This approach is attractive, since some types are inherently unrestricted (`Int`, `Bool`, etc.), while others (`File`, `Array`, `IO`, etc.) are unique.
Take for example `writeFile`. If this function is used in a function like `frugal`, then the result would be a kind error instead of a type error.
An example of a programming language that takes this 'kind' of approach, is Idris.

Arrows: Another way to indicate the distinction between restricted and unrestricted objects is to use the

arrow type. That is, there is both a restricted arrow and an unrestricted arrow. The Linear Haskell proposal [5] takes this approach. They introduce a new kind of function type to separate linear and unrestricted objects. As always we give `writeFile` as example:

```
fFrugal :: File ⊸ (File, File)
fFrugal f = (writeFile "1" f, writeFile "2" f)
```

Here the ⊸ operator, pronounced lollipop, is a really a function (`\(x ::`^{1}`) a -> ..`). The normal function type, `a -> b` is shorthand for (`\(x ::` a^ω`) -> ..`).
Since `f` is used twice in `fFrugal`, this will fail to type check.
Similar to `frugal`, we can give a type to `wasteful`:

```
wasteful :: a ⊸ b -> a
wasteful x y = x
```

2.1.2 Uniqueness and Linearity

In the previous section, two examples were given to ensure that `f` is used only once.
In the first example `writeFile` got the guarantee that its file argument would be used at most once, while in the second example `f` got the restriction that it will be used exactly once. This *guarantee* is called uniqueness typing, while the *restriction* is called linear typing.

Uniqueness: A unique value is one which has not been shared. It is safe to forget that a value is unique. However after a value is no longer unique, it can no longer be modified. The following function is thus allowed:

```
removeUnique :: a¹ -> aω
removeUnique a = a
```

This function demonstrates a relationship called subtyping. Subtyping will be explained in more detail in §2.2.1 and §2.2.4.

Linearity: A linear value is one which may not be shared. Linearity can for example be used to enforce that `dfree` can't close a handle twice.
While in uniqueness typing, we can coerce a unique type to a non-unique type, in linear typing we can coerce a non-linear type to a linear type. In linear logic this is called dereliction. The following example demonstrates this:

```
makeLinear :: aω -> a¹
makeLinear a = a
```

2.1.3 Uniqueness Typing

As discussed in previous section, uniqueness typing can be used to reject programs that will violate referential transparency. Since we can always approximate and say a program is unsafe, we want this analysis to be as precise as possible to be able to accept more programs as correct.
In the heap recycling analysis we don't reject programs, since the user cannot annotate the types in the program with uniqueness attributes. However, if the analysis marks an in-place update as unsafe, it won't be performed. Thus we want to accept as many programs as possible to maximize the performance benefit.
In §2.1.1 we described two possible ways of a possible implementation of uniqueness analysis. We will now go into more detail of both approaches.

Types: Annotations are placed on the type of an expression. Such a system is also called a type-and-effect system: an elegant way of extending a type system with the ability to track "things that may happen" (effects)

during evaluation.
Normal type systems have judgements in the form $\Gamma \vdash e : \tau$, indicating that within in context Γ, expression e has type τ. In type and effect systems, every judgement is extended with an effect. For uniqueness typing these effects are 1 and ω.

Kinds: In this approach, uniqueness attributes can be seen as type constructors of a special kind. For example the type of type variable a is $Type$, while a unique type variable a^1 is called $UniqueType$.
Since there are two separate kind of types, besides inferring types, kinds have to be inferred as well.

Since the language for which we will implement uniqueness analysis is already typed, we only need to infer annotations on the types. The kind-based approach will thus be too disruptive to the existing compiler. Also, the analysis we will discuss in next section, is a type-and-effect analysis. This makes the type-and-effect based approach the most suitable approach.

2.2 Uniqueness Analysis

The uniqueness analysis we will describe is based on the work by Hage, Holdermans and Middelkoop ([18], [15]). First we will describe the features of the type and effects system: its use of subeffecting §2.2.1 and polyvariance §2.2.2, its use of qualified types §2.2.3, and the absence of a subtype relation §2.2.4. In §2.3 we will explain the typing rules that are used.

2.2.1 Subeffecting

The function `removeUnique` as defined in §2.1.2 showed that it is safe to forget the uniqueness guarantee; there is a subtyping relation between the 1-annotation and the ω-annotation. We can formalize this relation, by giving a partial ordering on the annotations: $1 \sqsubset \omega$ is implied.
Using this ordering, subeffecting now states that it is safe to replace the effect component of an analysis by a bigger or equal effect value:

$$\frac{\Gamma \vdash t :^{\varphi'} \sigma \quad \Gamma \vdash \varphi' \sqsubseteq \varphi}{\vdash t :^{\varphi} \sigma} \qquad \text{(T-SUB)}$$

Here φ' is the effect of the argument passed in and φ the argument itself. Thus if $\varphi = \omega$ then φ' can be either 1 or ω, as stated by $\sqsubseteq$. In short, it thus safe to pass unique values to a functions with a ω-parameter.

2.2.2 Polyvariance

Instead of adapting the argument to the effect of the function argument, the function argument can also be adapted to the argument it is given. Take for example the following annotated function type:

```
double :: Int^ω -> Int^ω
```

By using the subeffecting rule, `double` can accept unique as well as non-unique variables. In other words, `double` doesn't care what the effect of its argument is. This can be expressed by assigning functions such as `double`, a type that is polymorphic in the effect of its parameter([18, §3.2]):

```
double :: ∀p. Int^p -> Int^ω
```

The same can be done for the result of a function, since the usage of the result of `double` is dependent on the context in which it is used. So the final type of `double` is:

```
double :: ∀p q. Int^p -> Int^q
```

Now `double` can be instantiated with all combinations of 1 and ω, which are all valid for `double`.

2.2.3 Constrained types

Using subeffecting and polyvariance allows the type and the effect of a term to be adapted to the context in which the term is used. This is especially useful when analyzing higher order functions. Take for example the following higher-order function: `apply f x = f x`. This function applies its first argument to its second. Assume that `f` is a shared function: it is used multiple times in the program. If function `f` requires that its argument must be unique then x must be unique as well. If `f` doesn't care, then x can be annotated with 1 or ω. Using these observations, `apply` can be given the following type:

`apply` :: $\forall \mathtt{a}\, p_1\, p_2\, q\, r.\ (p_2 \sqsubseteq p_1) \Rightarrow (\mathtt{a}^{p_1} \to \mathtt{a}^{q})^{\omega} \to (\mathtt{a}^{p_2} \to \mathtt{a}^{q})^{r}$

The effects on function `apply` are explained as follows:

- If the argument (p_1) of `f` has to to be unique then x (p_2) must be as well. This is reflected in $p_1 \sqsubseteq p_2$.
- The return value of function `f` requires that the return value must be used unique (q), then `f` applied to x must also be used unique.
- Function `apply` has no restrictions (r) placed on the returned value of the application of x to `f`.

This annotated type definition is still a bit imprecise, since we assume that `apply` works on shared functions. Abstracting over the usage property is more involved, since `apply` may be partially
applied. If `apply` is partially applied to a function `f`, and then used more than once in the program then the argument to `f` is also used more than once. So to maintain the relationship between the effect for `f` and the effect for partially applications of `apply`, another qualifier is needed: $r_1 \sqsubseteq r_2$. This results in the following type for `apply`:

`apply` :: $\forall \mathtt{a}\, p_1\, p_2\, q\, r_1\, r_2.\ (p_2 \sqsubseteq p_1, r_1 \sqsubseteq r_2) \Rightarrow (\mathtt{a}^{p_1} \to \mathtt{a}^{q})^{r_1} \to (\mathtt{a}^{p_2} \to \mathtt{a}^{q})^{r_2}$

Thus, the function argument (r_1) must be used as least as often as the associated partial application of `apply` (r_2).

2.2.4 Subtyping

In §2.2.1 we showed a subeffect relation between 1 and ω. This subeffect relation can be extended to a shape-conformant partial ordering on types to obtain a subtype relation. However, adding a subtyping to a type system complicates the design considerably [19, 17, 18], since an analysis not only has to deal with inequalities between effects, but also with inequalities between types. Given that the shape of a type, influences the inequalities that have to hold, instantiations of type variables may therefore introduce additional constraints. Fortunately, an explicit subtyping relation is in our case not necessary, since inequalities only arise between effects. Thus instantiating type variables does not lead to new inequalities. This means that if we carefully assign effects to types, the system can be just as expressive as a system with subtyping.
Take for example the following function: `two x = 2`, which ignores its argument and returns 2. A analysis based on subtyping assigns `two` its least restrictive type:

`two` :: $\forall \mathtt{a}.\ (\mathtt{a}^{1} \to \mathtt{Int}^{\omega})^{\omega}$

Now, if `two` is applied to a shared integer value, where the result must be stored as unique, subtyping coerces the type of `two` to: $\mathtt{Int}^{\omega} \to \mathtt{Int}^{1}$. This coercion, however, is unnecessary if `two` is given the following type instead: $\forall \mathtt{a}\, p\, q.\ (\mathtt{a}^{p} \to \mathtt{Int}^{q})^{\omega}$. Thus, if `two` is assigned the most polyvariant type instead of the least restrictive type, then subtyping is unnecessary.

2.2.5 Partial Applications

Uniqueness analysis without subtyping is less complex. Unfortunately, there is a catch: Since the type system does admit subeffecting, uniqueness annotations on typings can be enlarged in an arbitrary fashion. This can break referential transparency unless countermeasures are taken. The following two functions show that without special provisions [17], we can duplicate unique values:

```
const :: ∀a q r s. a^q -> (b^r -> a^q)^s

dup! :: ∀a v. a^1 -> (a^1, a^1)^v
dup! x = (\f -> (f ⊥, f ⊥)) (const x)
```

We first partially apply `const` to a unique value x, then we apply the resulting function `f` twice. Since there are no requirements (s) on how often we can call `f`, we claim to be able to duplicate unique values.
So Hage and Holdermans [15] adopt the following solution: add an attribute on the arrows [19]. This attribute says that if a function is curried, and its curried argument is unique, then the resulting function must be unique when applied. So `const` should actually have the following type:

```
const :: ∀a q r s. a^q -> (b^r ->^q a^q)^s
```

Now when `f` is applied we check if the annotation on the arrow adheres to the upper bound. In the case of `f` this is 1. However, f is applied twice (ω), thus `dup!` is ill-typed.
In §2.2.3 we defined `apply`. We now give the variant which also takes partial applications into account:

```
apply :: ∀a t p1 p2 q r1 r1 s1 s2. (p2 ⊑ p1, r1 ⊑ r2, s2 ⊑ r1) ⇒ (a^p1 ->^s1 a^q)^r1 ->^t (a^p2 ->^s2 a^q)^r2
```

2.3 Uniqueness Analysis: Type System

The language that Hage and Holdermans [15] use in the typing rules for uniqueness analysis is described in Figure 2.1 and is relatively straightforward.

Identifiers

$x \in$ **Var**		*(term variables)*
$\alpha \in$ **TyVar**		*(type variables)*
$\beta \in$ **AnnVar**		*(annotation variables)*

Terms

$a \in$ **Atm**	$:=$	$x \mid x@Nil \mid x@Cons(x\ x)$
$t \in$ **Tm**	$:=$	$a \mid \lambda x.\ t \mid t\ a \mid$ **let** $x = t$ **in** t
	$\mid$	$Nil \mid x@Cons(x\ x)$
	$\mid$	**case** x **of** $\{Nil \Rightarrow t; Cons\ x\ x \Rightarrow t\}$

Types

$\varphi \in$ **UnqAnn**	$:=$	$1 \mid \omega \mid \beta$
$\psi \in$ **ConAnn**	$:=$	$\epsilon \mid Nil \mid Cons$
$\pi \in$ **Qual**	$:=$	$\varphi \sqsubseteq \varphi$
$\tau \in$ **Ty**	$:=$	$\alpha \mid \tau^{\varphi} \xrightarrow{\varphi} \tau^{\varphi} \mid \text{List}\ \tau^{\varphi}$
$\rho \in$ **QTy**	$:=$	$\tau \mid \pi \Rightarrow \rho$
$\sigma \in$ **TyScheme**	$:=$	$\rho \mid \forall\alpha.\sigma \mid \forall\beta.\sigma$
$\Gamma \in$ **Ctx**	$:=$	$\emptyset \mid \Gamma, x :^{\varphi\mid\psi} \sigma \mid \Gamma, \pi$

Figure 2.1: Syntax [15, Figure 1]

Terms t can consist of variables, updates, lambda-abstractions, function application, local definitions, constructor expressions and case analyses. Constructors can only be applied to variables, and functions can only be applied to atomic terms. Such a term is either a variable or an update.
Types τ are constructed from type variables, function types, and list types. Each of these types are annotated with uniqueness annotations φ. A uniqueness annotation is either one of the constants 1 or ω or an annotation variable β. During typing, the order between annotations is captured in qualifiers of the form $\varphi_1 \sqsupseteq \varphi_2$. These can then be captured in type schemes σ by quantifying over types and annotation variables.
Typing contexts Γ are formed by bindings for variables and qualifiers. Each variable binding is mapped to type scheme σ and is annotated with both a uniqueness annotation φ and constructor annotation ψ. The value of ψ is either *Nil*, *Cons*, or the absence of information: ϵ.

2.3.1 Inference Rules

Using the described syntax we can define the typing rules. The judgments of the typing rules, seen in Figure 2.2, take the form $\Gamma \vdash t :^{\varphi} \sigma$, indicating that within typing context Γ, the term t can be assigned the type scheme σ and a uniqueness annotation φ.

Typing $\boxed{\Gamma \vdash t :^{\varphi} \sigma}$

$$\frac{\Gamma(x) =^{\varphi|\psi} \sigma}{\Gamma \vdash x :^{\varphi} \sigma} \quad \text{(T-Var)}$$

$$\frac{\begin{array}{c}\mathrm{fv}(t_1) - \{x\} = \{x_1, ..., x_n\} \\ \left.\begin{array}{c}\Gamma(x_i) =^{\varphi_{x_i}|\psi_{x_i}} \sigma_{x_i} \\ \hline \Gamma \Vdash \varphi_0 \sqsubseteq \varphi_{x_i}\end{array}\right\} \text{for each } i \in \{1, ..., n\} \\ \Gamma, x :^{\varphi_1|\epsilon} \tau_1 \vdash t_1 :^{\varphi_2} \tau_2\end{array}}{\Gamma \vdash \lambda x.\ t_1 :^{\varphi} \tau_1^{\varphi_1} \xrightarrow{\varphi_0} \tau_2^{\varphi_2}} \quad \text{(T-Abs)}$$

$$\frac{\begin{array}{c}\Gamma = \Gamma_1 \bowtie \Gamma_2 \\ \Gamma_1 \vdash t_1 :^{\varphi_1} \tau_2^{\varphi_2} \xrightarrow{\varphi_0} \tau^{\varphi} \quad \Gamma_2 \vdash x :^{\varphi_2} \tau_2 \\ \Gamma \Vdash \varphi_1 \sqsubseteq \varphi_0\end{array}}{\Gamma \vdash t_1\ x :^{\varphi} \tau} \quad \text{(T-AppVar)}$$

$$\frac{\begin{array}{c}\Gamma = \Gamma_1 \bowtie \Gamma_2 \\ \Gamma_1 \vdash t_1 :^{\varphi_1} \sigma_1 \quad \Gamma_2, x :^{\varphi_1|\epsilon} \sigma_1 \vdash t_2 :^{\varphi} \sigma\end{array}}{\Gamma \vdash \textbf{let } x = t_1 \textbf{ in } t_2 :^{\varphi} \sigma} \quad \text{(T-Let)}$$

$$\frac{\begin{array}{c}\Gamma \vdash x :^{\varphi_2} \text{List } \tau_1^{\varphi_1} \quad \Gamma, x :^{\varphi_2|Nil} \text{List } \tau_1^{\varphi_1} \vdash t_1 :^{\varphi} \sigma \\ \Gamma, x :^{\varphi_2|Cons} \text{List } \tau_1^{\varphi_1}, x_1 :^{\varphi_1|\epsilon} \tau_1, x_2 :^{\varphi_2|\epsilon} \text{List } \tau_1^{\varphi_1} \vdash t_2 :^{\varphi} \sigma\end{array}}{\Gamma \vdash \textbf{case } x \textbf{ of } \{Nil \Rightarrow t_1; Cons\ x_1\ x_2 \Rightarrow t_2\} :^{\varphi} \sigma} \quad \text{(T-Case)}$$

$$\Gamma \vdash Nil :^{1} \text{List } \tau_1^{\varphi_1} \quad \text{(T-Nil)}$$

$$\frac{\begin{array}{c}\Gamma = \Gamma_1 \bowtie \Gamma_2 \\ \Gamma_1 \vdash x_1 :^{\varphi_1} \tau_1 \quad \Gamma_2 \vdash x_2 :^{\varphi} \text{List } \tau_1^{\varphi_1} \\ \Gamma \Vdash \varphi \sqsubseteq \varphi_1\end{array}}{\Gamma_2 \vdash Cons\ x_1\ x_2 :^{\varphi} \text{List } \tau_1^{\varphi_1}} \quad \text{(T-Cons)}$$

$$\frac{\Gamma, \pi \vdash t :^{\varphi} \rho_1}{\Gamma \vdash t :^{\varphi} \pi \Rightarrow \rho_1} \quad \text{(T-Qual)}$$

$$\frac{\Gamma \vdash t :^{\varphi} \pi \Rightarrow \rho \quad \Gamma \Vdash \pi}{\Gamma \vdash t :^{\varphi} \rho} \quad \text{(T-Res)}$$

$$\frac{\Gamma \vdash t :^{\varphi} \sigma_1 \quad \alpha \notin \mathrm{ftv}(\Gamma)}{\Gamma \vdash t :^{\varphi} \forall \alpha.\ \sigma_1} \quad \text{(T-TyGen)}$$

$$\frac{\Gamma \vdash t :^{\varphi} \forall \alpha.\ \sigma_1}{\Gamma \vdash t :^{\varphi} [\alpha \mapsto \tau]\sigma_1} \quad \text{(T-TyInst)}$$

$$\frac{\Gamma \vdash t :^{\varphi} \sigma_1 \quad \beta \notin \mathrm{fav}(\Gamma)}{\Gamma \vdash t :^{\varphi} \forall \beta.\ \sigma_1} \quad \text{(T-EffGen)}$$

$$\frac{\Gamma \vdash t :^{\varphi} \forall \beta.\ \sigma_1}{\Gamma \vdash t :^{\varphi} [\beta \mapsto \varphi_0]\sigma_1} \quad \text{(T-EffInst)}$$

$$\frac{\Gamma \vdash t :^{\varphi_0} \sigma \quad \Gamma \Vdash \varphi_0 \sqsubseteq \varphi}{\Gamma \vdash t :^{\varphi} \sigma} \quad \text{(T-Sub)}$$

Figure 2.2: Typing rules of uniqueness analysis [15, Figure 4]

In T-Var the variable is retrieved from the environment, returning the type and usage.
In T-Abs, the lower bound on the usage of an abstraction is, besides its usage in the program, restricted by the annotations of the variables that appear free in the body of the lambda. The upper bound is given by the annotation on the arrow, since the type system admits subeffecting.
Now, in T-AppVar, when a function is applied to a variable, the annotation stored on the function must adhere to the upper bound. If this is indeed the case, this application is safe to do.
In the rule for case analysis, T-Case, we update the constructor annotation on the variable to a more informative constructor annotation of either *Cons* or *Nil*. The rules for list construction are T-Nil and T-Cons. A qualifier entailment in T-Cons is needed to meet the containment restriction.
The rules T-Qual and T-Res deal with introduction and elimination respectively of qualifiers.
Generalization and instantiation is handled by the rules T-TyGen, T-TyInst, T-EffGen, T-EffInst; T-Sub introduces subeffecting.

The typing rules make use of a number of subsidiary judgements and an entailment relation on qualifiers. Both are given in Figure 2.3. Context splitting judgements are of the form $\Gamma = \Gamma_1 \bowtie \Gamma_2$, indicating branching that can occur in a program's control-flow graph, such as function applications or let bindings.
To ensure that the order on uniqueness annotations is reflexive, transitive, and has 1 as least and ω as greatest element, an entailment relation on the qualifiers is employed: $\Gamma \Vdash \pi$.

Context splitting $\boxed{\Gamma = \Gamma_1 \bowtie \Gamma_2}$

$$\emptyset = \emptyset \bowtie \emptyset \quad \text{(C-Empty)}$$

$$\frac{\varphi \in \{\beta, 1\} \quad \Gamma_1 = \Gamma_{11} \bowtie \Gamma_{12}}{\Gamma_1, x :^{\varphi|\psi} \sigma = \Gamma_{11}, x :^{\varphi|\psi} \sigma \bowtie \Gamma_{12} \setminus x} \quad \text{(C-VarOnce1)}$$

$$\frac{\varphi \in \{\beta, 1\} \quad \Gamma_1 = \Gamma_{11} \bowtie \Gamma_{12}}{\Gamma_1, x :^{\varphi|\psi} \sigma = \Gamma_{11} \setminus x \bowtie \Gamma_{12}, x :^{\varphi|\psi} \sigma} \quad \text{(C-VarOnce2)}$$

$$\frac{\Gamma_1 = \Gamma_{11} \bowtie \Gamma_{12}}{\Gamma_1, x :^{\omega|\psi} \sigma = \Gamma_{11}, x :^{\omega|\psi} \sigma \bowtie \Gamma_{12}, x :^{\omega|\psi} \sigma} \quad \text{(C-VarMany)}$$

$$\frac{\Gamma_1 = \Gamma_{11} \bowtie \Gamma_{12}}{\Gamma_1, \pi = \Gamma_{11}, \pi \bowtie \Gamma_{12}, \pi} \quad \text{(C-Qual)}$$

Entailment $\boxed{\Gamma \Vdash \pi}$

$$\frac{\pi \in \Gamma}{\Gamma \Vdash \pi} \quad \text{(Q-Mono)}$$

$$\Gamma \Vdash \varphi \sqsubseteq \varphi \quad \text{(Q-Refl)}$$

$$\frac{\Gamma \Vdash \varphi_1 \sqsubseteq \varphi_2 \quad \Gamma \Vdash \varphi_2 \sqsubseteq \varphi_3}{\Gamma \Vdash \varphi_1 \sqsubseteq \varphi_3} \quad \text{(Q-Trans)}$$

$$\Gamma \Vdash 1 \sqsubseteq \varphi \quad \text{(Q-Bot)}$$

$$\Gamma \Vdash \varphi \sqsubseteq \omega \quad \text{(Q-Top)}$$

Figure 2.3: Context splitting and Entailment rules [15, Figure 4]

The rules T-VarOnce1 and T-VarOnce2 ensure that when there is branching, bindings for unique variables can go into either left or right context, but not both. Shared variables may go in both contexts (T-VarMany).

Typing (updates) $\boxed{\Gamma \vdash t :^{\varphi} \sigma}$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1(x) =^{1|Nil} \sigma_0 \quad \Gamma_2 \vdash Nil :^{\varphi} \sigma}{\Gamma \vdash x@Nil :^{\varphi} \sigma} \quad \text{(T-UpdNil)}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1(x) =^{1|Cons} \sigma_0 \quad \Gamma_2 \vdash Cons\ x_1\ x_2 :^{\varphi} \sigma}{\Gamma \vdash x@(Cons\ x_1\ x_2) :^{\varphi} \sigma} \quad \text{(T-UpdCons)}$$

$$\frac{\Gamma = \Gamma_1 \bowtie (\Gamma_{11} \bowtie \Gamma_{12}) \quad \Gamma_1 \vdash t_1 :^{\varphi_1} \tau_2^{\varphi_2} \xrightarrow{\varphi_0} \tau^{\varphi} \quad \Gamma \Vdash \varphi_1 \sqsubseteq \varphi_0 \quad \Gamma_{12}(x) =^{1|Nil} \sigma_0 \quad \Gamma_{22} \vdash Nil :^{\varphi_2} \tau_2}{\Gamma \vdash t_1\ x@Nil :^{\varphi} \tau} \quad \text{(T-AppNil)}$$

$$\frac{\Gamma = \Gamma_1 \bowtie (\Gamma_{11} \bowtie \Gamma_{12}) \quad \Gamma_1 \vdash t_1 :^{\varphi_1} \tau_2^{\varphi_2} \xrightarrow{\varphi_0} \tau^{\varphi} \quad \Gamma \Vdash \varphi_1 \sqsubseteq \varphi_0 \quad \Gamma_{12}(x) =^{1|Cons} \sigma_0 \quad \Gamma_{22} \vdash Cons\ x_1\ x_2 :^{\varphi_2} \tau_2}{\Gamma \vdash t_1\ x@(Cons\ x_1\ x_2) :^{\varphi} \tau} \quad \text{(T-AppCons)}$$

Figure 2.4: Typing rules of uniqueness analysis (updates) ([15, Figure 5])

Figure 2.4 shows the typing rules for in-place updates. They ensure that the value to be updated is guaranteed to be unique and is built with the right constructor.

3

Related Work

Probably the earliest known work on usage analysis is in the late 80's and earlier 90's. These analyses used abstract interpretation to derive information about the usage of partial application [9] or to measure the number of updates performed by a naive implementation of call-by-need evaluation [21].
Abstract interpretation is one of the three well-known frameworks used for the static analysis of programs, the other being type-and-effect systems.
The first framework uses the program control-flow to approximate the runtime behaviour and is mostly used in non-functional programming languages, the second framework uses the program structural syntax and is used to describe analysis that work on functional languages. Vouillon and Jouvelot [1] showed in 1995 that both approaches are actually not that different and that both can be used to derive analyses for a problem in question. However, most analyses use the type and effect approach since it is difficult to analyze higher order functions using abstract interpretation. On the other hand, the Cardinality analysis [9] used in GHC shows that abstract interpretation can be used to give a performing analysis.
In §2 we gave an introduction to Uniqueness typing/analysis and Linearity. Strictness, Sharing and Absence analysis try to achieve something similar as well: reduce the number of thunks. A lot of work has been done in studying these analyses individually. So in the next sections we will discuss the work which has been done on these analyses.
Since these analyses show many similarities, recent work puts effort into combining these analyses. Usage analysis [18] combines Sharing and Uniqueness, Cardinality analysis [9] combines Sharing + Strictness + Absence, and Counting Analysis combines all described analyses.

3.1 Absence Analysis

Absence Analysis [31] is an analysis that determines which expressions are defined as used. It is similar to dead code elimination. This analysis is especially useful for functions that only use a part of an argument. For example: `fst`, `length`, etc. These functions only look at the constructors of the respective data type (lists, pairs) and are plenty used in container-like data structures. Analyzing the usage of data types is thus very important.
Another use case is compiler generated code. Template Haskell comes to mind. This sort of code is often less "smart" and thus may contain a lot of dead code.

3.2 Sharing Analysis

Turner, Wadler and Mossin extend the Hindler-Milner type systems to include 'uses' [30], to determine whether values are used at most once. Their type system is based on ideas on linear logic. Unfortunately, their analysis performed rather poorly.
Take for example the following expression: `a + (f a) + (f b)`. Clearly `a` gets annotation ω, since it is used more than once. So a non-subsumptive type system must give `f` the type `Int`$^{\omega}$ `->` ... But since `f` is applied to `b` as well, `b` gets annotation ω. Because of this 'poisoning' problem, one call to `f` 'poisons' all others.
Wansbrough and Jones [28] extend the analysis to include polymorphism and user-defined algebraic data types. To make the analysis, to some form, context sensitive, they include a subtyping relation. However when applied to a full scale system, the results turned out to be unsatisfactory. It turned out that curried functions, which are used extensively in languages like Haskell, received rather poor types.
In a subsequent paper, Wansbrough and Jones, describe a solution to the 'currying' problem. They introduce a limited form of polyvariance called 'simple polymorphism'. They observe that usage polymorphism is critical for the accuracy of the analysis of large programs.
The work of Wansbrough and Jones [27] is extended by the PhD thesis of Wansbrough [25]. It contains the same analysis, but more attention is given to data types, implementation and proofs. Furthermore he introduces annotation schemes, a way to encapsulate a variety of approaches to assigning usage annotations to algebraic data types. However, this resulted in data types with many tens of usage parameters.
Gustavsson and Sveningsson [26] take a different approach to solve above problem. They state that 'simple polymorphism' [27] is rather inaccurate when it comes to data structures, since the analysis doesn't take the context where a function is called into account, which is crucial to analyze large problems. The number of constraints in 'bounded usage polymorphism' may be exponentially in the size of the program. So to deal with the huge number of constraints generated by this analysis, they represent the constraints compactly through calls to constraint abstractions.
Gedell, Gustavsson and Sveningsson study in [20] the impact of polymorphism, subtyping, whole program analysis or user defined data types on type based usage analyses. They came to the conclusion that all these features increase the precision of the analysis, but that acceptable results can still be achieved if some features are left out. The usage analysis is based on the type based analyses described in [30, 29, 28, 27, 25].

3.3 Uniqueness Analysis

A functional programming language that incorporates uniqueness typing as part of the type system is Clean [11]. It uses uniqueness typing for dealing with mutable state and I/O. De Vries, Plasmeijer and Abrahamson [19] modify Clean's uniqueness type system to give a more precise types. The typing rules they present depend on a sharing analysis that marks variable uses as exclusive or shared.
What makes their analysis different, is that their analysis is defined over the lambda calculus. while Clean's system uses graph rewrite rules. Another important difference, is their treatment of curried functions. While in Clean functions that are partially applied to a unique argument are necessarily unique (they cannot lose their uniqueness), they require that they must be unique when applied. They achieve this by adding an extra annotation on the function arrow, which is set to unique if the function has access to a unique parameter. Second, they extend the type system to allow for higher-rank types.
In a subsequent paper [17] they give an analysis that is simpler than Clean and what they described earlier. Uniqueness attributes become special type constructors, constraints are removed and encoded as Boolean expressions. Their simplified implementation however does not include higher rank types and impredicativity. They implement their analysis in an experimental functional programming language called Morrow.
An implementation of uniqueness typing for Haskell inspired by Clean is described by A. Middelkoop [22]. Compared to Clean they treat the components independently. This makes the analysis complex. The constraints that get produced are complicated and dealing with constraint duplication in the presence of polyvariance is hard (graphs with hyper edges, graph reduction).

3.4 Strictness Analysis

In Haskell, the programmer can force strict evaluation through the `seq` function or by annotating data type constructors with the bang pattern (!). Adapting strictness analysis to take these explicit strictness annotations into account, however, is not easy.
Hage and Holdermans describe a monomorphic, monovariant strictness analysis that includes subeffecting [12]. Besides demand propagation the analysis also keeps track of what they call 'applicativeness': which terms are guaranteed to be applied to an argument or may not be applied to an argument. They implement their analysis in a small language that includes booleans and integers and of course strict function application.

3.5 Combined Analyses

Sharing analysis is very similar to uniqueness analysis [28]: both analyses determine which terms are used once. There is however an important difference. In uniqueness typing, a function with a unique argument type places a demand on the caller, while sharing analysis places an demand on the callee. Thus in uniqueness analysis the function gets the guarantee that the argument is not used more than once, while in sharing analysis the function promises to not use the argument more than once.
This difference is made explicit by giving the subeffecting rule for sharing analysis:

$$\frac{\Gamma \vdash t :^{\varphi'} \sigma \quad \Gamma \vdash \varphi' \sqsupseteq \varphi}{\vdash t :^{\varphi} \sigma} \qquad \text{(T-SUB)}$$

The only difference with the uniqueness rule, is the direction of the effect: $\sqsupseteq$ instead of $\sqsubseteq$.
Hage, Holdermans, and Middelkoop make this duality more precise, by describing an expressive generic analysis that can be instantiated (with a parameter) to both sharing and uniqueness typing [18]. It is a polymorphic, polyvariant analysis which includes subeffecting. Subtyping is deemed unnecessary for two reasons: inequalities only arise between effects and types are given a most polyvariant type, making coercion unnecessary. The analysis has a parameter that specifies how the subeffecting rule works, and is the only difference they mention between the analyses, all other differences are intentionally ignored.
De Vries et al. point out that ignoring these differences can lead to problems [17]. Because of subeffecting, it is possible to coerce unique terms to non-unique terms. One must thus be careful with partially applied functions which can have unique elements in their closure. Hage and Holdermans suggest that this problem can be remedied by introducing an additional attribute on the arrows [15], as De Vries et al suggested [19].
Harrington [21] suggests a different solution to his problem. Two distinct sorts of functions are introduced: Ones that can have unique elements in their closure and ones that cannot. This solution is however not sufficient to guarantee referential transparency. De Vries et al [17] give an example when this is the case.
More recent research on combining Sharing, Uniqueness, and Strictness into one analysis is given by Verstoep [10]. The implemented analysis is very precise but isn't very fast. They don't guarantee that their algorithm for solving the constraints is correct. They also present static and operational semantics, to integrate the heap recycling part of [15] into the analysis.
Sergey et al. describe a Modular, higher-order Cardinality Analysis as is implemented in GHC [9]. Roughly Cardinality Analysis translates to combining Strictness, Sharing, and Absence analysis. To sidestep the complexity of using a type system based approach, they use an approach based on abstract interpretation. Since the existing strictness analysis in GHC was already a backwards analysis, they make this analysis a backwards analysis as well.
Compared to a polymorphic effect system, this approach has as consequence that third- and higher-order functions can't be given 'rich' demand signatures and that demand usage can't be tracked within data structures.

3.6 Heap Recycling

While uniqueness typing is primarily used to incorporate side-effects in lieu of monads, it also gives rise to certain performance opportunities. Clean, for example, uses the results of the uniqueness analysis to emit code that performs destructive updates in an automatic way. These optimizations show to be quite effective.
Hage and Holdermans bring the destructive updates feature to Haskell [15]. However, instead of doing it in an automatic way they add a syntactically light assignment operator. To check if the operator is applied correctly (the variables are used uniquely), they present a polymorphic, polyvariant uniqueness analysis with subeffecting. The system they describe requires duplication of marked functions
Dijksta et al. [14] implement the analysis of Hage and Holdermans in the Utrecht Haskell compiler. They extend the analysis to support user-defined data types.
To translate the uniqueness annotations they prepend for each function argument, an extra argument representing the uniqueness annotation. Now in the body of a function, one can pattern match on that extra argument and check if a destructive update is safe to do. An additional optimization pass is then run, that generates optimized functions with the pattern matchings removed. This will result in several versions of the same function, depending on the number of unique arguments and occurrences of application instances.
Yasunao Takano and Hideya Iwasaki [7] take a different approach on thunk recycling. Instead of reusing thunks that otherwise would have to be garbage collected, thunk recycling suppresses thunk generation by reusing and updating already allocated thunk at the tail of a list, under the condition that the thunk is singly referred.

3.7 Reference Counting

In §1.3, we explained the various ways memory management can be achieved. All the analyses described so far, assume that a lazy language is used, which uses some form of garbage collection. Ullrich and Moura[3] have explored reference counting as a memory management technique in the context of an eager and pure functional language. While reference counting has a higher overhead at runtime, it does enable optimizations such as destructive updates, since the exact reference count of each value is known.
To minimize the overhead of reference counting they describe an approach for minimizing the number of borrowed references and a heuristic for automatically inferring borrow annotations. The notion of borrowed references and explicit RC instructions for incrementing/decrementing come from the Swift compiler.
They implemented these techniques in an eager and purely functional programming language called Lean.
To analyze the performance of their compiler, they compare the runtime of various programs, which they have implemented in Swift, ML, Ocaml, Haskell, and Lean. They show, even though these programming languages are fundamentally different besides using a garbage collector or not, that both the garbage collection and the overall runtime and compiler implementation of Lean are very competitive with the other languages/compilers. In most analyzed programs, the runtime is similar to or outperforms the other implementations.

4

Helium

The Helium compiler is a Haskell compiler being developed at the University of Utrecht. The back-end that is still currently used in Helium is called LVM, which stands for Lazy Virtual Machine. It interprets the Haskell program using a stack-based instruction set.
Unfortunately LVM has some bugs and only supports the x86 architecture. This makes it hard to maintain. So currently, the compiler is in a phase where LVM is replaced by a new back-end called Iridium. Iridium is then compiled to LLVM.
The analysis and associated changes will be implemented in the new back-end of Helium. To understand the changes we will make, this chapter will give an introduction to the Helium compiler. We will explain the pipeline of Helium and the two intermediate representations, Core and Iridium.
Every phase that Helium performs, is described in the *Compiler.hs* file. The Phase{phase}.hs files represent the various phases.
For the intermediate representations Core and Iridium, we will give an example of the program `head`.

4.1 Haskell

4.1.1 Lexing & Parsing

The lexer and parser are based on the Haskell Language Report. Take for example the production rules for expressions:

```
exp  -> \apat1 ... apatn -> exp -- (lambda abstraction, n>=1)
      | let decls in exp -- (let expression)
      | if exp then exp else exp -- (conditional)
      | case exp of { alts } -- (case expression)
      | do { stmts } -- (do expression)
      | fexp -- (application)

fexp -> [fexp] aexp (application)

aexp -> var -- (variable)
      | con
      | literal
      | "[" "]"
      | "[" exp1 "," ... "," expk "]" -- (list, k>=1)
```

```
| "[" exp1 ( "," exp2 )? ".." exp3? "]" -- (arithmetic sequence)
| "[" exp "|" qual1 "," ... "," qualn "]" -- (list comprehension, n>=1)
| ()
| (op fexp) -- (left section)
| (fexp op) -- (right section)
| ( exp ) -- (parenthesized expression)
| ( exp1 , ... , expk ) -- (tuple, k>=2)
```

Using these definitions `Node a b` is parsed as the application of *a* and *b* to `Node`, while expressions such as `a `Node` b` and `x:xs` are parsed as list expressions. The phase 'phaseResolveOperators' transforms both to infix application.
For every non-terminal, a similar named Haskell function exists. For example, the rule *aexp*, is implemented in the `aexp` function. The implementation can be found in the folder *src/Helium/Parser*.
Not all production rules mentioned in this report are supported, but a reasonable subset is implemented.
The abstract syntax itself, can be found in *UHA_Syntax.ag* in the *src/Helium/Syntax* folder.

4.1.2 Importing

Every Haskell file has imports that need to be resolved. Some of these imports are implicit such as the Prelude. These imports are added for every Haskell file. Then, for every import the corresponding *.iridium* file is parsed into a Module.
The exported functions from a module are then added to the current compiling module and an environment is created that maps these functions to their types.

4.1.3 Static Checking

Some programs are syntactically valid, but semantically invalid. Static Checks reject those programs that are semantically invalid. Examples of warnings implemented are the following:

- Scope warnings such as unused and shadowed variables
- Functions without a function binding
- Instance declarations with missing functions

Examples of errors implemented are the following:

- Scope errors such as duplicate variables and functions.
- Checking for undefined variables and functions.

4.1.4 Type & Kind Inference

Type inference on a Haskell file is done in two phases. First constraints are generated by a traversal over the AST representation. The collected contraints are then solved using the Top framework.
Optionally kind inference can be performed before doing typing inference

4.2 Core

After running type inference on the AST representation of a Haskell file, it is desugared to Core. Core is a small, explicitly-typed, variant of System F.
The data types that Core consists of and the functions that operate over it, can be found in the 'heaprecycling' branch of 'lvm'. To give an impression of Core, we will first show the data types (§4.2.1) that make up Core

and then show how the Haskell function `head` is represented in Core.
On Core various passes are run for correctness (§4.2.3) and optimizations (§4.2.4). A simple type checker exists to check if the generated Core file is type correct. This is to make sure that the various passes don't generate invalid or type incorrect code.

4.2.1 Data Types

The language, Core, is essentially made of two data types: `Type` and `Expr`. The first data type describes the type system of Core, the second data type describes what kind of expressions Core has.
We will start with describing the type system of Core
Type.hs: Each expression has a type. The following data types, represent that type:

```
data Type = TAp !Type !Type
          | TForall !Quantor !Type
          | TVar !Int
          | TCon !TypeConstant
          | TAnn !SAnn

data SAnn = SStrict | SNone
```

```
data Kind = KFun !Kind !Kind | KStar

data Quantor = Quantor !Int !Kind !(Maybe String)

data TypeConstant = TConDataType !Id
                  | TConTuple !Int
                  | TConTypeClassDictionary !Id
                  | TConFun
```

We will discuss the most interesting constructors of `Type`, `Quantor` and `Kind`:

- `TCon` represents the different type of constructors a data type can have. Examples are normal data types, tuples and functions. It also includes typeclasses since typeclasses in Helium are represented using dictionary translation.
- `TVar` is used for type variables.
- `TForall` represents quantified types. For example the function `id :: a -> a` is represented in core as `id :: forall a. a -> a`.
- `Quantor` is used for the type variable name
- `Kind` is used for the type of type constructors, which for functions is *: `KStar`
- `TAnn` is used for strictness annotations on a type. An annotation is either strict (`SStrict`) or absence of strictness (`SNone`).

To support the primitive types `Int` and `Char`, `IntType` is introduced with constructors `IntTypeInt` and `IntTypeChar` for `Int` and `Char` respectively.

Expr.hs: The data type to represent expresssions is very similar to the one used in GHC Core:

```
data Expr = Let !Binds Expr
          | Match !Id Alts
          | Ap Expr Expr
          | ApType !Expr !Type
          | Lam !Bool !Variable Expr
          | Forall !Quantor !Expr
          | Con !Con
          | Var !Id
          | Lit !Literal

data Con  = ConId !Id | ConTuple !Arity
```

```
data Binds = Rec ![Bind]
           | Strict !Bind
           | NonRec !Bind
data Bind = Bind !Variable !Expr

type Alts = [Alt]
data Alt  = Alt !Pat Expr

data Pat = PatCon !Con ![Type] ![Id]
         | PatLit !Literal
         | PatDefault
```

```
data Variable =
      Variable { variableName :: !Id,
                 variableType :: !Type }
```

```
data Literal = LitInt !Int !IntType
             | LitDouble !Double
             | LitBytes  !Bytes
```

We will discuss the most interesting constructors of `Expr`:

- `Let` handles both recursive and non-recursive let-bindings; see the three constructors for `Bind`
- `Match` represents case expressions and pattern matches. The `Pat` data type represents the pattern that is matched upon. This pattern is either a constructor, `Con` or a literal, `Literal`.
- `Lam` is used for both term and type abstraction.
- `Forall` is the term variant of `TForall`. It binds type variable on the term level. This is different from GHC, which uses explicit (big) lambdas to bind type variables.
- `Con` is used for constructors.
- `Var` represents a variable. The Id is a hashed representation of the variable name.
- `Lit` is used for literals. A literal can be an `Int` or a `Double`. `Int` has an extra argument `IntType` to tell if this literal is a `Int` or a `Char`.

4.2.2 Example

Using the type system and AST defined in §4.2.1 we can give `head` a representation in Core:

```
1  head :: forall a. [a] -> a : public
2    = forall a: *. \''u$0'' : [a] -> let ''nextClause$'' :  a =
3      error { a } (''$primPackedToString'' "Prelude.head: empty list");
4      in let! ''u$0'' : [a] = ''u$0'' ;
5        in match ''u$0'' with {
6        ''::'' { a } x ''_'' -> x;
7          _ -> ''nextClause$'' ;
8        };
```

Every user-defined function is marked as public by default, as can be seen in the function binding of `head`. If a function matches on a data type, for example the empty list, then Helium makes this pattern match exhaustive. For every pattern that the user has not matched against, Helium inserts a call to `''nextClause$''`. A call to `''nextClause$''` means that the program will panic and print an error that pattern matching failed. In `head` we match on the wildcard, i.e. don't care pattern. This means that Helium can use the definition of this pattern for `''nextClause$''` instead of generating a `''nextClause$''`. The argument of `head` is always evaluated, thus the argument is evaluated strictly, as can be seen at line 5. The error of the wildcard pattern is not evaluated strictly, since it only occurs in one of the match arms.

4.2.3 Passes: Correctness

The following correctness passes are run:

- *Saturate*: Saturates all calls to externals and constructors. Take for example:

  ```
  data Foo = Foo Int Bool String

  x = Foo 1 true
  ```

In this example `x` has a partial application of `Foo`. The *Saturate* pass transforms `x` into the following: `x = \y -> Foo 1 true y`

- *Normalize*: Add let declarations for all non-trivial subexpressions except the binding of a let expression and the expression of an alternative in a match expression.
  Take for example: `x = f (g y)`. This is transformed to `x = let z = g y in f z`.

- *Rename*: Makes all declarations globally unique.

- *Lift*: Lifts lambda and non-strict let declarations to the toplevel. Transforms the program such that all non-strict let declarations have a function application as binding. This is done by adding new toplevel functions for those let declarations. Furthermore lambda lifting is applied: all lambda expressions will be moved to toplevel declarations.
  Take for example `a = \x -> let y = expr in \z -> y + z`. This becomes:

  ```
  a = \x -> let y = b x in c x y
  b = \x -> expr
  c = \x -> \y -> \z -> y + z
  ```

- *FunctionType*: Certain arguments of a function are strict. But this isn't reflected in the type yet. So the strictness of the type that is associated with the function binding is updated.
  This pass is special, since it is run on Core, but used in the generation of Iridium. So the result of this pass is not visible in Core.

4.2.4 Passes: Optimization

The following optimization passes are run:

- *LetInline*: Thunks are evaluated at most once. This may prevent inlining, as that duplicates work. So *LetInline* inlines lazy non-recursive let bindings if one of the following holds:
  - The definition of the variable is an unsaturated call
  - The result of the thunk is not shared
  - Variable is not used

- *LetSort*: There are three kinds of let declarations: recursive, non-recursive, and strict. The goal of *LetSort* is to determine which of these are really recursive and which are not and sorting these let declarations in the order they are used. Take for example:

  ```
  let a = h b c
      b = f c
      c = g b
  in [a, b, c]
  ```

  Here `a` depends on `b` and `c`, but both are defined after `a` is defined. Secondly, `b` and `c` are recursive, while `a` is not. So LetSort transforms this example to the following:

  ```
  let b = f c
      c = g b
  in
    let a = h b c
    in [a, b, c]
  ```

Thus recursive let-declarations are separated from the non-recursive, and the let-declarations are sorted according through their uses.

- *Strictness*: Analyze which expressions will always be used.
  `x = let z = g y in f z` is transformed to `x = let! z = g y in f z`, since we know that `z` will always be used.
- *RemoveAliases*: The *Normalize pass* introduces a lot of let bindings. Some of these let bindings are unnecessary and introduce aliasing of variables. The *RemoveAlias pass*, remove aliasing of variables.
  For examples `let x = y in f x` becomes `f y`.
- *ReduceThunks*: Some expressions are less expensive if there are evaluated strictly instead of lazy. *ReduceThunks* analyses which expressions are cheap to evaluate strictly.
  For example `let a = 0 in f a` becomes `let! a = 0 in f a`. In this example it is more expensive to put `a` in a thunk, then evaluating immediately.
- *RemoveDead*: Removes dead declarations.

4.3 Iridium

In Core two important details are still left implicit: memory allocation and laziness. So Core is desugared to Iridium[4]. Iridium is a strict, imperative language where laziness and allocation of memory is explicit. The type system of Core is used in Iridium as well.
Control and data flow of Core is transformed from lambda calculus to Static Single Assignment form, shortened to SSA[34]. Since SSA can express more, namely loops, it allows for more optimizations such as tail-call optimization.
The language that Iridium describes is a bit bigger than Core. So we will only describe the language on a high level.

4.3.1 Language

All functions must be top level, can have multiple arguments and may be partially applied.
To support laziness thunks are used. The runtime presentation of a thunk in Iridium is an object consisting of the following fields:

- **next**: a pointer to a function or another thunk
- **given**: number of given arguments
- **remaining**: number of remaining arguments of a function call. It may also contain a magic number to denote some state.
- **arguments**: the arguments

To evaluate a thunk we check if the number of arguments in **remaining** is zero. If the thunk is being evaluated, a magic number (32767) is written to **remaining**. Then we call the function pointer **next**, and replace the result by a pointer to the computed value. Finally, we again write a magic number (32766) to **remaining**, to indicate the thunk is fully evaluated.
A method in Iridium consists of blocks. The first block of a method always starts with *entry*. Since Iridium just like LLVM uses SSA, *phi* nodes are used when diverged control flow of a function merges back together. Consider `x = case a of True -> f; False -> g`. Here `x` is assigned either `f` or `g` depending on whether `a` is `True` or `False`. In Iridium this is written as follows:

```
entry.1:
  case %a.2: !Bool constructor (
    @True: Bool to match_case.3,
    @False: Bool to match_case.4)
match_after.7:
  %x = phi (match_case.3 => %.5: !Int, match_case.4 => %.6: !Int)
match_case.3:
  %.5 = eval %f: Int
  jump match_after.7
match_case.4:
  %.6 = eval %g: Int
  jump match_after.7
```

In this example, we first do a case match on the constructors of `Bool`. If this is true, we go to match case 3 otherwise we go to match case 4. Then in the match case we end up in, we evaluate the body.
After evaluating the body, we jump to match case 7. Since we could arrive either from 3 or 4, the *phi*-instruction assigns x either the result from 3 or 4.
In this example we used several constructs and instructions. In next section we will explain these instructions and constructs.

4.3.2 Instructions

Iridium has various instructions. We start with the instructions that can occur anywhere inside a block, but not as final instruction.

let: A let instruction binds an expression to a variable name. It is written as `let %name = expr`.

letalloc: Written as `letalloc` $\overline{bind}$. Is used for normal let bindings, but also for mutually recursive let bindings.
A *bind* describes the construction of an object in a 'letalloc' instruction. It consists of the variable to which the object is bound, the target and argument. A target represents what kind of object is created: function, thunk, or constructor.

match: A match instruction extracts fields from a constructor or a tuple. The type which is matched on is written down explicitly.

At the end of each block one of the following instructions is used:

return: Is similar to return in other languages. a `return var` returns a value from the method and gives control back to the previous method on the call stack.

jump: Is used when a function can be made tail-call optimized. written as `jump block`, it jumps unconditionally to the block.

case: matches on the constructor of the argument and jumps to the block to which the argument matches. It is written as `case var alts`.

4.3.3 Expressions

A let instruction binds an expression to a variable name. This expression can be one of the following:

literal: Denotes an integer, float, or string literal.

call: A call expression, calls a method. It takes the name of the called function, the type and arity of the function, and the method arguments.

instantiate: Instantiates a polymorphic value *var* with type τ.

eval: Evaluates the variable *var* to WHNF or yields the previously computed value.

castthunk: Takes a strict value *var* and converts it to a non-strict value by wrapping it in a thunk.

var: Evaluates to the value *var* of the argument.

phi: Represents a *phi* node in the control flow.

primitive: Is used for primitive operations like addition, comparisons etc.

4.3.4 Example

In §4.3.1 we already gave a small example on what Iridium looks like. Since that example was a bit short, we now a give a bigger example, again using `head` to demonstrate:

```
export_as @"head'" define @"simple#head'": { (forall a. !([a]) -> a) }
  $ (forall a, %u$0.0: !([a])): a [trampoline] {
entry.6:
  %u$0.1 = var %u$0.0: !([a])
  case %u$0.1: !([a]) constructor (
    @":": (forall a. a -> [a] -> [a]) to match_case.9,
    @"[]": (forall a. [a]) to match_default.11)
match_case.9:
  match %u$0.1: !([a]) on @":": (forall a. a -> [a] -> [a]) {a}(%x, _)
  %.10 = eval %x: a
  return %.10: !a
match_default.11:
  %.12 = literal str "Prelude.head: empty list"
  %.5 = var %.12: !String
  %.13 = call @HeliumLang#$primPackedToString[1]: (!String -> String)
         $ (%.5: !String)
  %.3 = var %.13: !String
  %.3.17 = castthunk %.3: !String
  %.14 = call @LvmException#error[1]: (forall v$0. [Char] -> v$0)
         $ ({a}, %.3.17: ([Char]))
  return %.14: !a
}
```

While `head` in Core was relatively short, and similar to the Haskell version, `head` in Iridium is more verbose. The reason of course, is the explicit memory management, evaluating thunks, and Iridium being in SSA form. Function `head` in Iridium is made of three parts:

- `entry`: This is the block which is equal to the left-hand side of a function. Since `head` matches a list on the left-hand side, `entry.6` also pattern-matches a list using the `case` instruction.

- `match_case`: The list turns out be non empty, so we extract the head from the list by using a `match` instruction. Then we evaluate `%x` to WHNF and return the result.
- `match_default`: Unfortunately our list is empty. So we call the error function on the provided string. Since Haskell is lazy, the packed string is cast to a thunk using the `castthunk` expression.

More complex functions have the same pattern as `head`. First we enter an entry block then we go to either a `match_case` or a `match_default`. Both match blocks can then contain further pattern matches.

4.3.5 Passes: optimizations

On Iridium two optimization passes are run:

- Dead code elimination: Removes all variables whose value is not needed and tries to remove unused arguments from functions where possible.
- Tail Recursion: turns recursions into loops where possible

After generating Iridium, it is lowered to LLVM and compiled to a binary if the program has a `main` function.

4.4 LLVM & Helium runtime

The last desugaring pass Helium performs is that from Iridium to LLVM. LLVM is designed to be a "universal" IR of sorts, and supports many CPU targets. It is also typed, but uses a different type system than that of Core and Iridium.
Every Iridium instruction and expression is translated to a sequence of LLVM instructions. In the case of the `letalloc` instruction, this also includes a call to the C function `helium_global_alloc(int size)`, to request the right amount of memory.
In §1.3 we described the various ways memory management can be done. Since memory allocation in Haskell is implicit, to allocate memory, a runtime is needed.
While the new Iridium/LLVM backend in Helium currently has no garbage collector, it does have a simple runtime:

- allocation: blocks with size of 128kb are allocated at once. Every time, a program requests a piece memory, for example for thunk allocation or constructing a data type, a small piece of memory is returned from the current allocated memory of block. If a block of memory is fully used, a new block of memory is allocated.
- deallocation: not supported yet. This means that a program that repeatedly allocates memory, will eventually run out of memory.

The implementation of the runtime can be found in *lib/runtime/runtime.c*.

5

Heap recycling analysis in Helium

Implementing Heap Recycling Analysis in a compiler such as Helium can be divided in roughly three parts:

1. Syntax and Semantic analysis: This includes lexing/parsing of the new construct and giving warnings/errors if it used at places that we don't support.

2. Uniqueness analysis: The actual analysis to place uniqueness annotations on the types. The analysis will be implemented on Core, which we will discuss briefly in §5.3.3. In §6 we will go in more detail on our implementation of uniqueness analysis

3. Code generation: This part consists of generating Core from UHA (§5.3.1), Iridium from Core (§5.4) and LLVM from Iridium (§5.5).

What makes implementing uniqueness and heap recycling analysis more difficult to implement in a real compiler such as Helium, is separate compilation. This means that we need to store the results of the analyses even after compiling a Haskell file. So we need to adjust the module system so that information that modules need is available. Thus, the code generation part includes this as well.
In this chapter we will explain the implementation of item 1 and 3. We will describe the syntax changes needed on Core and Iridium, how the type system of Core and Iridium will be changed, and finally how we changed the runtime of Helium so that we can measure the memory allocation difference when in-place updates are performed.

5.1 Syntax & Parser

In §4.1.1 we showed the production rules of expressions. These production rules don't support the @-construct on the right hand side. Thus without changes, we will get a parse error. Since '@' can be seen as an application to an expression, we will extend `aexp` with the following production rule:

```
aexp -> ...
      | var@( aexp )
```

However, since non-terminal `aexp` includes other terminals besides `con`, we cannot only parse `l@(x:xs)` but `l@(if r then (Right x) else (Right y))` as well. The second example is, according to the production rules, syntactically valid, however the semantics state that this is not a valid use case. So static analysis should reject such cases as invalid.
That said, a correct version of the second example would be: `if r then l@(Right x) else l@(Right y)`.

5.2 Static Checks

After checking if the @-construct is used correctly syntactically on the right-hand side we need to check if it is correct semantically as well. Since Uniqueness analysis will check valid use of @-application, we will keep the static checks rather simple and let uniqueness analysis do all the work.
We will check for the following three things:

- Undefined use of variable binding: `f = l@(x:xs)`
- Application of @ to unsupported expression:

  ```
  f l@(Right r) x y = l@(if r then (Right x) else (Right y))
  ```

- Wrong constructor application: `f l@(Right r) = l@(Left r)` or `f l@(a,b) c = l@(a, b, c)`. Even though we can see that `Right` and `Left` have the same memory occupation, we still need to check if this is the case. So for now we will require that both constructors have to be the same.

5.3 Core

After verifying that the program is syntactically and semantically correct, we can generate and check Core for the new construct. We will discuss the following things:

- How the new construct is lowered (§5.3.1).
- How the type system will be extended to support uniqueness analysis (§5.3.2).
- Where in the pipeline on Core, uniqueness analysis will be implemented (§5.3.3).

Finally, since the Core representation has changed, other areas that operate on Core have to change as well.

5.3.1 From UHA to Core

To lower UHA to Core we will need to add the new construct to Core. Since we apply the @ to the constructor of a type, we will extend `Con` with the identifier it has been associated with. So we change `Expr` to the following:

```
data Expr = .. | Con !Con !(Maybe Id) | ..
```

Using this definition `l@(x:xs)` will translate to the following Core expression:

```
Ap (Ap (Con ":" (Just l)) x) xs
```

When the second argument of `Con` contains an identifier, in this case `l`, no fresh heap binding is created, but the result of `(x:xs)` is written to the heap location associated with `l`.

5.3.2 Type System

To be able to store the annotations that uniqueness analysis introduces on the type we need to add support in Core. We have two options on how to adjust Core:

- Introduce a standalone annotated type system: create a new data type that supports the shape of a type, and only supports annotations. This way uniqueness analysis is truly separate from the existing analyses and other analyses have no knowledge of the annotations that get introduced. This has as advantage that we do not need to adjust other parts of the compiler, apart from the parser. The disadvantage however, is that the analysis becomes more complex. Since the existing types have no annotations, we somehow need to map annotations to the type they annotate. For functions this is rather simple. For data types, however, such types become incomprehensible pretty quickly. This makes debugging and testing the analysis hard to do.

- Extend the type system: Use the existing type system and add annotations on the type itself. Since Core already supports strictness annotations, this requires very little change to the type system of Core. However, now all analyses have information about these extra annotations, thus require changes. Since Core is typed, it has a type checker, and this needs to be adjusted to deal with these annotations.

We have chosen for the second approach, for two reasons. The first reasons is that adding annotations on the type itself follows the typing rules of uniqueness analysis and it is more intuitive. The second reason is because of the disadvantages of using a separate type system: translating the annotations in the separate system to the type it is applied to.
Since Core already supports annotations in the form of strictness annotations, we only need to extend the `TAnn` of `Type`. We introduce a new data type called `UAnn`, to represent the various possibilities an annotation can have: 1, ω, annotation variable, and absence of uniqueness.
To support polyvariant types, we add an extra constructor called `KAnn` in `Kind`. Now when a `forall` occurs in a type, we can distinguish type quantification from uniqueness quantification.
To support uniqueness qualifiers on types ($p \sqsubseteq q$), we add the `TQTy` constructor. Qualifiers will be used in the definition of functions and for constructor definitions. In §6 we will go in more detail on how qualifiers are used in constructor definitions.
These changes result in the following definition for `Type`:

```
data Type = TAp !Type !Type
          | TForall !Quantor !Type
          | TVar !Int
          | TCon !TypeConstant
          | TQTy Type [(UAnn, UAnn)]
          | TAnn !SAnn !UAnn

type UVar = Int
data Kind = KFun !Kind !Kind | KStar | KAnn

data UAnn = UUnique
          | UShared
          | UVar !UVar
          | UNone
```

Using these extensions, types can be annotated using the `TAp` constructor, the same as is done for applying strictness annotations. We give an example using the `reverse` function:

```
reverse :: ∀a p q. [a^ω]^p ->^ω [a^ω]^q
```

Here **a** is a type variable, and p, q are annotation variables. Each variable is introduced using quantification, where `Quantor` distinguishes what each variable entails.

5.3.3 Uniqueness Analysis

Now that Core has been adjusted to support in-place updates and uniqueness annotations can be applied to a `Type`, uniqueness analysis can be implemented. Since Core has various analyses that are performed for optimizations and correctness, deciding where to apply uniqueness analysis is important. Each pass could disrupt the results of uniqueness analysis. So to prevent prevent passes from intervening with uniqueness analysis, we run uniqueness analysis as the last pass on Core. There are various reasons for doing so:

- The Inlining, Renaming and Alias reduction pass reduce the size of the program, and thus the amount of constraints that get generated. Also all function/constructor applications are unnested, simplifying the analysis.
- The Lifting pass moves all nested functions to the top-level. This means that all let expressions are either variables or function/constructor applications. This also simplifies the analysis.

Since the various analyses reduce not only the complexity of implementing the analysis, but also the runtime complexity of the analysis itself, running the analysis as the last pass on Core, is the most sensible approach. The actual implementation of the uniqueness analysis will be explained in §6.

5.4 Iridium

Translating information from Core to Iridium is relatively simple. Because Iridium uses the same type system as Core, uniqueness information can be copied verbatim. The only thing left, is to translate the @-syntax from Core to Iridium. Since we extended `Con` to include the identifier (§5.3.1) , we are going to do something similar. In Iridium, the **letalloc** instruction is used for constructor and function application and allocate memory. Thus we will adjust the *bind* for constructor allocation.
In Iridium **letalloc** is represented by the following data types:

```
data Instruction = .. | LetAlloc ![Bind] !Instruction | ..

data Bind = Bind {
    bindVar :: !Id, bindTarget :: !BindTarget
  , bindArguments :: ![Either Type Variable]
  }

data BindTarget
  = ..
  | BindTargetConstructor !DataTypeConstructor
  | BindTargetTuple !Arity
```

In §4.3.3 we explained that **letalloc** can consist of multiple binds. Each `Bind` then consists of a variable to which the object is bound, the target and argument. The target for constructors is called `BindTargetConstructor` and for tuples `BindTargetTuple`. We extend both to include an extra argument: the identifier. This results in the following definition for `BindTarget`:

```
data BindTarget
  = ..
  | BindTargetConstructor !DataTypeConstructor !(Maybe Id)
  | BindTargetTuple !Arity !(Maybe Id)
```

Besides adding this simple change to the syntax of Iridium, the parser of Iridium needs to be adjusted as well. The reason is of course that Iridium is the representation that is used to support the module system.

5.5 LLVM & Helium runtime

Until now we only have made syntactic changes to pass information from Haskell through Core and finally to Iridium. To actually reuse memory we need to not allocate memory when we have the information to do so.
In §4.4 we explained the runtime that Helium uses in a compiled program. When a **letalloc** instruction is translated to LLVM, it includes the call to `helium_global_alloc`. So when `BindTargetConstructor` includes the identifier for memory reuse, we simply have to leave out this call to not allocate memory.
Using this simple change, already allocated memory, used for storing the information associated with the identifier can be reused.
To measure the reduced memory pressure by reusing heap space, we change the runtime as well. Since, `helium_global_alloc` is used for thunk, function and constructor application, we split this function in two: `helium_global_ds_alloc` for constructor allocation and `helium_global_fn_alloc` for function and thunk allocation. In this way, we can measure the amount of allocated memory for constructors separately from other type of allocations.

6

Uniqueness Analysis in Helium

In chapter 5 we described the changes necessary to perform in-place updates in Helium. To perform these changes safely we needed two implement two analyses: constructor and uniqueness analysis. We decided to let constructor analysis be a static check and uniqueness analysis a pass on Core. In this chapter we will explain our implementation of uniqueness analysis. The analysis we have implemented is polymorphic and includes subeffecting, but no constrained types. In Results we will explain what the consequences are of a system without constaints types.
The standard way of translating typing rules to a deterministic implementation is to use Algorithm $\mathcal{W}$:

- Given an expression e and a typing environment Γ (mapping of program variables to augmented types), the output is an augmented type τ for the expression and a substitution θ telling how the type environment has to be refined in order to obtain an typing. If it fails to find a typing, it returns a unification error.

In Algorithm $\mathcal{W}$, unification is based on the fact that two types are equal if and only if their syntactic representations are the same. For annotated types, this does not hold. Two annotated types may be equal even if their syntactic representations are different. Also, for analyses containing not only equality between types, but also inequalities, unification can not be used.
To solve this problem, Algorithm $\mathcal{W}$ can be extended to include constraints. During the inference algorithm not only substitutions are gathered but constraints as well. These constraints are then solved, either during type inference or at the end, to obtain additional substitutions. Just like in unification, if a constraint is violated, a constraint error is returned.
Algorithm $\mathcal{W}$ is however not suitable for our use case. This has one simple reason: the language (Core) for which we implement our analysis on, is already typed. This means that we only have to infer the annotations on the types and not also the types of all expressions. This simplifies the analysis, since we know how many annotations, for example a function application, needs. Thus we remove the type inference part of Algorithm $\mathcal{W}$ and only gather constraints by bottom-up traversal. We then solve the constraints per function.
In the next sections we will explain how and which constraints we generate (§6.1), how to solve them (§6.3), and finally how to solve the problem of when constraints are violated (§6.4).

6.1 Constraint Generation

Even though the typing rules as defined in §2.3 can give an almost direct translation to an implementation, there are still some rules left implicit: subeffecting, generalizing/instantiation and environment splitting. The first two rules can be incorporated quite easily. We generalize in a let-binding or function definition and instantiate in function or constructor application. The subeffecting rule is used in function application as well,

to allow application of a unique variable to a function expecting a shared variable.
Environment splitting is more subtle. The rules that define how an environment must be split are not deterministic: How can we split the environment in such a way that unique variables will only end up in one environment and all constraints are satisfied? A simple solution would be to generate all possible ways an environment can be split and check if a solution exists. This solution of course takes exponential time. Fortunately, more efficient solutions exist. We describe two ways, the first being top-down, the second bottom-up:

- If an environment Γ has to be split into Γ_1 and Γ_2, such as in function/constructor application and let bindings, for each variable generate two fresh annotations: one for Γ_1 and for Γ_2. The resulting Γ_1 is passed down to the left branch and Γ_2 to the right branch.
  At every binding site such as let and lambda-bindings, the introduced variable is added to the environment with a fresh annotation.

- Instead of splitting environments, we merge environments. We call this environment the 'usage environment'. Two environments Γ_1 and Γ_2 are merged in function/constructor application and let bindings. Now, when merging, if a variable occurs in both environments, generate a fresh annotation for that variable. The resulting Γ is passed up.
  Since environments are merged, a variable is added to the environment with a fresh annotation at the usage site and removed at the binding site.

The first approach follows the typing rules most closely. It is however far from efficient. Since we don't know which variables in the type environment will be used in a particular branch we have to generate fresh annotations for all variables in the environment. For small programs this impact will not be that great, but for big programs the amount of fresh annotations could potentially become huge. This approach also means that besides having 1,ω and β as annotations we need a 0 annotation as well, to capture variables which are not used.
Thus we will use the second approach: merging of 'Usage environments'. In this case we only generate annotations for variables that are used. The consequence however, is that we have to generate extra constraints at binding sites. This impact however is minimal.
Before we dive into our constraint language (§6.1.2, §6.1.3), we will explain the kind of environments we use in the analysis besides the 'usage environment'. Since the typing rules only support lists and the syntactic representation of constructors in Core is a different than the rules describe, we explain in §6.1.4 how to deal with data types.

6.1.1 Environments

In the previous section we introduced the 'usage environment', to store the usage of variables. However, to create all constraints necessary to infer the uniqueness of the variables and functions, we need other environments as well. We distinguish these environments in those that are passed down the syntax tree (inherited) and those that are passed up (synthesized). The inherited environments are extended at the binding site of variables, the synthesized environments are extended at the occurrence of a variable.
The environments we have are now as follows:

- Inherited environments:
  - **Type environment**: A mapping from identifiers to types in the current module and to annotated types for imported functions and data type constructors.
  - **MSet**: Stores all annotation variables occurring in nested types introduced in lambda bindings. We use this environment in instantiation constraints, explained in §6.1.3.

- Synthesized environments:

- **Assumptions**: Contains for every function call or variable introduced in a let binding, the annotated type. Is used to create, at the binding site of a variable, instantiation constraints.
- **Constraints**: The set of annotation and type constraints.
- **Usage environment**: The usage of every variable.

Using the described environment all information is available which is necessary to create annotation and type constraints.

6.1.2 Annotation Constraints

The constraints we generate can be split into two categories: type and annotation constraints. We first discuss the annotation constraints.
The first two annotation constraints are rather trivial and have already shown up in the type rules:

- Inequality, (p $\sqsubseteq q$): We generate this constraint in the lambda rule and application rule, but we also use it for subeffecting in function application, let binding usage, and in variables introduced in case binding.
- Equality, ($p == q$): If a function uses in-place updates, the variables involved must be used once. We enforce uniqueness by using an equality constraint. We also use equality constraints at lambda- and let-bindings, to enforce that the usage of the binder is the same in the whole function. The reason of course is that we build the usage environment bottom-up. The last place where we use equality constraint is at pattern bindings: the usage of the variable on which is pattern matched upon must be the same as the usage of the constructor. In §6.4 we go into more detail.

The last two annotation constraints we generate have to do with the merging of usage environments:

- And, (p == q && r): Take for example the addition of two numbers: `e1 + e2`. To calculate the result of this addition we need to calculate both expressions. Now, suppose that a variable x occurs in both branches, then the combined usage of x is shared, since x is used more than once.
  The &&-Constraint captures the usage of a variable which occurs in both branches of an application, but also for a let-binding: `\a -> let b = a in (b, a)`. Here a occurs in both branches of the let, and is thus used more than once.
- Or, (p = q || r): This constraint is generated when only one of the branches gets executed. This only happens in pattern bindings. Take for example the expression `if e then e1 else e2`. In this case only `e1` or `e2` gets executed, but certainly not both. It is thus safe to update a variable x in-place in one branch, while in another branch x is used shared. The ||-Constraint captures the usage of a variable that occurs in both branches, but where only one branch is executed.

Using these four kinds of constraints we can derive usage of variables in a function.

6.1.3 Type Constraints

Annotation constraints are only sufficient for top level annotations on types, but not for nested types like lists or trees. Take for example the list type: `[a]`. To be able to track the usage of a list, we not only need the usage on top of the list definition, but also of the type variable a. Thus, just like we had (in)-equality constraints on annotations, we have them on types as well.
To support polyvariance we also need type constraints. We generate instantiation constraints at let definitions. For this we capture, besides the type of the function to generalized and instantiated, the following information:

- A Boolean indicating if this constraint must be solved in a monovariant or polyvariant way. If a function is member of a recursive binding group, then we can only instantiate monovariantly, otherwise we can solve polyvariantly.

- The visibility of the function to be instantiated. If it is a global function then this constraint can be solved immediately, since the annotated type of the function is known. For local functions i.e. let bindings, this is not the case. Other constraints have to be solved first.
- The set of annotations which must not be generalized. This includes the Mset, but also the usage of the let-definition itself.

Using the described type and annotation constraints, we can generate constraints for a polyvariant uniqueness analysis with subeffecting. What is still missing is annotating constructor definitions, which are described in the next section.

6.1.4 Data types

Giving precise annotations to data types for a type-and-effect based analysis is hard. If we want to be very precise, then number of annotations will grow out of proportion. We can also be pessimistic, and annotate all arguments and the data type with the same annotation. In the case of uniqueness analysis: ω.
Before we describe how we annotate data types, we first explain how data type definitions are defined in Core. These definitions determine mostly on which annotations we can place. Since the typing rules give a rule on how to annotate lists, we start with lists as example.
In Haskell we can give the following definition for lists:

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

This definition is desugared to Core by giving a separate definition for each constructor together with the type of the constructor. For lists this results in two definitions, for `Cons` and `Nil` respectively:

```
Cons :: ∀a. a -> List a -> List a
Nil  :: ∀a. List a
```

This way of writing constructors is also called GADT syntax. Another example, this time for trees that can only contain integers:

```
data Tree = Node Tree Int Tree | Leaf

Node :: Tree -> Int -> Tree -> Tree
Leaf :: Tree
```

Now, to annotate lists or trees, given the typing rules for lists, we will adhere to the following restrictions:

- Every occurrence of the data type name in the constructor must have the same annotation. If the name of the data type occurs in one of the arguments of the constructor, we have a recursive data type.
- The usage of every argument in the constructor must be as least as big the whole constructor. This is again the containment restriction.

We tighten these restrictions even further by saying that all arguments have the same annotation, except type variables: each fresh occurrence of a type variable gets a different annotation variable. The reason for these extra restrictions has to do with the return type of a constructor: It only lists the data type name, not the arguments the constructor has. So after we have constructed a new data type, we lose the usage of the individual arguments. Thus introducing separate annotations for each argument, will only increase the amount of annotations but will not increase the precision. Furthermore, if a resulting data type is used more than once, then the elements are probably also used more than once.
Thus, using the rules we have defined for annotating constructors, we annotate the constructors of `List` and `Tree` in the following way:

```
Cons :: ∀apq. a^q -> (List a^q)^p -> (List a^q)^p, p ⊑ q
Nil  :: ∀apq. (List a^q)^p, p ⊑ q

Node :: ∀q. Tree^q -> Int^q -> Tree^q -> Tree^q
Leaf :: ∀q. Tree^q
```

As you can see, in the annotated type of `Node`, all arguments receive the same annotation. This means that after constructing a `Node` we cannot distinguish usage for every argument independently, since the return type of Node can only contain one annotation. If we were to change the definition of `Tree` to take a type variable instead of `Int`, the annotation of the argument would be available. In the type of `Cons` this is the case. Here we can infer separate usage for the head of the list and the spine of the list.
This observation, shows that the return type of a constructor decides how precise the annotations on a data type are. Take for example the following definition of a Matrix:

```
data Matrix a = Matrix ([[a]])
```

In this definition we can only infer usage for the type variable a, and the whole definition of `Matrix`. So if we were to update only the rows `Matrix`, then the columns must be unique as well, while they could have been shared.
One possible change we could make, is to allow multiple annotations on the return type of the constructor, i.e. the data type definition itself. This way we can infer usage inside data type definitions. The consequence of this approach, however, is that the number of annotations will grow very quickly, possibly exponentially, including the number of constraints. Since in most cases, data types are constructed as whole at once, the chance that one of the elements in a data type has different usage is probably pretty low. So, even though our approach may be pragmatic, we think for our use case it is enough.
Wansbrough [25], Section 5.4 gives a detailed report on how data types can be annotated algorithmically and what subtleties can arise if a particular algorithm is chosen.

6.1.5 Implementation

Using the constraints and the algorithm for annotating data types, we can now explain the constraint gathering algorithm. The whole implementation is located in 'Uniqueness' under 'Helium/Codegeneration/Core'.
The algorithm is roughly a three step process:

1. Annotating data type constructors using described algorithm. We walk over all *con* declarations in the current module.

2. Create per function type and annotation constraints.

3. Create annotation constraints to determine usage of global function definitions. Since functions can be used more than once, can be exported and written in arbitrary order, we have to be pragmatic in the usage of a function definition. For exported functions the usage is shared, for private functions it depends on how often it is used.
   Thus for each private function, we first generate a fresh annotation. Equality constraints are then generated between the annotation variable returned from a function and this fresh annotation.
   If a private function is used more than once in different functions, it occurs in multiple usage environments. Therefor we create &&-constraints for the multiple usage and merge the usage environments.

The only step we have not explained yet, is the last step. We now explain per type of expression, the type of operations we perform. We start with simple expressions: literals and variables, then we will explain lambda bindings, let expression and pattern matches. Finally we explain function and constructor application.

- **Constant**: We infer the type of the literal and generate a fresh annotation variable u.

- **Variable**: First we retrieve the type of the variable from the type environment. Then we annotate the type with fresh annotations and add it to the assumption environment. lastly we create a fresh annotation variable u to note the usage of this variable and add it to the usage environment.

- **Lambda**: The annotation on a lambda variable equals that of the total usage of this variable in a function. Since the type of a lambda variable is monomorphic, the type cannot change. Thus we annotate the type with fresh annotations immediately and add it to the type environment. This reduces the amount of type constraints we have to create.
  We then collect the annotations from the annotated type and add them to the MSet. Besides using this set in a **Let**, we also use this set to for inequality constraints between the usage of the lambda variable and the annotations in this set.
  After collecting constraint from the body of the lambda, we proceed according the lambda rule. We generate a fresh annotation which will be place on top of the arrow and use it to generate the containment constraints.
  The annotation on the argument of the lambda is retrieved from the usage environment and becomes an annotation on the type of the lambda variable. The annotation we return from collecting the constraints in the body of the lambda, is placed on top of the return type of the lambda.

- **Let**: The let is probably the most complicated rule to implement. We start with generating constraints for every bind in a **Let**. If an identifier of a bind occurs in multiple binds, thus in multiple usage environments, we create &&-constraints. For recursive bindings we also create monomorphic instantiation constraints. After collecting the constraints for the body of the let, we create polymorphic instantiation constraints. Since variables can be used in both the binds and the body of a let, we again create &&-constraints.
  Finally we add inequality constraints between the annotation of a bind and the annotation of that bind occurring in the usage environment.

- **Match**: First we retrieve and instantiate the type of the scrutinee. Then we generate constraints for each alternative of a match. Since the return type and usage of the body of a match arm has to be equal, we create type and annotation equality constraints. If a variable occurs in both branches, and thus occurs in two usage environments, we generate ||-constraints.
  Now for each match arm we collect constraints. The type of the constructor on which is matched in instantiated. The return type and usage of the constructor must match the type and usage of the scrutinee, thus we add equality constraints to reflect that.
  Finally, for every argument of the constructor in a match arm, the usage of that argument must match the usage in the body. So we add annotation equality constraints for that.

Even though the types of functions and constructors are similar, there is a distinction between function and constructor application: functions have an extra annotation on the arrow. So we will discuss function and constructor application separately.
Since variables are applied to constructors and functions, we first check if the current expression is a function or constructor application. Using this knowledge, we create constraints in the following way:

- **Function**: If the expression we analyze is a function application, we go into recursion until we reach the identifier of the called function. We then retrieve the type of the function and annotate it with fresh annotations.
  For every argument we apply the T-AppVar rule. We add an inequality constraint between the usage of the function (φ_1) and the annotation on the arrow (φ_0). To achieve subeffecting we create an inequality constraint between the usage annotation of the callee (φ_3) and the usage annotation on the type of the function argument (φ_2). The same happen, for nested types: we add a type inequality constraint.
  Finally, since a function with multiple arguments can be applied to same identifier more than once, the variable that is currently applied can already occur in the usage environment. We thus generate a &&-

constraint if this is the case.
The usage we return (φ) is that of the return type.

- **Constructor**: If the expression we analyze turns out to be constructor application, we go into recursion until we reach a *Con*. We now have two cases:
  - **Reuse**: The constructor that gets applied contains a reuse. Thus we create an equality constraint with the annotation variable we applied to the constructor (`((Cons @l) { u0 } ..`), stating that the value of this annotation variable must be 1.
    We then instantiate the constructor and add the annotation variable $u0$ to the usage environment.
  - **Normal**: We have no reuse, so we only need to instantiate the constructor.

  After obtaining the instantiated type of a constructor, we proceed similar to function application. The only difference is that we have no annotation on the arrow, so no constraints are generated for that. Also, instead of using inequality type and annotation constraints, we use equality constraints. The reason for this is that we put values inside a container and the annotations must exactly match.

The result of the constraint gathering algorithm is a mapping from function identifiers to a set of assumptions, type and annotation constraints, and a global set of usage constraints.
In §6.3 we will explain how we solve the constraints and how we turn the set of assumptions, gathered per function, into instantiation constraints.

6.1.6 Example

To show the number of constraints and of what kind, we will use `reverse` as an example. We first translate `reverse` to Core. For simplification reasons, we use a custom List definition. The cleaned up Core version is as follows:

```
1  con Cons :: forall a. a -> List a -> List a;
2  con Nil :: forall a. List a;
3
4  reverse :: forall a. List a -> List a
5  reverse = forall a. \!xs: (List a) ->
6    let! .17: (List a) = Nil { a };
7    in rev { a } xs .17;
8
9  rev :: forall a. List a -> List a -> List a
10 rev = forall a. \!xs: (List a) -> \ys: (List a) ->
11   case xs of {
12     Nil { a } ->
13       let! ys.9 = ys;
14       in ys.9;
15     Cons { a } x xs.6 ->
16       let! .19: (List a) = (Cons @ xs) { a } x ys;
17       in rev { a } xs.6 .19;
18   };
```

Table 6.1 shows the kind of constraints we gather for `reverse`.

Equality	Inequality	&&	||
8	24	0	1

(a) Nr. of Annotation Constraints

Inequality	Equality	Instantiation
4	5	3

(b) Nr. of Type Constraints

Table 6.1: Constraint Gathering

As you can see, we generate exactly one ||-constraint, corresponding to the use of `ys` in both branches. We also generate three instantiation constraints, corresponding to the three let-bindings in the program.
In §6.4 we will explain how to solve these constraints, and how we create the missing instantiation constraints for `rev`.

6.2 Primitive operations

Until now we assumed that all modules we compile have Haskell as source language. These modules are then desugared to Core and are optimized. On the optimized Core we then run uniqueness analysis. There are however modules in Helium which are not written in Haskell but in Core or Iridium.
The Core modules are mostly there because of legacy reasons and are not that problematic, since the uniqueness analysis is applied to Core as well. These modules can also, for the most part be rewritten in Haskell, with some support glue in Iridium. They mostly contain functions on integers and strings.
The Iridium modules are more problematic, since we cannot run uniqueness analysis directly on these modules. For these modules we have to add annotations manually. This is relatively easy, since the functions in these modules are either function definitions to C functions or primitive operation definitions on integers and strings.

6.3 Constraint Solving

The constraints we have gathered per function using the constraint gathering algorithm can only be solved in a specific order. For example we can only instantiate the annotations on a type of a function until we have determined what those annotations are.
Besides annotation constraints we also have type constraints, which must be solved as well.
We start with explaining how we determine the dependencies between functions (§6.3.1). We then follow up with how we solve the annotation constraints (§6.3.2, §6.3.4) and type constraints (§6.3.3). Finally, we combine all solving methods to give our constraint solving algorithm (§6.3.6).

6.3.1 Binding Group Analysis

Using the constraint gathering algorithm described in §6.1, we have gathered for almost all expressions the necessary constraints. For occurrences of global functions, however, we haven't created instantiation constraints yet. The reason for this is that some functions may be recursive or a part of a group that are mutually recursive. During constraint gathering this fact is not known, so we don't know which functions are recursive.
To determine for which instantiation constraints we must solve monomorphic or polymorphic we run binding group analysis. This analysis is equivalent to calculating the 'strongly connected components' (SCC) on a graph. To calculate the SCC we use the following Haskell function, defined in `Data.Graph`:

```
data SCC node = AcyclicSCC node -- a single node that is not in any cycle.
              | CyclicSCC [node] -- a maximal set of mutually  reachable nodes.

stronglyConnComp :: Ord key => [(node, key, [key])] -> [SCC node]
```

The input is a list of nodes uniquely identified by keys, with a list of keys of nodes this node has edges to. The output is a list of `SCC`, reverse topologically sorted. Every node can contain extra information, which is in our

case the name and type of the function, and the type and annotation constraints.
Now for every function call in a binding group we determine if the function name occurs in the binding group. If this is the case, the resulting instantiation constraint must use equality

6.3.2 Global Constraints

In §6.1.5 we mentioned that we create constraints to determine the usage of functions definitions. Since these constraints are not function specific, they can be solved independently of any function.
The set of constraints only consists of &&-constraints and equality constraints. For the equality constraints we use substitution. For the &&-constraints we mark the annotation on the left side as well as those on the right side with ω. If for a particular function call, an &&-constraint exists, this function is used more than once.

6.3.3 Type Constraints

As mentioned earlier we have three kinds of type constraints: inequality, equality and instantiation constraints. Inequality constraints are rather trivial to solve: map over both types at the same time and create annotation equality constraints. Take for example the following type equality constraint between two integer lists: `[Int`^{p}`]` $\equiv$ `[Int`^{q}`]`. This constraint becomes the following annotation constraint: $p \equiv q$.
For inequality constraints we do something similar. However, to be sound with respect to subtyping, we have to be careful if the type in an inequality constraint is a function. If this is the case we take the following measure: We swap the arguments. Take for example the following type inequality constraint: `[Int`^{p}`] ->`^{q} `[Int`^{r}`]` $\sqsubseteq$ `[Int`^{s}`] ->`^{t} `[Int`^{u}`]`. We create the following three inequality annotation constraints: $r \sqsubseteq t$, $r \sqsubseteq u$, $s \sqsubseteq p$. As you can see, for the last inequality constraint the direction is flipped.
The last type of constraints, instantiation constraints, are solved using a three step process. As example we use the following instantiation constraint:

```
[q] -- Mset
[Int^ω]^q ->^q [Int^ω]^p -- function type
[Int^r]^s ->^t [Int^u]^v -- assumed function type
```

The three steps are now as follows:

1. Generalizing function type: $\forall p.$ `[Int`$^\omega$`]`^{q} `->`^{q} `[Int`$^\omega$`]`^{p}. Since q occurs in the Mset, it is not generalized.
2. Instantiation function type: We introduce fresh annotation variables for quantified annotation variables. We thus get the following function type: `[Int`$^\omega$`]`^{q} `->`^{q} `[Int`$^\omega$`]`^{o}
3. Create equality constraints: $r \equiv \omega, s \equiv q, t \equiv q, u \equiv \omega, v \equiv o$.

In the case of monomorphic instantiation constraints we can skip step 1 and 2, since no annotation variables are generalized.

6.3.4 Annotation Constraints

To solve annotation constraints we use a *worklist* algorithm, which is based on the algorithm described in Chapter 3 of Principles of Program Analysis [13]. This algorithm consists of three steps:

1. Initialization of the variables. For each variable a suitable value is chosen.
2. Building a graph of the constraints. Each constraint becomes a node, if multiple constraints contain the same variable an edge is created between those constraints.
3. Iteration. All constraints are added to the list, and one by one these constraints are handled. If a constraint changes the value of a variable, all constraints referring that variable are added to the head of the list. This continues, until the list is empty.

We, however, skip the first step of this algorithm since we want to keep our annotations as generic as possible. Also, the only constraint that changes the lower bound of a variable is the &&-constraint.
In the third step of the *worklist* algorithm, we must define operations for each of our annotation constraints. We handle constraints in the following way:

- $p \equiv q$: We lookup the values of both variables. If one or none of the variables have a value, we create a substitution. The variable for which we add an substitution now has a value, thus we can possibly solve some of the constraints referring this variable. These constraints are added to the *worklist*.
- $p \sqsubseteq q$: We lookup the values of both variables. If none of the variables have a value, we do not have enough information to solve this constraint, so we skip it for now. If one or both of the variables have a value then we have four cases:
  - $p \equiv \omega$: q can never be 1, thus q gets value ω as well. Constraints referring q are added to the *worklist*.
  - $q \equiv 1$: p can never become ω, thus p gets value 1 as well. Constraints referring p are added to the *worklist*.
  - $w \equiv 1$: We have a violation, we set q to ω.
  - otherwise: We have two cases, $1 \sqsubseteq 1, \omega \sqsubseteq \omega$ and $1 \sqsubseteq q, p \sqsubseteq \omega$. In the first case, we can ignore the constraint. In the second case p and q are unconstrained, so this constraint will possibly be considered again. However, since p and q have not changed value, we can skip it for now.
- $p \equiv q \,\&\&\, r$: The variable is used more than once, thus the variables p, q, r get all value ω. If now, the value of one of the variables changes, the constraints referring that variable are added to the worklist.
- $p \equiv q \,||\, r$: The value for p is determined by the minimum of p and q. Thus if q or r is 1, then p is also 1. Only constraints referring p are added to the *worklist* if p changes value.

6.3.5 Simplifying

Using our *worklist* algorithm we can only determine the value of an annotation variable, not which constraints have become potentially irrelevant. So we introduce a simplifying pass, that reduces the constraint set by removing those constraints that have become obsolete.
For equality constraints we used substitution, so they are not relevant anymore, while the annotation variables in &&-constraints all have value: ω. Thus, both type of constraints can be removed.
An ||-constraint can be removed if the value of the annotation variable on the left hand side is either 1 or ω. In both cases, the right hand side cannot change value anymore.
Inequality constraints of the form $p \sqsubseteq w, 1 \sqsubseteq p$ can be be removed as well. In these cases the value of p is unconstrained.
After simplifying the constraint set, the set only contains inequality constraints, for which the *worklist* algorithm could not determine a value. These constraints can then be added to the type of a function.

6.3.6 Solving Order

Although we have defined for each type of constraint how to solve it and how we simply the constraints, we haven't described in which order this takes place.
We start with running binding group analysis, to determine in which order we must handle each function. Then we solve the global constraints. Finally, we apply for each function, the earlier described methods in the following order [1]:

[1] This algorithm is not the most efficient, since we run the worklist algorithm twice. The reason is that we add equalities for memory reuse. Substitution would prevent us from determining if a reuse itself is used shared.

1. Solving type constraints: All type constraints except instantiation constraints are removed. To apply generalization to Instantiation constraints, we must know over which annotations we must not annotate. The Mset gives this answer only partially. Intermediate let bindings also introduce annotations.

2. Run the *worklist* algorithm: We try to determine for as many annotation variables as we can their value.

3. Run simplifier: all equality constraints and &&-constraints can be removed. Some inequality constraints can be removed as well (§6.3.5).

4. Solve instantiation constraints: Now that all equality constraints are gone, we know which annotations must not be generalized and can thus be instantiated.

5. Run *worklist* algorithm a second time: Solving instantiation constraints introduce extra equality constraints, which we must solve. The resulting substitutions allow us to determine for more annotation variables their value, by looking at the constraints the simplifier could not remove.

6. Run simplifier again: Solving instantiation constraints, introduced extra equality constraints, which can be removed. Some inequality constraints can be removed as well.

After step 6 we are done solving the constraints for a function. No extra constraints have been introduced, so running the *worklist* algorithm again will not change the result.
The result is a set of substitutions and per function a set of constraints. This set of constraints only contains inequality constraints and possibly ||-constraints. These ||-constraints can be removed: none of the annotation variables have a value and are thus not constrained in their value.
We apply the substitutions to the type of every function definition and generalize.
For the inequality constraints we use defaulting: they become equality constraints. These equality constraints then become substitutions.

6.3.7 Example

In section we gave an example on what kind of constraints were generated for `reverse`. Solving these constraints using the described algorithm results in the following type for `reverse` and `rev`:

```
reverse :: ∀a p q r s. (List a^p)^1 ->^q (List a^r)^s
rev :: ∀a p q r s. (List a^p)^1 ->^ω ((List a^q)^r ->^1 (List a^q)^s)^1
```

As you can see both `reverse` and `rev` require the first argument to be unique. Also, `rev` requires the partial application to be unique as well.
In §7.2 we give more example programs, where we briefly explain the precision of our analysis.

6.4 Constraint Violation

In most programming languages, a compiler will refuse to compile a type incorrect program. By using a type checker or type inference algorithm, programs that perform operations on expressions that do not make sense on the underlying type are rejected. For uniqueness typing and especially for our implementation of uniqueness analysis, such as case also exists:

1. A variable with annotation ω is passed to a function argument with annotation 1. Take for example a shared variable passed to an in-place reversal of a list:

   ```
   let t = [1,2,3,4]
   in (t, reverse t)
   ```

   Here `t` is used twice in the result, while `reverse` requires `t` to be unique.

2. The variable we have marked to use for in-place updates is used multiple times itself, thus has annotation ω. Again we will use an example:

   ```
   dupReuse l@(x:xs) = (l, l@(x : xs))
   ```

   Here we use `l` twice. But `l` is also used to perform in-place updates.

The standard way is to report these errors back to the user so that the user can fix the problem. In our case, we do not want to do this, for two reasons:

- Since our analysis runs on top of Core instead of Haskell, reporting errors back to the user is hard or sometimes even impossible. Variables are renamed, possibly multiple times and let expressions are inlined or lifted as top-level functions. Even if we somehow could report these errors, they would probably be difficult to understand for the programmer.
- The purpose of our analysis is only to verify if the intentions of the user are safe to perform. We thus want our analysis to never fail. This means that the user should never see errors related to our analysis.

To solve this problem and how to deal with constraint violation, we will use function duplication combined with extra constraints.

6.4.1 Extra Constraints

To deal with case (1), we will do the same as Hage and Holdermans [18] also propose: duplication of functions. For every function that performs in-place updates, a fallback is created that does not perform in-place updates. Now if a shared variable is passed to a function that performs in-place updates, the fallback is called instead. Since our analysis is run after the **Lift** pass, only duplicating functions which perform in-place updates is not enough. Therefore, we will duplicate all functions that have 1 in the argument position. Thus if a function returns a unique variable, this function is not liable for duplication.
Unfortunately, the constraint solver only has information about the constraints, not where they are generated. Thus if a constraint is violated, we do not know which function call is involved and where. Fortunately, there is a relatively easy solution: we annotate every function call with an annotation variable. If this variable has value shared, then one of the arguments passed to the function is used shared, this the fallback function must be called.
To make things more clear, we will use `reverse` as example:

```
let t = reverse xs in t
-- becomes:
let t = reverse { u0 } xs in t
```

If `u0` has value ω, then the fallback function must be called.
Now to determine the value of this annotation variable we introduce two extra type of constraints:

- Argument, $(r \sim p \sqsubseteq q)$: Is an Inequality constraint but with an extra annotation variable r as argument. Now if we pass a shared variable $(p = \omega)$ to a function requiring a unique one $(q = 1)$, the extra annotation variable r gets the value ω. It indicates, that for this argument we should call the fallback.
- Function[2] $(s = \omega \in qs \ ? \ \omega : \emptyset)$: This constraint captures the annotation variable r of each argument in a list qs. If one of the elements of qs has the value ω then s also receives the value ω, indicating that for this function call we should call the fallback.

Now when we apply the substitutions, returned from the constraint solver, we not only apply them to function type definitions but also to expressions.

[2]Since we only create one fallback, we do not necessarily need this constraint. We can use s directly in the argument constraint. Also, a function constraint could be replace by Inequality constraints for every argument.

6.4.2 Shared Reuse

While duplication of functions deals with the first case where a constraints can be violated, it does not deal with the case where the variable itself is shared. Again, our solution is to use extra annotations. We will annotate the reuse identifier that is present in constructor application. If we take `Cons` constructor as example, it will look as follows:

```
(Cons @ l) { Int } x xs
  -- becomes:
(Cons @ l) { u0 } { Int } x xs
```

Here `l` is the variable used for memory use. In the the normal case, we first instantiate the type variable in `Cons` to `Int`, then we apply the integer i and the list xs. In the adjusted case, we first apply an annotation variable, then we proceed as in the normal case.
Now if an annotation variable on the constructor resolves to shared, thus the constraint is violated, we do the following two things: We ignore the constraint and mark it as solved, and when we apply substitutions we remove the reuse identifier from the constructor.

7

Results

7.1 Memory pressure

To measure the reduction of memory allocations that in-place updates promises, we have analyzed the memory usage of the following 5 programs: Reverse, fmap and rotation over a tree, and quicksort and insertion on integer lists. For the list based algorithms we use a sorted list containing 100 integers, for trees we use the following definition:

```
data Tree = Node Tree Int Tree | Leaf
```

The algorithms are run on trees with depth 11.
Every algorithm is run once, twice and ten times, where every run receives the input from the output of the previous run. This way every run is deterministic.
The results of every program can be seen in Table 7.1. The left table shows the memory allocations of the 5 programs without in-place updates, the right table shows the same algorithms but with in-place updates.

Program	1 run	2 runs	10 runs
Reverse	19.0	21.4	40.6
FTree	131.1	196.6	720.6
Quicksort	196.6	376.6	1816.6
Rotation	0.5	0.6	1.1
Insertion Sort	137.8	259.0	1228.6

(a) Without memory reuse

Program	1 run	2 runs	10 runs
Reverse	16.6	16.6	16.6
FTree	65.6	65.6	65.6
Quicksort	16.6	16.6	16.6
Rotation	0.5	0.5	0.5
Insertion Sort	19.0	21.4	40.6

(b) With memory reuse

Table 7.1: Memory allocations in kb

As you can see in the left table, (a), the memory allocations increase almost linearly with every run. One algorithm that especially allocates a lot of memory is *quicksort*, with almost 2 megabyte of memory after 10 runs. Without a garbage collector or some other way of releasing memory, these kind of programs will quickly run out of memory.
The right table, (b), shows that in-place updates have a huge reduction in memory allocations. It doesn't matter if we run an algorithm once, twice or ten times: no extra allocations are made.
One algorithm, *insertion sort*, still shows extra allocations, though much reduced. The reason for why this is happening has to do with the definition of insertion sort we have given in Haskell:

```
1  sort :: [Int] -> [Int]
2  sort xs = inserts xs []
3
4  inserts :: [Int] -> [Int] -> [Int]
5  inserts [] r = r
6  inserts (x:xs) r = inserts xs (insert x r)
7
8  insert :: Int -> [Int] -> [Int]
9  insert x [] = [x]
10 insert x l@(y:ys) = if x < y then x:(l@(y:ys)) else l@(y:(insert x ys))
```

As you can see on line 10, if the element x we insert is smaller than the current element y at the head of the list, we append x to the head list. In this case we have two constructor applications, thus we can only reuse memory for one of the constructor applications. Fortunately, this only happens once per `insert` call. Whereas `insert` is called at most $O(n^2)$ times, the then-branch is only reached $O(n)$ times. Thus the space complexity has become $O(n)$ instead of $O(n^2)$.
The results confirm the assumption that the space complexity of *insertion sort* has become linear. As you can see, the number of allocations of in-place *insertion sort* equals that of *reverse* in the normal case.

7.1.1 Random sorting

The 5 program so far, are rather artificial. You are not going to run a sorting algorithm on a data structure you know is sorted. So we have also run Quicksort and Insertion sort on an unsorted list with 100 distinct integers between 0 and 100. The results are presented in in Table 7.2.

Program	Normal	In-place
Quicksort	40.6	16.2
Insertion Sort	74.0	18.6

Table 7.2: Memory allocations in kb

Table 7.2 shows that the amount of memory allocations is much lower for both *Quicksort* and *Insertion sort* compared to the variant where all numbers were sorted, while the amount of allocations between in-place version and the artificial version is similar. Typically, we can see a reduction in allocations for quick and insertion sort of about 2 to 3 times.

7.1.2 Thunk & Function versus Constructor allocation

In §5.5 we mentioned that we measure constructor allocation separately from other forms of allocation. To measure the contribution of constructor allocation to the total memory usage, we have analyzed the same programs, but now for thunk and function allocations.

Program	1 run	2 runs	10 runs
Reverse	61.2	61.2	61.2
FTree	945.6	1210.0	3322.2
Quicksort	1048.8	2036.4	9937.1
Rotation	2.3	2.4	3.1
Insertion Sort	842.9	1624.7	7878.7

Table 7.3: Thunk & function allocations in kb

The amount of thunk and function allocations (Table 7.3) turn out to be an order of magnitude higher than constructor allocations. This shows that many improvements can be made to reduce thunk and function allocations as well. This can be for example be done, by improving the current strictness analysis [12] or by implementing the cardinality analysis [9]. The results of those analyses should then be exploited to generate an optimized program.

7.2 Uniqueness Analysis

To verify the correctness and precision our analysis, we have tested various programs. To show what our analysis is capable of we will demonstrate four scenarios: the identity function `id`, shared usage on `reverse`, Indirection on reverse by using higher order function `apply`, and finally global variable usage.

7.2.1 Identity

We start with the function `id`, which has the following definition:

```
id :: a -> a
id a = a
```

Uniqueness analysis gives this program the following type: `id` :: $\forall p\, q.$ $\mathtt{a}^{p}$ $\mathtt{->}^{q}$ $\mathtt{a}^{p}$. It shows that the output has the same usage as the input. This makes sense, since `id` does nothing with its parameter, except returning it.

7.2.2 Shared Usage

In §6.1.6 and §6.3.7 we gave `reverse` as example, to demonstrate what kind of constraints we gather and what the resulting type is. Now we will apply `reverse` to a shared list, and show that the duplicated version will be called.
We have tested the following program:

```
dupList :: (List Int, List Int)
dupList = let xs = Cons 2 (Cons 1 Nil) in (reverse xs, xs)

reverse :: List a -> List a
reverse xs = rev xs Nil

rev :: List a -> List a -> List a
rev Nil ys = ys
rev l@(Cons x xs) ys = rev xs l@(Cons x ys)
```

As you can see, the variable `xs` is used twice. Thus we expect the fallback to be called.
Running the analysis on this program, gives the following result:

```
dupList :: ((List Int^ω)^ω, (List Int^ω)^ω)
dupList = let xs = Cons 2 (Cons 1 Nil) in (reverse_nonmut xs, xs)

reverse :: ∀p q r s. (List a^p)^1 ->^q (List a^r)^s
reverse xs = rev xs Nil

reverse_nonmut :: ∀p q r. (List a^ω)^ω ->^p (List a^q)^r
reverse_nonmut xs = rev_nonmut xs Nil
```

```
rev :: ∀a p q r s. (List a^p)^1 ->^ω ((List a^q)^r ->^1 (List a^q)^s)^1
rev Nil ys = ys
rev l@(Cons x xs) ys = rev_nonmut xs l@(Cons x ys)

rev_nonmut :: ∀a p q r s. (List a^p)^ω ->^ω ((List a^q)^r ->^ω (List a^q)^s)^ω
rev_nonmut Nil ys = ys
rev_nonmut (Cons x xs) ys = rev_nonmut xs (Cons x ys)
```

Since the first argument of `reverse` and `reverse_nonmut` are required to be unique, two duplicate functions are created which do not have this requirement. Furthermore, in `dupList` the fallback is called, since `xs` is not used.

7.2.3 Higher Order

To see what effect higher order functions have on our analysis, we introduce indirection. Instead of calling `reverse` directly on a list, we call it indirectly through the `apply` function:

```
apply :: (a -> b) -> a -> b
apply f a = f a

high :: List Int
high = let xs = Cons 2 (Cons 1 Nil) in apply reverse xs
```

The definition and type of `reverse` are the same as in the previous example.
The type our analysis gives to `apply` is as follows:

```
apply :: ∀p q r s t u v. (a^p ->^q b^r)^s ->^t (a^p ->^u b^r)^v
apply f a = f a
```

Two observations can be made:

- The usage of the argument and return value equals that of the function passed to `apply` (p, r).
- The usage of the function passed to `apply` equals that to the usage if apply is partially applied (q)

Thus, the annotated type of `apply` should pose no problem to correctly update the list `xs` in-place.
The type we calculate for `high` is rather simple: `high::` $\forall p$. `List Int`^{p}

7.2.4 Global Variables

Global variables are a bit special. Since they can be exported, we do not know how many times they are used. To demonstrate this we use the following program:

```
global :: List Int
global = Cons 1 (Cons 2 Nil)

globalWithArg :: Int -> List Int
globalWithArg n = Cons 1 (Cons 2 Nil)

globalReverse :: (List Int, List Int)
globalReverse = (reverse global, reverse (globalWithArg 0))
```

All definitions in this example are exported, thus their usage is shared. For `global` this means, that we cannot use the in-place `reverse`. Thus we must call the fallback.
If `global` is not in the export list, the usage of `global` equals that to the usage in this program. In above program, we use `global` only once, thus `reverse` can then be called and not the fallback.

7.2.5 Constrained Types

In §2.2.3 we explained constrained types and why they are necessary. We also use constrained types in datatype definitions. Our implementation however, does not support constrained function types. Thus all constraints that are left after solving, are defaulted to substitutions. This has as consequence that our types are less precise than they can be. The question is, if these less precise types have effect on our analysis. That is, are there programs for which we select the fallback, where we have could have used the in-place version.
We have tested the following functions:

```
apply :: (a -> a) -> a -> a
apply f a =  f (f a)

apply1 :: (a -> b) -> a -> b -> (a, b)
apply1 f a b =  (f a, f b)

apply2 :: (a -> b) -> (b -> c) -> a -> c
apply2 f g a =  f (g a)

apply3 :: (a -> b) -> (c -> d) -> a -> c -> (b, d)
apply3 f g a b =  (f a, g b)
```

The annotated types for these functions are to large to describe here. However, in all cases we could use the functions without being affected by poisoning. This can have two reasons. First, our analysis infers incorrect annotated types. We hope that this is not the case but we cannot guarantee it. Second, the type of these functions do not need to be constrained, since they are precise enough.
We hope that our second hypothesis is indeed correct. Further research will be needed to verify if this is the case.

8

Conclusion

Our goal of this research and our main research question was to see if heap recycling analysis is effective in a functional language. To answer this question, we presented three sub-questions.
First, we asked if we can implement uniqueness analysis in Helium. Our implementation shows that this can be done. However, we could not implement all parts of the analysis. Notably, we do not support constrained types in function definitions.
Since Helium has two intermediate representations, our second sub-question regarded what a suitable source language is to implement the analysis. We determined that Core is the most suitable representation to do so.
With our last research sub-question, we asked what kind of performance increase we can expect from performing in-place updates. In §7.1 we showed for some simple programs, that the amount of memory allocations is strongly reduced. We also showed that in the current state of the Helium compiler, constructor allocations are only a fraction of the total allocations and that thunk & function allocations take a bigger part in the total memory consumption.
Having answered our three sub-questions, we can give an answer to our main research question. For the programs we have tested, heap recycling analysis has shown be effective in performing safe in-place updates.

8.1 Future Work

Research is never finished. The same applies to our research. So we describe various areas which are possibilities for further research or can be used as inspiration for a experimentation project.

Constrained types: The annotated function types we infer in our analysis our not constrained, since we default the inequality constraints to equality constraints after we have solved the constraints. To support such types, changes to the constraint solver are needed, mostly in the area of generalization and instantiation.
For the examples we have tested, constrained polyvariant types, did not seem necessary: we could always call the in-place version. This could mean that our analysis is incorrect in such cases or they are indeed not needed. We do know that they are necessary to be able to separate usage for nested data types and for higher-order functions. Take for example the type `[[Int]]`. Without constrained polyvariant types, every level of this type gets the same annotation (`[[Int`^{p}`]`^{p}`]`). Thus we can't distinguish inner usage from outer usage.

Datatype annotations: Placing annotation on data types is always problematic for type-and-effect based analyses. The design space is big and it is not always possible to support all data types. That said, the annotations we place on data types are rather conservative. Improvements can thus be made in this area.

Feedback: Besides static checking, we currently provide no feedback for the user to see if a particular in-place update can be performed or not. Although error messages from uniqueness analysis are probably not suitable to show for the end user, some simple warning messages could be rather usable. A trade-off will have to found in what information the user should get.

Function duplication: One unfortunate fact that our analysis should never fail, is that functions have to be duplicated. These duplicated functions are used as fallback if a particular in-place update is not safe to perform. This duplication has as consequence that in the worst case we have a two times size increase of the program.
A solution could be to find the common part of the normal function and the duplicated function and lift this part into a separate function.
Another solution could be, is to pass an allocator function as extra argument. If uniqueness analysis determines that a particular function call is safe to perform, this allocator function becomes a no-op. The downside of this approach, is that an extra function pointer has to be allocated.
Other solutions to this problem are possible as well. The one with the fewest downsides should be chosen.

Soundness: The implementation we have given has not been tested for soundness. It could be that we perform in-place updates which are not actually safe to perform. It would thus be desirable to have a soundness proof.

Integration: Implementing uniqueness analysis and performing in-place updates in a compiler such as Helium requires a lot of engineering. Various areas of the compiler have to be changed, sometimes introducing bugs but also discovering bugs. These discovered bugs sometimes are trivial to fix, but some are not.
Some passes, notably the Lift pass and desugaring pass interfere with uniqueness analysis. This has as consequence that our implementation does not always give the desired results.
Thus to provide better integration with in-place updates and uniqueness analysis, improving these passes will not only help our implementation but also increase the performance of the compiler itself

Bibliography

[1] Jerome Vouillon & Pierre Jouvelot. *Type and Effect Systems via Abstract Interpretation.* http://www.cri.ensmp.fr/classement/doc/A-273.pdf.

[2] Ben Gamari and Laura Dietz. "Alligator Collector: A Latency-Optimized Garbage Collector for Functional Programming Languages". In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*. ISMM 2020. London, UK: Association for Computing Machinery, 2020, pp. 87–99. ISBN: 9781450375665. DOI: 10.1145/3381898.3397214. URL: https://doi.org/10.1145/3381898.3397214.

[3] L. de Moura S. Ullrich. "Counting immutable beans - reference counting optimized for purely functional programming". In: *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*. IFL19. 2019.

[4] Ivo Gabe de Wolff. "Higher Ranked Region Inference For Compile-Time Garbage Collection". MA thesis. Unversity of Utrecht, 2019.

[5] Jean-Philippe Bernardy et al. "Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158093. URL: https://doi.org/10.1145/3158093.

[6] E. De Vries. *Linearity, Uniqueness, and Haskell.* http://edsko.net/2017/01/08/linearity-in-haskell. 2017.

[7] Yasunao Takano and Hideya Iwasaki. "Thunk Recycling for Lazy Functional Languages: Operational Semantics and Correctness". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC '15. Salamanca, Spain: Association for Computing Machinery, 2015, pp. 2079–2086. ISBN: 9781450331968. DOI: 10.1145/2695664.2695693. URL: https://doi.org/10.1145/2695664.2695693.

[8] Nicholas D. Matsakis and Felix S. Klock. "The Rust Language". In: *Ada Lett.* 34.3 (Oct. 2014), pp. 103–104. ISSN: 1094-3641. DOI: 10.1145/2692956.2663188. URL: https://doi.org/10.1145/2692956.2663188.

[9] Ilya Sergey, Dimitrios Vytiniotis, and Simon Peyton Jones. "Modular, Higher-Order Cardinality Analysis in Theory and Practice". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: Association for Computing Machinery, 2014, pp. 335–347. ISBN: 9781450325448. DOI: 10.1145/2535838.2535861. URL: https://doi.org/10.1145/2535838.2535861.

[10] H. Verstoep. "Counting Analyses". MA thesis. University of Utrecht, 2013.

[11] John van Groningen Rinus Plasmeijer Marko van Eekelen. *Clean Language Report 2.2.* https://clean.cs.ru.nl/download/doc/CleanLangRep.2.2.pdf. 2011.

[12] Stefan Holdermans and Jurriaan Hage. "Making "Stricterness" More Relevant". In: *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM '10. Madrid, Spain: Association for Computing Machinery, 2010, pp. 121–130. ISBN: 9781605587271. DOI: 10.1145/1706356.1706379. URL: https://doi.org/10.1145/1706356.1706379.

[13] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.

[14] H. Ferreiro et al. "Implementing memory reuse in the Utrecht Haskell compiler". In: (2009).

[15] Jurriaan Hage and Stefan Holdermans. "Heap Recycling for Lazy Languages". In: *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. PEPM '08. San Francisco, California, USA: Association for Computing Machinery, 2008, pp. 189–197. ISBN: 9781595939777. DOI: 10.1145/1328408.1328436. URL: https://doi.org/10.1145/1328408.1328436.

[16] Simon Marlow et al. "Parallel Generational-Copying Garbage Collection with a Block-Structured Heap". In: *Proceedings of the 7th International Symposium on Memory Management*. ISMM '08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 11–20. ISBN: 9781605581347. DOI: 10.1145/1375634.1375637. URL: https://doi.org/10.1145/1375634.1375637.

[17] Edsko Vries, Rinus Plasmeijer, and David M. Abrahamson. "Uniqueness Typing Simplified". In: *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 201–218. ISBN: 9783540853725. URL: https://doi.org/10.1007/978-3-540-85373-2_12.

[18] Jurriaan Hage, Stefan Holdermans, and Arie Middelkoop. "A Generic Usage Analysis with Subeffect Qualifiers". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ICFP '07. Freiburg, Germany: Association for Computing Machinery, 2007, pp. 235–246. ISBN: 9781595938152. DOI: 10.1145/1291151.1291189. URL: https://doi.org/10.1145/1291151.1291189.

[19] Edsko De Vries, Rinus Plasmeijer, and David M. Abrahamson. "Uniqueness Typing Redefined". In: *Proceedings of the 18th International Conference on Implementation and Application of Functional Languages*. IFL'06. Budapest, Hungary: Springer-Verlag, 2006, pp. 181–198. ISBN: 9783540741299.

[20] Tobias Gedell, Jörgen Gustavsson, and Josef Svenningsson. "Polymorphism, Subtyping, Whole Program Analysis and Accurate Data Types in Usage Analysis". In: *Proceedings of the 4th Asian Conference on Programming Languages and Systems*. APLAS'06. Sydney, Australia: Springer-Verlag, 2006, pp. 200–216. ISBN: 3540489371. DOI: 10.1007/11924661_13. URL: https://doi.org/10.1007/11924661_13.

[21] Dana Harrington. "Uniqueness Logic". In: *Theor. Comput. Sci.* 354.1 (Mar. 2006), pp. 24–41. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2005.11.006. URL: https://doi.org/10.1016/j.tcs.2005.11.006.

[22] A. Middelkoop. "Improved uniqueness typing for Haskell". MA thesis. University of Utrecht, 2006.

[23] D. Walker. "Substructural Type Systems". In: *AdvancedTopics in Types and Programming Languages*. MIT Press, 2005, pp. 3–43.

[24] S.P. Jones. *Haskell 98 language and libraries: the revised report*. https://www.haskell.org/definition/haskell98-report.pdf. 2003.

[25] Keith Wansbrough. "Simple Polymorphic Usage Analysis". PhD thesis. Cambridge University, 2002.

[26] Jörgen Gustavsson and Josef Svenningsson. "A Usage Analysis with Bounded Usage Polymorphism and Subtyping". In: *Selected Papers from the 12th International Workshop on Implementation of Functional Languages*. IFL '00. Berlin, Heidelberg: Springer-Verlag, 2000, pp. 140–157. ISBN: 3540419195.

[27] Keith Wansbrough and Simon Peyton Jones. "Simple Usage Polymorphism". In: *3rd ACM SIGPLAN Workshop on Types in Compilation*. 2000. URL: https://www.microsoft.com/en-us/research/publication/simple-usage-polymorphism/.

[28] Keith Wansbrough and Simon Peyton Jones. "Once upon a Polymorphic Type". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, pp. 15–28. ISBN: 1581130953. DOI: 10.1145/292540.292545. URL: https://doi.org/10.1145/292540.292545.

[29] Jörgen Gustavsson. "A Type Based Sharing Analysis for Update Avoidance and Optimisation". In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP '98. Baltimore, Maryland, USA: Association for Computing Machinery, 1998, pp. 39–50. ISBN: 1581130244. DOI: 10.1145/289423.289427. URL: https://doi.org/10.1145/289423.289427.

[30] David N. Turner, Philip Wadler, and Christian Mossin. "Once upon a Type". In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA '95. La Jolla, California, USA: Association for Computing Machinery, 1995, pp. 1–11. ISBN: 0897917197. DOI: 10.1145/224164.224168. URL: https://doi.org/10.1145/224164.224168.

[31] Simon Peyton Jones and Will Partain. "Measuring the effectiveness of a simple strictness analyser". In: *Functional Programming, Glasgow 1993: Proceedings of the 1993 Glasgow Workshop on Functional Programming, Ayr, Scotland, 5–7 July 1993*. London: Springer London, 1994, pp. 201–221. ISBN: 978-1-4471-3236-3. DOI: 10.1007/978-1-4471-3236-3_17. URL: https://doi.org/10.1007/978-1-4471-3236-3_17.

[32] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Professional, 1994. ISBN: 978-0201543308.

[33] Paul Hudak. "Conception, Evolution, and Application of Functional Programming Languages". In: *ACM Comput. Surv.* 21.3 (Sept. 1989), pp. 359–411. ISSN: 0360-0300. DOI: 10.1145/72551.72554. URL: https://doi.org/10.1145/72551.72554.

[34] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Global Value Numbers and Redundant Computations". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 12–27. ISBN: 0897912527. DOI: 10.1145/73560.73562. URL: https://doi.org/10.1145/73560.73562.

[35] David Ungar. "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm". In: *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. SDE 1. New York, NY, USA: Association for Computing Machinery, 1984, pp. 157–167. ISBN: 0897911318. DOI: 10.1145/800020.808261. URL: https://doi.org/10.1145/800020.808261.