Implementing Counting Analysis in UHC

Tibor Bremer (ICA-3470679)

MSc Thesis

March 1, 2018

Universiteit Utrecht

Center for Software Technology
Dept. of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands

Supervisors:
dr. A. Dijkstra
dr. J. Hage

Contents

List of Figures

1. Introduction

This thesis describes and implements an extension to counting analysis (defined be Verstoep in [?]). We also describe and implement a strictness optimization based on the results of the extended counting analysis.

Counting analysis is a analysis that combines four analyses that count how many times a certain expression is used and demanded. These analyses are absence, sharing, strictness and uniqueness analysis.

Having a single analysis that can be used for multiple optimizations is useful in that it is easier to maintain and it prevents cross analysis bugs. It also saves compile time as less analyses are run[1].

The goal of this thesis is to extend counting analysis to the whole of UHC core[2] and give an implementation of counting analysis in the Utrecht Haskell Compiler[3].

Only running the analysis will not result in any faster runtimes, so a strictness optimization is also presented that when run will transform the source code in such a way that the resulting binary runs faster.

1.1 Example

An example will demonstrate better what this thesis will achieve. We take the following code for the fibonacci function[4].

```
fib :: Int → Int
fib x | x < 1 = 0
fib 1 = 1
fib n = fib (n − 1) + fib (n − 2)
```

This is a very slow[5] version, but we can easily make it faster by making the recursive calls strict. This will result that not a whole chain of thunks is generated. The strict version of fibonacci looks as follows.

```
fib :: Int → Int
fib x | x < 1 = 0
fib 1 = 1
fib n =
  let
    f1 = fib (n − 1)
    f2 = fib (n − 2)
  in f1 + f2
```

Now for us it is easy to see that we are allowed to make the recursive calls strict, because we know they are needed to calculate the result, but for the compiler this is not so easy. That is what counting analysis for. It will determine what parts of the code can be made strict and the strictness optimization will than take this information and transform the code at the places it is allowed to make the code stricter.

This transformation results in a lowering of runtimes by more than 25, showing that using counting analysis to drive optimizations is a very useful tool for a compiler to have.

1.2 Outline

Before we delve into the extended counting analysis we need some information to understand counting analysis and the compiler (UHC) in which the analysis and accompanying strictness optimization are implemented. This is discussed in chapter **??**.

After the preliminaries the extended counting analysis is formally explained in chapter **??**. After this in chapter **??** the strictness optimization is discussed that will be used to measure the success of the analysis.

[1] Although the combined analysis runs slower than the individual analyses
[2] UHC core is the intermediate compiler language of the Utrecht Haskell Compiler
[3] This will be abbreviated to UHC in the rest of the thesis
[4] This a not a fast implementation of fibonacci. See section **??** for the full breakdown and results for this example.
[5] Even ignoring that it runs in quadratic time.

Before we discuss the results of the strictness optimization in chapter **??**, the implementation of counting analysis is given in chapter **??**.

Chapter **??** will discuss some related work and finally chapter **??** concludes and give some pointers to future work.

2. Preliminaries

Before diving into the depth of counting analysis some prior knowledge is required. First the Utrecht Haskell Compiler[1] is introduced[2]. After that in section **??** is explained what counting analysis is and what it does.

2.1 UHC

First an overview is given of what UHC is. After this high level overview a more in depth discussion is given of how the internals of UHC work. UHC is available from https://github.com/uhc/uhc.

2.1.1 Overview

UHC is a Haskell compiler developed by Atze Dijkstra as an experimentation platform. It is part of EHC[3] which is a series of compilers where each variant is build on top of the previous variant. The last variant of EHC is UHC. Each variant of EHC adds features. Fhe first version is the untyped lambda calculus. Version 3 adds types and results in the simply typed lambda calculus. Version 8 adds type inference to this. The module system is not available up to version 50. Version 99 is the most feature complete version. UHC is version 101 and is equal to version 99 with debug features removed. EHC is written completely in shuffle[4] Shuffle is a wrapper language around any other language which allows the source code for multiple variants to be present inside the same source file. UHC is written mainly as attribute Grammars[5]. Attribute grammars are perfectly suited for syntax directed computations over treelike structures. Counting analysis itself is a perfect example of this.

2.1.2 Internals

The compile pipeline outlined in this section is for the default grin based bytecode backend. There exists multiple backends for UHC in various stages of feature completeness including a Javascript and Java backend[6]. For full details see the UHC documentation[7]. Compilation happens in multiple phases using multiple intermediate languages. A graphical representation of the compilation process is give in figure **??**. The pipeline starts with Haskell. This is translated into Essential Haskell (EH) which is just desugared Haskell. On this language binding group analysis and type checking is performed. After this EH is transformed into Core[8]. During this translation all the type information is removed. type classes are translated to dictionaries (records) and type class constraints are translated in additional parameters. On this language so called core transformations are executed. These include the translation to A-normal form[9], the implementation of counting analysis outlined in this thesis and the optimizations the results of counting analysis allow. After all the core transformations are executed the code is translated from Core to Grin. Grin is a low level language on which some more transformations are executed[10]. Grin s then finally translated to bytecode.

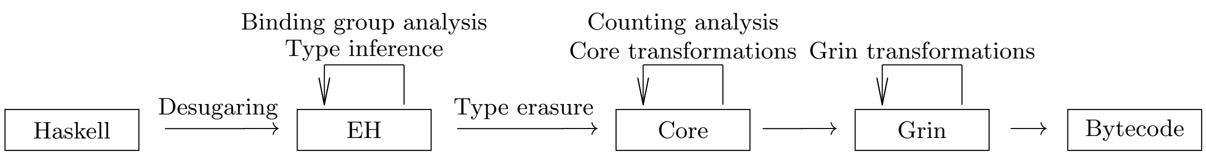

Figure 2.1: UHC compile pipeline

[1] After this abbreviated to UHC.
[2] Although knowing about UHC is not required to understand counting analysis it is a critical part of this thesis.
[3] EHC stands for Essential Haskell Compiler.
[4] for more information see http://foswiki.cs.uu.nl/foswiki/Ehc/Shuffle.
[5] For more information see http://foswiki.cs.uu.nl/foswiki/HUT/AttributeGrammarSystem.
[6] These two backends skip the Grin stage and generate their final code directly from Core.
[7] See http://foswiki.cs.uu.nl/foswiki/UHC/WebHome.
[8] Not to be confused with GHCs Core language which is a typed intermediate language.
[9] The right hand side of an application can only be a variable or a constant.
[10] Conveniently named Grin transformations.

2.2 Counting analysis

Counting analysis is, as the name already says, an analysis that counts. In this case the number of times an expression is being evaluated. Counting analysis is a combination of three or four analyses. It includes absence analysis, sharing analysis and strictness analysis. Whether or not uniqueness analysis is included depends on the subeffecting rule[11] being used. In this thesis uniqueness analysis will be excluded[12] as uniqueness analysis is not really useful if the user cannot supply type signatures[13].

First a short overview of the separate analyses is given followed by a high level overview of counting analysis. The full details of counting analysis can be found in section **??**.

All analyses annotate the types of expressions with some annotations. These differ for each analysis.

2.2.1 Absence analysis

Absence analysis tries to determine which parts of expressions are never going to be evaluated and can thus be safely replaced removed or replaced by undefined. A classic example is the constant function $const\ x\ y = x$. This functions does not use its second argument. It is thus safe to replace any expression of $const\ \mathbf{e_1}\ \mathbf{e_2}$ with $const\ \mathbf{e_1}\ \bot$[14].

Absence analysis has three main applications within functional languages:

Dead code removal This is particularly useful for generated code and for whole program analysis.

Unused parts of datatypes Nearly all projection functions don't use the whole data type.

Precision Enable the other analyses to be more precise. Especially the sharing analysis. See the last example of section **??**.

Lattice

Before we can give examples of the above three applications we need to define the annotations and the corresponding lattice. For absence analysis the annotations consist of $\mathbb{0}_A$ and ω_A. The lattice for these annotations is $\mathbb{0}_A \sqsubset \omega_A$. The annotation of $\mathbb{0}_A$ means it is not used while an annotation of ω_A means it is potentially used.

If a function argument is annotated as absent we still want to be able to pass in a non-absent value. E.g. we want **let** $x = 5$ **in** $const\ x\ x$ to be allowed. To allow this we use the following subeffecting rule:

$$\frac{\vdash \mathbf{e} : \tau^{\varphi'} \qquad \vdash \varphi \sqsubseteq \varphi'}{\vdash \mathbf{e} : \tau^{\varphi}}$$

Soundness of the subeffecting rule depends upon the optimizations the annotations allow. The optimization for absence analysis is to replace thunks and parameters annotated with $\mathbb{0}_A$ by $\bot$. Subeffecting allows that an argument annotated with $\mathbb{0}_A$ is bound by a value that is not replaced by $\bot$. E.g. the optimization would replace $const\ x\ y$ by $const\ x\ \bot$. This optimization is sound as long as the replaced thunk is not used. It does not matter how the thunk (y in this case) that is replaced is used. Hence the annotation on the replaced thunk can be anything. This is exactly what the subeffecting rule allows.

2.2.2 Dead code removal

There are two ways dead code can be removed:

1. Remove unused thunks and arguments completely
2. Replace unused thunks and arguments by $\bot$

Removing unused thunks completely is easy. For example take the following code:

$$\begin{aligned}&x =\\ &\quad \mathbf{let}\ y = 3\\ &\quad \mathbf{in}\ 5\end{aligned}$$

After running absence analysis we get the following types:

[11] See the subeffecting rules for the separate analyses.
[12] This will be done implicitly and the type rules (figure **??**) have holes for different instantiations of the subeffecting rule.
[13] Support for type signatures is future work.
[14] In practice this is seldom applicable as *const* is normally used partially applied instead of fully applied.

$$\begin{aligned} x &:: Int^{\omega_A} \\ y &:: Int^{\mathbb{0}_A} \end{aligned}$$

The $\mathbb{0}_A$ annotation on y indicates it is not used, which is also clear from the code. The code can be safely[15] transformed into $x = 5$.

For arguments it is very tricky to completely remove them. To see the problem we again take the *const* function as an example[16]:

$$\begin{aligned} &const :: a^{\omega_A} \overset{\omega_A}{\to} b^{\mathbb{0}_A} \overset{\omega_A}{\to} a^{\omega_A} \\ &const\ x\ y = x \end{aligned}$$

If the second argument would be removed the function would be the identity function. It can be very hard to make the code using the transformed function type correct. For example *map* (*const* 2) is valid, but *map* (*id* 2) is not type correct. It is completely unclear how removing the absent argument of *const* would be achieved in the caller code.

Another example of why removing absent arguments might not be the right thing to do can be seen using the following example:

$$\begin{aligned} &fail :: a^{\mathbb{0}_A} \overset{\omega_A}{\to} b^{\omega_A} \\ &fail\ x = \bot \end{aligned}$$

By removing the argument from *fail* it would be equal to $\bot$. However, without the argument removed, *fail* and undefined are not equal. This can be seen by the Haskell primitive *seq*. The code *seq fail* 3 will result in 3 while the code *seq* $\bot$ 3 will result in $\bot$. So removing the argument from *fail* would not be a safe transformation.

The removing unused thunks optimization is really useful for whole program analysis to reduce total code size, as probably most imported definitions are not used.

2.2.3 Data types

There are a lot of functions that only use parts of data structures. Examples are *length*, *fst*, *snd* and all record field functions.

A clear example of why data type absence analysis is desirable is removal of unused record fields of desugared class dictionaries. For example:

$$\begin{aligned} &\textbf{data}\ C\ a = \{ \\ &\quad f :: a \to a \to a, \\ &\quad g :: a \to a \to a\ \} \\ &func :: C\ a \to a \to a \to a \to a \\ &func\ d\ x\ y\ z = f'\ (f'\ x\ y)\ z \\ &\quad \textbf{where}\ f' = f\ d \end{aligned}$$

Instead of passing in a record which holds two functions, we want to pass in a record which has $\bot$ stored for g[17]. Especially for large class dictionaries where only a small part is used can this optimization decrease heap storage[18] needed.

2.3 Strictness analysis

Strictness analysis tries to determine if an expression is used at least once. This enables call by value semantics, resulting in far fewer thunks being created. For example:

$$fac\ n = \textbf{if}\ n \leqslant 1\ \textbf{then}\ 1\ \textbf{else}\ n * fac\ (n-1)$$

Without strictness analysis this will create a lot of thunks. Both for the $n-1$ computations as well as the recursive call. Using the fact that $-$ and $*$ are strict in both arguments we can transform the function in the much more efficient function[19]:

[15] In this case safe transformations mean transformations that do not alter the outcome of running the code.

[16] The annotations on the arrows are the annotations for the partial applications.

[17] We want to store $\bot$ instead of actually removing the field as we then have to change the data definition, all projection functions, etc... Basically the same reason why argument removal is not done, although in the case for records they can be avoided by generating a new data type holding only the used fields and calling functions that builds a record of the new type from the old type.

[18] Memory space.

[19] No *seq* is necessary for the $n-1$ because it will immediately be evaluated when evaluating the recursive call.

$$fac\ n = \textbf{if}\ n \leqslant 1\ \textbf{then}\ 1\ \textbf{else let}\ f = fac\ (n-1)\ \textbf{in}\ f\ `seq`\ n * f$$

This will only generate a single thunk for f which will immediately be evaluated[20,21].

2.3.1 Lattice

The annotations used for strictness analysis are $\mathbb{1}_S$ for strict[22] and ω_S for possibly lazy values[23]. The lattice for these annotations is $\mathbb{1}_S \sqsubset \omega_S$. If a function argument is annotated as strict we still want to be able to pass in a lazy value. To allow this we use the same subeffecting rule as for absence analysis.

The annotation on the value passed in does not really matter as only the annotations on the function parameters drive the optimizations (call by value and stack based argument passing), so the subeffecting rule is trivially sound.

2.4 Sharing analysis

Sharing analysis tries to determine whether an expression is used at most one time. This enables for example the following two optimizations:

let floating Float let bindings closer to the use site

non-updating thunks If a thunk is only used once then the computed value for the thunk does not need to be written back to the heap as it will never be inspected again

2.4.1 Lattice

The annotations used for this analysis are $\mathbb{1}_M$ for used at most once[24] and ω_M for possibly used multiple times[25]. The lattice for these annotations is $\mathbb{1}_M \sqsubset \omega_M$. If a function argument is annotated as used at most once we still want to be able to pass in a value that may be used multiple times. To allow this we use the same subeffecting rule as for absence and sharing analysis.

For the *non-updating thunk* optimization subeffecting means that it is allowed to pass in a thunk that is self-updating instead of a thunk that does not update. This is sound.

For the *let-floating* optimization subeffecting has no meaning. Let floating happens only locally and cannot happen through function arguments as the lambda to float the let into is not available at compile time.

2.4.2 Examples

An example of let floating optimizations is:

```
let f =
  let x = 1 + 2
  in
    λy → x + 5
in f
```

This code now creates two thunks. One for x and one for f. If the sharing analysis can find that f is used at most once then it is safe[26] to float the let inside the lambda like this:

```
let f = λy → let x = 1 + 2 in x + 5
in f
```

[20]GHC will not generate any thunks for *fac* as it will transform the type so that it works with unboxed integers. This will make it use only stack and no heap

[21]UHC supports **let**! which can also be used to remove the creation of the lazy thunk for f. Although here the heap is still used as opposed to GHCs optimized version.

[22]Strict in this case means that the value is guaranteed to be used during program execution

[23]The ω_S annotation for strictness has the same semantics as the ω_A annotation for absence analysis

[24]The $\mathbb{1}_S$ annotations of strictness and $\mathbb{1}_M$ of sharing analyses are not the same. By strictness it means used at least once, while for sharing it means at most once

[25]The ω_M annotation for sharing analysis is semantically equivalent to the ω_A annotation of absence analysis and the ω_S annotation of strictness analysis

[26]Safe means it does not result in more computation happening and the result stays the same.

Now the thunk for x is only created when f is applied, and not also when f is evaluated to WHNF. If f is shared[27] this transformation is of course not safe[28] as we lose sharing of x.

An example of the non-updating[29] thunk optimization[30] is the following example from [?]:

$$\begin{array}{l}\textbf{let } x = 1 + 2\\ \textbf{in}\\ \quad \textbf{let } y = (\lambda z \to z)\ x\\ \quad \textbf{in } y + y\end{array}$$

Running a sharing analysis will give the following annotations:

$$\begin{array}{l}\textbf{let}\\ \quad x :: \mathit{Int}^{\mathbb{1}_M}\\ \quad x = 1 + 2\\ \textbf{in}\\ \quad \textbf{let}\\ \quad\quad y :: \mathit{Int}^{\omega_M}\\ \quad\quad y = (\lambda z \to z)\ x\\ \quad \textbf{in } y + y\end{array}$$

The thunk for x does not need to be updating while the thunk for y needs to be updated.

The sharing analysis becomes more precise when coupled with the absence analysis[31]. For example:

$$\mathit{addLength}\ \mathit{xs} = \mathit{map}\ (+(\mathit{length}\ \mathit{xs}))\ \mathit{xs}$$

Without absence analysis the following types will be inferred:

$$\begin{array}{l}\mathit{length} :: [\,a^{\mathbb{1}_M}\,]^{\mathbb{1}_M} \xrightarrow{\omega_M} \mathit{Int}^{\omega_M}\\ \mathit{addLength} :: [\,a^{\omega_M}\,]^{\omega_M} \xrightarrow{\omega_M} [\,a^{\omega_M}\,]^{\omega_M}\end{array}$$

With absence analysis this can be improved to the following:

$$\begin{array}{l}\mathit{length} :: [\,a^{\mathbb{0}_M}\,]^{\mathbb{1}_M} \xrightarrow{\omega_M} \mathit{Int}^{\omega_M}\\ \mathit{addLength} :: [\,a^{\mathbb{1}_M}\,]^{\omega_M} \xrightarrow{\omega_M} [\,a^{\omega_M}\,]^{\omega_M}\end{array}$$

Now we can see that *addLength* only uses the elements of the list only once, while previously we got an annotation indicating it could be used multiple times[32].

2.5 Uniqueness analysis

Uniqueness analysis tries to determine whether certain annotated expressions are used at most once. This is a verifying analysis and not an optimizing analysis like the other three. This means that unlike the other three analyses this analysis can fail[33]. For the verifying part to be useful type annotations need to be given.

The main goal is to reject programs that use unique arguments multiple times. Even though the analysis is verifying there are optimizations it can enable. The heap recycling optimization as described in [?] is such an optimization.

2.5.1 Lattice

The annotations used for this analysis are $\mathbb{1}_U$ for unique values and ω_U for possibly non-unique values. The lattice for these annotations is $\mathbb{1}_U \sqsubset \omega_U$. If a function argument is annotated as not unique we still want to be able to pass in a value that is annotated as unique. To allow this we use the following subeffecting rule:

$$\frac{\vdash \mathbf{e} : \tau^{\varphi'} \qquad \vdash \varphi' \sqsubseteq \varphi}{\vdash \mathbf{e} : \tau^{\varphi}}$$

[27] I.e. used more than once.
[28] This results in duplicated thunks on the heap for x.
[29] Also called update avoiding.
[30] If a thunk is ever only needed once we can save time by not writing the result of evaluating the thunk back to trunk.
[31] The lattice is extended in the obvious way with $\mathbb{0}_M$. It is used in the same way as $\mathbb{0}_A$ during absence analysis.
[32] One use from applying *map* and one use from applying *length*. With absence analysis it does not get the use from applying *length*.
[33] I.e it rejects the programs for which the analysis fails.

This subeffecting rule is opposite to the subeffecting rule of absence, strictness and sharing analysis. The reason it is opposite is that the annotations of absence, strictness and sharing analysis indicate internal properties of the annotated symbols. The annotations for uniqueness analysis on the other hand indicate external guarantees that need to be met. In other words, absence, strictness and sharing analysis annotate how values *are* used, while uniqueness analysis annotate how values *should be* used.

2.5.2 Example

The following function requires its second argument to be unique:

$$\mathit{writeFile} :: \mathit{String}^{\omega_U} \xrightarrow{\omega_U} \mathit{File}^{\mathbb{1}_U} \xrightarrow{\omega_U} \mathit{File}^{\mathbb{1}_U}$$

For example the following should be rejected:

$$\lambda f \rightarrow (\mathit{writeFile}\ \texttt{"1"}\ f, \mathit{writeFile}\ \texttt{"2"}\ f)$$

While this should not:

$$\lambda f \rightarrow \mathit{writeFile}\ \texttt{"2"}\ (\mathit{writeFile}\ \texttt{"1"}\ f)$$

2.6 Combining the analyses : counting analysis

By combining the analyses to a single unifying analysis (the counting analysis defined in [?]) we can run the analysis once and use the resulting annotations to enable all the different optimizations for the previously defined optimizations[34].

2.6.1 Lattice

The annotations now consist of sets of values. We use the values 0, 1 and ∞ to represent not used, used once and used multiple times respectively. The lattice for this is the superset relation.

Depending on whether or not uniqueness analysis is included in the combination subeffecting is enabled or disabled[35], as the subeffecting rule for uniqueness is opposite to the subeffecting rules for the other three analyses.

We can represent all annotations from the four separate analyses as annotations for counting analysis:

Absence $\mathbb{0}_A$ and ω_A are now represented as $\{0\}$ and $\{0, 1, \infty\}$.

Sharing (and uniqueness) $\mathbb{1}_M$ $(\mathbb{1}_U)$ and ω_M (ω_U) are now represented as $\{0, 1\}$ and $\{0, 1, \infty\}$.

Strictness $\mathbb{1}_S$ and ω_S are now represented as $\{1, \infty\}$ and $\{0, 1, \infty\}$.

The new representation of using sets can express more properties (like used exactly once) than the old separate lattices.

An example showing that this lattice is more precise is the following:

$$\begin{array}{l} f :: \mathit{Bool} \rightarrow (a \rightarrow a) \rightarrow a \rightarrow a \\ f\ b\ g\ x = \textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ g\ (g\ x) \end{array}$$

This gets the very precise type[36]:

$$f :: \mathit{Bool}^{\beta_1} \xrightarrow{\beta_1} (a^{\beta_2} \xrightarrow{\top} a^{\beta_2})^{\{0,\infty\}} \xrightarrow{\top} a^{\beta_2} \xrightarrow{\top} a^{\beta_2}$$

The function parameter is either not used or used multiple times. None of the separate analyses can infer with this level of precision.

[34] A more complete and formal definition is given in section ??

[35] Enabled in this sense means using the subeffecting rule for sharing, strictness and absence analysis. Disabled means using equality as the subeffecting operator

[36] The $\top$ annotation is equal to $\{0, 1, \infty\}$.

2.6.2 Optimization problems

If care is not taken when implementing optimizations to keep the annotations sound, runtime problems can arise. For example assume we have the following:

$$\begin{array}{l}\textbf{let}\\ \quad x =^{1,1} \mathbf{e}\\ \quad f :: a^{1,1} \to b\\ \quad f\ x = \_\\ \textbf{in}\ f\ x\end{array}$$

If we transform this using the strictness optimization of call-by-value we get:

$$\begin{array}{l}\textbf{let} \cdots\\ \textbf{in}\ x \text{ ‘}seq\text{‘ } f\ x\end{array}$$

If we also now do the non-updating thunk (sharing) optimization we make x a non-updating thunk which results in x being evaluated twice: once in *seq* and once wherever f uses it.

This means that first strictness optimizations and then sharing optimizations have to be performed. And the annotations have to be updated to reflect the inserted *seq*’s, either by adding a demand whenever a *seq* is introduced or by rerunning the analysis after the strictness optimizations.

2.7 Demand driven analysis

Verstoep [?] not only gives a single combined analysis, it also extends the analysis to do demand driven analysis.

This demand is necessary to be precise for the Haskell *seq* primitive[37] as *seq* only demands (evaluates) it’s first argument but does not use it. Sergey [?] also introduced demand in the sharing analysis. Demand is there only present for functions and not for all types as in [?]. Demand in this case has to do with needing the value in WHNF. And only for functions is there a difference between using the value and evaluating it to WHNF. A function demanded but not used does not use its arguments. If demand would not be present at the annotation level any evaluation of the function to WHNF would contribute to the use annotation of its arguments as well.

[37] *seq* can also be represented by strict application or **let**!

3. Counting Analysis

3.1 Introduction

The analysis described here is based upon work by Verstoep [?] and extended to work for UHC Core. Also the algorithm is changed to allow a modular analysis. The changes made for UHC Core can be seen in section **??** and figures **??**, where there are additional constructs present to deal with constants and *FFI*, and the *seq* primitive is replaced by **let!**[1], and in figure **??**, where there is type application present to deal with type variables that have kind $* \to *$[2]. The changes made to allow modular analysis can be found in section **??** and figure **??**, where there are additional environments present to deal with imported and exported symbols[3].

3.2 Definitions

Before the analysis can be explained there are first some definitions needed. The definitions and notation given here is copied from [?] where possible and extended with new constructs where necessary.

The expressions in the subset[4] of UHC Core used here consist of variables x, constants $\mathbf{c}$ (integers and characters), abstraction and application, mutually recursive lets and single binding let! and fully applied datatype constructors[5] and case expressions. Also an FFI binding is allowed. Both abstraction and application is only allowed in A-normal form[6]. The same holds for let bindings. Binding to a pattern is not allowed[7]. The syntax for expressions is given in figure **??**.

$$
\begin{array}{lcl}
\mathbf{e} & ::= & v \mid \lambda x \to \mathbf{e} \mid \mathbf{e}\ v \\
 & \mid & \mathbf{let}\ \overline{x_i = \mathbf{e_i}}\ \mathbf{in}\ \mathbf{e} \mid \mathbf{let}\,!\ x = \mathbf{e_1}\ \mathbf{in}\ \mathbf{e_2} \\
 & \mid & K\ \overline{v_i} \mid (v_1, v_2, .., v_n) \mid \mathbf{case}\ x\ \mathbf{of}\ \overline{p_i \to \mathbf{e_i}} \mid \mathit{FFI}\ n\ \tau^{Core} \\
v & ::= & x \mid \mathbf{c} \\
p & ::= & K\ \overline{x_i} \mid (x_1, x_2, .., x_n) \mid \mathbf{c}
\end{array}
$$

Figure 3.1: Expressions

The lattice of annotations[8] φ is $\mathcal{P}\ (\{0, 1, \infty\})$, with set union being the join for the lattice. This means that the smaller the set the more precise the annotation is. The values represent how many times something is used: 0 means not used, 1 means used once and ∞ means used more than once. The meaning of the annotations is the same as the combination of the values inside the sets. For example $\{1\}$ means it is used exactly once, and $\{1, \infty\}$ means it is used at least once. For the basic annotations that consist of the singleton sets and the annotation that holds no information (the set of all annotation primitives) have a synonym defined for easy reading. These synonyms are given in figure **??**.

The syntax for annotations φ is given in figure **??**. An annotation is either an annotation variable β or an annotation value ϖ. An annotation value is a set of annotation primitives π. For usage annotations the symbol ν is used while for demand annotations the symbol δ is used. Even though ν and δ are both

[1]This is the strict version of the **let**. It will not create a lazy thunk. Instead it will evaluate the thunk and store the result on the heap. The **let!** construct is more powerful than *seq* as every *seq* $\mathbf{e_1}$ $\mathbf{e_2}$ can be rewritten into an equivalent $\mathbf{let}\,!\ x = \mathbf{e_1}\ \mathbf{in}\ \mathbf{e_2}$. The expression $\mathbf{e_1}$ will be evaluated to WHNF (weak head normal form) before it is bound to x and before evaluation of $\mathbf{e_2}$ happens. A **let!** can only be rewritten as a *seq* if the bound variable x is not used inside the body $\mathbf{e_2}$. If the bound variable is used in the body the **let!** can be rewritten to a **let** and a *seq* ($\mathbf{let}\ x = \mathbf{e_1}\ \mathbf{in}\ \mathit{seq}\ x\ \mathbf{e_2}$) but this has a slightly different runtime semantics as it allocates a lazy thunk for x which will then immediately be forced. The **let!** does not allocate the lazy thunk.

[2]For example Monad Transformers all have a parameter for the inner Monad that has kind $* \to *$.

[3]The Var and App rules have multiple variation dealing with imported and local symbols. The Let rule has an antecedent to change some annotations on the bindings of exported symbols.

[4]Although not everything of UHC Core is supported all the constructs that are used in the standard libraries is supported.

[5]It is possible to use partially applied data constructors in Haskell. Inside the compiler a wrapper lambda is created and all constructor calls use this lambda instead of directly using the constructor. This makes sure that in UHC Core the constructors are always fully applied.

[6] This last restriction is not enforced by UHC Core itself, but the analysis described here needs it. There exists a core transformation that brings UHC Core into A-normal form.

[7]At the surface language it is allowed but not a Core level. It is completely gone after desugaring.

[8]See also section **??**

synonyms for φ the meaning inside the type system is different. The ν annotation is for how often an expression is used, while the δ annotation is for how often the expression is demanded to be in WHNF[9]. For constants, FFI and constructors the use and demand annotations coincide[10] and have the same meaning[11]. For abstractions usage and demand mean different things. Usage annotations talk about how often the body of the function is used, while demand annotations talk about how often the function is demanded to be in WHNF. So demand talks about how often the function is evaluated to function form while usage talks about how often the function is applied. The difference between usage and demand is only observable using the primitive **let!**[12,13]. Without **let!** there would have been no need to differentiate between usage and demand annotations as they would always coincide. This is because without **let!** the only way to demand an expression is by using it.

$$\begin{array}{rcl}
\pi & ::= & 0 \mid 1 \mid \infty \\
\varpi & ::= & \emptyset \mid \{\pi\} \mid \varpi_1 \cup \varpi_2 \\
\varphi & ::= & \beta \mid \varpi
\end{array}$$

Figure 3.2: Annotations

$$\begin{array}{rcl}
\bot & ::= & \emptyset \\
\mathbb{0} & ::= & \{0\} \\
\mathbb{1} & ::= & \{1\} \\
\omega & ::= & \{\infty\} \\
\top & ::= & \{0,1,\infty\} \\
\mathcal{P}\,(\varpi) & ::= & \text{powerset of } \varpi \\
\nu & ::= & \varphi \\
\delta & ::= & \varphi
\end{array}$$

Figure 3.3: Annotation synonyms

Types can be type variables α, datatypes, type application[14], a function or a n-tuple of types. The n-tuple of types is used as the type for n-tuples[15]. Annotated datatypes are explained in more detail in section **??**. Function arguments always have both usage and demand annotations as the argument is available to the body as a variable. The value produced by a function only has an usage annotation attached[16]. This usage can mean two things depending on whether the produced value is a function or not. If the produced value is a function the usage annotation is for the usage of the partial application. If it is not a function it is for the use of the result. In neither situations is there a need for a demand annotation to be attached. The left hand side of a type application can only be a variable or another type application. In all other cases the application can be simplified directly and the result inlined.

Type schemes are used for polymorphic and polyvariant types[17]. They quantify over both type variables and annotation variables. The types are given in figure **??**. In the figure there are also type scheme variables γ. These variables are used during type inference for unknown type schemes and are completely removed from the types for expressions during constraint solving. Most of the time it is not allowed to freely instantiate all the annotation variables inside a type scheme. For example, it would be wrong to instantiate the usage of the elements of the list by ω and the spine by $\mathbb{0}$ as the spine is used as least as often as the elements of the list. This type of dependency is captured by constraints and stored inside

[9]Weak head normal form.

[10]Theoretically speaking this is the case. In the implementation however we don't do this and the usage of everything except for functions and variables are set to $\mathbb{0}$. This is because we only add usage when we apply a function. In all other cases no use is added and we end up with a $\mathbb{0}$ annotation in the end. Any optimizations are only interested in the usage of functions. For all other types they only look at the demand.

[11]This is because using a constant or constructor evaluates to WHNF and nothing more. FFI is already in WHNF.

[12]**let!** evaluates its binding to WHNF before evaluating the body of the **let**.

[13]Or equivalently the *seq* primitive. In UHC core this primitive is defined using **let!**.

[14]This is necessary for partial type application. Without partial type application, type application is never present.

[15]This is equivalent with the data type version of tuples ($\mathit{Tup}^n\ [\nu_1, \cdots, \nu_n, \delta_1, \cdots \delta_n]\ [\tau_1, \cdots \tau_n]$). Having a separate representation simplifies the type rule in the case of tuples.

[16]The reason that no demand annotation is attached is that it is not useful for a demand other than $\mathbb{1}$ to be present there. A demand of $\mathbb{0}$ would mean we can attach $\mathbb{0}$ to the argument as they will never be used in that case. Demanding a result multiple times can never result in more demand on the argument. Multiple usage of the result can of course lead to more demand on the argument so the usage annotation is present.

[17]A type scheme can be both polymorphic and polyvariant at the same time.

the type scheme. Constraints will be explained later in this section. Usage and demand annotations can be attached to types as well as type schemes.

$$
\begin{array}{rcl}
\tau & ::= & \alpha \mid T\,\overline{\varphi_l}\,\overline{\tau_k} \mid \tau_1\,\tau_2 \mid \tau_1{}^{\nu_1,\delta_1} \to \tau_2{}^{\nu_2} \mid (\tau_1{}^{\nu_1,\delta_1}, \tau_2{}^{\nu_2,\delta_2}, \cdots, \tau_n{}^{\nu_n,\delta_n}) \\
\sigma & ::= & \gamma \mid \forall\,\overline{\alpha}\,\overline{\beta}\,.\;C \Rightarrow \tau \\
\mu^{\nu} & ::= & \text{Attach an usage annotation } \nu \text{ to } \mu\,(\text{where } \mu \in \{\tau, \sigma\}) \\
\mu^{\nu,\delta} & ::= & \text{Attach both an usage annotation } \nu \text{ and a demand annotation } \delta \text{ to } \mu\,(\text{where } \mu \in \{\tau, \sigma\})
\end{array}
$$

Figure 3.4: Types

An environment is simply a mapping of type variables to usage and demand annotated type schemes. The definition is given in figure ??. With *keys* (Γ) the variables are extracted from the environment.

$$
\Gamma \quad ::= \quad \epsilon \mid \Gamma, x : \sigma^{\nu,\delta}
$$

$$
\begin{array}{rcl}
\mathit{keys}\ (\epsilon) & ::= & \emptyset \\
\mathit{keys}\ (\Gamma, x : \sigma^{\nu,\delta}) & ::= & \mathit{keys}\ (\Gamma) \cup \{x\}
\end{array}
$$

Figure 3.5: Environments

To support polyvariance in a precise way, constraints on the instantiation of the annotation variables are needed. For example, assume we have a polyvariant list that has one annotation for the elements and one annotation for the spine[18]. If both annotations would be polyvariant annotation variables, without constraints a possible instantiation would be that the elements will be used multiple times and the spine will not be used. This is, like we said before, of course not a valid instantiation[19]. So without constraints we would be forced to give the spine a $\top$ annotation, which is not very precise. Another example is that without constraints the demand annotations belonging to quantified usage annotations[20] would need to be set to $\top$. This is because it is invalid if the demand is ever instantiated to less use than the usage annotation. For all these dependencies between annotation variables it is necessary for constraints to be present that define valid instantiations. Also the counting analysis algorithm is a two stage algorithm that first collects constraints and then solves them. In figure ?? the language of constraints are given. For annotations there are five different constraints: equality $(\equiv)$, sum $(\oplus)$, union $(\sqcup)$, product $(\cdot)$ and conditional[21] $(\triangleright)$. Sum constraints express multiple usages of the same variable. Union constraints express the usage combination of multiple independent branches. Times constraints express the use of free variables inside function bodies. The usage of these free variables depends upon how many times the function is used. Conditional constraints express the fact that some let bindings are only used inside other let bindings and not inside the body of the let. So the usage of those bindings depends upon the usage of the bindings that use them. For types and type schemes only equality constraints are needed[22]. Besides these constraints we also have instantiation and generalization constraints to support let bound polymorphic and polyvariant types.

$$
\begin{array}{rcl}
C & ::= & \varphi \equiv \varphi_1 \oplus \varphi_2 \mid \varphi \equiv \varphi_1 \sqcup \varphi_2 \mid \varphi \equiv \varphi_1 \cdot \varphi_2 \mid \varphi \equiv \varphi_1 \triangleright \varphi_2 \mid \varphi_1 \equiv \varphi_2 \\
 & \mid & \tau_1 \equiv \tau_2 \mid \sigma_1 \equiv \sigma_2 \mid \mathit{inst}\ (\sigma) \equiv \tau \mid \mathit{gen}\ (\tau^{\nu_1,\delta_1}, C, \Gamma) \equiv \sigma^{\nu_2,\delta_2} \\
 & \mid & C_1 \cup C_2 \mid \emptyset
\end{array}
$$

Figure 3.6: Constraints

[18]For simplicity assume the usage and demand annotations are the same, so we can deal with only two annotation variables instead of 4.

[19]No actual constraint for this is generated. It is however implicitly captured by the generated constraints as it is impossible to access the elements without first pattern matching on the spine. So constraints for the demand on the spine will always be generated when the elements are accessed. It is of course possible that the spine is demanded once while the elements are demanded multiple times. However it will not be possible for the spine to be demanded zero times while the elements are accessed.

[20]Usage and demand annotations are always paired. A demand annotation cannot occur without an usage annotation. An usage annotation can appear without a demand annotation, for example in the result type of a function.

[21]This is explained in more detail in section ??.

[22]This is explained in more detail in section ??.

$$\mu_1 \sqsubseteq \mu_2 \quad ::= \quad \mu_1 \sqcup \mu_2 \equiv \mu_2$$

Figure 3.7: Subeffecting

For subeffecting a constraint synonym $\sqsubseteq$ is used for readability. The definition is given in figure **??**.

To illustrate the type schemes, constraints and annotations an example[23] is given in figure **??**. The function is given in both idiomatic Haskell and the desugared A-normal form required for the analysis. As we do not know anything about how g is used[24] the result pessimistically gets an usage annotation of $\top$ to indicate this. The second argument gets an annotation of $\mathbb{1}$[25] to indicate that it is used exactly once in the body of the function. The annotations on the first argument and the first partial application are variables. We cannot give a concrete annotation to the partial application as again it is unknown[26] how often it is used, and as we want to have a precise a type as is possible we do not want it to be set to $\top$[27]. The usage of the first argument is determined by how many times the first partial application is used. That is why the usage of the result of the first partial application is equal to the usage and demand of the first argument.

$$\begin{aligned}
&g, g' :: \forall \emptyset\ \{\nu_1\}\ .\ \cdot\ \mathit{Int}\ \nu_1, \nu_1 \rightarrow (\mathit{Int}^{\mathbb{1},\mathbb{1}} \rightarrow \mathit{Int}^{\top})^{\nu_1} \\
&g\ x\ y = x + y - 1 \\
&g' = \lambda x \rightarrow \lambda y \rightarrow \\
&\quad \textbf{let}\ z = (+)\ x\ y \\
&\quad \textbf{in}\ (-)\ z\ 1
\end{aligned}$$

Figure 3.8: Example

3.3 Counting analysis

Now that most formal definitions are out the way, counting analysis can finally be defined. We still need one final part of the puzzle: the annotation operators, and the variants of those that result in constraints. These operators will be used throughout this section for both the constraint generation as well as the constraint solving. These operators can be found in section **??** and match the different constraints in figure **??**.

The analysis is performed in two stages:

1. Constraint generation. This can be found in section **??**

2. Constraint solving. This can be found in section **??**

During constraint generation no annotated types are actually computed. Only constraints that these types should satisfy are generated. These constraints are then fed into the constraint solver to produce the annotated types for all the expressions.

3.3.1 Annotation operations

There are four different kind of operators used in the constraints:

$\oplus$ This operator is used to combine the usage annotations of multiple usages of a variable. E.g. in $x + x$. Here the usage of the first use of x needs to be added to the usage of the second use of x resulting in the fact that x is used multiple times.

$\sqcup$ This operator is used to combine usages of multiple branches. E.g. in **if** b **then** x **else** $x + x$. Here the resulting usage of x is that it is either used once or used multiple times.

[23] The example is taken from [**?**].
[24] E.g. This is the case if g is exported.
[25] As neither argument is a function, the usage and demand annotation coincide.
[26] The code using this function does know for each partial application how often it is used, but the definition site is not allowed to assume anything.
[27] As the code using the partial application knows the usage we would lose information by forcing it to $\top$.

$\cdot$ This operator is used for function application. The usage of the free variables inside a lambda are dependent upon how many times the lambda is applied. The usage of the free variables is multiplied by the usage of the lambda.

$\triangleright$ This operator is used for let bindings where certain let bindings are only used inside other bindings. The usage of that binding then depends upon whether the other let binding is used.

The computational definition of these operators is given in figure **??**. The operator $+$ is just the normal plus for integers with the fact that any number larger than one is mapped to ∞[28].

The given definition for $(\cdot)$ may not be very intuitive. The computational meaning[29] is: for every element m in ϖ_1, for all combinations of m elements[30] from ϖ_2, take the sum of the m elements. Example: suppose a function (which is applied twice) uses a value at most once[31], then that value is used: $\{\infty\} \cdot \{0,1\} = \{0+0, 0+1, 1+0, 1+1\} = \{0,1,\infty\}$ times. This aligns with the declarative definition that each repeated use of a function may use the free variables in a different way. When the first argument is $\mathbb{0}$ then the result will always be $\mathbb{0}$ independently of what the second argument is[32].

The operator $(\triangleright)$ can do three things depending upon whether the first argument is $\mathbb{0}$, contains 0 or does not contain 0. If it is $\mathbb{0}$ the result is always $\mathbb{0}$, if it contains 0 then it result is the union of $\mathbb{0}$ and the second argument. If it does not contain 0 then the result is equal to the second argument. This aligns with the intuition that when a binding which uses another binding is not demanded then this does not lead to an additional demand on the other binding. This means that the usages of bindings inside other bindings is only taken into account when those other bindings are actually used. This means that unused bindings do not influence the demand on other bindings.

$$\begin{array}{rcl}
\varpi_1 \oplus \varpi_2 & ::= & \{m + n \mid m \in \varpi_1, n \in \varpi_2\} \\
\varpi_1 \sqcup \varpi_2 & ::= & \varpi_1 \cup \varpi_2 \\
\varpi_1 \cdot \varpi_2 & ::= & \{\sum_{i=1}^{min\ (m,2)} n_i \mid m \in \varpi_1, \forall\, i\, .\ n_i \in \varpi_2\} \\
\varpi_1 \triangleright \varpi_2 & ::= & \bigcup_{m \in \varpi_1} (m \equiv 0\ ?\ \mathbb{0} : \varpi_2)
\end{array}$$

Figure 3.9: Annotation value operators

The fact that these operators are only well defined for annotation values and not for annotation variables means that the constraint generator cannot use these computational form of the operators, as most annotations will be variables and not values, and will only generate the constraints for it. The constraint solver will actually use these definitions to solve the appropriate constraints.

Constraint generation makes use of abstract versions of these operators that generate the appropriate constraint for that operator[33]. These abstract versions are given in figure **??** for the base cases and figure **??** for the lifted versions[34]. For the types (type schemes) with annotations attached the lifted version just breaks down the components and uses the operations for types (type schemes) and annotations. For environments the operations are just point wise lifted. Only the rules for $(\oplus)$ and $(\cdot)$ are given. The rules for($\sqcup$) are similar to the rules for $(\oplus)$. The same holds for $(\triangleright)$ and $(\cdot)$.

The cases for types and type schemes in figure **??** might look a bit weird. The reason only equality constraints are generated has to do with the fact that the counting analysis described here only uses subeffecting and not subtyping. In other words only the top level annotations, which are handled in the lifted versions of the operators, are allowed to be different and the rest of the type and annotations need to be exactly the same. Poisoning might occur here[35].

Constraint generation makes use of the following syntax $A \rightsquigarrow B$ meaning that B is generated by A. Most of the time that what is generated will be constraints. For example in figures **??** and **??** the left hand side is an abstract operator while at the right hand side the generated constraints are given.

[28] The same holds for the definition of $\sum$ in $(\cdot)$.

[29] ∞ is mapped to 2.

[30] An element from ϖ_2 may be selected multiple times.

[31] E.g. $\lambda x \to$ **if** $x \equiv 0$ **then** y **else** x. The y is a free variable in scope for which the usage is calculated using $(\cdot)$.

[32] Instead of $\emptyset$.

[33] Each operator has an accompanying constraint for annotations. Without these constraints the constraint generator would become stuck if it encountered a variable instead of concrete values.

[34] Lifted to environments and types and type schemes with annotations attached.

[35] It is future work to determine how much poisoning is actually happening as for polyvariant type schemes equality only means that the quantified variables in both type schemes have the same name. This case does not yield poisoning. For instantiated type schemes this equality is only used when they are actually the same instantiation. In this case they should be equal. For type equalities the likely place for poisoning to be introduced is when using datatypes and then the poisoning is more likely to come from annotation approximations in the annotated datatypes.

$$\frac{}{\varphi_1 \oplus \varphi_2 = \varphi_3 \rightsquigarrow \{\varphi_3 \equiv \varphi_1 \oplus \varphi_2\}}\ \varphi\text{-ADD}$$

$$\frac{}{\varphi_1 \cdot \varphi_2 = \varphi_3 \rightsquigarrow \{\varphi_3 \equiv \varphi_1 \cdot \varphi_2\}}\ \varphi\text{-MUL}$$

$$\frac{}{\tau_1 \oplus \tau_2 = \tau \rightsquigarrow \{\tau \equiv \tau_1, \tau \equiv \tau_2\}}\ \tau\text{-ADD}$$

$$\frac{}{\varphi_1 \cdot \tau_2 = \tau \rightsquigarrow \{\tau \equiv \tau_2\}}\ \tau\text{-MUL}$$

$$\frac{}{\sigma_1 \oplus \sigma_2 = \sigma \rightsquigarrow \{\sigma \equiv \sigma_1, \sigma \equiv \sigma_2\}}\ \sigma\text{-ADD}$$

$$\frac{}{\varphi_1 \cdot \sigma_2 = \sigma \rightsquigarrow \{\sigma \equiv \sigma_2\}}\ \sigma\text{-MUL}$$

Figure 3.10: Base abstract constraint operators

$$\frac{\mu_1 \oplus \mu_2 = \mu \rightsquigarrow C_1 \qquad \nu_1 \oplus \nu_2 = \nu \rightsquigarrow C_2}{{\mu_1}^{\nu_1} \oplus {\mu_2}^{\nu_2} = \mu^{\nu} \rightsquigarrow C_1 \cup C_2}\ \mu^{\nu}\text{-ADD}$$

$$\frac{\varphi_1 \cdot \mu_2 = \mu \rightsquigarrow C_1 \qquad \varphi_1 \cdot \nu_2 = \nu \rightsquigarrow C_2}{\varphi_1 \cdot {\mu_2}^{\nu_2} = \mu^{\nu} \rightsquigarrow C_1 \cup C_2}\ \mu^{\nu}\text{-MUL}$$

$$\frac{\mu_1 \oplus \mu_2 = \mu \rightsquigarrow C_1 \qquad \nu_1 \oplus \nu_2 = \nu \rightsquigarrow C_2 \qquad \delta_1 \oplus \delta_2 = \delta \rightsquigarrow C_3}{{\mu_1}^{\nu_1,\delta_1} \oplus {\mu_2}^{\nu_2,\delta_2} = \mu^{\nu,\delta} \rightsquigarrow C_1 \cup C_2 \cup C_3}\ \mu^{\nu,\delta}\text{-ADD}$$

$$\frac{\varphi_1 \cdot \mu_2 = \mu \rightsquigarrow C_1 \qquad \varphi_1 \cdot \nu_2 = \nu \rightsquigarrow C_2 \qquad \varphi_1 \cdot \delta_2 = \delta \rightsquigarrow C_3}{\varphi_1 \cdot {\mu_2}^{\nu_2,\delta_2} = \mu^{\nu,\delta} \rightsquigarrow C_1 \cup C_2 \cup C_3}\ \mu^{\nu,\delta}\text{-MUL}$$

$$\frac{x \in \mathit{keys}\ (\Gamma_1) \cup \mathit{keys}\ (\Gamma_2) \cup \mathit{keys}\ (\Gamma) \qquad \Gamma_1\ (x) \oplus \Gamma_2\ (x) = \Gamma\ (x) \rightsquigarrow C_x \qquad C = \bigcup_x C_x}{\Gamma_1 \oplus \Gamma_2 = \Gamma \rightsquigarrow C}\ \Gamma\text{-ADD}$$

$$\frac{x \in \mathit{keys}\ (\Gamma_2) \cup \mathit{keys}\ (\Gamma) \qquad \varphi_1 \cdot \Gamma_2\ (x) = \Gamma\ (x) \rightsquigarrow C_x \qquad C = \bigcup_x C_x}{\varphi_1 \cdot \Gamma_2 = \Gamma \rightsquigarrow C}\ \Gamma\text{-MUL}$$

$$\Gamma\ (x) = \begin{cases} \sigma^{\nu,\delta} & \text{if } x : \sigma^{\nu,\delta} \in \Gamma \\ \mathit{fresh}_{\sigma}{}^{\mathbb{0},\mathbb{0}} & \text{otherwise} \end{cases}$$

Figure 3.11: Lifted abstract constraint operators

Most of the time the declarative versions[36] of the operators are not suitable as the right hand side is unknown. This can of course be solved by creating a fresh variable and using that as the right hand side. This works, but for types and type schemes this is actually unnecessary as only equality constraints are generated and these can immediately be solved. The computational rules are given in figure **??**. The difference between the declarative versions in figures **??** and **??** and the computational versions in **??** is basically that the result is moved from the left hand side of the $\rightsquigarrow$ to the right hand side indicating that the result is being generated by the rule. For types (type schemes) one of the input types (type schemes) is returned together with a constraint making all the types (type schemes) equal.

With these computational versions in place we can now define folds over collections using these operators that we denote $\bigoplus$ and $\bigsqcup$. The definition of $\bigoplus$ is given in figure **??**[37]. The definition of $\bigsqcup$ is similar. These folds are only well defined over non empty collections.

[36]The versions in figures **??** and **??** are declarative in the sense that both the operants and the result need to be given before a constraint can be generated.

[37]For simplicity this is given as a function over lists.

$$\frac{\varphi = fresh_{\varphi}}{\varphi_1 \oplus \varphi_2 \rightsquigarrow (\varphi, \{\varphi \equiv \varphi_1 \oplus \varphi_2\})}\ \text{COMP-ANN-ADD}$$

$$\frac{\varphi = fresh_{\varphi}}{\varphi_1 \oplus \varphi_2 \rightsquigarrow (\varphi, \{\varphi \equiv \varphi_1 \cdot \varphi_2\})}\ \text{COMP-ANN-MUL}$$

$$\frac{}{\tau_1 \oplus \tau_2 \rightsquigarrow (\tau_1, \{\tau_1 \equiv \tau_2\})}\ \text{COMP-}\tau\text{-ADD}$$

$$\frac{}{\sigma_1 \oplus \sigma_2 \rightsquigarrow (\sigma_1, \{\sigma_1 \equiv \sigma_2\})}\ \text{COMP-}\sigma\text{-ADD}$$

$$\frac{}{\varphi_1 \cdot \gamma \rightsquigarrow (\gamma, \emptyset)}\ \text{COMP-}\sigma\text{-MUL}$$

$$\frac{x \in (keys\ (\Gamma_1) \cup keys\ (\Gamma_2)) \qquad \Gamma_1\ (x) \oplus \Gamma_2\ (x) = (\Gamma\ (x), C_x) \qquad C = \bigcup_x C_x}{\Gamma_1 \oplus \Gamma_2 \rightsquigarrow (\Gamma, C)}\ \text{COMP-}\Gamma\text{-ADD}$$

$$\frac{x \in keys\ (\Gamma_2) \qquad \varphi_1 \cdot \Gamma_2\ (x) \rightsquigarrow (\Gamma\ (x), C_x) \qquad C = \bigcup_x C_x}{\varphi_1 \cdot \Gamma_2 \rightsquigarrow (\Gamma, C)}\ \text{COMP-}\Gamma\text{-MUL}$$

$$\Gamma\ (x) = \begin{cases} \sigma^{\nu,\delta} & \text{if } x : \sigma^{\nu,\delta} \in \Gamma \\ fresh_{\sigma}{}^{\mathbb{0},\mathbb{0}} & \text{otherwise} \end{cases}$$

Figure 3.12: Computational operators

$$\begin{array}{rcl} \bigoplus[y] & ::= & (y, \emptyset) \\ \bigoplus(y : ys) & ::= & \textbf{let}\ (y_1, C_1) = \bigoplus ys; (y_2, C_2) = y \oplus y_1\ \textbf{in}\ (y_2, C_1 \cup C_2) \end{array}$$

Figure 3.13: Fold operator: y is some type for which $\oplus$ is defined

3.3.2 Constraint generation

Using the previously defined operators we can now define how the constraint generator works. The static semantics[38] are given in figure **??**.

The following environments are present in the type rules:

Υ is the import environment. This contains the annotated types for imported symbols.

Γ is the local or module environment[39]. It is defined bottom up.

Δ is the list of symbols the local module exports. This is used to determine if the annotations for these symbols are $\top$ (meaning the usage of these symbols is unknown) or are defined only by the uses of that symbol by the local module.

Π is the annotated data environment used for looking up the types for data constructors.

The environments Υ, Δ, Π are global variables which are constant throughout the constraint generation and constraint solving.

The VAR-1 and VAR-2 rules differ depending on whether or not it is a locally defined symbol or an imported symbol[40].

If it is a locally defined symbol (VAR-1) then the type scheme σ, type τ and usage ν are freshly generated. Note the demand of $\mathbb{1}$ that is present in the environment. This demand is ignored in the APP rules[41], but it is used in the right hand side of bindings. This demand is also implicitly used in all places where an expression is allowed. Finally a constraint is generated that specifies the type must be an instantiation of the type scheme.

[38] A large part of this is the same here as it is in [?]. The differences deal with modular vs whole program analysis and the added constructs of FFI, constants and Let!.

[39] Also named just environment if it is clear that is not one of the other environments.

[40] Note that we don't have to deal with shadowing as all names in UHC core are already resolved to fully qualified unique names.

[41] In the APP rules the demand depends upon how the function uses the applied value and not on the occurrence of the variable. E.g. in $\mathbf{e}\ x$ the use of x only depends upon how $\mathbf{e}$ uses its argument and not on the fact that the variable is present inside the expression.

Basics $\boxed{\Gamma \vdash \mathbf{e} : \tau^{\nu} \rightsquigarrow C}$

$$\frac{x \notin \Upsilon}{x : \sigma^{\nu,\mathbb{1}} \vdash x : \tau^{\nu} \rightsquigarrow \{\mathit{inst}\ (\sigma) \equiv \tau\}}\ \textsc{Var-1}$$

$$\frac{x \in \Upsilon \qquad \Upsilon\ (x) = \sigma}{\epsilon \vdash x : \tau^{\top} \rightsquigarrow \{\mathit{inst}\ (\sigma) \equiv \tau\}}\ \textsc{Var-2}$$

$$\frac{\mathbf{c} \text{ has type } \tau}{\epsilon \vdash \mathbf{c} : \tau^{\nu} \rightsquigarrow \emptyset}\ \textsc{Const}$$

$$\frac{\Gamma_1, x : (\forall \emptyset\, \emptyset\, .\ \emptyset \Rightarrow \tau)^{\nu,\delta} \vdash \mathbf{e} : {\tau_1}^{\nu_1} \rightsquigarrow C_1 \qquad \nu_2 \cdot \Gamma_1 = \Gamma_2 \rightsquigarrow C_2}{\Gamma_2 \vdash \lambda x \rightarrow \mathbf{e} : (\tau^{\nu,\delta} \rightarrow {\tau_1}^{\nu_1})^{\nu_2} \rightsquigarrow C_1 \cup C_2}\ \textsc{Abs}$$

$$\frac{x \notin \Upsilon \qquad \Gamma_1 \vdash_{\sqsubseteq} \mathbf{e} : ({\tau_2}^{\nu_2,\delta_2} \rightarrow {\tau_3}^{\nu_3})^{\mathbb{1}} \rightsquigarrow C_1 \qquad x : {\tau_4}^{\nu_4 \mathbb{1}} \vdash_{\diamond} x : {\tau_2}^{\nu_2} \rightsquigarrow C_2 \qquad \Gamma_1 \oplus x : {\tau_4}^{\nu_4 \delta_2} = \Gamma_2 \rightsquigarrow C_3}{\Gamma_2 \vdash \mathbf{e}\ x : {\tau_3}^{\nu_3} \rightsquigarrow C_1 \cup C_2 \cup C_3}\ \textsc{App-1}$$

$$\frac{x \in \Upsilon \qquad \Gamma \vdash_{\sqsubseteq} \mathbf{e} : ({\tau_2}^{\nu_2,\delta_2} \rightarrow {\tau_3}^{\nu_3})^{\mathbb{1}} \rightsquigarrow C_1 \qquad \epsilon \vdash_{\diamond} x : {\tau_2}^{\nu_2} \rightsquigarrow C_2}{\Gamma \vdash \mathbf{e}\ x : {\tau_3}^{\nu_3} \rightsquigarrow C_1 \cup C_2}\ \textsc{App-2}$$

$$\frac{\Gamma \vdash_{\sqsubseteq} \mathbf{e} : ({\tau_2}^{\nu_2,\delta_2} \rightarrow {\tau_3}^{\nu_3})^{\mathbb{1}} \rightsquigarrow C_1 \qquad \epsilon \vdash_{\diamond} \mathbf{c} : {\tau_2}^{\nu_2} \rightsquigarrow C_2}{\Gamma \vdash \mathbf{e}\ \mathbf{c} : {\tau_3}^{\nu_3} \rightsquigarrow C_1 \cup C_2}\ \textsc{App-3}$$

$$\frac{\begin{array}{c}\Gamma_0, \overline{x_i : {\sigma_i}^{\nu_i,\delta_i}} \vdash \mathbf{e} : \tau^{\nu} \rightsquigarrow C_0 \qquad \Gamma_i, \overline{x_j : (\forall \emptyset\, \emptyset\, .\ \emptyset \Rightarrow \tau_{ij})^{\nu_{ij},\delta_{ij}}} \vdash \mathbf{e_i} : {\tau_i}^{\nu_i} \rightsquigarrow C_{1i} \\ \Gamma_0 \oplus (\bigoplus_i\ (\delta_i \triangleright \Gamma_i)) = \Gamma \rightsquigarrow C_2 \qquad \delta_i \oplus (\bigoplus_j\ (\delta_j \triangleright \delta_{ij})) = \delta_i \rightsquigarrow C_{3i} \\ \nu_i \oplus (\bigoplus_j\ (\delta_j \triangleright \nu_{ij})) = \nu_i \rightsquigarrow C_{4i} \qquad \bigoplus_j\ (\delta_j \triangleright \tau_{ij}) = \tau_i \rightsquigarrow C_{5i} \qquad C_1 = \bigcup_i C_{1i} \\ C_3 = \bigcup_i C_{3i} \qquad C_4 = \bigcup_i C_{4i} \qquad C_5 = \bigcup_i C_{5i} \qquad C_6 = \bigcup_i \{\mathit{gen}\ ({\tau_i}^{\nu_i,\delta_i}, C_1 \cup C_3 \cup C_4 \cup C_5, \Gamma) \equiv \sigma_i\} \\ \textbf{if } x_i \in \Delta \textbf{ then } C_{7i} = \{\nu_i \equiv \top, \delta_i \equiv \top\} \textbf{ else } C_{7i} = \emptyset \qquad C_7 = \bigcup_i C_{7i}\end{array}}{\Gamma \vdash \textbf{let } \overline{x_i = \mathbf{e_i}} \textbf{ in } \mathbf{e} : \tau^{\nu} \rightsquigarrow C_0 \cup C_2 \cup C_6 \cup C_7}\ \textsc{Let}$$

Sequential evaluation $\boxed{\Gamma \vdash \mathbf{e} : \tau^{\nu} \rightsquigarrow C}$

$$\frac{\begin{array}{c}\Gamma_0, x : {\sigma_0}^{\nu,\delta} \vdash \mathbf{e_2} : {\tau_2}^{\nu_2} \rightsquigarrow C_0 \\ \Gamma_1, x : (\forall \emptyset\, \emptyset\, .\ \emptyset \Rightarrow \tau)^{\nu_1,\delta_1} \vdash_{\sqsubseteq} \mathbf{e_1} : \tau^{\mathbb{0}} \rightsquigarrow C_1 \qquad \Gamma_0 \oplus \Gamma_1 = \Gamma \rightsquigarrow C_2 \qquad \mathbb{1} \oplus \delta = \delta_0 \rightsquigarrow C_3 \\ \delta_0 \oplus \delta_1 = \delta_0 \rightsquigarrow C_4 \qquad \nu \oplus \nu_1 = \nu \rightsquigarrow C_5 \qquad C_6 = \{\mathit{gen}\ (\tau^{\nu,\delta_0}, C_1 \cup C_3 \cup C_4 \cup C_5, \Gamma) \equiv \sigma_0\}\end{array}}{\Gamma \vdash \textbf{let}\ !\ x = \mathbf{e_1} \textbf{ in } \mathbf{e_2} : {\tau_2}^{\nu_2} \rightsquigarrow C_0 \cup C_2 \cup C_6}\ \textsc{Let!}$$

Datatypes $\boxed{\Gamma \vdash \mathbf{e} : \tau^{\nu} \rightsquigarrow C}$

$$\frac{\begin{array}{c}\textbf{data } T\ \overline{u_l}\ \overline{\alpha_k} = \overline{K_i\ \overline{{\tau_{ij}}^{\nu_{ij},\delta_{ij}}}} \in \Pi \qquad \tau^{\nu_j,\delta_j} = {\tau_{ij}}^{\nu_{ij},\delta_{ij}}\ [{}^{\overline{\varphi_l}}/_{\overline{u_l}}, {}^{\overline{\tau_k}}/_{\overline{\alpha_k}}] \qquad \Gamma_j \vdash v_j : \tau^{\nu_j} \rightsquigarrow C_{1j} \\ C_1 = \bigcup_j C_{1j} \qquad \bigoplus_j \Gamma_j = \Gamma \rightsquigarrow C_2 \qquad \textbf{if } v_j \in \Gamma_j \textbf{ then } \delta_j \text{ is the demand on } v_j \text{ in } {\Gamma_j}^{a}\end{array}}{\Gamma \vdash K_i\ \overline{v_j} : (T\ \overline{\varphi_l}\ \overline{\tau_k})^{\nu} \rightsquigarrow C_1 \cup C_2}\ \textsc{Con}$$

$$\frac{\begin{array}{c}\Gamma_i \vdash v_i : {\tau_i}^{\nu_i} \rightsquigarrow C_{1i} \\ C_1 = \bigcup_i C_{1i} \qquad \bigoplus_i \Gamma_i = \Gamma \rightsquigarrow C_2 \qquad \textbf{if } v_i \in \Gamma_i \textbf{ then } \delta_i \text{ is the demand on } v_i \text{ in } {\Gamma_i}^{b}\end{array}}{\Gamma \vdash (v_1, v_2, .., v_n) : ({\tau_1}^{\nu_1,\delta_1}, {\tau_2}^{\nu_2,\delta_2}, .., {\tau_n}^{\nu_n,\delta_n})^{\nu} \rightsquigarrow C_1 \cup C_2}\ \textsc{Tup}$$

$$\frac{\begin{array}{c}\textbf{data } T\ \overline{u_l}\ \overline{\alpha_k} = \overline{K_i\ \overline{{\tau'_{ij}}^{\nu'_{ij},\delta'_{ij}}}} \in \Pi \qquad \overline{{\tau_{ij}}^{\nu_{ij},\delta_{ij}} = {\tau'_{ij}}^{\nu'_{ij},\delta'_{ij}}\ [{}^{\overline{\varphi_l}}/_{\overline{u_l}}, {}^{\overline{\tau_k}}/_{\overline{\alpha_k}}]} \\ \Gamma_0 \vdash_{\sqsubseteq} x : (T\ \overline{\varphi_l}\ \overline{\tau_k})^{\mathbb{0}} \rightsquigarrow C_1 \qquad \Gamma_i, x_{ij} : (\forall \emptyset\, \emptyset\, .\ \emptyset \Rightarrow \tau_{ij})^{\nu_{ij},\delta_{ij}} \vdash \mathbf{e_i} : \tau^{\nu} \rightsquigarrow C_2 \qquad \bigsqcup_i \Gamma_i = \Gamma \rightsquigarrow C_3\end{array}}{\Gamma \vdash \textbf{case } x \textbf{ of } \overline{K_i\ \overline{x_{ij}} \rightarrow \mathbf{e_i}} : \tau^{\nu} \rightsquigarrow C_1 \cup C_2 \cup C_3}\ \textsc{Case-1}$$

$$\frac{\Gamma_0 \vdash_{\sqsubseteq} x : ({\tau_1}^{\nu_1,\delta_1}, {\tau_2}^{\nu_2,\delta_2}, .., {\tau_n}^{\nu_n,\delta_n})^{\mathbb{0}} \rightsquigarrow C_1 \qquad \Gamma, \overline{x_i : (\forall \emptyset\, \emptyset\, .\ \emptyset \Rightarrow \tau_i)^{\nu_i,\delta_i}} \vdash \mathbf{e_1} : \tau^{\nu} \rightsquigarrow C_2}{\Gamma \vdash \textbf{case } x \textbf{ of } (x_1, x_2, .., x_n) \rightarrow \mathbf{e_1} : \tau^{\nu} \rightsquigarrow C_1 \cup C_2 \cup C_3}\ \textsc{Case-2}$$

$$\frac{\mathbf{c} \text{ has type } \tau \qquad \Gamma_0 \vdash_{\sqsubseteq} x : \tau^{\mathbb{0}} \rightsquigarrow C_1 \qquad \Gamma_i \vdash \mathbf{e_i} : {\tau_2}^{\nu_2} \rightsquigarrow C_{2i} \qquad C_2 = \bigcup_i C_{2i} \qquad \bigsqcup_i \Gamma_i = \Gamma \rightsquigarrow C_3}{\Gamma \vdash \textbf{case } x \textbf{ of } \overline{\mathbf{c} \rightarrow \mathbf{e_i}} : {\tau_2}^{\nu_2} \rightsquigarrow C_1 \cup C_2 \cup C_3}\ \textsc{Case-3}$$

Figure 3.14: Static semantics

[a] In that case Γ_j looks like $v_j : {\sigma_j}^{\nu_j,\delta_j}$.
[b] In that case Γ_i looks like $v_i : {\sigma_i}^{\nu_i,\delta_i}$.

Foreign function call $\boxed{\Gamma \vdash \mathbf{e} : \tau^{\nu} \rightsquigarrow C}$

$$\frac{\mathit{annotate}\ \tau^{\mathit{Core}} = \sigma^{\nu}}{\epsilon \vdash \mathit{FFI}\ n\ \tau^{\mathit{Core}} : \tau^{\nu} \rightsquigarrow \{\mathit{inst}\ (\sigma) \equiv \tau\}}\ \text{FFI}$$

Subeffecting $\boxed{\Gamma \vdash_{\backsim} \mathbf{e} : \tau_{\nu} \rightsquigarrow C}$

$$\frac{\Gamma \vdash \mathbf{e} : \tau^{\nu_2} \rightsquigarrow C}{\Gamma \vdash_{\backsim} \mathbf{e} : \tau^{\nu_1} \rightsquigarrow C \cup \{\nu_1 \backsim \nu_2\}}\ \text{SUB}(\backsim)$$

Figure 3.14: Static semantics

If it is an imported symbol (VAR-2) the type scheme is looked up in the import environment Υ. A fresh type variable is created and an instantiation constraint is generated. The local environment is empty as the use of an imported symbol can never contribute to the type (or annotations) of the imported symbol. Imported symbols always have $\top$ as their usage annotation. This is because the usage annotation is part of the usage annotation for the declaration site (see the LET rule) and for exported symbols[42] the usage annotation is always $\top$.

The CONST rule is trivial. A fresh usage annotation variable is created for the usage annotation and no constraints are generated.

The ABS rule only allows monomorphic arguments. This restriction keeps the types of function at rank 1 and avoids the loss of principal typing for higher-rank types[43]. The usage of all free variables in the body are multiplied by the usage of the abstraction itself.

The APP-1, APP-2 and APP-3 rules differ depending on whether it is application to a locally defined symbol, an imported symbol or a constant. In all three rules the function is used exactly once[44]. When it is a locally defined symbol (APP-1) the demand of $\mathbb{1}$ on the variable is ignored and changed to the demand δ_2 of the function argument. This is also the only place where the analysis dependent subeffecting rule is present. When it is an application to an imported symbol (APP-2) or to a constant (APP-3) the typing rules are simplified by the fact that the variable has an empty environment. In the rule for constant application the constraints for the constant can only contain a single subeffecting constraint as a normal constant derivation does not generate any constraints.

The LET rule is by far the largest. Fortunately most antecedents are trivial. The first antecedent types the body of the let. The types for the bindings are allowed to be polymorphic type schemes. The second antecedent types each binding, but here the types for the bindings can only be monomorphic types[45]. The next four antecedents are the definition for the environment, demand, usage and type respectively[46]. Then we have four antecedents which just collects the constraints from each binding. Next are the generalization constraints for each binding. The generated constraints that are collected per binding are put inside the generalization constraint and do not appear in the constraints generated for the let expression as a whole. The next to last antecedent sets the demand and usage of each exported binding to $\top$. This is crucial as we are not allowed to assume any use and demand for exported symbols. For example it would not be sound to derive a demand of $\mathbb{0}$ as that would imply the symbol is never used, even in modules that import the current module. The last antecedent again just collects the constraints for the exported bindings.

The LET! rule is a cross between the LET rule and the SEQ rule defined in [**?**]. The rule is basically a simplified version[47] of the LET rule, with two differences. The binding is typed with subeffecting setting the usage to $\mathbb{0}$, and $\mathbb{1}$ is added to δ to tell the system that the binding is demanded at least once[48]. The $\mathbb{0}$ comes from the fact that although the binding is evaluated (this is where the $\mathbb{1}$ comes from) to WHNF it does not imply a use. Subeffecting is used to ensure that it is still possible to use the binding inside the body of the **let**. As δ_0 cannot contain a $\mathbb{0}$ all the $\delta_i \triangleright$ can be removed from the rules for LET simplifying the rules even more.

The CON rule is for fully applied data types. First a completely fresh copy of the data type is generated and then each constructor field type is matched to the corresponding given argument type. Some care

[42] To be able to import the symbol it needs to be exported by another module.
[43] Counting analysis also does type inference at the same time as calculating the annotations on the type. Even when the types are already present we do not want polyvariant arguments for the same reason we do not want polymorphic arguments during type inference. This restriction to monomorphic arguments makes sense as nearly all polymorphic functions are also polyvariant.
[44] This uses subeffecting so it can potentially be used multiple times, but the application uses the function exactly once.
[45] This is the same reason as why function arguments needs to be monomorphic. It prevents loss of principal typing.
[46] The definitions for the usage and demand annotations are fixpoints. The ones for environment and types are folds.
[47] A *Let*! can only have a single binding.
[48] In [**?**] this is achieved by the $\mathbb{1}$ present inside the var rule.

needs to be taken here to ensure that the demand in the environment for each argument matches the demand of the fresh data type.

The TUP rule is basically the same as the CON rule but without the need to create a fresh instance and without the complication of matching types and annotations correctly. This is because a tuple is fully polymorphic in each field.

The CASE rule is different based upon whether pattern matching is on a data type, tuple or constant. The pattern match must be complete[49]. The variable being pattern matched on is neither used nor demanded by the **case** expression. Subeffecting is enable here so that if it is used in other places that is allowed by the generated constraints[50]. In all cases the Γ_0 for the case variable is ignored. This is valid as it contains a fresh type scheme variable with both a usage and demand annotation of lzero. So adding this environment to Γ would not do anything[51] The rule for data types first creates a fresh copy of the data type. Each constructor field is assigned the type of the corresponding field in the data type. This rule mostly boils down to carefully matching the generated correct types and annotations. The rule for tuples is slightly simpler as there is only one alternative and no fresh copy needs to be generated[52]. The rule for constants is basically the same as for data types with the exception that now no fresh copy needs to be generated. It is also simpler as there are no fields present.

The FFI rule is nearly identical to the rule for imported symbols. The only difference is that the type comes from annotating the τ^{core}[53]. The function *annotate* will pessimistically[54] annotate the given type.

The SUB($\backsim$) rule is the only not syntax directed rule. It is however only applied in specified places[55] and cannot apply in others.

3.3.3 Constraint solving

After the gathering of constraints the constraints need to be solved[56]. The declarative solving rules are given in figure **??**. An annotation constraint can only be solved when the operants are annotation values. In that case the definitions in figure **??** can be applied to solve the constraint.

The notation $A \rightsquigarrow B$ means that constraint A generates the constraints B to also be solved before A is considered to be solved. Only instantiation constraints generate new constraints to be solved.

$$\frac{}{\mu \equiv \mu \rightsquigarrow \emptyset}\ \textsc{Solve-}\mu\textsc{-EQ}$$

$$\frac{\varpi_1 \diamond \varpi_2 = \varphi}{\varphi \equiv \varpi_1 \diamond \varpi_2 \rightsquigarrow \emptyset}\ \textsc{Solve-}\varphi$$

$$\frac{\begin{array}{c} C_2 = \mathit{simplify}\ C_1 \\ V_\alpha = ((\mathit{ftv}\ C_2) \cup (\mathit{ftv}\ \tau)) - (\mathit{ftv}\ \Gamma) \qquad V_\beta = ((\mathit{fav}\ C_2) \cup (\mathit{fav}\ \tau)) - ((\mathit{fav}\ \Gamma) \cup \{\nu, \delta\}) \end{array}}{\mathit{gen}\ (\tau^{\nu,\delta}, C_1, \Gamma) \equiv \forall\, V_\alpha\ V_\beta\ .\ C_2 \Rightarrow \tau \rightsquigarrow \emptyset}\ \textsc{Solve-Gen}$$

$$\frac{\phi = [\,{}^{\overline{alpha'}}/_{\overline{\alpha}}, {}^{\overline{beta'}}/_{\overline{\beta}}]}{\mathit{inst}\ (\forall\, \overline{\alpha}\ \overline{\beta}\ .\ C \Rightarrow \tau) \equiv \tau\ [\phi] \rightsquigarrow C\ [\phi]}\ \textsc{Solve-Inst}$$

Figure 3.15: Solving rules

The SOLVE-μ-EQ rule just specifies that an equality constraint is solved if both sides are equal.

The SOLVE-φ rule specifies that an annotation constraint is solved when both operants are annotation values ϖ and the result of applying the operator to the operants is equal to the annotation stored inside the constraint[57].

The SOLVE-GEN rule specifies that a generalization constraint is solved when the generalization of $\tau^{\nu,\delta}$ under constraints C_1 and environment Γ is equal to $\forall\, V_\alpha\ V_\beta\ .\ C_2$. It generalizes over the free type

[49] Every constructor must match. Also there must be at least one constructor for a data type. Things like **data** *Void* without any constructors are not supported.

[50] Without subeffecting here the expression would not be allowed to be used anywhere.

[51] The type schemes would be made equal and for the annotation we would sum a $\mathbb{0}$ which would not change anything as $\mathbb{0}$ is the identity for $(\oplus)$

[52] It can be done but as this than boils down to making all the types and annotations in the tuple fresh variables which then needs to be equal to other things it is just simpler to omit this and not generate a bunch of equality constraints.

[53] τ^{core} is basically the same as τ^{EH}. The only difference is that the subset of τ^{core} allowed in an *FFI* is monomorphic.

[54] In other words all annotations will be $\top$.

[55] Only in the APP, CASE and LET! rules.

[56] As the constraint definitions match the constraint definitions in [?] the following section contains the same as in [?], although presented differently.

[57] The φ present in the type rule.

variables V_α and the free annotation variables V_β. The constraint set C_2 only contains constraints that cannot be solved. This is what *simplify* does[58]. The only type of constraints that can not be solved are annotation constraints that have annotation variables as one or more of the operants. These constraints define the limitations quantified annotation variables should obey. An example of these kind of constraints are constraints that specify that the usage is always less than or equal to the demand.

The Solve-Inst rule specifies that an instantiation constraint is solved when there exists a substitution ϕ such that the substitution applied to the the type inside the type scheme is equal to the instantiated type. The substitution applied to the constraints in the type scheme is the set of new constraints that still needs to be solved.

In section **??** an algorithm is given that will solve the constraints if they are not yet solved according to the specifications given here.

[58]In a declarative system *simplify* is equal to *id*. The reason it is still present in the type rule is that it makes it explicit that the constraint set C_2 only contains annotation variable constraints. In the algorithm in section **??** *simplify* is equal to $\mathcal{S}_{fix}$, the function that solves the constraints as much as possible.

4. Annotation driven optimizations

4.1 Introduction

Now that the annotations are in place we can define source code transformations using these annotations. There are two ways these annotation can be used to optimize code. The first is source code to source code transformations. The second way is for source code to executable code. Examples of the first kind are strictness optimizations ([?]) and lambda/let floating ([?]). An example of the second kind is the non-updating thunk optimization ([?]). Here a strictness transformation will be given[1]. Other optimizations will be for future work.

4.2 Stricter code

The transformation described here will introduce **let**! to enable call by value evaluation for strict applications and transform normal *Let* in **let**! whenever there is only a single binding which is guaranteed to be used at least once. In figure **??** the declarative rules for the transformation are given. Only the rules for actual transformations are given, the identity rules for all other expressions are not presented. A **let**! is only introduced when the expression being made strict is not already guaranteed to be in normal form. This is the case when the expression is already in normal form (see figure **??**) or when there is already an **let**! introduced for that expression. The variables which are already in a **let**! in scope are stored in the environment Ω. A final part is when pattern matching on a strict field of a data constructor. This field is by definition in normal form. A tuple has by definition lazy fields so no check needs to be performed there.

The crucial part in all the rules is the $\delta \subseteq \{1, \infty\}$ antecedent. This is met when delta is $\mathbb{1}$, ω or $\{1, \infty\}$. This tells the rule the expression is demanded at least once and so it is safe to introduce a **let**! for the expression. As soon as we introduced a **let**! the variable now made strict is added to the set of variables in normal form in the body of the **let**!. This ensures that we never introduce another **let**! for the same variable in the same scope. It can still happen that a variable is forced multiple times. This is the case if the **let**! for a variable is introduced inside multiple **let** bindings. Also **let**! introduced inside a binding does not mean it is forced inside the body of the **let**.

$$\zeta \quad ::= \quad \mathbf{c} \;\mid\; \lambda x \rightarrow \mathbf{e} \;\mid\; K\ \overline{v_i} \;\mid\; (v_1, v_2, .., v_n)$$

Figure 4.1: Normal forms

The declarative rules are sound when the annotations are sound. By transforming the expressions only demands on variables that were already guaranteed to be demanded are added. The declarative rules do not change the annotations but if another transformation would transform the strictness transformed code with the old annotations then problems[2] occur. The fix is simple: whenever a **let**! is introduced change the demand on the variable that is forced to ω[3].

[1] Based on [?]
[2] In this case this means unsound transformations
[3] Now it is demanded at least twice

Strictness transformation $\boxed{\Omega \vdash \mathbf{e} \leadsto \mathbf{e}}$

$$\frac{\textit{fresh}\ x' \qquad \mathbf{e} : ({\eta_1}^{\delta} \to \eta_2)^{\nu_2} \qquad \delta \subseteq \{1, \infty\} \qquad x \not\equiv \zeta \land x \notin \Omega \qquad \Omega, x \vdash \mathbf{e} \leadsto \mathbf{e'}}{\Omega \vdash \mathbf{e}\ x \leadsto \mathbf{let}\,!\,x' = x\ \mathbf{in}\ \mathbf{e'}\ x}\ \textsc{App}$$

$$\frac{x : \eta^{\delta} \qquad \delta \subseteq \{1, \infty\} \qquad \mathbf{e_1} \notin \Omega \qquad \Omega \vdash \mathbf{e_1} \leadsto \mathbf{e'_1} \qquad \Omega, x \vdash \mathbf{e} \mapsto \mathbf{e'}}{\Omega \vdash \mathbf{let}\ x = \mathbf{e_1}\ \mathbf{in}\ \mathbf{e} \leadsto \mathbf{let}\,!\,x = \mathbf{e'_1}\ \mathbf{in}\ \mathbf{e'}}\ \textsc{Let}$$

$$\frac{\Omega \vdash \mathbf{e_1} \leadsto \mathbf{e'_1} \qquad \Omega, x \vdash \mathbf{e} \mapsto \mathbf{e'}}{\Omega \vdash \mathbf{let}\,!\,x = \mathbf{e_1}\ \mathbf{in}\ \mathbf{e} \leadsto \mathbf{let}\,!\,x = \mathbf{e'_1}\ \mathbf{in}\ \mathbf{e'}}\ \textsc{Let!}$$

$$\frac{\textit{fresh}\ \overline{x'_k} \qquad \overline{x_k} \subseteq \overline{v_i} \qquad \forall\,\mathbf{k}\,.\ x_k \not\equiv \zeta\ \lor\ x_k \notin \Omega \qquad \forall\,\mathbf{k}\,.\ x_k : \textit{etak}^{\delta_k} \qquad \forall\,\mathbf{k}\,.\ \delta_k \subseteq \{1, \infty\}}{\Omega \vdash K\ \overline{v_i} \leadsto \mathbf{let}\,!\,x'_1 = v_1\ \mathbf{in} \cdots \mathbf{let}\,!\,x'_n = v_n\ \mathbf{in}\ K\ \overline{v_i}}\ \textsc{Con}$$

$$\frac{\textit{fresh}\ \overline{x'_i} \qquad \overline{x_i} \subseteq \overline{v_i} \qquad \forall\,i\,.\ x_i \not\equiv \zeta\ \lor\ x_i \notin \Omega \qquad \forall\,i\,.\ x_i : \textit{etai}^{\delta_i} \qquad \forall\,i\,.\ \delta_i \subseteq \{1, \infty\}}{\Omega \vdash (v_1, v_2, \cdots, v_n) \leadsto \mathbf{let}\,!\,x'_1 = v_1\ \mathbf{in} \cdots \mathbf{let}\,!\,x'_n = v_n\ \mathbf{in}\ (v_1, v_2, \cdots, v_n)}\ \textsc{Tup}$$

Case alternatives $\boxed{\Omega \vdash p \to \mathbf{e} \leadsto p \to \mathbf{e}}$

$$\frac{\begin{array}{c}\textit{fresh}\ \overline{yj'} \qquad \overline{y_i} \subseteq \overline{x_i} \qquad \forall\,j\,.\ y_j \notin \Omega \qquad \forall\,j\,.\ y_j : {\tau_j}^{\nu_j, \delta_i} \qquad \forall\,j\,.\ \delta_j \subseteq \{1, \infty\} \\ \forall\,j\,.\ \textit{field}\ j\ \mathbf{of}\ K\ \textit{is}\ \neg\ \textit{strict} \qquad \overline{\textit{elzs}} \subseteq \overline{x_i} \qquad \forall\,s\,.\ \textit{field}\ s\ \mathbf{of}\ K\ \textit{is strict} \qquad \Omega, \overline{y_j}, \overline{zs} \vdash \mathbf{e} \mapsto \mathbf{e'}\end{array}}{\Omega \vdash K\ \overline{x_i} \to \mathbf{e} \leadsto K\ \overline{x_i} \to \mathbf{let}\,!\,\textit{y1}' = y_1\ \mathbf{in} \cdots \mathbf{let}\,!\,\textit{yn}' = y_n\ \mathbf{in}\ \mathbf{e'}}\ \textsc{Pat-Con}$$

$$\frac{\textit{fresh}\ \overline{y'_i} \qquad \overline{y_i} \subseteq \overline{x_i} \qquad \forall\,i\,.\ y_i \not\equiv \zeta \qquad \forall\,i\,.\ y_i : \textit{etai}^{\delta_i} \qquad \forall\,i\,.\ \delta_i \subseteq \{1, \infty\} \qquad \Omega, \overline{y_i} \vdash \mathbf{e} \mapsto \mathbf{e'}}{\Omega \vdash (x_1, x_2, \cdots, x_n) \to \mathbf{e} \leadsto (x_1, x_2, \cdots, x_n) \to \mathbf{let}\,!\,\textit{y1}' = y_1\ \mathbf{in} \cdots \mathbf{let}\,!\,\textit{yn}' = y_n\ \mathbf{in}\ \mathbf{e'}}\ \textsc{Pat-Tup}$$

Figure 4.2: Strictness transformation

5. Implementation

The type rules in chapter **??** do not readily match an implementation. In this chapter the type rules are refined into algorithms that generate and solve constraints (chapter **??**) . All the algorithms described in this section are fully implemented inside a branch[1] of UHC.

5.1 Datatype annotation algorithm

5.1.1 Introduction

Nearly all Haskell code uses datatypes in some form, so it is imperative for any analysis that it is precise for datatypes. As it is allowed for programmers to define new datatypes it is necessary to have an algorithm that automatically annotates any data declaration. The datatype annotation algorithm described here is based upon work by Wansbrough [**?**]. The algorithm has full support for both data and type declarations, and all forms of recursive declarations[2]. Newtype declaration are also supported and are treated the same way as data declarations[3]. In the rest of this section newtype declarations will therefore be omitted from the text and code fragments.

The algorithm takes EH[4] data and type declarations and transforms them into annotated declarations. First the some formal definitions are given, then the algorithm is presented and finally some examples are given to illustrate how the annotation algorithm performs.

5.1.2 Definitions

The relevant EH syntax[5] is given in figure **??**. Although not enforced by the syntax[6] the input AST to the algorithm is structured such that every let binding group is mutually recursive.

Fields of a datatype have an optional strictness annotation on the types, notated as $!\tau^{EH}$. Type declaration do not have fields and so can never have any strictness annotations. The types in the declaration are the same types as in figure **??** but with the annotations removed.

$$
\begin{array}{rcl}
\mathbf{e}^{EH} & ::= & \mathbf{let}\ \overline{D^{EH}}\ \mathbf{in}\ \mathbf{e}^{EH} \quad |\ \epsilon \\
D^{EH} & ::= & \mathbf{data}\ T\ \overline{\alpha} = \overline{K :: \overline{\pi^{EH}}} \quad |\ \mathbf{type}\ T\ \overline{\alpha} = \tau^{EH} \\
\pi^{EH} & ::= & \tau^{EH} \quad |\ !\,\tau^{EH} \\
\tau^{EH} & ::= & \alpha \quad |\ T\ \overline{\tau_k{}^{EH}} \quad |\ \alpha\ \tau^{EH} \quad |\ \tau_1{}^{EH} \to \tau_2{}^{EH} \quad |\ (\tau_1{}^{EH}, \tau_2{}^{EH}, \cdots, \tau_n{}^{EH})
\end{array}
$$

Figure 5.1: EH

The annotation algorithm both consumes and produces a data environment Π which holds the annotated data and type declarations. The definition is given in figure **??**. The strictness annotation from the EH types are carried over into the data environment.

$$
\begin{array}{rcl}
\Pi & ::= & \epsilon \quad |\ \Pi, T\ \overline{\alpha}\ \overline{\beta} = \overline{K_i\ \overline{\pi_{ij}{}^{\nu_{ij},\delta_{ij}}}} \quad |\ \Pi, T\ \overline{\alpha}\ \overline{\beta} = \tau \\
\pi & ::= & \tau^{\nu,\delta} \quad |\ !\,\tau^{\nu,\delta}
\end{array}
$$

Figure 5.2: Data environment

[1]https://github.com/UU-ComputerScience/uhc/tree/TiborCountingAnalysis

[2]This includes direct and indirect recursive datatypes, mutual recursive datatypes and data declarations mutual recursive with type declarations. Recursive types are not allowed as that would lead to infinite types during type expansion.

[3]During runtime there is a difference between data and newtype declaration, however during type inference they are the same.

[4]The intermediate language of UHC on which type inference is performed

[5]Only the subset needed during the annotation algorithm is shown. ϵ stands for any expression that is irrelevant here

[6]A dependently typed language is needed for this to be even possible

5.1.3 Annotation algorithm

The annotation algorithm takes the data environment from imported modules, possibly already extended with local declarations and an EH expression. It produces a data environment which hold both the imported and the local definitions. The algorithm is presented in figures **??** up to and including **??**. The idea is to transform every EH type into an annotated type. Fresh annotation variables are generated wherever an annotation needs to be present. This fresh variable introduction happens up to a fixed depth ι inside the type. By setting ι to zero no annotation variables will be introduced. By setting it to one only for each field will there be fresh annotation variables be generated, no variables are generated inside the types. If ι is very high than there will be annotation variables generated all over the types. For large datatypes with a large type depth, e.g. ASTs, this will be a lot. The parameter ι is a performance trade-off. On the one hand having a high ι will make the analysis very precise, but it will come with a performance penalty as a lot more constraints are generated.

To deal with (mutual) recursive definitions, no annotations are generated whenever a datatype is found that is part of the current binding group. All (data)types in the binding group will have the same annotations. This ensures that an annotation variable traced through the recursion will end up pointing to the same place as is defined locally[7].

The algorithm is heavily circular in that the collected annotations for each declaration are used to annotate the declaration itself. In lazy languages this is not a problem as the annotations can be collected independently of annotating the declaration[8]

In figure **??** the algorithm for expressions is given. This is the top level function that is used. When there are no declarations left (ϵ) then the data environment is just returned. If there are declarations present it runs the algorithm separately for each declaration in the binding group. This is also the place where the circularity of the algorithm is shown: the collected annotations of each declaration is passed to each declaration. After collecting the annotated declarations of each declaration these are merged with the input data environment to produce the data environment for the body of the **let**.

$\mathbf{e}^{EH}$ $\boxed{\mathcal{D}\ (\Pi, \mathbf{e}^{EH}) \leadsto \Pi}$

$$\begin{aligned}
&\mathcal{D}\ (\Pi, \epsilon) = \Pi \\
&\mathcal{D}\ (\Pi, \mathbf{let}\ \overline{D_i^{EH}}\ \mathbf{in}\ \mathbf{e}^{EH}) = \\
&\quad \mathbf{let}\ \overline{(\Pi_i, \overline{\beta_{ij}})} = [\mathcal{D}'\ (\Pi, \textstyle\bigcup_i \beta_i, D_i^{EH}) \mid D_i^{EH} \leftarrow \overline{D_i^{EH}}] \\
&\quad \mathbf{in}\ \mathcal{D}\ (\Pi \cup (\textstyle\bigcup_i \Pi_i), \mathbf{e}^{EH})
\end{aligned}$$

Figure 5.3: Datatype annotation algorithm (expression)

In figure **??** the algorithm for a single declaration is given. It has as input the current data environment, the annotations for the declaration and the declaration itself. It produces the singleton data environment containing the annotated version of the declaration and a list of annotation variables generated while annotating the current declaration. It runs the algorithm for each data type field declared inside the declaration.

D^{EH} $\boxed{\mathcal{D}'\ (\Pi, \overline{\beta}, D^{EH}) \leadsto (\Pi, \overline{\beta})}$

$$\begin{aligned}
&\mathcal{D}'\ (\Pi, \overline{\beta}, \mathbf{data}\ T\ \overline{\alpha} = \overline{K_i :: \overline{{\pi_{ij}}^{EH}}}) = \\
&\quad \mathbf{let}\ \overline{\overline{(\pi_{ij}{}^{\nu_{ij}, \delta_{ij}}, \overline{\beta_{ijk}})}} = [[\mathcal{T}'\ (\Pi, \overline{\beta}, {\pi_{ij}}^{EH}) \mid {\pi_{ij}}^{EH} \leftarrow \overline{{\pi_{ij}}^{EH}}] \mid \overline{{\pi_{ij}}^{EH}} \leftarrow \overline{\overline{{\pi_{ij}}^{EH}}}] \\
&\quad \mathbf{in}\ (T\ \overline{\alpha}\ \overline{\beta} = \overline{K_i\ \overline{\pi_{ij}{}^{\nu_{ij}, \delta_{ij}}}}, \textstyle\bigcup_i (\bigcup_j (\bigcup_{\mathbf{k}} \beta_{ijk}))) \\
&\mathcal{D}'\ (\Pi, \overline{\beta}, \mathbf{type}\ T\ \overline{\alpha} = \tau^{EH}) = \\
&\quad \mathbf{let}\ (\tau, \overline{\beta_i}) = \mathcal{T}\ (\Pi, \overline{\beta}, \tau^{EH}, 1) \\
&\quad \mathbf{in}\ (T\ \overline{\alpha}\ \overline{\beta} = \tau, \overline{\beta_i})
\end{aligned}$$

Figure 5.4: Datatype annotation algorithm (declaration)

In figure **??** the algorithm possibly strictness annotated fields is done. It just runs the algorithm on

[7]The recursive examples at the end of this section might clarify this

[8]In strict languages this two phases needs to be separated in different functions.

the underlying type (using the version for datatype fields) and re-adds the strictness annotation when needed.

π^{EH} $\boxed{\mathcal{T}'\ (\Pi, \overline{\beta}, \pi^{EH}) \rightsquigarrow (\pi^{\nu,\delta}, \overline{\beta})}$

```
$\mathcal{T}'\ (\Pi, \overline{\beta}, \tau^{EH}) =$
  **let** $(\tau^{\nu,\delta}, \overline{\beta_i}) = \mathcal{T}_{field}\ (\Pi, \overline{\beta}, \tau^{EH})$
  **in** $(\tau^{\nu,\delta}, \overline{\beta_i})$
$\mathcal{T}'\ (\Pi, \overline{\beta}, !\tau^{EH}) =$
  **let** $(\tau^{\nu,\delta}, \overline{\beta_i}) = \mathcal{T}_{field}\ (\Pi, \overline{\beta}, \tau^{EH})$
  **in** $(!\tau^{\nu,\delta}, \overline{\beta_i})$
```

Figure 5.5: Datatype annotation algorithm (strict data field)

In figure **??** the algorithm for data fields is given. It just runs the algorithm on the underlying type and add usage and demand annotation to the return type. Whether these annotations are variables or $\top$ depends on whether or not ι admits generating annotation variables.

τ^{EH} $\boxed{\mathcal{T}_{field}\ (\Pi, \overline{\beta}, \tau^{EH}) \rightsquigarrow (\tau^{\nu,\delta}, \overline{\beta})}$

```
$\mathcal{T}_{field}\ (\Pi, \overline{\beta}, \tau^{EH}) =$
  **let** $(\tau, \overline{\beta_i}) = \mathcal{T}\ (\Pi, \overline{\beta}, \tau^{EH}, 1)$
  **in**
    **if** $(0 < \iota)$ **then**
      **let**
        $\nu = fresh_\nu$
        $\delta = fresh_\delta$
      **in** $(\tau^{\nu,\delta}, [\nu, \delta] \cup \overline{\beta_i})$
    **else**
      $(\tau^{\top,\top}, [\,])$
```

Figure 5.6: Datatype annotation algorithm (data field)

In figure **??** the algorithm for types is given. This is where the most work is done. For variables nothing is to be done and the variable is just returned. This is the only case where the annotated type and the EH type are exactly the same. The case for (data)types is the most complex. For every type applied to the type constructor T the algorithm is run at one deeper level. Then we check whether the type constructor is defined in the current binding group or not. If it is in the data environment then it is not in the current binding group. If it is not in the current binding group we lookup the number of annotation variables for the type and generate the same amount of fresh annotation variables, or $\top$s if the current level does not allow the generation of variables anymore. If it is in the current binding group the input annotation list is used. It is converted to a list of $\top$s if the current level demands it. For type application the algorithm is applied recursively on the right side of the application with the level increased by one to signal that we get deeper into the type. For function types only the right component is annotated at one deeper level, the left component is annotated at the current level. This is to ensure that all the argument types of a function with multiple arguments are annotated at the same level. Annotation variables are generated when appropriate. For tuples each field is annotated at one deeper level and, when appropriate, a fresh usage and demand annotation is generated for each field.

5.1.4 Type expansion

Even though the annotation algorithm computes annotated data declarations this is not yet suitable for the analysis as there are still type synonyms present. These need to be expanded. Expansion is simply taking the annotated type declaration and substituting the annotations and type variables in the type for the applied annotations and types in the data declaration. This is done until no references to a type synonym are present in the data declarations. The type declarations however need to be preserved as

τ^{EH}

$\boxed{\mathcal{T}\ (\Pi, \overline{\beta}, \tau^{EH}, Int) \rightsquigarrow (\tau, \overline{\beta})}$

```
$\mathcal{T}\ (\Pi, \overline{\beta}, \alpha, level) = (\alpha, [\,])$
$\mathcal{T}\ (\Pi, \overline{\beta}, T\ \overline{\tau_i{}^{EH}}, level) =$
  let
    $\overline{(\tau_i, \overline{\beta_{ij}})} = [\mathcal{T}\ (\Pi, \overline{\beta}, \tau_i{}^{EH}, level + 1) \mid \tau_i{}^{EH} \leftarrow \overline{\tau_i{}^{EH}}]$
  in
    if $(T \in \Pi)$ then
      let
        $n$ = number of annotation variables for $T$ in $\Pi$
        $\overline{\beta_i}$ = replicate $n$ (if $level < \iota$ then $fresh_\varphi$ else $\top$)
      in $(T\ \overline{\tau_i}\ \overline{\beta_i}, (\overline{\beta_i} \cup (\bigcup_i (\bigcup_j \beta_{ij}))) - \{\top\})$
    else
      let $\overline{\beta_i}$ = [if $level < \iota$ then $\beta$ else $\top \mid \beta \leftarrow \overline{\beta}$]
      in $(T\ \overline{\tau_i}\ \overline{\beta_i}, \bigcup_i (\bigcup_j \beta_{ij}))$
$\mathcal{T}\ (\Pi, \overline{\beta}, \alpha\ \tau^{EH}, level) =$
    let $(\tau, \overline{\beta_i}) = \mathcal{T}\ (\Pi, \overline{\beta}, \tau^{EH}, level + 1)$
    in $(\alpha\ \tau, \overline{\beta_i})$
$\mathcal{T}\ (\Pi, \overline{\beta}, \tau_1{}^{EH} \rightarrow \tau_2{}^{EH}, level) =$
  let
    $(\tau_1, \overline{\beta_{1i}}) = \mathcal{T}\ (\Pi, \_\ \beta, \tau_1{}^{EH}, level + 1)$
    $(\tau_2, \overline{\beta_{2i}}) = \mathcal{T}\ (\Pi, \_\ \beta, \tau_2{}^{EH}, level)$
  in
    if $(level < \iota)$ then
      let
        $\nu_1 = fresh_\nu$
        $\delta = fresh_\delta$
        $\nu_2 = fresh_\nu$
      in $(\tau_1{}^{\nu_1, \delta} \rightarrow \tau_2{}^{\nu_2}, [\nu_1, \delta, \nu_2] \cup \overline{\beta_{1i}} \cup \overline{\beta_{2i}})$
    else
      $(\tau_1{}^{\top,\top} \rightarrow \tau_2{}^{\top}, [\,])$
$\mathcal{T}\ (\Pi, \overline{\beta}, (\tau_1{}^{EH}, \tau_2{}^{EH}, \cdots, \tau_n{}^{EH}), level) =$
  let
    $\overline{(\tau_i, \overline{\beta_{ij}})} = [\mathcal{T}\ (\Pi, \overline{\beta}, \tau_i{}^{EH}, level + 1) \mid \tau_i{}^{EH} \leftarrow \overline{\tau_i{}^{EH}}]$
  in
    if $(level < \iota)$ then
      let
        $\overline{\nu_i} = [fresh_\nu \mid \_ \leftarrow \overline{\tau_i}]$
        $\overline{\delta_i} = [fresh_\delta \mid \_ \leftarrow \overline{\tau_i}]$
      in $((\tau_1{}^{\nu_1, \delta_1}, \tau_2{}^{\nu_2, \delta_2}, \cdots, \tau_n{}^{\nu_n, \delta_n}), [\nu_1, \cdots \nu_n, \delta_1, \cdots, \delta_n] \cup (\bigcup_i \bigcup_j \beta_{ij}))$
    else
      $((\tau_1{}^{\top,\top}, \tau_2{}^{\top,\top}, \cdots, \tau_n{}^{\top,\top}), [\,])$
```

Figure 5.7: Datatype annotation algorithm (type)

other modules may use the type synonym there[9].

5.1.5 Examples

The examples here are divided in four different groups based on level of recursion. The examples in the first two groups are taken from [?]. All examples are annotated with a ι of two.

The first group in figure **??** has no recursion or only direct recursion and no function fields or nested types.

The second group in figure **??** has nested recursion, nested types and function fields.

The third group in figure **??** has mutual recursion, both direct and nested.

The last group in figure **??** has a mutual recursion with a type declaration. Type expansion is also shown here.

$\textbf{data}\ \mathit{Bool}\ [\,] = \mathit{True} \mid \mathit{False}$

$\textbf{data}\ [a]\ [\nu_1,\nu_2,\delta_1,\delta_2] = [\,] \mid a^{\nu_1,\delta_1} : ([a]\ [\nu_1,\nu_2,\delta_1,\delta_2])^{\nu_2,\delta_2}$

$\textbf{type}\ \mathit{String}\ [\nu_1,\nu_2,\delta_1,\delta_2] = [\mathit{Char}]\ [\nu_1,\nu_2,\delta_1,\delta_2]$

$\textbf{data}\ \mathit{Tree}\ a\ [\nu_1,\nu_2,\nu_3,\delta_1,\delta_2,\delta_3] = \mathit{Leaf}$

$\quad \mid \mathit{Node}\ (\mathit{Tree}\ a\ [\nu_1,\nu_2,\nu_3,\delta_1,\delta_2,\delta_3])^{\nu_1,\delta_1}\ a^{\nu_2,\delta_2}\ (\mathit{Tree}\ a\ [\nu_1,\nu_2,\nu_3,\delta_1,\delta_2,\delta_3])^{\nu_3,\delta_3}$

$\textbf{data}\ \mathit{Term}\ a\ [\nu_1,\cdots,\nu_5,\delta_1,\cdots,\delta_5] = \mathit{Var}\ a^{\nu_1,\delta_1}$

$\quad \mid \mathit{App}\ (\mathit{Term}\ a\ [\nu_1,\cdots,\nu_5,\delta_1,\cdots,\delta_5])^{\nu_2,\delta_2}\ (\mathit{Term}\ a\ [\nu_1,\cdots,\nu_5,\delta_1,\cdots,\delta_5])^{\nu_3,\delta_3}$

$\quad \mid \mathit{Lam}\ a^{\nu_4,\delta_4}\ (\mathit{Term}\ a\ [\nu_1,\cdots,\nu_5,\delta_1,\cdots,\delta_5])^{\nu_5,\delta_5}$

$\textbf{data}\ \mathit{Skew}\ [\,] = \mathit{SLeft} \mid \mathit{SNone} \mid \mathit{SRight}$

$\textbf{data}\ \mathit{AVLTree}\ a\ [\nu_1,\cdots,\nu_4,\delta_1,\cdots,\delta_4] = \mathit{ALeaf}$

$\quad \mid \mathit{ANode}\ (\mathit{AVLTree}\ a\ [\nu_1,\cdots,\nu_4,\delta_1,\cdots,\delta_4])^{\nu_1,\delta_1}\ a^{\nu_2,\delta_2}\ \mathit{Skew}^{\nu_3,\delta_3}\ (\mathit{AVLTree}\ a\ [\nu_1,\cdots,\nu_4,\delta_1,\cdots,\delta_4])^{\nu_4,\delta_4}$

Figure 5.8: Simple datatypes

$\textbf{data}\ \mathit{Customer}\ [\nu_1,\cdots,\nu_{11},\delta_1,\cdots,\delta_{11}] =$

$\quad \mathit{MkCustomer}\ (\mathit{Int}\ [\,])^{\nu_1,\delta_1}\ (\mathit{String}\ [\nu_6,\nu_7,\delta_6,\delta_7])^{\nu_2,\delta_2}\ (\mathit{String}\ [\nu_8,\nu_9,\delta_8,\delta_9])^{\nu_3,\delta_3}$

$\qquad ([(\mathit{String}\ [\top,\top,\top,\top])]\ [\nu_{10},\nu_{11},\delta_{10},\delta_{11}])^{\nu_4,\delta_4}\ \mathit{Bool}^{\nu_5,\delta_5}$

$\textbf{data}\ \mathit{Rose}\ a\ [\nu_1,\cdots,\nu_4,\delta_1,\cdots,\delta_4] = \mathit{RLeaf}\ a^{\nu_1,\delta_1} \mid \mathit{RNode}\ ([\mathit{Rose}\ a\ [\top,\top,\top,\top,\top,\top,\top,\top]]\ [\nu_3,\nu_4,\delta_3,\delta_4])^{\nu_2,\delta_2}$

$\textbf{data}\ R\ a\ b\ [\nu_1,\cdots,\nu_4,\delta_1,\cdots,\delta_3] = \mathit{R1}\ a^{\nu_1,\delta_1} \mid \mathit{R2}\ ((\mathit{Int}\ [\,])^{\nu_3,\delta_3} \to b^{\nu_4})^{\nu_2,\delta_2}$

Figure 5.9: Complex non mutual recursive datatypes

$\textbf{data}\ \mathit{Test}\ [\nu_1,\cdots,\nu_{16},\delta_1,\cdots,\delta_{16}] = \mathit{Test}\ (M_1\ [\nu_3,\cdots,\nu_9,\delta_3,\cdots,\delta_9])^{\nu_1,\delta_1}\ (M_2\ [\nu_{10},\cdots,\nu_{16},\delta_{10},\cdots,\delta_{16}])^{\nu_2,\delta_2}$

$\textbf{data}\ M_1\ [\nu_1,\cdots,\nu_7,\delta_1,\cdots,\delta_7] = \mathit{M11}\ (M_2\ [\nu_1,\cdots,\nu_7,\delta_1,\cdots,\delta_7])^{\nu_1,\delta_1}\ (\mathit{Int}\ [\,])^{\nu_2,\delta_2}$

$\textbf{data}\ M_2\ [\nu_1,\cdots,\nu_7,\delta_1,\cdots,\delta_7] = \mathit{M22}\ ([(M_3\ (\mathit{replicate}\ 14\ \top))]\ [\nu_4,\nu_5,\delta_4,\delta_5])^{\nu_3,\delta_3}$

$\textbf{data}\ M_3\ [\nu_1,\cdots,\nu_7,\delta_1,\cdots,\delta_7] = \mathit{M33}\ (M_1\ [\nu_1,\cdots,\nu_7,\delta_1,\cdots,\delta_7])^{\nu_6,\delta_5}\ (\mathit{Bool}\ [\,])^{\nu_7,\delta_7}$

Figure 5.10: Mutual recursive datatypes

$\textbf{type}\ \mathit{MT1}\ [\nu_1,\cdots,\nu_6,\delta_1,\cdots,\delta_6] = [(\mathit{MD1}\ (\mathit{replicate}\ 12\ \top))]\ [\nu_1,\nu_2,\delta_1,\delta_2]$

$\textbf{data}\ \mathit{MD1}\ [\nu_1,\cdots,\nu_6,\delta_1,\cdots,\delta_6] = \mathit{MD1}\ (\mathit{MT1}\ [\nu_1,\cdots,\nu_6,\delta_1,\cdots,\delta_6])^{\nu_3,\delta_3}\ (\mathit{String}\ [\nu_5,\nu_6,\delta_5,\delta_6])^{\nu_4,\delta_4}$

Figure 5.11: Mutual recursive datatype and type

[9]Meaning we keep the annotated type declarations inside the data environment even though no annotated data declaration references any of these anymore

5.2 Counting analysis

5.2.1 Constraint generation

The algorithm is presented in figure ?? and is based on the typing rules of figure ??. This is in no way a practical implementation and the real implementation in UHC is a heavily optimized worklist variant of the algorithm presented here. The algorithm has as input an expression and the import and export environments. It produces an usage annotated type and the local environment. It also produces an environment Ψ that holds the type of all the defined symbols. This environment will be used with the result of constraint solving to produce the final result. Even though it is an algorithm it leaves certain details unspecified. This includes the way fresh variables are generated and how certain information is obtained from child nodes.

The environment lookup function returns the type if it is in the environment otherwise a complete fresh usage and demand annotated type scheme is returned. This is different from the lookup used inside the definition of the constraint operators (section ??). The definition is given in figure ??. This function is in the algorithm never explicitly called but it is needed to implement the environment pattern matches made in the result of recursively calling $\mathcal{G}$. A desugared version of this pattern matching is presented in figure ??. The first line calls $\mathcal{G}$. The second line removes the variable form the environment. The final line does the lookup using the version defined in figure ?? and returns everything. This is how the algorithm works. In $\mathcal{G}$ the more readable version is used $(\Gamma, x : \sigma^{\nu,\delta}, \eta_\tau, C) = \mathcal{G}\ (\mathbf{e}, \Upsilon, \Delta)$ is used instead of the three lines defined here.

The VAR-1 and VAR-2 rules differ depending on whether or not it is a locally defined symbol or an imported symbol[10]. If it is a locally defined symbol (VAR-1) then fresh variables for the annotation, type and type scheme are created and an instantiation constraint is generated. Note the demand of $\mathbb{1}$ put in the environment. This demand is ignored in the APP rules but it is used in the right hand side of bindings. This is crucial in the LET! rule if $\mathbf{e_2}$ is a variable. If it is an imported symbol the type is looked up from the import environment. A fresh type variables is created and an instantiation constraint is generated. Nothing is put in the local environment as the use of the use of the symbol can never contribute to the type (or annotations) of the imported symbol. Imported symbols have always $\top$ as the usage annotation.

The CONST rule is trivial. A fresh usage annotation variable is created for the usage annotation and no constraints are generated.

The ABS rule only allows monomorphic arguments. The usage of all free variables in the body are multiplied by the usage of the abstraction itself.

The APP rule is different depending on whether it is application to a locally defined symbol, an imported symbol or a constant. When it is a locally defined symbol the function is used exactly once[11]. The demand of $\mathbb{1}$ on the variable is ignored and changed to the demand of the function argument. This is also the only place where the analysis dependent subeffecting rule is present. When it is an application to an imported symbol or to a constant the typing rules are simplified by the fact that the variable has an empty environment. In the rule for constant application the constraints for the constant can only contain a single subeffecting constraint as a normal constant derivation does not generate any constraints.

The LET rule is by far the most complex. However most antecedents are trivial. The first antecedent types the body of the let. The types for the bindings are allowed to be polymorphic type schemes. The second antecedent types each binding, but here the types for the bindings can only be monomorphic types. The next four antecedents are the definition for the environment, demand, usage and type respectively[12]. Then we have four antecedents which just unions the constraints from each binding. Next are the generalization constraints for each binding. The generated constraints that are the unions of the constraints generated per binding are put inside the generalization constraint and do not appear in the constraints generated for the let expression as a whole. And finally there is the antecedent that sets the demand and usage of each exported binding to $\top$.

The LET! rule is a cross over between the LET rule and the SEQ rule defined in [?]. The rule is basically a simplified version[13] of the LET rule, with two differences. The binding is typed with subeffecting setting the usage to $\mathbb{0}$ and $\mathbb{1}$ is added to δ to tell the system that the binding is demanded at least once[14]. The $\mathbb{0}$ comes from the fact that although the binding is evaluated (this is where the $\mathbb{1}$ comes from) to normal form it is not used. Subeffecting is used to ensure that it is still possible to use the binding inside the

[10] Note that we don't have to deal with shadowing as all names in UHC core are already resolved to fully qualified unique names.
[11] This uses subeffecting so it can potentially be used multiple times, but the application uses the function exactly once.
[12] The definitions for the usage and demand annotations are fixpoints. The ones for environment and types are folds.
[13] A *Let*! can only have a single binding.
[14] In [?] this is achieved by the $\mathbb{1}$ present inside the var rule.

body of the **let**. As δ_0 cannot contain a $\mathbb{0}$ all the $\delta_i \rhd$ can be removed from the rules for LET simplifying the rules even more.

The CON rule is for fully applied data types. First a completely fresh copy of the data type is generated and then each constructor field type is matched to the corresponding given argument type. Some care needs to be taken here to ensure that the demand in the environment for each argument matches the demand of the fresh data type.

The TUP rule is basically the same as the CON rule but without the need to create a fresh instance and without the complication of matching types and annotations correctly. This is because a tuple is fully polymorphic in each field.

The CASE rule is different based upon whether pattern matching is on a data type, tuple or constant. The pattern match must be complete[15]. The expression being pattern matched on is not used by the **case** expression. That is why the Γ_0 are not used in the rest of the code. Subeffecting is enable here so that if it is used in other places that is allowed by the generated constraints[16]. The rule for data types first creates a fresh copy of the data type. Each constructor field is assigned the type of the corresponding field in the data type. This rule mostly boils down to carefully matching the generated correct types and annotations. The rule for tuples is slightly simpler as there is only one alternative and no fresh copy needs to be generated[17]. The rule for constants is basically the same as for data types with the exception that now no fresh copy needs to be generated. It is also simpler as there are no fields present.

The FFI rule is nearly identical to the rule for imported symbols. The only difference is that the type comes from annotating the τ^{core}[18]. The function *annotate* will pessimistically[19] annotate the given type[20].

Environment lookup $\boxed{\Gamma\ (x) : \sigma^{\nu,\delta}}$

$$\Gamma\ (x) = \begin{cases} \sigma^{\nu,\delta} & \text{if } x : \sigma^{\nu,\delta} \in \Gamma \\ \mathit{fresh}_{\sigma}{}^{\mathit{fresh}_{\nu},\mathit{fresh}_{\delta}} & \text{otherwise} \end{cases}$$

Figure 5.12: Environment lookup

Environment pattern matching $\boxed{(\Gamma, x : \sigma^{\nu,\delta}, \tau^{\nu'}, C) = \mathcal{G}\ (\mathbf{e}, \Upsilon, \Delta)}$

$$\begin{array}{l} (\Gamma_1, \tau'^{\nu'}, C', \Psi') = \mathcal{G}\ (\mathbf{e}, \Upsilon, \Delta) \\ \Gamma_2 = \{x' : \sigma^{\nu,\delta} \mid (x' : \sigma^{\nu,\delta}) \in \Gamma_1, x' \not\equiv x\} \\ (\Gamma, \sigma^{\nu,\delta}, \tau^{\nu_1}, C, \Psi) = (\Gamma_2, \Gamma_1\ (x), \tau'^{\nu'}, C', \Psi) \end{array}$$

Figure 5.13: Environment pattern matching

5.2.2 Constraint solving

An algorithm to solve the constraints is presented in figure **??**. The algorithm has as input the constraint to solve[21], the solution[22] ϕ that is computed up to this point and a partial solution ψ constraining the solution for annotation variables[23]. It returns an updated solution and partial solution and the constraints to be solved later.

A common pattern is that most rules do not update the solution at all. Instead they generate equality constraints and let the rule for equality figure out how to put it in the solution.

Except for the equality rules no variable in the constraint is bound by the solution. This is enforced in the rules for $C_1 \cup C_2$ and $\mathcal{S}_{fix}$.

[15]Every constructor must match. Also there must be at least one constructor for a data type. Things like **data** *Void* without any constructors are not supported

[16]Without subeffecting here the expression would not be allowed to be used anywhere.

[17]It can be done but as this than boils down to making all the types and annotations in the tuple fresh variables which then needs to be equal to other things it is just simpler to omit this and not generate a bunch of equality constraints

[18]τ^{core} is basically the same as $\_\ \tau\ \mathit{EH}$. The only difference is that the subset of τ^{core} allowed in an *FFI* is monomorphic.

[19]In other words all annotations will be $\top$

[20]This basically is equal to $\mathcal{T}$ (figure **??**) with an ι of zero

[21]This can be a compound constraint or a single constraint

[22]A solution is a mapping from variables to types, annotations and schemes

[23]This is a mapping from annotation variables to sets of annotation values.

Algorithm $\boxed{\mathcal{G}\ (\mathbf{e}, \Upsilon, \Delta) \leadsto (\Gamma, \eta_\tau, C, \Psi)}$

(a) Variables and constants

```
$\mathcal{G}\ (x, \Upsilon, \Delta) \mid x \notin \Upsilon =$
  let
    $\tau = fresh_\tau$
    $\sigma = fresh_\sigma$
    $\nu = fresh_\nu$
  in $(x : \sigma^{\nu,\mathbb{1}}, \tau^{\nu}, \{inst\ (\sigma) \equiv \tau\}, \epsilon)$
$\mathcal{G}\ (x, \Upsilon, \Delta) \mid x \in \Upsilon =$
  let
    $\tau = fresh_\tau$
    $\sigma = \Upsilon\ (x)$
  in $(\epsilon, \tau^{\top}, \{inst\ (\sigma) \equiv \tau\}, \epsilon)$
$\mathcal{G}\ (\mathbf{c}, \Upsilon, \Delta) =$
  let
    $\tau = lookupType\ \mathbf{c}$
    $\nu = fresh_\nu$
  in $(\epsilon, \tau^{\nu}, \emptyset, \epsilon)$
```

(b) Abstraction and application

```
$\mathcal{G}\ (\lambda x \to \mathbf{e}, \Upsilon, \Delta) =$
  let
    $(\Gamma_1, x : \sigma^{\nu,\delta}, \eta, C_1, \Psi') = \mathcal{G}\ (\mathbf{e}, \Upsilon, \Delta)$
    $\nu_2 = fresh_\nu$
    $\tau = fresh_\tau$
    $(\Gamma_2, C_2) = \nu_2 \cdot \Gamma_1$
    $\Psi = \Psi' \cup \{x : \sigma^{\nu,\delta}\}$
  in $(\Gamma_2, (\tau^{\nu,\delta} \to \eta)^{\nu_2}, C_1 \cup C_2 \cup \{\sigma \equiv (\forall \emptyset\, \emptyset\, .\ \emptyset \Rightarrow \tau)\}, \Psi)$
$\mathcal{G}\ (\mathbf{e}\ x, \Upsilon, \Delta) \mid x \notin \Upsilon =$
  let
    $\delta_2 = fresh_\delta$
    $(x : eta4^{\mathbb{1}}, eta2', C_2, \_) = \mathcal{G}\ (x, \Upsilon, \Delta)$
    $\eta_3 = fresh_{\eta_\tau}$
    $(\eta_2, C_2') = sub_\diamond\ eta2'$
    $(\Gamma_1, eta', C_1, \Psi) = \mathcal{G}\ (\mathbf{e}, \Upsilon, \Delta)$
    $(\tau_1{}^{\nu_1}, C_1') = sub_\sqsubseteq\ eta'$
    $(\Gamma_2, C_3) = \Gamma_1 \oplus x : eta4^{\delta_2}$
  in $(\Gamma_2, \eta_3, C_1 \cup C_1' \cup C_2 \cup C_2' \cup C_3 \cup \{\tau_1 \equiv (\eta_2{}^{\delta_2} \to \eta_3), \nu_1 \equiv \mathbb{1}\}, \Psi)$
$\mathcal{G}\ (\mathbf{e}\ x, \Upsilon, \Delta) \mid x \in \Upsilon =$
  let
    $\delta_2 = fresh_\delta$
    $(\epsilon, eta2', C_2, \_) = \mathcal{G}\ (x, \Upsilon, \Delta)$
    $\eta_3 = fresh_{\eta_\tau}$
    $(\eta_2, C_2') = sub_\diamond\ eta2'$
    $(\Gamma, eta', C_1, \Psi) = \mathcal{G}\ (\mathbf{e}, \Upsilon, \Delta)$
    $(\tau_1{}^{\nu_1}, C_1') = sub_\sqsubseteq\ eta'$
  in $(\Gamma, \eta_3, C_1 \cup C_1' \cup C_2 \cup C_2' \cup \{\tau_1 \equiv (\eta_2{}^{\delta_2} \to \eta_3), \nu_1 \equiv \mathbb{1}\}, \Psi)$
$\mathcal{G}\ (\mathbf{e}\ \mathbf{c}, \Upsilon, \Delta) \mid x \in \Upsilon =$
  let
    $\delta_2 = fresh_\delta$
    $(\epsilon, eta2', \emptyset, \_) = \mathcal{G}\ (\mathbf{c}, \Upsilon, \Delta)$
    $\eta_3 = fresh_{\eta_\tau}$
    $(\eta_2, C_2) = sub_\diamond\ eta2'$
    $(\Gamma, eta', C_1, \Psi) = \mathcal{G}\ (\mathbf{e}, \Upsilon, \Delta)$
    $(\tau_1{}^{\nu_1}, C_1') = sub_\sqsubseteq\ eta'$
  in $(\Gamma, \eta_3, C_1 \cup C_1' \cup C_2 \cup \{\tau_1 \equiv (\eta_2{}^{\delta_2} \to \eta_3), \nu_1 \equiv \mathbb{1}\}, \Psi)$
```

Figure 5.14: Constraint generation

Algorithm $\boxed{\mathcal{G}\ (\mathbf{e}, \Upsilon, \Delta) \rightsquigarrow (\Gamma, \eta_\tau, C, \Psi)}$

(c) Let and Let!

```
$\mathcal{G}\ (\mathbf{let}\ \overline{x_i = \mathbf{e_i}}\ \mathbf{in}\ \mathbf{e}, \Upsilon, \Delta) =$
  $\mathbf{let}$
    $(\Gamma_0, \overline{x_i : {\sigma_i}^{\nu_i, \delta_i}}, \eta, C_0, \Psi_0) = \mathcal{G}\ (\mathbf{e}, \Upsilon, \Delta)$
    $\overline{\overline{\tau_{ij}}} = [[fresh_\tau \mid \_ \leftarrow \overline{x_j}] \mid \_ \leftarrow \overline{\mathbf{e_i}}]$
    $\overline{\overline{\sigma_{ij}}} = [[\forall \emptyset\, \emptyset\, .\ \emptyset \Rightarrow \tau'_{ij} \mid \tau'_{ij} \leftarrow \overline{\tau'_{ij}}] \mid \overline{\tau'_{ij}} \leftarrow \overline{\overline{\tau_{ij}}}]$
    $\overline{(\Gamma_i, \overline{x_j : {\sigma'_{ij}}^{\nu_{ij}, \delta_{ij}}}, {\tau_i}^{\nu_{xi}}, C_{1i}, \Psi_i)} = [\mathcal{G}\ (x, \Upsilon, \Delta) \mid x \leftarrow \overline{\mathbf{e_i}}]$
    $C'_0 = \{\overline{\nu_i} \equiv \overline{\nu_{xi}}\}$
    $C'_1 = \{\overline{\overline{\sigma_{ij}}} \equiv \overline{\overline{\sigma'_{ij}}}\}$
    $C_1 = C'_1 \cup (\bigcup_i C_{1i})$
    $\overline{(\Gamma'_i, C_{2i})} = [di \triangleright \Gamma_x \mid (\Gamma_x, di) \leftarrow \overline{(\Gamma_i, \delta_i)}]$
    $(\Gamma', C_2) = \bigoplus_i \Gamma'_i$
    $(\Gamma, C'_2) = \Gamma_0 \oplus \Gamma'$
    $\overline{\overline{(\delta'_{ij}, C_{3ij})}} = [[di \triangleright \delta \mid (\delta, di) \leftarrow \overline{(\delta, di)}] \mid \overline{(\delta, di)} \leftarrow \overline{\overline{(\delta_{ij}, \delta_i)}}]$
    $\overline{(\delta'_i, C_{3i})} = [\bigoplus_j \delta_{ij} \mid \overline{\delta_{ij}} \leftarrow \overline{\overline{\delta'_{ij}}}]$
    $\overline{(\delta_{0i}, C'_{3i})} = [di \oplus \delta \mid (\delta, di) \leftarrow \overline{(\delta'_i, \delta_i)}]$
    $C'_3 = \overline{\delta_{0i}} \equiv \overline{\delta_i}$
    $C_3 = C'_3 \cup (\bigcup_i (\bigcup_j C_{3ij})) \cup (\bigcup_i C_{3i}) \cup (\bigcup_i C'_{3i})$
    $\overline{\overline{(\nu'_{ij}, C_{4ij})}} = [[di \triangleright \nu \mid (\nu, di) \leftarrow \overline{\nu}] \mid \overline{(\nu, di)} \leftarrow \overline{\overline{(\nu_{ij}, \delta_i)}}]$
    $\overline{(\nu'_i, C_{4i})} = [\bigoplus_j \nu_{ij} \mid \overline{\nu_{ij}} \leftarrow \overline{\overline{\nu'_{ij}}}]$
    $\overline{(\nu_{0i}, C'_{4i})} = [n_i \oplus \nu \mid (\nu, n_i) \leftarrow \overline{(\nu'_i, \nu_i)}]$
    $C'_4 = \overline{\nu_{0i}} \equiv \overline{\nu_i}$
    $C_4 = C'_4 \cup (\bigcup_i (j \cup C_{4ij})) \cup (\bigcup_i C_{4i}) \cup (\bigcup_i C'_{4i})$
    $\overline{\overline{(\tau'_{ij}, C_{5ij})}} = [[di \triangleright \tau \mid (\tau, di) \leftarrow \overline{\tau}] \mid \overline{(\tau, di)} \leftarrow \overline{\overline{(\tau_{ij}, \delta_i)}}]$
    $\overline{(\tau'_i, C_{5i})} = [\bigoplus_j \tau_{ij} \mid \overline{\tau_{ij}} \leftarrow \overline{\overline{\tau'_{ij}}}]$
    $C'_5 = \overline{\tau'_i} \equiv \overline{\tau_i}$
    $C_5 = C'_5 \cup (\bigcup_i (j \cup C_{5ij})) \cup (\bigcup_i C_{5i})$
    $C_6 = \bigcup_i \{gen\ ({\tau_i}^{\nu_i, \delta_i}, C_1 \cup C_3 \cup C_4 \cup C_5, \Gamma) \equiv \sigma_i\}$
    $\overline{C_{7i}} = [\mathbf{if}\ x \in \Delta\ \mathbf{then}\ \{\nu \equiv \top, \delta \equiv \top\}\ \mathbf{else}\ \emptyset \mid (x, \nu, \delta) \leftarrow \overline{(x_i, \nu_i, \delta_i)}]$
    $\Psi = \Psi_0 \cup (\bigcup_i \Psi_i) \cup (\bigcup_i \{x_i : {\sigma_i}^{\nu_i, \delta_i}\})$
  $\mathbf{in}\ (\Gamma, \eta, C_0 \cup C'_0 \cup C_2 \cup C'_2 \cup (\bigcup_i C_{2i}) \cup C_6 \cup (\bigcup_i C_{7i}), \Psi)$
$\mathcal{G}\ (\mathbf{let}\ !\ x = \mathbf{e_1}\ \mathbf{in}\ \mathbf{e_2}, \Upsilon, \Delta) =$
  $\mathbf{let}$
    $(\Gamma_0, x : {\sigma_0}^{\nu, \delta}, \eta, C_0, \Psi_1) = \mathcal{G}\ (\mathbf{e_2}, \Upsilon, \Delta)$
    $(\Gamma_1, x : {\sigma'}^{\nu_1, \delta_1}, \tau^{\nu_2}, C_1, \Psi_2) = \mathcal{G}\ (\mathbf{e_1}, \Upsilon, \Delta)$
    $C'_1 = \{\sigma' \equiv \forall \emptyset\, \emptyset\, .\ \emptyset \Rightarrow \tau, \mathbb{0} \sqsubseteq \nu_2\}$
    $(\Gamma, C_2) = \Gamma_0 \oplus \Gamma_1$
    $(\delta_0, C_3) = \mathbb{1} \oplus \delta$
    $(\delta', C_4) = \delta_0 \oplus \delta_1$
    $C'_4 = \{\delta_0 \equiv \delta'\}$
    $(\nu', C_5) = \nu \oplus \nu_1$
    $C'_5 = \{\nu \equiv \nu'\}$
    $C_6 = \{gen\ (\tau^{\nu, \delta_0}, C_1 \cup C'_1 \cup C_3 \cup C_4 \cup C'_4 \cup C_5 \cup C'_5, \Gamma) \equiv \sigma_0\}$
    $\Psi = \Psi_1 \cup \Psi_2 \cup \{x : {\sigma_0}^{\nu, \delta}\}$
  $\mathbf{in}\ (\Gamma, \eta, C_0 \cup C_2 \cup C_6, \Psi)$
```

Figure 5.14: Constraint generation

The solving rules are repeated until no change in either one of the solution or in the constraint are happening. This is presented in figure ??. It can be the case that the constraint and the partial solution are not empty after running the fixpoint iteration. In that case some defaulting needs to happen to solve the final constrains. This is discussed in section ??. This defaulting cannot be built into $\mathcal{S}_{fix}$ as defaulting should not happen when this is called during the solving of generalization constraints.

Algorithm $\boxed{\mathcal{G}\ (\mathbf{e}, \Upsilon, \Delta) \rightsquigarrow (\Gamma, \eta_\tau, C, \Psi)}$

(d) Data types

```
$\mathcal{G}\ (K_i\ \overline{v_j}, \Upsilon, \Delta) =$
  let
    $T\ \overline{u_l}\ \overline{\alpha_k} = lookupData\ K_i$
    $\overline{rhoj} = lookupDataCon\ K_i$
    $\overline{\varphi_l} = [fresh_\varphi \mid \_ \leftarrow \overline{u_l}]$
    $\overline{\tau_k} = [fresh_\tau \mid \_ \leftarrow \overline{\alpha_k}]$
    $\overline{rhoj'} = \overline{rhoj}\ [^{\overline{\varphi_l}}/_{\overline{u_l}}, ^{\overline{\tau_k}}/_{\overline{\alpha_k}}]$
    $(\Gamma, \overline{rho'}, C_1, \Psi) = \mathcal{H}\ (\overline{v_i}, \Upsilon, \Delta)$
    $C_2 = \overline{rhoj'} \equiv \overline{rho'}$
    $\nu = fresh_\nu$
  in $(\Gamma, (T\ \overline{\varphi_l}\ \overline{\tau_k})^\nu, C_1 \cup C_2, \Psi)$
$\mathcal{G}\ ((v_1, v_2, \cdots, v_n), \Upsilon, \Delta) = \mathcal{H}\ (\overline{v_i}, \Upsilon, \Delta)$
$\mathcal{G}\ (\mathbf{case\ e\ of}\ \overline{K_i\ \overline{x_{ij}} \rightarrow \mathbf{e_i}})$
  let
    $T\ \overline{u_l}\ \overline{\alpha_k} = lookupData\ K_1$
    $\overline{\overline{rhoij}_\tau} = lookupDataCons\ \overline{K_i}$
    $\overline{\overline{rhoij'}_\tau} = \overline{\overline{rhoij}_\tau}\ [^{\overline{\varphi_l}}/_{\overline{u_l}}, ^{\overline{\tau_k}}/_{\overline{\alpha_k}}]$
    $\overline{\overline{rhoij}_\sigma} = [[(\forall \emptyset\ \emptyset\ .\ \emptyset \Rightarrow \tau_{ij})^{\nu_{ij}, \delta_{ij}} \mid {\tau_{ij}}^{\nu_{ij}, \delta_{ij}} \leftarrow \overline{rhoij}] \mid \overline{rhoij} \leftarrow \overline{\overline{rhoij'}_\tau}]$
    $\overline{\varphi_l} = [fresh_\varphi \mid \_ \leftarrow \overline{u_l}]$
    $\overline{\tau_k} = [fresh_\tau \mid \_ \leftarrow \overline{\alpha_k}]$
    $(\Gamma_0, {\tau_0}^{\nu_0}, C_1, \Psi') = \mathcal{G}\ (\mathbf{e}, \Upsilon, \Delta)$
    $C_1' = \{\tau_0 \equiv T\ \overline{\varphi_l}\ \overline{\tau_k}, \mathbb{1} \sqsubseteq \nu_0\}$
    $\overline{(\Gamma_i, \overline{x_{ij} : rhoij'}, etai, C_{2i}, \Psi_i)} = [\mathcal{G}\ (x, \Upsilon, \Delta) \mid x \leftarrow \overline{\mathbf{e_i}}]$
    $C_2 = allEqual\ \overline{etai}$
    $C_2' = \overline{\overline{rhoij}_\sigma} \equiv \overline{\overline{rhoij'}}$
    $(\Gamma, C_3) = \bigsqcup i\ \Gamma_i$
    $\Psi = \Psi' \cup (\bigcup_i \Psi_i) \cup (\bigcup_i (\bigcup_j \{x_{ij} : rhoij'\}))$
  in $(\Gamma, \eta_1, C_1 \cup C_1' \cup C_2 \cup C_2' \cup (\bigcup_i C_{2i}) \cup C_3 \cup C_3', \Psi)$
$\mathcal{G}\ (\mathbf{case\ e\ of}\ (x_1, x_2, \cdots, x_n), \Upsilon, \Delta) =$
  let
    $\overline{rhoi_\tau} = replicate\ n\ ({fresh_\tau}^{fresh_\nu, fresh_\delta})$
    $\overline{rhoi_\sigma} = [(\forall \emptyset\ \emptyset\ .\ \emptyset \Rightarrow \tau_i)^{\nu_i, \delta_i} \mid {\tau_i}^{\nu_i, \delta_i} \leftarrow \overline{rhoi_\tau}]$
    $(\Gamma_0, {\tau_0}^{\nu_0}, C_1, \Psi_1) = \mathcal{G}\ (\mathbf{e}, \Upsilon, \Delta)$
    $C_1' = \{\tau_0 \equiv \overline{rhoi}, \mathbb{1} \sqsubseteq \nu_0\}$
    $(\Gamma, \overline{x_i : rhoi'}, \eta, C_2, \Psi_2) = \mathcal{G}\ (\mathbf{e_1}, \Upsilon, \Delta)$
    $C_2' = \overline{rhoi_\sigma} \equiv \overline{rhoi'}$
    $\Psi = \Psi_1 \cup \Psi_2 \cup (\bigcup_i \{x_i : rhoi'\})$
  in $(\Gamma, \eta, C_1 \cup C_1' \cup C_2 \cup C_2' \cup C_3, \Psi)$
$\mathcal{G}\ (\mathbf{case\ e\ of}\ \overline{\mathbf{c} \rightarrow \mathbf{e_i}}, \Upsilon, \Delta) =$
  let
    $\tau = lookupType\ \mathbf{c_1}$
    $(\Gamma_0, {\tau_0}^{\nu_0}, C_1, \Psi') = \mathcal{G}\ (\mathbf{e}, \Upsilon, \Delta)$
    $C_1' = \{\tau_0 \equiv \tau, \mathbb{1} \sqsubseteq \nu_0\}$
    $\overline{(\Gamma_i, etai, C_{2i}, \Psi_i)} = [\mathcal{G}\ (x, \Upsilon, \Delta) \mid x \leftarrow \overline{\mathbf{e_i}}]$
    $C_2 = allEqual\ \overline{etai}$
    $(\Gamma, C_3) = \bigsqcup i\ \Gamma_i$
  in $(\Gamma, \eta_1, C_1 \cup C_1' \cup C_2 \cup (\bigcup_i C_{2i}) \cup C_3 \cup C_3', \Psi' \cup (\bigcup_i \Psi_i))$
```

Figure 5.14: Constraint generation

In figure **??** the rules for the compound constraint are given. The rules for the empty constraint is trivial. When solving the union of two constraints, the first constraint is solved and the resulting solution is applied to the second constraint before it is solved. The resulting solutions are returned and the left over constraints are combined[24].

[24]It can be the case that one of these is the empty constraint. For the correctness this does not matter. For efficiency this does matter so in any real implementation you would filter these out.

Algorithm $\boxed{\mathcal{G}\ (\mathbf{e}, \Upsilon, \Delta) \rightsquigarrow (\Gamma, \eta_\tau, C, \Psi)}$

(e) Foreign function call

```
$\mathcal{G}\ (FFI\ n\ \tau^{core}, \Upsilon, \Delta) =$
  let
    $\tau = fresh_\tau$
    $\sigma = annotate\ \tau^{core}$
  in $(\epsilon, \tau^\top, \{inst\ (\sigma) \equiv \tau\}, \epsilon)$
```

(f) Algorithm helpers

Lists $\boxed{\mathcal{H}\ (\overline{v_i}, \Upsilon, \Delta) \rightsquigarrow (\Gamma, rho_\tau, C, \Psi)}$

```
$\mathcal{H}\ (\overline{v_i}, \Upsilon, \Delta) =$
  let
    $\overline{(\Gamma_i, etai, C_i, \Psi_i)} = [\mathcal{G}\ (v, \Upsilon, \Delta) \mid v \leftarrow \overline{v_i}]$
    $(\Gamma, C') = \bigoplus_i \Gamma_i$
    $C = \bigcup_i C_i$
    $\overline{rhoi} = [\eta^{(getDemand\ \Gamma')} \mid (\eta, \Gamma') \leftarrow \overline{(etai, \Gamma_i)}]$
  in $(\Gamma, \overline{rhoi}, C \cup C', \bigcup_i \Psi_i)$
```

getDemand $\boxed{\Gamma \rightsquigarrow \delta}$

```
$getDemand\ (x : \eta^\delta) = \delta$
$getDemand\ (\Gamma) = fresh_\delta$
```

allEqual $\boxed{\overline{x} \rightsquigarrow C}$

```
$allEqual\ (x : xs@(y : \_)) = \mathbf{let}\ C = allEqual\ xs\ \mathbf{in}\ C \cup \{x \equiv y\}$
$allEqual\ \_ = \emptyset$
```

replicate $\boxed{(n, x) \rightsquigarrow \overline{x}}$

```
$replicate\ n\ x$
  $\mid n > 0 = \mathbf{let}\ xs = replicate\ (n-1)\ x\ \mathbf{in}\ x : xs$
  $\mid otherwise = [\,]$
```

$sub_\diamond$ $\boxed{\eta \rightsquigarrow (\eta, C)}$

```
$sub_\diamond\ (\tau^\nu) =$
  let $\nu' = fresh_\nu$
  in $(\tau^{\nu'}, \{\nu' \diamond \nu\})$
```

Figure 5.14: Constraint generation

The rules for equality are presented in figure ??. This the only place where the solution is updated except for the annotation solving rules. The rules for when the two things are the same are trivial[25]. When they are not the same and the constraint contains variables a check is made to see if these variables are already present in the environment. If that is the case solving continues with the found value. If no variable is already present in the solution then it is added to the solution.

The rules for annotation constraints[26] are given in figure ??. When the right hand side contains only annotation values the constraint can be solved by computing using the definition. In these rules $\diamond$ stands for any of the four operators[27]. If there is a variable in the right hand side then solving proceeds as follows:

1. For each variable retrieve the possible values it can have from ψ
2. Instantiate the constraint in all possible ways with the values retrieved

[25] There are two versions, one for when both are variables and one for when both are non variables
[26] No code is given here as it does not help to understand the function. A complex version is given in appendix ??
[27] These are $\oplus$, $\sqcup$, $\cdot$ and $\triangleright$

3. Remove all the invalid constraints
4. Limit the values of each variable to the values it has in the valid constraints and put these into ψ as the new partial solution for these variables
5. Move any variable that now only has a single solution into ϕ and remove it from ψ
6. See if there are two or more variables always equal by testing whether the annotation values are always the same in the valid constraints. If there are, generate equality constraints between them.
7. Return the new ϕ, ψ and the union of the generated equality constraints of the previous step and the annotation constraint[28].

Normally the constraint is returned as is except in the following case:

- When there is only a single variable present and the possible solutions after solving is equal to all possible annotation values except $\bot$ then the constraint did not constrain the variable at all and the constraint is not returned for additional solving.

The rules for generalization and instantiation are given in figure ??. Generalization starts by solving the constraints inside the generalization constraint. This uses the fixpoint solving to ensure the constraints are solved as much as possible. It then applies the new solution to the remaining parts of the constraint[29]. It determines the variables over which to generalize and returns the new solutions together with an equality constraint between the scheme in the constraint and the computed scheme. Instantiation simply replaces every quantified variable with a fresh variable and returns the constraints inside the scheme (with the fresh variables). If the scheme is still a variable no solving can happen and the constraint is simply returned to be dealt with later.

5.2.3 Defaulting

When after solving the partial solution is not empty a value needs to be chosen for these remaining unsolved variables. Defaulting takes care of selecting a single variable and value. This variable is then removed from ψ and added to ϕ with the selected value and solving continues on. So defaulting takes care of forcing the solving to finish with a full solution. Defaulting selects the variable which has the best value as possible value, where best is defined in the following order:

1. $\mathbb{0}$: This represents this is not used at all and symbols with a demand of $\mathbb{0}$ can be removed.
2. $\mathbb{1}$: Choosing this enables both strictness and sharing optimizations
3. $\{1, \infty\}$: Arbitrarily chosen that knowing something is strict is better than knowing that something is not shared
4. ω: Also strict
5. $\{0, 1\}$: Not shared
6. $\{0, \infty\}$: Basically as useless as $\top$ but it is more precise
7. $\top$: Will never be chosen as in that case there was only a single value possible and that would not be present in ψ to begin with

Valuing enabling strictness optimizations over enabling sharing optimizations is purely arbitrarily and depends upon the actual optimizations happening. It is for future work to see which ordering is better.

The defaulting algorithm is made more precise in figure ??. It uses the ordering defined above.

5.2.4 Combining generation, solving and defaulting

Now that constraint generation, constraint solving and defaulting are defined we can give a final algorithm that computes the types of all defined symbols in the expression. It is presented in figure ??. It gets as input the expression and the import and export environments. It starts by running the constraint generation algorithm, followed by the solving with defaulting algorithm. It then applies the solution to the Ψ from the generation and returns the result.

The solving with defaulting is defined in a lazy way. In a strict version the $\mathcal{F}$ call needs to be inlined inside the then branch. It also uses the fact that whenever ψ is empty the returned constraint is also empty.

[28]The order here is important for efficiency reasons. Solving first the equality constraint and then resolve the annotation constraint prevents unnecessary work and duplicate constraints generated

[29]The returned constraint already has the solution applied to it

Solving $\boxed{\mathcal{S}\ (C, \phi, \psi) \rightsquigarrow (\phi, \psi, C)}$

(a) Basics

```
$\mathcal{S}\ (\emptyset, \phi, \psi) = (\phi, \psi, \emptyset)$
$\mathcal{S}\ (C_1 \cup C_2, \phi, \psi) =$
  let
    $(\phi_1, \psi_1, C_1') = \mathcal{S}\ (C_1, \phi, \psi)$
    $C_2' = C_2\ [\phi_1]$
    $(\phi_2, \psi_2, C_3) = \mathcal{S}\ (C_2', \phi_1, \psi_1)$
  in $(\phi_2, \psi_2, C_1' \cup C_3)$
```

(b) Equality

```
$\mathcal{S}\ (x_1 \equiv x_2, \phi, \psi) =$
  if $(x_1 \equiv x_2)$ then
    $(\phi, \psi, \emptyset)$
  else if $((x_1 : mu') \in \phi)$ then
    $\mathcal{S}\ (x_2 \equiv mu', \phi, \psi)$
  else if $((x_2 : mu') \in \phi)$ then
    $\mathcal{S}\ (x_1 \equiv mu', \phi, \psi)$
  else
    $(\phi; x_1 : x_2, \psi, \emptyset)$
$\mathcal{S}\ (x \equiv \mu, \phi, \psi) \mid !isvar\ \mu =$
  if $((x : mu') \in \phi)$ then
    $\mathcal{S}\ (\mu \equiv mu', \phi, \psi)$
  else
    $(\phi; x : \mu, \psi, \emptyset)$
$\mathcal{S}\ (\mu \equiv x, \phi, \psi) \mid !isvar\ \mu = \mathcal{S}\ (x \equiv \mu, \phi, \psi)$
$\mathcal{S}\ (\mu_1 \equiv \mu_2, \phi, \psi) \mid !isvar\ \mu_1 \wedge !isvar\ \mu_2 =$
  if $(\mu_1 \not\equiv \mu_2)$ then
    error "Unsatisfiable constraint"
  else
    $(\phi, \psi, \emptyset)$
```

(c) Annotation

```
$\mathcal{S}\ (\varphi \equiv \varpi_1 \diamond \varpi_2, \phi, \psi) =$
  let
    $\varpi = \varpi_1 \diamond \varpi_2$
  in $\mathcal{S}\ (\varphi \equiv \varpi)$
$\mathcal{S}\ (\varphi_3 \equiv \varphi_1 \diamond \varphi_2, \phi, \psi) \mid isvar\ \varphi_1 \vee isvar\ \varphi_2 =$
"(code is not provided)"
```

Figure 5.15: Constraint solving

Solving $\boxed{\mathcal{S}\ (C, \phi, \psi) \rightsquigarrow (\phi, \psi, C)}$

(d) Generalization and instantiation

```
$\mathcal{S}\ (gen\ (\tau^{\nu,\delta}, C_1, \Gamma) \equiv \sigma, \phi, \psi) =$
  **let**
    $(\phi_1, \psi_1, C_2) = \mathcal{S}_{fix}\ (C_1, \phi, \psi)$
    $\tau' = \tau\ [\phi_1]$
    $\Gamma' = \Gamma\ [\phi_1]$
    $\nu' = \nu\ [\phi_1]$
    $\delta' = \delta\ [\phi_1]$
    $V_\alpha = ((ftv\ C_2) \cup (ftv\ \tau')) - (ftv\ \Gamma')$
    $V_\beta = ((fav\ C_2) \cup (fav\ \tau')) - ((fav\ \Gamma') \cup (fav\ \nu') \cup (fav\ \delta'))$
  **in** $(\phi_1, \psi_1, \sigma \equiv (\forall V_\alpha\ V_\beta\ .\ C_2\ \tau'))$
$\mathcal{S}\ (inst\ (\forall \overline{\alpha_1}\ \overline{\beta_1}\ .\ C \Rightarrow \tau_1) \equiv \tau_2, \phi, \psi) =$
  **let**
    $\overline{\alpha_2} = [fresh_\tau \mid \_ \leftarrow \overline{\alpha_1}]$
    $\overline{\beta_2} = [fresh_\varphi \mid \_ \leftarrow \overline{\beta_1}]$
    $C' = C\ [^{\overline{\alpha_2}}/_{\overline{\alpha_1}}, ^{\overline{\beta_2}}/_{\overline{\beta_1}}]$
    $\tau' = \tau\ [^{\overline{\alpha_2}}/_{\overline{\alpha_1}}, ^{\overline{\beta_2}}/_{\overline{\beta_1}}]$
  **in** $(\phi, \psi, C' \cup \{\tau' \equiv \tau_2\})$
$\mathcal{S}\ (inst\ (\sigma) \equiv \tau), \phi, \psi) = (\phi, \psi, inst\ (\sigma) \equiv \tau)$
```

(e) Fixpoint

Solving $\boxed{\mathcal{S}_{fix}\ (C, \phi, \psi) \rightsquigarrow (\phi, \psi, C)}$

```
$\mathcal{S}_{fix}\ (C, \phi, \psi) =$
  **let**
    $C' = C\ [\phi]$
    $(\phi_1, \psi_1, C_1) = \mathcal{S}\ (C', \phi, \psi)$
    $changed = C_1 \not\equiv C' \vee \phi_1 \not\equiv \phi \vee \psi_1 \not\equiv \psi$
  **in if** $changed$ **then** $\mathcal{S}_{fix}\ (C_1, \phi_1, \psi_1)$ **else** $(\phi_1, \psi_1, C_1)$
```

Figure 5.15: Constraint solving

Defaulting $\boxed{\mathcal{F}\ (\psi) \rightsquigarrow (\beta, \varpi)}$

$$\mathcal{F}\ ((\beta : \overline{\varpi_i}) : \psi) = \mathcal{F}\ (\psi, \beta, max\ \overline{\varpi_i})$$

Defaulting $\boxed{\mathcal{F}\ (\psi, \beta, \varpi) \rightsquigarrow (\beta, \varpi)}$

```
$\mathcal{F}\ (\epsilon, \beta, \varpi) = (y, \varpi)$
$\mathcal{F}\ ((\beta_1 : \overline{\varpi_i}) : \psi, \beta_2, \varpi_2) =$
  **let** $\varpi_1 = max\ \overline{\varpi_i}$
  **in if** $(\varpi_2 < \varpi_1)$ **then**
      $\mathcal{F}\ (\psi, \beta_1, \varpi_1)$
    **else**
      $\mathcal{F}\ (\psi, \beta_2, \varpi)$
```

Figure 5.16: Defaulting

Annotation algorithm $\boxed{\mathcal{A}\ (\mathbf{e}, \Upsilon, \Delta) \rightsquigarrow \Psi}$

```
$\mathcal{A}\ (\mathbf{e}, \Upsilon, \Delta) =$
    **let**
        $(\_, \_, C, \Psi) = \mathcal{G}\ (\mathbf{e}, \Upsilon, \Delta)$
        $\phi = \mathcal{S}_{def}\ (C, \epsilon, \epsilon)$
    **in** $(\Psi\ [\phi])$
```

Solving with defaulting $\boxed{\mathcal{S}_{def}\ (C, \phi, \psi) \rightsquigarrow \phi}$

```
$\mathcal{S}_{def}\ (C, \phi, \psi) =$
    **let**
        $C' = C\ [\phi]$
        $(\phi_1, \psi_1, C_1) = \mathcal{S}_{fix}\ (C', \phi, \psi)$
        $(\beta, \varpi) = \mathcal{F}\ (\psi_1)$
    **in if** $\psi_1 \not\equiv \epsilon$ **then** $\mathcal{S}_{def}\ (\{\beta \equiv \varpi\} \cup C_1, \phi_1, \psi_1)$ **else** $\phi_1$
```

Figure 5.17: Annotation algorithm

6. Results

6.1 Introduction

Now that all the theory is out of the way we can start applying it to some test programs. This section is divided in three parts. We start by doing some small tests to see if the annotations gathered are indeed what we want them to be. The second part tests some recursive functions. For these also some speed improvement tests are run. The last part consists of some tests using a binary tree. These tests have a lot of potential to gain speed improvements by the analysis and strictness optimizations.

6.2 Annotation tests

The first test is to check whether the analysis can correctly determine whether the elements of a list are going to be used or not. We use the test code presented in figure **??**.

$$\begin{array}{l}
g :: [\,a\,] \to \mathit{Int} \\
g\ \mathit{xs} = \mathit{length}\ (\mathit{id}\ \mathit{xs}) \\
h :: [\,\mathit{Int}\,] \to \mathit{Int} \\
h\ \mathit{xs} = \mathit{sum}\ (\mathit{id}\ \mathit{xs})
\end{array}$$

Figure 6.1: Annotation list tests code

The main part of interest here are the types of the input lists of g and h. We use id here to show that information can flow through functions using polyvariance.

This results in the annotations presented in figure **??**.

$$\begin{array}{l}
g = (\lambda \mathit{xs} \to \\
\quad (\mathbf{let} \\
\qquad u = (\mathit{id} :: ((([\,a^{(\nu_1,\delta_1)}\,]^{\,(\nu_2,\mathbb{1})})^{(\mathbb{0},\mathbb{1})} \to ([\,a^{(\nu_1,\delta_1)}\,]^{\,(\nu_2,\mathbb{1})})^{\mathbb{0}})^{\mathbb{1}}\ \mathit{xs}) :: ([\,a^{(\nu_1,\delta_1)}\,]^{\,(\nu_2,\mathbb{1})})^{\mathbb{0}} \\
\quad \mathbf{in}\ (\mathit{length} :: ((([\,a^{(\nu_1,\delta_1)}\,]^{\,(\nu_2,\mathbb{1})})^{(\mathbb{0},\mathbb{1})} \to \mathit{Int}^{\mathbb{0}})^{\top}\ \ u) :: \mathit{Int}^{\mathbb{0}} \\
\quad) :: \mathit{Int}^{\mathbb{0}} \\
\ ) :: ((([\,a^{(\nu_1,\delta_1)}\,]^{\,(\nu_2,\mathbb{1})})^{(\nu_3,\mathbb{1})} \to \mathit{Int}^{\mathbb{0}})^{\top} \\
h = (\lambda \mathit{xs} \to \\
\quad (\mathbf{let} \\
\qquad u = (\mathit{id} :: (((([\,\mathit{Int}^{(\nu_1,\mathbb{1})}\,]^{\,(\nu_2,\mathbb{1})})^{(\mathbb{0},\mathbb{1})} \to ([\,\mathit{Int}^{(\nu_1,\mathbb{1})}\,]^{\,(\nu_2,\mathbb{1})})^{\mathbb{0}}))^{\mathbb{1}}\ \mathit{xs}) :: ([\,\mathit{Int}^{(\nu_1,\mathbb{1})}\,]^{\,(\nu_2,\mathbb{1})})^{\mathbb{0}} \\
\quad \mathbf{in}\ ((\mathit{sum} :: ((\mathit{Num}\ \overline{\beta}\ \mathit{Int})^{(\mathbb{0},\top)} \to ((([\,\mathit{Int}^{(\nu_1,\mathbb{1})}\,]^{\,(\nu_2,\mathbb{1})})^{(\mathbb{0},\mathbb{1})} \to \mathit{Int}^{\mathbb{0}})^{\top})^{\top}\ \ \mathit{NumInt})\ u) :: \mathit{Int}^{\mathbb{0}} \\
\quad) :: \mathit{Int}^{\mathbb{0}} \\
\ ) :: ((([\,\mathit{Int}^{(\nu_1,\mathbb{1})}\,]^{\,(\nu_2,\mathbb{1})})^{(\nu_3,\mathbb{1})} \to \mathit{Int}^{\mathbb{0}})^{\top}
\end{array}$$

Figure 6.2: Annotation list tests

This might be a little bit hard to parse. Each sub-expression gets annotated with an usage annotated type and the code is in A-Normal form. So lets break down the code:

- Because the code is in A-Normal form the binding for u is introduced to hold the application $\mathit{id}\ \mathit{xs}$.
- Class constraints are transformed into dictionaries and functions that have a class constraint receive an additional parameter of this type. For example the *sum* function expects a *Num Int* as its first argument. The concrete dictionary *NumInt* is passed as the first argument. For each instance a concrete dictionary is created and used whenever a function requires such an instance[1].

[1] In later tests also the instances for *Enum* and *Show* are used.

- In u only id and $id\ xs$ are annotated with a type. These types are the locally instantiated types and not the global type for the expression[2]
- The types presented on the last two lines[3] are the types of the **let** expressions and of the function respectively.
- It is in the types for the function that we find the typing difference we actually were interested in from the start. We see that g only demands the spine and not the elements inside of it[4]. In h however both the spine and the elements are demanded. As in all cases only a $\mathbb{1}$ is present we even see that the spine (and the elements in h) is only demanded once.

Some notes to keep in mind when reading these annotations are:

- As only functions can be used anything that is not a function will have a usage of $\mathbb{0}$ attached.
- As these functions are analysed without a main function context the usage of these functions is unknown and they get annotated with a $\top$ to signify this.

6.3 Recursive function tests

We test four recursive functions: *map*, *fold*, *fac* and *fib*. *map* is chosen as it is the most commonly used higher order function. *foldl* is chosen to represent higher order functions as normally you never want to use it, normally you should prefer *foldl′* except when the supplied function is not strict in its first argument. In an ideal world you would want the compiler to optimize *foldl* to *foldl′* if it is only used with a strict function. The results for these two functions can be generalized to arbitrary higher order functions. *fac* and *fib* are two function that should profit greatly from a strictness optimization.

Because we are also interested in runtime speeds the function will be accompanied by a main function. Contrary to the tests in section **??** this means that the functions will only be demanded if the main function does so[5].

All tests are compiled both with and without optimizations and the programs are each run 10 times and the timings are collected. In the results the average time and the minimum time and maximum time are presented.

6.3.1 Map

Map is a basic higher order function. The code we use to test is presented in figure **??**.

```
map :: (a → b) → [a] → [b]
map _ [] = []
map f (x : xs) = f x : map f xs
main = print (sum (map succ [1 . . 1000000 :: Int]))
```

Figure 6.3: Map

We can clearly see in figure **??** which bindings can be optimized[6]. The optimized code is shown in figure **??**. Of the bindings that are optimized two are useless (*u3* and *uhcMain*) as the strict binding will only be evaluated when the binding is demanded from the **in** part. Of the remaining bindings none are inside the *map* function. The reason this is the case is that whether or not the bindings *u2* and *u3* can be made strict is depending on external usages. For *u2* this depends whether or not the spine is used of the resulting list. For *u3* this depends on whether or not the elements of the resulting list are used. This is generally the case for functions that return data structures. Most of the time the function can internally not be optimized because it is not allowed to assume usage of the result[7].

[2] I.e. they are annotated types and not annotated type schemes.

[3] The lines that start with a) followed by a type

[4] Remember that the first two annotations of a list are for the elements and the last two for the spine. See the example in figure **??** in section **??**.

[5] Of course we will define a main function that actually does, but it will result in actual useful demands on the functions as opposed to the default $\top$

[6] These are *u3*, *u4*, *u6*, *u7*, *main* and *uhcMain*

[7] Internally the only thing assumed about the result is that it is demanded. For data structures this only means the outer constructor is forced but no such conclusions can be for the internal fields.

```
let
  $map =^{(\mathbb{1},\mathbb{1})} (\lambda f \rightarrow$
            $(\lambda ls \rightarrow$
              let ! $ls' =^{(\mathbb{1},\omega)} ls :: ([a^{(\mathbb{0},\mathbb{0})}]^{(\mathbb{0},\{0,1\})})^{\nu_4}$
              in
                (case $ls' :: ([a^{(\mathbb{0},\mathbb{0})}]^{(\mathbb{0},\{0,1\})})^{\mathbb{1}}$ of
                  $: x\ xs \rightarrow$
                    let
                      $u1 =^{(\mathbb{0},\delta_4)} map :: ((a^{(\mathbb{0},\delta_3)} \rightarrow b^{\nu_1})^{(\top,\top)} \rightarrow ((([a^{(\mathbb{0},\mathbb{0})}]^{(\mathbb{0},\{0,1\})})^{(\mathbb{0},\mathbb{1})} \rightarrow ([b^{(\nu_2,\delta_1)}]^{(\nu_3,\delta_2)})^{\mathbb{0}})^{\{0,1\}})^{\top}\ f\ xs$
                    in
                      let $u2 =^{(\nu_1,\mathbb{0})} (f :: (a^{(\mathbb{0},\delta_3)} \rightarrow b^{\nu_1})^{\mathbb{1}}\ x) :: b^{\nu_1}$
                      in
                        let
                          $u3 =^{(\top,\mathbb{1})} (: :: (b^{(\mathbb{0},\mathbb{0})} \rightarrow ((([b^{(\mathbb{0},\mathbb{0})}]^{(\mathbb{0},\delta_4)})^{(\mathbb{0},\delta_4)} \rightarrow ([b^{(\mathbb{0},\mathbb{0})}]^{(\mathbb{0},\delta_4)})^{\top})^{\mathbb{1}})^{\top}\ u2\ u1) :: ([b^{(\mathbb{0},\mathbb{0})}]^{(\mathbb{0},\delta_4)})^{\top}$
                        in
                          $u3 :: ([b^{(\mathbb{0},\mathbb{0})}]^{(\mathbb{0},\delta_4)})^{\top}$
                  $[\,] \rightarrow [\,] :: ([b^{(\mathbb{0},\mathbb{0})}]^{(\mathbb{0},\delta_4)})^{\top}$
                $) :: ([b^{(\mathbb{0},\mathbb{0})}]^{(\mathbb{0},\delta_4)})^{\top}$
            $) :: ((([a^{(\mathbb{0},\mathbb{0})}]^{(\mathbb{0},\{0,1\})})^{(\nu_4,\mathbb{1})} \rightarrow ([b^{(\mathbb{0},\mathbb{0})}]^{(\mathbb{0},\delta_4)})^{\top})^{\nu_6}$
        $) :: ((a^{(\mathbb{0},\delta_3)} \rightarrow b^{\nu_1})^{(\nu_5,\delta_5)} \rightarrow ((([a^{(\mathbb{0},\mathbb{0})}]^{(\mathbb{0},\{0,1\})})^{(\nu_4,\mathbb{1})} \rightarrow ([b^{(\mathbb{0},\mathbb{0})}]^{(\mathbb{0},\delta_4)})^{\top})^{\nu_6})^{\mathbb{1}}$
in
  let
    $u4 =^{(\mathbb{0},\mathbb{1})} enumFromTo :: ((Enum\ \overline{\beta}\ Int)^{(\mathbb{0},\mathbb{1})} \rightarrow (Int^{(\mathbb{0},\mathbb{0})} \rightarrow (Int^{(\mathbb{0},\{0,1\})} \rightarrow ([Int^{\top\ \top}]^{\top\ \top})^{\mathbb{0}})^{\mathbb{1}})^{\mathbb{1}})^{\top}\ EnumInt\ 1\ 1000000$
  in
    let
      $u5 =^{(\top,\top)} (succ :: ((Enum\ \overline{\beta}\ Int)^{(\mathbb{0},\mathbb{1})} \rightarrow (Int^{(\mathbb{0},\delta_6)} \rightarrow Int^{\top})^{\top})^{\top}\ EnumInt) :: (Int^{(\mathbb{0},\delta_6)} \rightarrow Int^{\top})^{\top}$
    in
      let
        $u6 =^{(\top,\mathbb{1})} map :: ((Int^{(\mathbb{0},\mathbb{0})} \rightarrow Int^{\top})^{(\top,\top)} \rightarrow ((([Int^{(\mathbb{0},\mathbb{0})}]^{(\mathbb{0},\{0,1\})})^{(\mathbb{0},\mathbb{1})} \rightarrow ([Int^{(\mathbb{0},\mathbb{0})}]^{(\mathbb{0},\delta_7)})^{\top})^{\mathbb{1}})^{\mathbb{1}}\ u5\ u4$
      in
        let
          $u7 =^{(\mathbb{0},\mathbb{1})} (sum :: ((Num\ \overline{\beta}\ Int)^{(\mathbb{0},\top)} \rightarrow (([Int^{(\nu_7,\mathbb{1})}]^{(\mathbb{0},\mathbb{1})})^{(\mathbb{0},\mathbb{1})} \rightarrow Int^{\mathbb{0}})^{\top})^{\top}\ NumInt\ u6) :: Int^{\mathbb{0}}$
        in
          let
            $main =^{(\mathbb{1},\mathbb{1})} (print :: ((Show\ \overline{\beta}\ Int)^{(\mathbb{1},\mathbb{1})} \rightarrow (Int^{(\mathbb{0},\mathbb{1})} \rightarrow (IO\ ())^{\mathbb{1}})^{\mathbb{1}})^{\top}\ ShowInt\ u7) :: (IO\ ())^{\mathbb{1}}$
          in
            let
              $u8 =^{(\mathbb{0},\mathbb{0})} () :: ()^{\mathbb{0}}$
            in
              let
                $uhcMain =^{(\mathbb{1},\mathbb{1})} (ehcRunMain :: ((IO\ ())^{(\mathbb{1},\mathbb{1})} \rightarrow (()^{(\mathbb{0},\mathbb{0})} \rightarrow (IO\ ())^{\mathbb{1}})^{\mathbb{1}})^{\top}\ main\ u8) :: (IO\ ())^{\mathbb{1}}$
              in
                $uhcMain :: (IO\ ())^{\mathbb{1}}$
```

Figure 6.4: Map annotated

```
let
  map = λf →
          λls →
            let ! ls' = ls
            in case ls' of
              : x xs →
                let u1 = map f xs
                in
                  let u2 = f x
                  in
                    let ! u3 = :u2 u1
                    in u3
              []     → []
in let ! u4 = enumFromTo EnumInt 1 1000000
  in
    let u5 = succ EnumInt
    in
      let ! u6 = map u5 u4
      in
        let ! u7 = sum NumInt u6
        in
          let ! main = print ShowInt u7
          in
            let u8 = ()
            in
              let ! uhcMain = ehcRunMain main u8
              in uhcMain
```

Figure 6.5: Map optimized

When we analyze the results in the table **??** we see that we still get a reasonable speed gain even though the function we wanted to optimize (*map*) did not get any meaningful optimizations[8]. The reason is that the **let** introduced for the main function have enough impact to be noticeable. In general for all further tests the resulting speed gain should be lowered with around 2% to gain the speed gain of the function under consideration[9].

	Average	Minimum	Maximum
Normal	4.012	3.789	4.266
Optimized	3.784	3.710	3.875
Speed gain	5.7%	2.1%	9.2%

Table 6.1: Results map

6.3.2 Fold

foldl is another basic recursive function. It has one more argument than *map* and when given a strict function can be heavily optimized. The code is given in figure **??**

There are unfortunately only a few optimizations happening in figure **??**. The most important binding (binding for z') is not made strict. Even though we can clearly convince ourselves that it should be strict, as $(+)$ is strict in its first argument. However the compiler has no way to use this information as only the usage and demand on the binding are determined by the rest of the code. For the internal code the only assumption is that the result will be demanded and the annotations are calculated accordingly. This means that we analyze *lgo*, and generate a type scheme for it, before we analyze the body of *foldTest*[10]. This results in the most general type for a binding. The side effect of this is thus that we get annotation variables on the places of which the body of the binding does not specify anything. Even when we later

[8] In fact, the **let!** should make it slightly slower

[9] The timings for *map* tend to fluctuate a lot. For the *foldl* they are much more stable and the 2% can be more easily seen

[10] The main reason for this is that we need the generalized types of the bindings to correctly type the body of a let.

$$\begin{array}{l}
\mathit{foldTest} :: \mathit{Int} \to \mathit{Int} \\
\mathit{foldTest}\ x = \mathit{foldl}\ (+)\ 0\ [1\mathinner{\ldotp\ldotp}x] \\
\mathit{foldl} :: (a \to b \to a) \to a \to [b] \to a \\
\mathit{foldl} = \lambda f\ \mathit{z0}\ \mathit{xs0} \to \textbf{let} \\
\quad \mathit{lgo} = \lambda z\ \mathit{ys} \to \textbf{case}\ \mathit{ys}\ \textbf{of} \\
\quad\quad [\,] \to z \\
\quad\quad (x : \mathit{xs}) \to \textbf{let}\ z' = f\ z\ x\ \textbf{in}\ \mathit{lgo}\ z'\ \mathit{xs} \\
\quad \textbf{in}\ \mathit{lgo}\ \mathit{z0}\ \mathit{xs0} \\
\mathit{main} = \mathit{print}\ (\mathit{sum}\ (\mathit{replicate}\ 100\ (\mathit{foldTest}\ 1000000)))
\end{array}$$

Figure 6.6: Foldl

$$\begin{array}{l}
\mathit{lgo} = \lambda f \to \\
\quad \lambda z \to \\
\quad\quad \lambda \mathit{ys} \to \\
\quad\quad\quad \textbf{let}\ !\ \mathit{ys}' = \mathit{ys} \\
\quad\quad\quad \textbf{in case}\ \mathit{ys}'\ \textbf{of} \\
\quad\quad\quad\quad :\ x\ \mathit{xs} \to \\
\quad\quad\quad\quad\quad \textbf{let}\ !\ \mathit{xs}' = \mathit{xs} \\
\quad\quad\quad\quad\quad \textbf{in} \\
\quad\quad\quad\quad\quad\quad \textbf{let}\ z' = f\ z\ x \\
\quad\quad\quad\quad\quad\quad \textbf{in}\ \mathit{lgo}\ f\ z'\ \mathit{xs} \\
\quad\quad\quad\quad [\,] \quad\ \to z \\
\mathit{foldl} :: ((a^{(\mathbb{0},\mathbb{0})} \to (b^{(\mathbb{0},\nu_2)} \to a^{\nu_1})^{\nu_4})^{(\top,\top)} \to (a^{(\mathbb{0},\{0,1\})} \to (([\,b^{(\mathbb{0},\delta_1)}]^{\,(\nu_3,\mathbb{1})})^{(\nu_5,\mathbb{1})} \to a^{\mathbb{0}})^{\mathbb{1}})^{\mathbb{1}})^{(\mathbb{0},\mathbb{0})} \\
\mathit{foldl} = \lambda f \to \\
\quad \lambda z \to \\
\quad\quad \lambda \mathit{xs} \to \\
\quad\quad\quad \mathit{lgo}\ f\ z\ \mathit{xs} \\
\mathit{foldTest} :: (\forall\, \nu_1, \nu_2\ .\ \mathit{Int}^{(\nu_2,\mathbb{0})} \to \mathit{Int}^{\nu_1})^{(\mathbb{0},\mathbb{0})} \\
\mathit{foldTest} = \lambda x \to \\
\quad \textbf{let}\ !\ \mathit{u1} = \mathit{enumFromTo}\ \mathit{EnumInt}\ 1\ x \\
\quad \textbf{in} \\
\quad \textbf{let}\ \mathit{u2} = +\mathit{NumInt} \\
\quad \textbf{in}\ \mathit{foldl}\ \mathit{u2}\ 0\ \mathit{u1} \\
\mathit{u3} :: (\forall\ .\ \mathit{Int})^{(\mathbb{0},\mathbb{0})} \\
\mathit{u3} = \mathit{foldTest}\ 1000000 \\
\mathit{u4} :: (\forall\, \nu_1, \delta_1\ .\ [\mathit{Int}^{(\mathbb{0},\mathbb{0})}]^{\,(\nu_1,\delta_1)})^{(\top,\mathbb{1})}) \\
!\ \mathit{u4} = \mathit{replicate}\ 100\ \mathit{u3} \\
\mathit{u5} :: (\forall\ .\ \mathit{Int})^{(\mathbb{0},\mathbb{1})} \\
!\ \mathit{u5} = \mathit{sum}\ \mathit{NumInt}\ \mathit{u4} \\
\mathit{main} :: (\forall\ .\ \mathit{IO}\ ())^{(\mathbb{1},\mathbb{1})} \\
!\ \mathit{main} = \mathit{print}\ \mathit{ShowInt}\ \mathit{u5} \\
\mathit{u6} :: (\forall\ .\ ())^{(\mathbb{0},\mathbb{0})} \\
\mathit{u6} = () \\
\mathit{uhcMain} :: (\forall\ .\ \mathit{IO}\ ())^{(\mathbb{1},\mathbb{1})} \\
!\ \mathit{uhcMain} = \mathit{ehcRunMain}\ \mathit{main}\ \mathit{u6}
\end{array}$$

Figure 6.7: Foldl optimized

at the use site of a function we pass in strict annotations the function itself can only be optimized by the annotations locally present which are variables. This gives us the general result that bindings that depend on annotations of a supplied function[11] or argument[12] cannot assume anything of annotations the supplied function has[13]. So any binding that depend on an annotation of a function argument cannot be optimized[14].

As we can see in the results in table **??**, the speed gains from optimizing the fold program are very low.

	Average	Minimum	Maximum
Normal	2.272	2.250	2.334
Optimized	2.325	2.290	2.375
Speed gain	2.3%	1.7%	1.7%

Table 6.2: Results foldl

6.3.3 Factorial

Factorial is a simple strict recursive function that in all cases should be optimized to avoid creating a chain of un-evaluated chunks. The code is given in figure **??**.

$$\begin{array}{l}
\mathit{fac} :: \mathit{Int} \rightarrow \mathit{Int} \\
\mathit{fac}\ n = \mathbf{if}\ n \leqslant 1\ \mathbf{then}\ 1\ \mathbf{else}\ n * \mathit{fac}\ (n - 1) \\
\mathit{main} = \mathit{print}\ (\mathit{sum}\ (\mathit{replicate}\ 10000000\ (\mathit{fac}\ 12)))
\end{array}$$

Figure 6.8: Factorial

Running the analysis results in the annotations given in figure **??**.

$$\begin{array}{l}
\mathit{fac} = \\
\quad (\lambda n \rightarrow \\
\quad\quad \mathbf{let} \\
\quad\quad\quad b =^{(\top,\mathbb{1})} (\leqslant ::((\mathit{Ord}\ \overline{\beta}\ \mathit{Int})^{(\mathbb{0},\mathbb{1})} \rightarrow (\mathit{Int}^{(\mathbb{0},\mathbb{1})} \rightarrow (\mathit{Int}^{(\mathbb{0},\mathbb{1})} \rightarrow \mathit{Bool}^{\top})^{\mathbb{1}})^{\mathbb{1}})^{\top}\ \mathit{OrdInt}\ n\ 1) :: \mathit{Bool}^{\top} \\
\quad\quad \mathbf{in} \\
\quad\quad\quad \mathbf{let}\ !\ b' =^{(\mathbb{1},\omega)} b :: \mathit{Bool}^{\top} \\
\quad\quad\quad \mathbf{in} \\
\quad\quad\quad\quad (\mathbf{case}\ b' :: \mathit{Bool}^{\mathbb{1}}\ \mathbf{of} \\
\quad\quad\quad\quad\quad \mathit{False} \rightarrow \\
\quad\quad\quad\quad\quad\quad \mathbf{let}\ \mathit{u3} =^{(\top,\{1,\infty\})} (- :: ((\mathit{Num}\ \overline{\beta}\ \mathit{Int})^{(\mathbb{0},\mathbb{1})} \rightarrow (\mathit{Int}^{(\mathbb{0},\mathbb{1})} \rightarrow (\mathit{Int}^{(\mathbb{0},\mathbb{1})} \rightarrow \mathit{Int}^{\top})^{\mathbb{1}})^{\mathbb{1}})^{\top}\ \mathit{NumInt}\ n\ 1) :: \mathit{Int}^{\top} \\
\quad\quad\quad\quad\quad\quad \mathbf{in} \\
\quad\quad\quad\quad\quad\quad\quad \mathbf{let}\ \mathit{u2} =^{(\top,\mathbb{1})} (\mathit{fac} :: (\mathit{Int}^{(\mathbb{0},\{1,\infty\})} \rightarrow \mathit{Int}^{\top})^{\mathbb{1}}\ \mathit{u3}) :: \mathit{Int}^{\top} \\
\quad\quad\quad\quad\quad\quad\quad \mathbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{let}\ \mathit{u1} =^{(\top,\mathbb{1})} (* :: ((\mathit{Num}\ \overline{\beta}\ \mathit{Int})^{(\mathbb{0},\mathbb{1})} \rightarrow (\mathit{Int}^{(\mathbb{0},\mathbb{1})} \rightarrow (\mathit{Int}^{(\mathbb{0},\mathbb{1})} \rightarrow \mathit{Int}^{\top})^{\mathbb{1}})^{\mathbb{1}})^{\top}\ \mathit{NumInt}\ n\ \mathit{u2}) :: \mathit{Int}^{\top} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{in}\ \mathit{u1} :: \mathit{Int}^{\top} \\
\quad\quad\quad\quad\quad \mathit{True} \rightarrow 1 :: \mathit{Int}^{\top} \\
\quad\quad\quad\quad) :: \mathit{Int}^{\top} \\
\quad) :: (\mathit{Int}^{(\mathbb{0},\{1,\infty\})} \rightarrow \mathit{Int}^{\top})^{\mathbb{0}}
\end{array}$$

Figure 6.9: Factorial annotated

So what is happening here:

[11]This basically applies to all higher order functions

[12]Datatypes are an example of this. If the internal usages of certain bindings depend upon the usage of the result then it can not be optimized at those places.

[13]The annotations on the function argument are either a variable over which the higher order function quantifies or it needs to be $\top$. This is because the caller can give a function with any kind of annotations.

[14]This for example applies to the *foldl* function, which cannot be automatically optimized to *foldl'*

- Just as in the previous section most usage annotations of non-function types are $\mathbb{0}$. Except this time there are a few $\top$'s present. This has to do with the fact that this time we are using imported functions which have always $\top$ for both usage and demand annotations. These $\top$'s are in two places: they are present as the usage of the imported functions and as the usage of the final result of the imported functions. This directly demonstrates that exporting (and using imported) functions has relevance for the precision of the types. However polyvariance does result in the fact that the arguments are correctly annotated to be demanded once and that the partial functions are all used exactly once.

- We clearly can see that *fac* usage its argument one or multiple times. It is used exactly one times when the argument is smaller than 1. If the argument is larger than 1 then the argument is used exactly thrice. One time for the comparison, one time when subtracting one and one time when multiplying.

- Every **let** binding is annotated with its usage and demand. And we can clearly see that all bindings are in fact strict. This enables us to optimize these single **let** bindings into **let**! bindings. The result is presented in figure **??**. There are no normal **let** bindings left. Every binding is optimized to a strict **let**! binding.

```
λn →
  let ! b =⩽ OrdInt n 1
  in
    let ! b′ = b
    in
      case b′ of
        False →
          let ! u3 = −NumInt n 1
          in
            let ! u2 = fac u3
            in
              let ! u1 = ∗NumInt n u2
              in u1
        True → 1
```

Figure 6.10: Factorial optimized

Now that we have optimizations lets see what the actual speed gains are. The results are given in table **??**.

	Average	Minimum	Maximum
Normal	8.778	8.749	8.895
Optimized	8.654	8.626	8.760
Speed gain	1.4%	1.4%	1.5%

Table 6.3: Results factorial

The speed gains are underwhelming. This probably has to with the additional overheads of **let**! that are not really necessary. For example this is the case for binding *u3*. This is first evaluated in a **let**! and then passed as the recursive argument. For the comparison it is needed so it is evaluated basically immediately after it was evaluated by the **let**!. The same holds for the bindings *b* and *u1*.

6.3.4 Fibonacci

Fibonacci is a function that should get a nice boost from a strictness optimization. The version presented in figure **??** is the simple inefficient version[15].

This code results in the optimized code presented in figure **??**.

[15]The algorithm presented here is $\mathcal{O}(n^2)$. There exists algorithms that are $\mathcal{O}(n)$.

```
fib :: Int → Int
fib x | x < 1 = 0
fib 1 = 1
fib n = fib (n − 1) + fib (n − 2)
main = print (fib 30)
```

Figure 6.11: Fibonacci

```
f = λx →
  let u3 = −NumInt x 2
  in
    let u2 = fib u3
    in
      let u1 = −NumInt x 1
      in
        let u5 = fib u1
        in
          let u4 = +NumInt u5 u2
          in
            let ! u6 =≡ EqInt 1 x
            in case u6 of
              False → u4
              True → 1
fib = λx →
  let y = f x
  in
    let ! z = <OrdInt x 1
    in
      let ! z′ = z
      in case z′ of
          False → y
          True → 0
```

Figure 6.12: Fibonacci optimized

There are surprisingly few[16] optimizations happening here. In fact the only **let**! introduced is for the *z* binding. The problem is that the bindings for *u1* to *u5* are before the *u6* binding. At that point there is indeed no guarantee that those bindings are indeed needed[17].

If however we would have specified *fib* differently such that the recursive bindings are positioned in such a way that we can guarantee their use. The code[18] in figure **??** is a rewritten version where the relevant **let** bindings are moved to the *False* branches.

```
fib2 x =
  let u1 = x < 1
  in case u1 of
    True → 0
    False →
      let u2 = x ≡ 1
      in case u2 of
        True → 1
        False →
          let
            u3 = x − 1
            u4 = x − 2
          in
            let
              f1 = fib2 u3
              f2 = fib2 u4
            in f1 + f2
```

Figure 6.13: Fibonacci better code

The code form figure **??** results in the optimized code presented in **??**.

```
fib2 = λx →
  let ! u1 = <OrdInt x 1
  in
    let ! u1′ = u1
    in case u1′ of
      False →
        let ! u2 =≡ Eq_@DCT_@u81_24_0 x 1
        in
          let ! u2′ = u2
          in case u2′ of
            False →
              let ! u3 = −NumInt x 1
              in
                let ! u4 = fib2 u3
                in
                  let ! u5 = −NumInt x 2
                  in
                    let ! u6 = fib2 u5
                    in + NumInt u4 u6
            True    → 1
      True → 0
```

Figure 6.14: Fibonacci better optimized

Because this time the recursive bindings are indeed inside the branch where they are guaranteed to be used, they are optimized to **let**! bindings.

Table **??** shows the results for both versions of the fibonacci function.

[16]Before studying the code and seeing that in fact there are no more optimization allowed

[17]As indeed they are not when x is 1

[18]This code is already written in A-normal form to ensure that the desugaring does not create the previous inefficient version. It should be possible to write a more concise function that still result in the same optimized code.

	Average	Minimum	Maximum
Normal *fib*	2.104	1.884	2.529
Optimized *fib*	1.950	1.738	2.528
Speed gain *fib*	7.3%	7.7%	0.0%
Normal *fib2*	1.453	1.373	1.791
Optimized *fib2*	1.047	1.001	1.160
Speed gain *fib2*	27.9%	27.1%	35.2%
Speed gain *fib* vs. *fib2* normal	30.9%	27.1%	29.2%
Speed gain *fib* vs. *fib2* optimized	46.2%	42.4%	54.1%

Table 6.4: Results fibonacci

This results show the unfortunate fact that it can really matter how some functions are written in the surface language.

6.4 Binary tree

6.4.1 Introduction

Here we test some bigger programs concerning binary trees which consists of multiple functions. We use some shared code for all the tests. The shared code consists of a function to generate a balanced tree, a mirror function and a single instance map function. It is single instance because we saw earlier that higher order functions cannot be optimized.

After a short discussion of the shared functions we test two separate functions. We count the number of nodes in the tree and we sum the leafs together. The first function does not use the values inside the tree so it cannot be optimized to evaluate the internal values early. The second function however does use the internal values and it can be optimized[19] to compute the internal values eagerly.

The counting of the nodes happens in two ways: directly, and by calculating the length of a flattened tree. Finally we test the combination of all functions and see what optimization gains we achieve there. For completeness we also compare this with a version that uses a higher order map function.

6.4.2 Shared code

We will use the code presented in figure **??** in all the following tests. The annotated types for these functions are given in figure **??**.

```
data BinTree a = Leaf a | BinTree (BinTree a) a (BinTree a)

mapTreeUnitToFac :: BinTree () → BinTree Int
mapTreeUnitToFac (Leaf _) = Leaf (fac 12)
mapTreeUnitToFac (BinTree l _ r) = BinTree l′ (fac 12) r′
  where
    l′ = mapTreeUnitToFac l
    r′ = mapTreeUnitToFac r

genBalancedBinTree :: Int → BinTree ()
genBalancedBinTree n | n < 1 = Leaf ()
genBalancedBinTree n = BinTree s () s
    where s = genBalancedBinTree (n − 1)

mirror :: BinTree a → BinTree a
mirror (BinTree l a r) = BinTree r a l
mirror x = x

testTree :: BinTree Int
testTree = mirror (mapTreeUnitToFac (genBalancedBinTree 20))
```

Figure 6.15: Shared BinTree code

[19] Even though this does not happen as the function does not know how the internal values are being used due to polyvariance.

$$
\begin{array}{l}
\mathit{Data}\ \mathit{BinTree}\ [\nu_1, \delta_1 \mathinner{..} \nu_4, \delta_4]\ a \\
\quad = \mathit{Leaf}\ \ a^{(\nu_1, \delta_1)} \\
\quad \mid \mathit{BinTree}\ (\mathit{BinTree}\ [\nu_1, \delta_1 \mathinner{..} \nu_4, \delta_4]\ a)^{(\nu_2, \delta_2)}\ \ a^{(\nu_3, \delta_3)}\ (\mathit{BinTree}\ [\nu_1, \delta_1 \mathinner{..} \nu_4, \delta_4]\ a)^{(\nu_4, \delta_4)} \\
\mathit{mapTreeUnitToFac} :: \\
\ \ \forall \nu_3, \delta_3, \nu_4, \delta_2, \nu_5, \delta_1, \nu_1, \nu_2, a\ .\ (\mathit{BinTree}\ (\nu_1\ \delta_1\ \mathbb{0}\ \mathbb{0}\ \nu_2\ \delta_2\ \mathbb{0}\ \mathbb{0})\ a)^{(\mathbb{0}, \mathbb{1})} \rightarrow (\mathit{BinTree}\ (\nu_3\ \delta_3\ \mathbb{0}\ \mathbb{0}\ \mathbb{0}\ \mathbb{0}\ \nu_4\ \mathbb{0})\ \mathit{Int})^{\nu_5} \\
\mathit{genBalancedBinTree} :: \forall \delta\ .\ \mathit{Int}^{(\mathbb{0}, \{1, \infty\})} \rightarrow (\mathit{BinTree}\ (\mathbb{0}\ \delta\ \mathbb{0}\ \mathbb{0}\ \mathbb{0}\ \mathbb{0}\ \mathbb{0}\ \mathbb{1})\ (()))^{\mathbb{0}} \\
\mathit{mirror} :: \forall \nu_1, \delta_1, \nu_2, \delta_2, a\ .\ (\mathit{BinTree}\ (\nu_1\ \delta_1\ \mathbb{0}\ \mathbb{0}\ \mathbb{0}\ \delta_2\ \mathbb{0}\ \mathbb{0})\ a)^{(\top, \{1, \infty\})} \rightarrow (\mathit{BinTree}\ (\nu_1\ \delta_1\ \mathbb{0}\ \mathbb{0}\ \mathbb{0}\ \delta_2\ \mathbb{0}\ \mathbb{0})\ a)^{\nu_2} \\
\quad \Rightarrow \mathbb{0}\ U\ \nu_2 \equiv \top \\
\mathit{testTree} :: \forall \delta_2, \nu_4, \delta_1, \nu_3, \delta_3, \nu_1, \nu_2\ .\ \ \mathit{BinTree}\ (\nu_1\ \delta_1\ \nu_2\ \delta_2\ \nu_3\ \delta_3\ \nu_4\ \delta_2)\ \mathit{Int}
\end{array}
$$

Figure 6.16: Shared BinTree types

6.4.3 CountNodes

Here we count the number of nodes inside the tree. The code is presented in figure **??**.

```
countNodes :: BinTree a → Int
countNodes (Leaf _) = 1
countNodes (BinTree l _ r) = countNodes l + 1 + countNodes r
testBinTreeNoInternals :: Int
testBinTreeNoInternals = countNodes testTree
main = print testBinTreeNoInternals
```

Figure 6.17: Count nodes

This code results in the following optimized code presented in figure **??**. We can see that the sub-trees, the recursive calls and the additions are made strict.

```
countNodes = λx →
  let ! x′ = x
  in case x′ of
    BinTree l _ r →
      let ! u1 = l
      in
        let ! u2 = r
        in
          let ! u3 = countNodes r
          in
            let ! u4 = countNodes l
            in
              let ! u5 = +NumInt u4 1
              in
                let ! u6 = +NumInt u5 u3
                in u6
    Leaf _ → 1
```

Figure 6.18: Count nodes optimized

The results are presented in table **??**.

6.4.4 Length . flatten

Instead of directly counting the nodes we first convert the the tree into a list and then count the number of elements inside the list. The code is presented in figure **??**.

The optimized code is presented in figure **??**. Due to bugs in the types for the imported symbols (:) and (++) the recursive calls cannot be made strict.

	Average	Minimum	Maximum
Normal	1.221	1.192	1.262
Optimized	0.957	0.949	0.964
Speed gain	21.6%	20.4%	23.6%

Table 6.5: Results count nodes

```
flatten :: BinTree a → [a]
flatten (BinTree l a r) = flatten l ++ a : flatten r
flatten (Leaf a) = [a]

testBinTreeNoInternalsViaList :: Int
testBinTreeNoInternalsViaList = length (flatten testTree)

main = print testBinTreeNoInternalsViaList
```

Figure 6.19: Length . flatten

```
flatten = λx →
  let ! x' = x
  in case x' of
    BinTree l a r →
      let u1 = flatten r
      in
        let u2 = (:) a u1
        in
          let u3 = flatten l
          in
            let ! u4 = (++) u3 u2
            in u4
    Leaf a →
      let ! u5 = (:) a []
      in u5
```

Figure 6.20: Length . flatten optimized

The results are presented in table ??. Due to the previously described bugs the results are not very good. However we still get a sizable speed gain nonetheless.

	Average	Minimum	Maximum
Normal	6.021	5.909	6.140
Optimized	5.569	5.458	5.690
Speed gain	7.5%	7.6%	7.3%

Table 6.6: Results length . flatten

We see a large increase in time and a lower speed gain. This mainly has to do with the fact that a very inefficient *flatten* function is used[20]. As this cannot be prevented by making the recursive calls strict this is present in both versions. If we remove the extra time and re-compare the timings then the speed gains are about the same as the version that counts the nodes directly.

If we write the flatten function using a difference list it is still slower than the direct version but a lot faster than the simple flatten version. As we can see in the code in figure ?? and results in table ??.

```
flatten :: BinTree a → [a]
flatten t = flattenDiff t []

flattenDiff :: BinTree a → [a] → [a]
flattenDiff (BinTree l a r) xs = flattenDiff l (a : flattenDiff r xs)
flattenDiff (Leaf a) xs = [a] ++ xs
```

Figure 6.21: Flatten difference list

	Average	Minimum	Maximum
Normal	2.051	2.020	2.082
Optimized	2.291	2.266	2.366
Speed gain	10.5%	10.9%	12.0%

Table 6.7: Results flatten difference list

6.4.5 Sum

Here we sum the internal values of the tree. This should be the case that can be optimized the most. The code is presented in figure ??.

```
sumTree :: BinTree Int → Int
sumTree (Leaf x) = x
sumTree (BinTree l x r) = sumTree l + x + sumTree r

testBinTreeInternals :: Int
testBinTreeInternals = sumTree testTree

main = print testBinTreeInternals
```

Figure 6.22: Sum tree

This results in the optimized code presented in figure ??. The analysis correctly determined that all fields of the BinTree are indeed strict and both recursive calls are also made strict.

This results in the very good speed gains presented in table ??.

6.4.6 Combined

We now run the three previous tests as a single program. This allows us to analyse the combination of multiple functions and looks more like real code. The code is presented in figure ??.

[20]The *flatten* presented here is quadratic.

```
sumTree = λx →
  let ! x′ = x
  in case x′ of
    BinTree l y r →
      let ! u1 = l
      in
        let ! u2 = y
        in
          let ! u3 = r
          in
            let ! u4 = sumTree r
            in
              let ! u5 = sumTree l
              in
                let ! u6 = (+) NumInt u5 y
                in
                  let ! u7 = (+) NumInt u6 u4
                  in u7
    Leaf y →
      let ! u8 = y
      in y
```

Figure 6.23: Sum tree optimized

	Average	Minimum	Maximum
Normal	15.100	13.796	15.618
Optimized	7.980	7.763	8.124
Speed gain	47.2%	43.7%	48.0%

Table 6.8: Results sum tree

```
testBinTreeInlinedMap :: Int
testBinTreeInlinedMap = countNodes testTree + length (flatten testTree) + sumTree testTree
main = print testBinTreeInlinedMap
```

Figure 6.24: Combined

When we analyse the results in table ?? we see that the timings are comparable to the sum of timings from the three tests above and the speed gain is the weighed average of the speed gains of the stand alone tests. We conclude from this that the optimizations are locally and are not really impacted by the surrounding code.

	Average	Minimum	Maximum
Normal	20.755	19.510	22.230
Optimized	13.497	12.858	14.398
Speed gain	35.0%	34.1%	35.2%

Table 6.9: Results

6.4.7 Combined with higher order map

We finally test the combined version again, but this time with a higher order map function. The code is given in figure ??. We expect to see less speed gains than using the inlined map used in the previous tests.

```
mapTree :: (a → b) → BinTree a → BinTree b
mapTree f (Leaf a) = Leaf (f a)
mapTree f (BinTree l a r) = BinTree (mapTree f l) (f a) (mapTree f r)
testTreeHigherOrderMap :: BinTree Int
testTreeHigherOrderMap = mirror (mapTree (const (fac 12)) (genBalancedBinTree 20))
testBinTreeHigherOrderMap :: Int
testBinTreeHigherOrderMap = countNodes testTreeHigherOrderMap + length (flatten testTreeHigherOrderMap) + sumTree test
main = print testBinTreeHigherOrderMap
```

Figure 6.25: Combined with higher order map

When we analyse the results in table ?? we see that indeed we get a lower speed gain. However the total time is drastically lower. I have no idea why this is the case.

	Average	Minimum	Maximum
Normal	7.167	6.966	7.412
Optimized	6.345	6.091	6.574
Speed gain	11.5%	12.6%	11.3%

Table 6.10: Results

7. Related work

The main related work is of course [?] by Hidde Verstoep as this whole thesis is based upon it. In [?] counting analysis is presented formally for a simpler language than what is presented in this thesis. Verstoep presented the theory and a heap recycling optimization. In this thesis the focus was on making counting analysis a practical analysis and extending it to work for UHC core. Without the theoretical work of Verstoep this thesis would never have been started. Even though both theses describe counting analysis the focus was completely different. In this thesis the focus was on making runtimes lower by using counting analysis combined with a strictness optimization.

Although counting analysis is defined in [?], the idea for it is sketched inside [?]. Wansbrough describes a fully functional polymorphic usage analysis, complete with a datatype annotation algorithm. The datatype annotation algorithm in this thesis is derived from [?] and extended to work for the annotations used by counting analysis.

[?] introduces a backwards analysis to do a more precise sharing analysis, especially for one-shot lambda's. This is done by not only keeping track of the usage of functions but also of their demand[1]. The reason for separate demands is the fact that *seq* does evaluate its first argument but does not use it. This is a simple analysis that performs well on first order functions. As [?] targets GHC core and GHC's inliner removes a lot of higher order functions, the analysis in [?] does well for GHC[2].

[?] introduces a monomorphic, monovariant strictness analysis with subeffecting. The language contains a strict application as a construct to force evaluation[3]. Normal strictness analysis only track the demand of values. To improve the analysis here another property is tracked in the analysis. They call this applicativeness, or whether or not a function will be applied to a value. This basically is the inverse of what [?] does. There usage is tracked and for functions demand is tracked separately, here demand is tracked and for functions usage is tracked separately. The strictness optimization described in [?] is implemented in this thesis. It is mostly identical except for the necessary extensions to make it work for UHC core.

A polyvariant strictness analysis is given in [?]. Here the idea is that polyvariant functions get additional formal parameters for the annotation variables. These are then later compiled away to specialized monovariant versions of the function. As a result we can make specialized functions for given annotations. This is one way to solve the problem that functions cannot be optimized if the annotations are unknown.

The idea for a single unifying analysis is first sketched in [?] and fully worked out for a prototype language in [?]. The latter also extended the analysis to include uniqueness analysis. [?] gives a polymorphic, polyvariant generic analysis that can be instantiated with or without subeffecting. This includes or excludes uniqueness analysis from the resulting combined analysis.

[1]Using a function means applying it to an argument, demanding a function requires it to be in whnf. So demanding a function does evaluate the function, but does not use the function.

[2]There are some limitations exposed in the form version of the paper.

[3]Equivalent to *seq*

8. Conclusion and further work

This has been a long and bugful experience. Although in the end we really did achieve some good speed ups from the strictness optimization, there are still some problems left over. The problems fall in three catagories:

Theoretical These are problems with the type theory. For example the theory of dealing with imports exports might or might not be correct.

Implementation These are problems with mapping the type theory to the implementation inside UHC. For example there are still some bugs with multi argument functions and datatypes resulting in incorrect absence annotations.

UHC These are problems within the UHC compiler which are exposed by the implementation. For example the core generated by type classes and instances is not correctly ordered.

8.1 Theoretical

As the theoretical work of Verstoep [**?**] only dealt with a simplified language instead of the full fledged core of UHC there needed be some extensions to the type system made. Most extensions were straightforward additions to the type system[1]. However there are a few extensions that were not straightforward. These are the modular analysis mode [2] and strict fields of datatypes.

The modular part gives errors[3] during the analysis when imprecise information is retrieved from an imported symbol, while the analysis can deduce locally more precise annotations. This happens mostly with class function as they tend to end up with $\top$ in most places[4].

The theory for strict datatypes quickly became a mess and took too much time away from the main parts of the thesis, so in the end they were removed from the theory. The only part where it is still present is in the datatype annotation algorithm[5]. This is because it is used inside the standard libraries of UHC and they needed to be supported, although that support only means that it does not crash the analysis. The information for strict fields is just ignored. This means that we end up with a too low count of demands on these fields. For the strictness optimization this means that we err on the safe side[6]. This does however block the analysis results from being useful for a sharing optimization.

There are some UHC core structures which are just ignored in the type system and it is hoped that they do not break the algorithm when encountered. These are existential types and higher ranked types. Luckily these are not commonly used and any type errors are ignored when encountered during solving. As UHC has already done type checking, the analysis just assumes the types are correct and ignores the errors. As there is also no way to give type signatures[7] to the analysis, all types are inferred during constraint solving, higher ranked types cannot work at all. Also without user supplied type signatures the uniqueness typing analysis part of counting analysis is not really useful.

8.2 Implementation

Although the constraint generation implementation was a straightforward mapping from the type rules to Haskell AG code, the constraint solver implementation was a lot less straightforward. Not because the algorithm outlined in section **??** was not straightforward to map, but because the presented algorithm is way to inefficient to be used as an actual implementation[8].

Due to the severe rewrite of the constraint solver there are still some bugs in the analysis. All known bugs result in a lower count[9] so the wrong annotations are still safe for the strictness optimization. This

[1]This includes the transforming of *seq* to **let!**, *FFI* and constants

[2]E.g. dealing with imports and exports

[3]It generates a message that $\top$ is not equal to some more precise annotation value. It does not seem to generate wrong type annotations in other parts of the code.

[4]This is because they are translated into datatype definitions. The algorithm to give annotations to datatypes does not descend very deep into the types. This is to have reasonable types without hurting performance much.

[5]See section **??**.

[6]As with a lower count we might not be able to optimize. Not optimizing is always safe.

[7]Neither with nor without annotations are possible in the current system.

[8]For comparison, the algorithm in section **??** took around six hours to solve 45 thousand constraints, while the worklist algorithm used in the final implementation takes around six minutes for 75 thousand constraints.

[9]Actually it results in $\mathbb{0}$ annotations.

also means that in the current form the analysis is not ready to work for sharing, uniqueness or absence optimizations.

There are two known specific bugs and some known unspecific bugs. The specific bugs are both illustrated in chapter **??**. The first one is present in section **??**. There is a bug present that the argument of the function argument of *lgo* gets annotated with $\mathbb{0}$ instead of with $\top$ or an annotation variable. It does not happen for all higher order functions. It seems to happen on the first argument of a two argument function, but it does not happen on all those functions. The second one is presented in section **??**. It is most clearly visible in the type for *mirror*. The annotations on it somehow specify the sub-trees are not demanded at all. As it is clearly not the case this is a bug. I believe these two bugs are related but cannot find what exactly goes wrong[10].

There are also some unspecific bugs in the analysis. Some functions crash during the analysis when defined inside one module, but when moved to another module they work just fine. Certain definitions of functions which involve guards result in a crash during the analysis. Rewriting those functions using an **if then else** expression or a **case** expression avoids the crash.

8.3 UHC

UHC is not the most stable compiler and there are bugs in it. The most glaring of those is that when using classes and instances the instance record is bound by a let before the base class record is bound by a let. During the analysis this results in an out of scope error[11].

Some programs just crash when compiled with UHC. For example the program $main = foldl\ (+)\ 0\ [1\,..\,10000000]$ results in a crash[12]. The end result is not outside of the range of *Int* nor does the program run out of memory.

As the UHC.Base module was too big for the analysis to analyse in a reasonable amount of time, the module was split into multiple modules. The first attempt named these modules UHC.Base# where # was a single digit. Turns out digits inside module names is not supported by UHC. It compiles the package fine, but any package with a module that has a name which contains a digit is hidden from other packages. This made the UHC.Base modules unusable for every module not inside the UHCBase package, including the Base package that includes the Prelude module.

8.4 Further work

Although the analysis and implementation as they are currently achieve significant speedups in most situations. It needs some further work to make it fully useable:

- Formalize the theory about imports and exports.
- Formalize the theory about strict data types.
- Formalize the theory about user supplied type signatures. Both with and without usage and demand annotations.
- Formalize the theory about existential types in data type declarations.
- Fix the bugs with annotations incorrectly becoming $\mathbb{0}$, so other optimizations can safely be implemented.

When the above points are done this enables more further work:

- Uniqueness typing. Although this disables subeffecting it might not be that bad[13] in most cases due to polyvariance.
- With uniqueness typing a heap recycling optimization can safely be implemented.
- More optimizations based on the annotations of counting analysis. This include dead code removal, lambda/let floating, unboxed types and non-updating thunks.

[10]It is hard to see where in over 2000 constraints it goes wrong.
[11]In the implementation there is now code present that fixes these out of scope errors.
[12]The program $main = foldl\ (+)\ 0\ [1\,..\,1000000]$ does not results in a crash.
[13]Resulting in useless $\top$ annotations.

- Compilation of polyvariant functions using code duplication for different strictness annotations[14]. For example *foldl* where we can generate two versions. One where the supplied function is strict in its first argument and one the most general annotated version. Then at call site point to the correct version of *foldl*.

[14] See chapter 5 of [**?**]

A. Annotation constrain solving

Here is presented the algorithmic style version (figure **??**) of the complex annotation constraint solving rule. The helper functions are mostly presented using Haskell style. Definition of Haskell prelude functions are omitted.

```
$\mathcal{S}\ (\varphi_3 \equiv \varphi_1 \diamond \varphi_2, \phi, \psi) \mid isvar\ \varphi_1 \vee isvar\ \varphi_2 =$
  let
    $favs = fav\ (\varphi_3 \equiv \varphi_1 \diamond \varphi_2)$
    $\overline{(\varphi, \overline{\varpi})} = [\psi\ (x) \mid x \leftarrow favs]$
    $\overline{\overline{(\varphi', \varpi')}} = [[(a, av) \mid av \leftarrow \overline{av}] \mid (a, \overline{av}) \leftarrow \overline{(\varphi, \overline{\varpi})}]$
    $\overline{\overline{(\varphi_4, \varpi_4)}} = sequence\ \overline{\overline{(\varphi', \varpi')}}$
    $\overline{\overline{(\varphi_5, \varpi_5)}} = [\overline{(\varphi_4, \varpi_4)} \mid (\overline{(\varphi_4, \varpi_4)}, \phi_1)\ \overline{(\overline{(\varphi_4, \varpi_4)}, {}^{\overline{\varpi_4}}/_{\overline{\varphi_4}})}, \varphi_3\ [\phi_1] \equiv \varphi_1\ [\phi_1] \diamond \varphi_2\ [\phi_1]]$
    if $(\overline{\overline{(\varphi_5, \varpi_5)}}) \equiv [\,]$ then $error$ "Unsatisfiable constraint"
    $\overline{(\varphi_6, \overline{\varpi_6})} = combine\ (concat\ (\overline{\overline{(\varphi_5, \varpi_5)}}))$
    $\psi_2 = [{}^{x}/_{y} \mid {}^{x}/_{y} \in \psi, y \notin \overline{\varphi_6}]$
    $(\phi_2, \psi_3) = partition\ (\lambda({}^{x}/_{y}) \rightarrow size\ x \equiv 1)\ \psi_2$
    $\psi_4 = [{}^{\overline{\varpi_6}}/_{\overline{\varphi_6}}] +\!\!+\ \psi_3$
    $useless = size\ favs \equiv 1 \wedge \psi_4\ (favs) \equiv \mathcal{P}\ (\top) - \emptyset$
    $C_3 =$ if $useless$ then $\emptyset$ else $\varphi_3 \equiv \varphi_1 \diamond \varphi_2$
    $\phi_3 = equalvars\ (\overline{\overline{(\varphi_5, \varpi_5)}})$
    $\phi_5 = [{}^{x}/_{y} \mid {}^{x}/_{y} \in \phi_3, {}^{-}/_{y} \notin \phi_2]$
    $\phi_5 = \phi +\!\!+\ \phi_2 +\!\!+\ \phi_5$
    $\psi_5 = [{}^{x}/_{y} \mid {}^{x}/_{y} \in \psi_4, {}^{-}/_{y} \notin \phi_5]$
  in $(\phi_6, \psi_5, C_3)$


$\psi\ (x)$
  $\mid x : \overline{\varpi} \in \psi = \overline{\varpi}$
  $\mid ohterwise = \mathcal{P}\ (\top) - \emptyset$
$combine :: [[(a, b)]] \rightarrow [(a, [b])]$
$combine\ [\,] = [\,]$
$combine\ ((a, b) : xs) =$ let $ys = combine\ xs$ in $insert\ a\ b\ ys$
$insert :: a \rightarrow b \rightarrow [(a, [b])] \rightarrow [(a, [b])]$
$insert\ a\ b\ [\,] = (a, [b])$
$insert\ a\ b\ ((x, y) : xs) =$ if $(x \equiv a)$ then $(x, b : y) : xs$ else $(x, y) : insert\ a\ b\ xs$
$size :: [a] \rightarrow Int$
$size\ [\,] = 0$
$size\ (\_ : xs) = 1 + size\ xs$
$partition :: (a \rightarrow Bool) \rightarrow [a] \rightarrow ([a], [a])$
$partition\ \_\ [\,] = ([\,], [\,])$
$partition\ f\ (x : xs) =$ let $(ys, zs) = partition\ f\ xs$ in if $f\ x$ then $(x : ys, zs)$ else $(ys, x : zs)$
$equalVars :: [[(Var, AnnVal)]] \rightarrow Map\ Var\ (Set\ Var)$
$equalVars\ [\,] = Data.Map.\emptyset$
$equalVars\ (xs : xss) = Data.Map.unionWith\ Data.Set.intersection\ toM\ \$\ equalVars\ xss$
  where
    $toM = toM2\ \$\ map\ snd\ \$\ Data.Map.toList\ \$\ revMap\ \$\ Data.Map.fromList\ xs$
    $toM2\ [\,] = Data.Map.\emptyset$
    $toM2\ (y : ys) = Data.Map.union\ (toM2\ ys)\ \$\ Data.Map.fromList\ \$\ map\ (\lambda v \rightarrow (v, y))\ \$\ Data.Set.toList\ y$
$revMap :: (Ord\ a, Ord\ b) \Rightarrow Map\ a\ b \rightarrow Map\ b\ (Set\ a)$
$revMap = revMap' \circ Data.Map.toList$
  where $revMap'\ [\,] = Data.Map.\emptyset$
    $revMap'\ ((\mathbf{k}, v) : xs) = Data.Map.insertWith\ (Data.Set.union)\ v\ (Data.Set.singleton\ \mathbf{k})\ \$\ revMap'\ xs$
```

Figure A.1: Annotation constraint solving