Utrecht University
Faculty of Science
Department of Information and Computing Sciences

Strictness analysis in Helium

Computing Science MSc Thesis
August 10, 2021

Marco van de Weerthof
ICA-5871476

Supervised by:
Jurriaan Hage
Ivo Gabe de Wolff

CONTENTS

ABSTRACT iii

1 INTRODUCTION 1
- 1.1 Problem description . . . 1
- 1.2 Outline . . . 2

2 PRELIMINARIES 3
- 2.1 Strict, non-strict and lazy evaluation . . . 3
- 2.2 Strictness in Haskell . . . 5
- 2.3 Motivations for strictness analysis . . . 6
- 2.4 Type and effect systems . . . 7

3 STRICTNESS ANALYSIS 9
- 3.1 Relevance typing . . . 9
- 3.2 Relevance and applicativeness typing . . . 12

4 HELIUM 15
- 4.1 Design philosophy . . . 15
- 4.2 Architecture . . . 17
- 4.3 Core . . . 17

5 RELATED WORK 21
- 5.1 Related analyses . . . 21
- 5.2 Strictness analysis in UHC . . . 23
- 5.3 Recent work in Helium . . . 23

6 RESEARCH QUESTIONS AND SETUP 25
- 6.1 Research questions . . . 25
- 6.2 Setup . . . 26

7 IMPLEMENTATION 29
- 7.1 Transformation rules . . . 29
- 7.2 Algorithm . . . 37
- 7.3 Adaptation to polyvariance . . . 42

8 EXPERIMENT 45

9 DIFFERENT APPROACHES 47
- 9.1 Counting arguments . . . 47
- 9.2 Monotypes . . . 55

10 Conclusion **57**

11 Future work **59**
11.1 Strictness analysis . . . 59
11.2 Helium . . . 60

A Datatypes **63**
A.1 Tuples . . . 63
A.2 Lists . . . 64
A.3 Other datatypes . . . 66
A.4 Results . . . 66

ABSTRACT

Strictness analysis is an important optimization in compilers for lazy languages. Expressions which are guaranteed to be used should preferably be evaluated in a strict manner to improve performance. This thesis implements strictness analysis in the Helium compiler for Haskell. The analysis is based on relevance and applicativeness typing. A monovariant and polyvariant analysis are proposed, with an optimal precision-speed balance required. The monovariant analysis with monotypes provides the best trade-off, while the polyvariant analysis provides better precision but at an increased analysis cost. An alternative system without applicativeness annotations was proposed which could have improved the balance even further, but this turned out to be unsound.

INTRODUCTION

The introduction contains a description of the problem (1.1) and the outline for the thesis (1.2).

1.1 PROBLEM DESCRIPTION

Haskell is a functional programming language which uses lazy evaluation. Consider the program

$$\textbf{let}\ f = \lambda x\ y \rightarrow x + 1$$
$$\textbf{in}\ \ f\ (5 * 5)\ (10 + 2)$$

Haskell creates a thunk in memory for every subexpression. The thunks are only evaluated (to weak head normal form) when they need to be. The thunk containing $(5 * 5)$ will be evaluated to 25, as it needs to be calculated for the function to return the final answer of 26. Note that the second argument in f is never used, which means its thunk is also never evaluated.

The opposite of lazy evaluation is strict evaluation. In the previous example, it would first evaluate the arguments of f to 25 and 12, pass both arguments to f and return the result of 26. Even though the second argument is not used, it has to be evaluated before passing it to the function, which is avoided in lazy evaluation. The advantage of strict evaluation over lazy evaluation is memory usage, as storing unevaluated expressions in thunks is more costly in that regard.

A lazy language like Haskell would benefit from strictness analysis. This is a static analysis to determine which parts of the program can be evaluated strictly rather than lazily. It would infer that f is strict in its first argument, meaning it is safe to evaluate $(5 * 5)$ immediately instead of creating a thunk, thus saving memory. The second argument is not used and therefore remains lazy, as it is beneficial to not waste any time on evaluation there. Haskell has ways to force strict evaluation of expressions, with the special function *seq* which always evaluates its first argument to weak head normal form and returns its second.

Not every argument which is used can be made strict. Consider the function

$$f :: Bool \rightarrow a \rightarrow a \rightarrow a$$
$$f\ x\ y\ z = \textbf{if}\ x\ \ \textbf{then}\ y\ \ \textbf{else}\ z$$

The first argument is guaranteed to be evaluated, as the if-statement needs to know the result of the conditional to know which branch to pick. The other two arguments might be evaluated, but this depends on x.

If x is *True*, y will be evaluated. If x is *False*, z will be evaluated. It is not possible for both to be evaluated. This means at least one argument is never going to be evaluated, which means making that argument strict would result in unnecessary computation. Because we do not know which argument is given to x beforehand, we do not know which of y or z will be used, and thus we cannot make either of them strict. Furthermore, if all arguments were to be made strict, the application f *True* `3` (*error* `"Crash"`) would crash. However, if only x is made strict, the function returns 3 as expected in a lazy language. The strictness analysis needs to be conservative and only infer strictness when an argument is guaranteed to be evaluated under all circumstances, such that the meaning of the program does not change.

The goal of this thesis is to implement strictness analysis in Helium. Helium is a Haskell compiler developed at Utrecht University with a focus on high-quality error messages [9]. This thesis seeks to implement a monovariant and polyvariant strictness analysis based on relevance and applicativeness typing proposed by Holdermans and Hage [10]. Both variants will be compared in precision, analysis cost, and the trade-offs between them. A more elaborate analysis might return a better result, but might take longer to compute, or uses more memory. The implementations are also compared against the strictness analysis which is already present in Helium, with the goal to find an improvement over the existing analysis.

1.2 Outline

The outline of this thesis is as follows:

- **Preliminaries** (Chapter 2): Background information such as the definition of strictness and type and effect systems.
- **Strictness analysis** (Chapter 3): A description of strictness analysis using relevance and applicativeness.
- **Helium** (Chapter 4): The architecture of the Helium compiler.
- **Related work** (Chapter 5): Previous research into strictness analysis, related analyses and Helium.
- **Research question and setup** (Chapter 6): The main research question, subquestions and the setup of the experiment.
- **Implementation** (Chapter 7): The implementation of strictness analysis in the Helium compiler.
- **Experiment** (Chapter 8): Results of the experiments on the analyses.
- **Different approaches** (Chapter 9): Two different approaches to improve the trade-off of the analysis.
- **Conclusion** (Chapter 10): Which strictness analysis should be used in Helium going forward.
- **Future work** (Chapter 11): Improvements which can still be made to the analysis and the compiler itself.
- **Datatypes** (Appendix A): An extension of the analysis to include datatypes.

PRELIMINARIES

The preliminaries give an overview of some topics of importance to this thesis. This includes the difference between strict and lazy evaluation (2.1), strictness in Haskell (2.2), practical motivations for strictness analysis in a lazy language (2.3), and an introduction to type and effect systems (2.4). The chapters on strictness analysis (3) and the Helium compiler (4) also contain background information, but due to their size and importance they have their own chapters.

2.1 STRICT, NON-STRICT AND LAZY EVALUATION

Strict, non-strict and lazy evaluation are three different evaluation strategies. They differ in how they handle arguments to functions. Subsection 2.1.1 describes the difference between strict and non-strict evaluation, and subsection 2.1.2 describes the difference between non-strict and lazy evaluation.

2.1.1 STRICT VERSUS NON-STRICT EVALUATION

Strict or eager evaluation first evaluates all arguments before passing them to the function body. This is opposed to non-strict evaluation, in which arguments are passed to the function body unevaluated. Strict evaluation is also known as call-by-value, while non-strict evaluation is known as call-by-name.

Consider the function

$succ :: Int \rightarrow Int$
$succ\ x = x + 1$

Strict evaluation first evaluates $(2 + 3)$ to 5, and passes that to the function:

$succ\ (2 + 3)$
$=$
$succ\ 5$
$=$
$5 + 1$
$=$
6

Non-strict evaluation on the other hand immediately proceeds to the function and evaluates $(2 + 3)$ in there:

$succ\ (2 + 3)$
$=$
$(2 + 3) + 1$
$=$
$5 + 1$
$=$
6

In the previous example, both strategies return the same answer, and the difference in execution is minimal. Differences between the strategies occur when arguments are not used.

Consider the function *const*, which takes two arguments and returns its first, ignoring the second:

$const :: a \to b \to a$
$const\ x\ y = x$

Strict evaluation would evaluate both arguments, but evaluating the second argument is a waste of time as it is never used. The non-strict approach forwards the second argument to the function, where it is never evaluated. This means the outcome of a function could change depending on what is given to the second argument. In *const* $3\ \bot$, where $\bot$ is an expression which causes abnormal termination, the strict approach would crash on $\bot$ while the non-strict approach returns 3, as $\bot$ is never evaluated [10].

Another benefit of strict evaluation is reduced memory usage. If *const* is given a complex expression as first argument, non-strict evaluation has to build a huge thunk containing all (sub-)expressions. Strict evaluation immediately evaluates the entire expression and only has to store the result, which typically takes less space. In the case of the first argument, strict evaluation is preferable as the argument will be evaluated later anyway, while the second argument is better off in non-strict evaluation as it is not evaluated at all.

Formally, for any function f, if f diverges when given a divergent argument, f is strict in that argument:

$$f \bot \rightsquigarrow \bot$$

where $\bot$ is a divergent argument. A computation is divergent when it does not terminate or terminates in an exceptional state. Examples of diverging arguments in Haskell are an infinite loop, the error function or undefined. If f converges given a divergent argument, then that argument is not necessary for the computation, otherwise the function would have diverged. Therefore, it does not need to be evaluated [12].

2.1.2 Non-strict versus lazy evaluation

In the function square, non-strict evaluation would need to evaluate its argument twice.

$square :: Int \to Int$
$square\ x = x * x$
$square\ (2 + 3)$
$=$
$(2 + 3) * (2 + 3)$
$=$
$5 * (2 + 3)$
$=$
$5 * 5$
$=$
25

As the name call-by-name suggests, it replaces the occurrences of variable x by the entire expression, which is then evaluated twice. However, it is more efficient to share evaluations between the occurrences of the argument. Lazy evaluation, or call-by-need, is a non-strict evaluation strategy which avoids repeated evaluation. Instead of passing the full argument, a memory thunk is created which stores the

(computation of the) argument. Upon reaching the first occurrence of the variable, the expression in the thunk is evaluated and the result is written back to the thunk. The second time the argument is used, it can directly take the result and does not need to evaluate the expression again.

$$
\begin{aligned}
& \mathit{square}\ (2+3) \\
& = \\
& x * x\ (\textbf{where}\ x = (2+3)) \\
& = \\
& 5 * x\ (\textbf{where}\ x = 5) \\
& = \\
& 5 * 5 \\
& = \\
& 25
\end{aligned}
$$

In general, lazy evaluation performs better than non-strict analysis, except when the thunk is used exactly once. In lazy languages, sharing analysis is used to avoid an evaluated expression being written to the thunk if it is not used anymore [6].

2.2 Strictness in Haskell

Haskell is a functional language which uses lazy evaluation. However, it also has constructs to force the evaluation of expressions to weak head normal form (WHNF). An expression is in WHNF if it is either

- A constructor (*True*, $[1, 2, 3]$).
- A function applied to too few arguments (*square*, $(+)\ 2$).
- A lambda abstraction ($\lambda x \rightarrow \mathit{expression}$).

Note that arguments of partially applied functions are not evaluated, only the head of the expression is reduced. This is the difference between normal form and weak head normal form [1].

Haskell has multiple ways of enforcing strictness [17]:

seq: x \`*seq*\` y evaluates x to WHNF and returns y.

$!: f \$! x evaluates x to WHNF and applies f to x (f \$! $x = x$ \`*seq*\` $f\ x$).

Bang patterns: data fields and variables can be made strict by annotating them with an exclamation mark; For instance, **data** $D = D\ !\ \mathit{Int}\ \mathit{Int}$ will store its first argument strictly. The expression

$$
\begin{aligned}
& \textbf{let}\ !\, x = (5+5) \\
& \textbf{in}\ x + 1
\end{aligned}
$$

will first evaluate x to 10 before proceeding with the body. This requires the BangPatterns language extension[1].

StrictData/Strict: since GHC 8.0.1, the StrictData language extensions[2] makes datatype declarations strict by default, and the Strict language extension[3] makes everything binding strict by default. The tilde is used to annotate bindings which should still be handled lazily.

[1] https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/strict.html#bang-patterns-and-strict-haskell
[2] https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/strict.html#strict-by-default-data-types
[3] https://downloads.haskell.org/ghc/latest/docs/html/users_guide/exts/strict.html#strict-by-default-pattern-bindings

2.3 Motivations for strictness analysis

Section 2.1 described the advantages and disadvantages of strict evaluation over lazy evaluation. Section 2.2 described how a user could manually influence evaluation in Haskell, a lazy language. To give a motivation as to why this benefits the execution of the code, we sketch an example using folds. Folds are higher-order functions which fold a list of values into one single value using a combining function and a base value. In Haskell, the right-associative fold (*foldr*) can be defined as

$foldr :: (a \to b \to b) \to b \to [a] \to b$
$foldr\ f\ z\ [\,] \quad = z$
$foldr\ f\ z\ (x : xs) = x\ `f`\ foldr\ f\ z\ xs$

For instance, $foldr\ (+)\ 0\ [1,2,3,4,5]$ returns 15, the sum of all elements in the list. The result is calculated in a right-associative manner, namely $(1 + (2 + (3 + (4 + (5 + 0)))))$. This means that the additions can only be applied once all elements of the list are pushed on the stack. For very large lists, this could result in a stack overflow.

Alternatively, the left-associative fold (foldl) can be defined as

$foldl :: (b \to a \to b) \to b \to [a] \to b$
$foldl\ f\ z\ [\,] \quad = z$
$foldl\ f\ z\ (x : xs) = foldl\ f\ (z\ `f`\ x)\ xs$

The function starts with the base value and continuously applies f to the first item in the remaining list until the list is empty. Using the previous example, the list unfolds to $(((((0{+}1){+}2){+}3){+}4){+}5)$, and also evaluates to 15. Note that *foldr* and *foldl* do not always give the same result, as $foldr\ (-)\ 0\ [1,2,3,4,5] = 3$ and $foldl\ (-)\ 0\ [1,2,3,4,5] = -15$.

In this version, it looks like it would be possible to apply the function during the calculation, which would prevent the stack overflow. However, since Haskell is a lazy language, expressions are only evaluated once they are needed. In the example, Haskell would create a thunk for every element and only starts reducing the expression when it cannot expand further, which is when the list becomes empty. It then starts with the final addition, and works it way backwards. Executing the function would result in

$foldl\ (+)\ 0\ [1,2,3,4,5]$
  -- apply second branch of foldl
$foldl\ (+)\ (0 + 1)\ [2,3,4,5]$
  -- apply second branch of foldl
$foldl\ (+)\ ((0 + 1) + 2)\ [3,4,5]$
...
$foldl\ (+)\ (((((0 + 1) + 2) + 3) + 4) + 5)\ [\,]$
  -- apply first branch of foldl
$(((((0 + 1) + 2) + 3) + 4) + 5)$
  -- inner reduction
$((((1 + 2) + 3) + 4) + 5)$
...
15

If the list is large enough it will once again trigger a stack overflow. This can be solved by forcing evaluation of the accumulating value. $foldl'$ is the left-associative fold which uses strict function application. This function can be defined as

$foldl' :: (b \to a \to b) \to b \to [a] \to b$
$foldl'\ f\ z\ [\,] \quad = z$
$foldl'\ f\ z\ (x : xs) = (foldl'\ f\ \$!\ f\ z\ x)\ xs$

$foldl'\ (+)\ 0\ [1,2,3,4,5]$ returns 15, just like *foldr* and *foldl*. However, instead of unfolding the entire list before performing any addition, $! forces the evaluation after every step:

$foldl'\ (+)\ 0\ [1,2,3,4,5]$
  -- apply second branch of foldl’

$(foldl' \ (+) \ \$! \ (0 + 1)) \ [2, 3, 4, 5]$
-- strict application
$foldl' \ (+) \ 1 \ [2, 3, 4, 5]$
-- apply second branch of foldl'
$(foldl' \ (+) \ \$! \ (1 + 2)) \ [3, 4, 5]$
-- strict application
$foldl' \ (+) \ 3 \ [3, 4, 5]$
...
$foldl' \ (+) \ 15 \ [\,]$
-- apply first branch of foldl'
15

As a result, $foldl'$ can take the sum of large lists without running into a stack overflow. In almost all cases, *foldl* can be replaced by $foldl'$. There are some cases in which *foldl* is more efficient, or has a different result. A function which is lazy in its first argument might only need the last element of the list, meaning evaluation of earlier elements will not happen using *foldl* but will be done anyway in $foldl'$, with the possibility of diverging [18].

To be able to receive the benefits, the programmer has to use $foldl'$ or any other function with strictness measures manually. It would be better if the compiler could determine which expressions can be evaluated strictly rather than lazily. This gives rise to strictness analysis. By doing a static analysis of the program, we can determine which parts of the code would benefit from being executed strictly, which would have a positive effect on memory usage, without needing the programmer to manually specify where strictness should be applied.

We should be careful that we do not infer strictness on parts of the program where this changes the outcome of the program. The original meaning of the program should be preserved, and the additional strictness is only there to improve the performance. If there is even the slightest of doubt that strictness changes the outcome, we cannot infer it and the expression remains lazy. With more elaborate strictness analyses, we could infer more parts of the program as strict at the cost of performance of the analyzer itself.

2.4 Type and effect systems

Haskell is a statically typed language. Its type system is based on the Hindley-Milner type inference algorithm [15], extended with type classes [8]. Type polymorphism can be achieved in three different ways:

Ad-hoc polymorphism is achieved by functions which can be applied to arguments of different types. For instance, addition can be applied to integers or to floating point values with the $(+)$ operator. The function has a different implementation depending on the types of the arguments, but can be used with the same operator. In Haskell, addition can be performed on any member of the Num typeclass, of which Int and Float are instances.

Parametric polymorphism is achieved by giving functions and data types a type parameter which can be instantiated with any type. For instance, the *id* function in Haskell can take and return an argument of any type, and lists have a type parameter such that every type of list can be handled via one constructor instead of requiring a separate datatype for lists of integers, booleans and so on.

Subtyping relations can be defined between different types. For instance, any integer value can be treated as a floating point value as well (2 becomes 2.0), or Square is a subtype of Rectangle. If an Int is given to a function expecting a Float, the function can convert the argument to Float. Haskell does not support explicit subtyping; a function which expects a Float cannot be given an

Int. However, it is useful in type inference. Formally, for two types τ_1 and τ_2, if there is a partial order $\sqsubseteq$ where $\tau_1 \sqsubseteq \tau_2$, then τ_1 can be weakened to τ_2.

The opposite of polymorphism is monomorphism, which does not allow type variables to occur. This means the type can only contain ground terms. Monotypes are in between polymorphism and monomorphism, as it does allow for type variables but does not allow quantification over them.

To perform analysis on programs, we have to extend the type inference to include annotations [16]. To illustrate this, consider parity analysis, which analyzed the parity of numbers. These can be either even or odd, which are represented by E and O. If we want to perform this analysis without using annotations, we would have to adapt the type system to have even and odd integers, which means every arithmetic version needs to have different implementations depending on the exact type, which is unmaintainable. A function double takes a number which is annotated with either E or O, and returns a number annotated with E, as any number multiplied by two is even. The type signature of the function, including the annotations on the arguments, becomes

$$double :: Int^{\{E,O\}} \rightarrow Int^{\{E\}}$$

Effects describe the computational properties of arguments. In the previous example, the annotations are not effects because they say something about the type itself. An example of effects would be an annotation to describe how often an argument is used [6].

Without polymorphism on the annotations, the previous function would only take arguments of which the parity is unknown, which means information might be lost if we use arguments where the parity is known. Subeffecting solves this problem, as the individual type annotations are subsets of the set of acceptable types. We can define a partial order $\sqsubseteq$ on the annotations, with $E \sqsubseteq \{E, O\}$ and $O \sqsubseteq \{E, O\}$. A number of any parity can now be passed to double. Using subeffecting, a function annotated with effect T can be used with a value of effect S, as long as S can be weakened to T ($S \sqsubseteq T$). Formally, an effect system is a complete lattice (**Ann**, $\sqsubseteq$) with **Ann** being the set of annotations and $\sqsubseteq$ a partial order on the annotations [16].

Other problems arise for a function like *id*, which can take any annotation and returns that same annotation. However, it can only have one annotation associated, which needs to be the most general in case multiple annotations are possible. In parity analysis, the function *id* (restricted to integers) would become

$$id :: Int^{\{E,O\}} \rightarrow Int^{\{E,O\}}$$

This is a monovariant or context-insensitive analysis [16]. A monovariant analysis has monomorphic types.

However, annotating the variables like this loses information. If *id* is given an even argument it returns an even number, and it returns an odd number given an odd number. According to the annotations, *id* can only take an argument which is even or odd and returns a number which is even or odd. To solve this we introduce polymorphism on effects as well. An analysis using polymorphic effects is called a polyvariant or context-sensitive analysis [16]. Using polyvariance, the *id* example becomes:

$$id :: \forall \beta . Int^{\beta} \rightarrow Int^{\beta}$$

where β can be instantiated by any subset of {E,O}.

STRICTNESS ANALYSIS

This chapter discusses an implementation of strictness analysis using relevance typing (3.1) and an improved version using applicativeness annotations (3.2).

3.1 RELEVANCE TYPING

Holdermans and Hage propose a strictness analysis based on relevance typing [10]. A variable x is relevant to term t if any term bound to x is guaranteed to be evaluated whenever t is evaluated. S describes terms which produce relevant abstractions when the term is made strict, and L describes terms where it is not known if it will produce relevant abstractions. These annotations are placed on the function arrows. In terms of a type and effect system, relevance typing has two annotations (S for strict and L for lazy, **Ann** = {S,L}) and a partial order S $\sqsubseteq$ L. The complete lattice can be written as (**Ann**, $\sqsubseteq$) with S as least element, L as greatest element, the join defined as

$$S \sqcup \varphi = \varphi$$

$$L \sqcup \varphi = L$$

and the meet defined as

$$S \sqcap \varphi = S$$

$$L \sqcap \varphi = \varphi$$

where φ ranges over the annotations.

To introduce their analysis, they provide a simple, non-strict language, with $x \in$ **Var** containing all variables, $t \in$ **Term** containing all terms and $\hat{\tau} \in \widehat{\textbf{Type}}$ containing all annotated types. The language contains booleans, natural numbers, variables, lambda abstraction, strict and non-strict application, conditionals and the error keyword which fails on evaluation. The difference between strict application ($t_1 \bullet t_2$) and non-strict application ($t_1 t_2$) is that the former forces t_2 to WHNF before applying it to t_1. A term is in WHNF if it is a boolean (true or false), 0 or a lambda.

$$\begin{aligned} t &::= \text{false} \mid \text{true} \mid 0 \mid x \mid \lambda x.t_1 \\ &\mid t_1 t_2 \mid t_1 \bullet t_2 \mid \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \\ &\mid \text{succ } t_1 \mid \text{pred } t_1 \mid \text{iszero } t_1 \mid \text{error} \\ \hat{\tau} &::= \text{bool} \mid \text{nat} \mid \hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2 \end{aligned}$$

The goal of the analysis is to transform as many non-strict applications to strict applications as possible. Holdermans and Hage describe transformation rules of the form $\widehat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{\varphi}$, where $\widehat{\Gamma}$ represents the type environment, t represents the original term, t' represent the transformed term, $\hat{\tau}$ represent the annotated type of t (and t') and φ represents the relevance context. A type environment $\widehat{\Gamma} \in \textbf{TEnv} = \textbf{Var} \to_{\text{fin}} \widehat{\textbf{Type}} \times \textbf{Ann}$ maps from variables x to a pair containing an annotated type and an annotation $(\hat{\tau}, \varphi)$. [] represents the empty environment, $[x \mapsto (\hat{\tau}, \varphi)]$ represents a singleton environment and $\widehat{\Gamma}[x \mapsto (\hat{\tau}, \varphi)]$ extends $\widehat{\Gamma}$ with a binding from x to $(\hat{\tau}, \varphi)$. The transformation rules are described in figure 3.1.

$$\frac{}{[] \vdash \text{false} \rhd \text{false} : \text{bool}^{\varphi}}\ \text{[r-false]} \qquad \frac{}{[] \vdash \text{true} \rhd \text{true} : \text{bool}^{\varphi}}\ \text{[r-true]} \qquad \frac{}{[] \vdash 0 \rhd 0 : \text{nat}^{\varphi}}\ \text{[r-zero]}$$

$$\frac{}{[x \mapsto (\hat{\tau}, \varphi)] \vdash x \rhd x : \hat{\tau}^{\varphi}}\ \text{[r-var]} \qquad \frac{\varphi \blacktriangleright \widehat{\Gamma} \quad \widehat{\Gamma}[x \mapsto (\hat{\tau}_1, \varphi_1)] \vdash t_1 \rhd t_1' : \hat{\tau}_2^{\text{S}}}{\widehat{\Gamma} \vdash \lambda x.t_1 \rhd \lambda x.t_1' : (\hat{\tau}_1 \xrightarrow{\varphi_1} \hat{\tau}_2)^{\varphi}}\ \text{[r-abs]}$$

$$\frac{\widehat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\hat{\tau}_2 \xrightarrow{\text{S}} \hat{\tau})^{\varphi} \quad \widehat{\Gamma}_2 \vdash t_2 \rhd t_2' : \hat{\tau}_2^{\varphi}}{\widehat{\Gamma}_1 \diamond \widehat{\Gamma}_2 \vdash t_1 t_2 \rhd t_1' \bullet t_2' : \hat{\tau}^{\varphi}}\ \text{[r-app1]}$$

$$\frac{\widehat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\hat{\tau}_2 \xrightarrow{\text{L}} \hat{\tau})^{\varphi} \quad \widehat{\Gamma}_2 \vdash t_2 \rhd t_2' : \hat{\tau}_2^{\text{L}}}{\widehat{\Gamma}_1 \diamond \widehat{\Gamma}_2 \vdash t_1 t_2 \rhd t_1' t_2' : \hat{\tau}^{\varphi}}\ \text{[r-app2]}$$

$$\frac{\widehat{\Gamma}_1 \vdash t_1 \rhd t_1' : \text{bool}^{\varphi} \quad \widehat{\Gamma}_2 \vdash t_2 \rhd t_2' : \hat{\tau}^{\text{L}} \quad \widehat{\Gamma}_2 \vdash t_3 \rhd t_3' : \hat{\tau}^{\text{L}}}{\widehat{\Gamma}_1 \diamond \widehat{\Gamma}_2 \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rhd \text{if } t_1' \text{ then } t_2' \text{ else } t_3' : \hat{\tau}^{\varphi}}\ \text{[r-if]}$$

$$\frac{\widehat{\Gamma} \vdash t_1 \rhd t_1' : \text{nat}^{\text{L}}}{\widehat{\Gamma} \vdash \text{succ } t_1 \rhd \text{succ } t_1' : \text{nat}^{\varphi}}\ \text{[r-succ]} \qquad \frac{\widehat{\Gamma} \vdash t_1 \rhd t_1' : \text{nat}^{\varphi}}{\widehat{\Gamma} \vdash \text{pred } t_1 \rhd \text{pred } t_1' : \text{nat}^{\varphi}}\ \text{[r-pred]}$$

$$\frac{\widehat{\Gamma} \vdash t_1 \rhd t_1' : \text{bool}^{\varphi}}{\widehat{\Gamma} \vdash \text{iszero } t_1 \rhd \text{iszero } t_1' : \text{nat}^{\varphi}}\ \text{[r-iszero]} \qquad \frac{}{[] \vdash \text{error} \rhd \text{error} : \hat{\tau}^{\varphi}}\ \text{[r-error]}$$

$$\frac{\widehat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{\text{L}}}{\widehat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{\text{S}}}\ \text{[r-sub]} \qquad \frac{\widehat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{\varphi}}{\widehat{\Gamma}[x \mapsto (\hat{\tau}_0, \text{L})] \vdash t \rhd t' : \hat{\tau}^{\varphi}}\ \text{[r-weak]}$$

$$\frac{}{\varphi \blacktriangleright []}\ \text{[c-nil]} \qquad \frac{\text{S} \blacktriangleright \widehat{\Gamma}_1}{\text{S} \blacktriangleright \widehat{\Gamma}_1[x \mapsto (\hat{\tau}, \varphi_0)]}\ \text{[c-cons-s]} \qquad \frac{\text{L} \blacktriangleright \widehat{\Gamma}_1}{\text{L} \blacktriangleright \widehat{\Gamma}_1[x \mapsto (\hat{\tau}, \text{L})]}\ \text{[c-cons-l]}$$

FIGURE 3.1: *Relevance typing, call-by-value transformations and containment. Adapted from [10]*

Rules [r-false], [r-true] and [r-zero] require the environment to be empty. The weakening rule [r-weak] states that any binding which has a relevance annotation L can be dropped from the environment. This means that every variable in the type environment has to be set to L for these transformation rules. Rule [r-var] demands the environment to be empty except for the variable itself, which also leads to the conclusion that any other variable in the expression under scrutiny has to be set to L. The annotation of

the variable itself is set to the context. If we are in a strict context (the first argument of *const*), the variable is guaranteed to be used and can thus be set to S, but in a non-strict context (the second argument of *const*) the variable is not guaranteed to be used.

New bindings are created via the [r-abs] rule. The type environment for the body is extended by the variable. We reset the context to S to measure whether the variable is relevant to its body. However, any information related to other variables might not be sound if the input context is L. Hence we perform containment ([c-nil], [c-cons-s] and [c-cons-l]), which states that all annotations in the environment before the abstraction have to be set to L if the context was L, thus preventing other variables to be erroneously annotated with S. An L-context does not guarantee that the term is never executed, (we might still evaluate the second argument of (&) if the first argument is True), thus it is still beneficial to determine whether an abstraction is relevant when defined within an L-context.

There are two rules for application, [r-app1] and [r-app2]. Rule [r-app1] transforms a non-strict application to a strict application, and rule [r-app2] leaves the non-strict application as is. An application can be transformed to a strict application if the function is relevant in its argument. If the function is relevant, the argument is evaluated under the current context, otherwise the context is set to L as the argument is not guaranteed to be used. The type environments used for the function and the argument are different environments, as it is possible that a variable is relevant in one side and not relevant in the other. In this case, if a variable is strict in either environment it is strict in the combined environment. This is represented by the context split $\diamond$, which takes the pointwise meet of environments $\widehat{\Gamma}_1$ and $\widehat{\Gamma}_2$.

$$[] \diamond [] = []$$
$$\hat{\Gamma}_1[x \mapsto (\hat{\tau}, \varphi_1)] \diamond \hat{\Gamma}_2[x \mapsto (\hat{\tau}, \varphi_2)] = (\hat{\Gamma}_1 \diamond \hat{\Gamma}_2)[x \mapsto (\hat{\tau}, \varphi_1 \sqcap \varphi_2)]$$

Rule [r-if] handles if-statements. The conditional always has to be evaluated, which means we can take the current context. However, only one of the branches is picked depending on the value of the conditional, meaning neither is guaranteed to be evaluated, meaning the context for the branches is set to L. Rules [r-succ], [r-pred] and [r-iszero] handle operations on natural numbers. Note that the successor operation, unlike the predecessor operation, produces a weak head normal form, and thus the context of the term is set to L. As the term error fails on evaluation, its transformation rule [r-error] can have any type or context. Finally, [r-sub] allows for subeffecting. Whenever we are in a S-context, we can subeffect to an L-context if the situation requires so. This will not lead to unsound transformations, as any strict function can also be treated as a non-strict function without changing the outcome. This is not the case the other way around, meaning we cannot go from an L-context to an S-context.

As example, the type signature for *const* can be derived as $\hat{\tau}_1 \xrightarrow{S} \hat{\tau}_2 \xrightarrow{L} \hat{\tau}_1$.

$$\dfrac{\dfrac{}{\mathrm{S} \blacktriangleright []} \quad \dfrac{\dfrac{}{\mathrm{S} \blacktriangleright [x \mapsto (\hat{\tau}_1, \mathrm{S})]} \quad \dfrac{\dfrac{}{[x \mapsto (\hat{\tau}_1, \mathrm{S})] \vdash x \rhd x : \hat{\tau}_1^{\mathrm{S}}}}{[x \mapsto (\hat{\tau}_1, \mathrm{S}), y \mapsto (\hat{\tau}_2, \mathrm{L})] \vdash x \rhd x : \hat{\tau}_1^{\mathrm{S}}}}{[x \mapsto (\hat{\tau}_1, \mathrm{S})] \vdash \lambda y.x \rhd \lambda y.x : (\hat{\tau}_2 \xrightarrow{\mathrm{L}} \hat{\tau}_1)^{\mathrm{S}}}}{[] \vdash \lambda x.\lambda y.x \rhd \lambda x.\lambda y.x : (\hat{\tau}_1 \xrightarrow{\mathrm{S}} \hat{\tau}_2 \xrightarrow{\mathrm{L}} \hat{\tau}_1)^{\mathrm{S}}}$$

Using this definition of *const*, we can see that the first argument will be transformed into a strict application, while the second argument remains a non-strict application:

$$\dfrac{\dfrac{\dfrac{\ldots}{[] \vdash \text{const} \rhd \text{const} : (\text{bool} \xrightarrow{\mathrm{S}} \text{bool} \xrightarrow{\mathrm{L}} \text{bool})^{\mathrm{S}}} \quad \dfrac{}{[] \vdash \text{true} \rhd \text{true} : \text{bool}^{\mathrm{S}}}}{[] \vdash \text{const true} \rhd \text{const} \bullet \text{true} : (\text{bool} \xrightarrow{\mathrm{L}} \text{bool})^{\mathrm{S}}} \quad \dfrac{}{[] \vdash \text{false} \rhd \text{false} : \text{bool}^{\mathrm{L}}}}{[] \vdash \text{const true false} \rhd \text{const} \bullet \text{true false} : \text{bool}^{\mathrm{S}}}$$

Note that strictness and relevance are not equal. The expression $(\lambda x.\text{error})$ is strict in its argument, as it trivially diverges given a diverging argument, but is not relevant as x does not occur in the body of the abstraction. On the other hand, the expression $((\lambda x.\lambda y.x)\ \text{error})$ is not strict in its first argument, as the function is not fully applied, but is relevant, as x occurs in the body of the abstraction. This leads to an unsound transformation, as $((\lambda x.\lambda y.x)\ \text{error})$ returns a function, but $((\lambda x.\lambda y.x) \bullet \text{error})$ evaluates error and fails:

$$\dfrac{\dfrac{\dots}{[] \vdash \lambda x.\lambda y.x \rhd \lambda x.\lambda y.x : (\widehat{\tau}_1 \xrightarrow{\text{S}} \widehat{\tau}_2 \xrightarrow{\text{L}} \widehat{\tau}_1)^{\text{S}}} \quad \dfrac{}{[] \vdash \text{error} \rhd \text{error} : \widehat{\tau}_1^{\text{S}}}}{[] \vdash (\lambda x.\lambda y.x)\ \text{error} \rhd (\lambda x.\lambda y.x) \bullet \text{error} : (\widehat{\tau}_2 \xrightarrow{\text{L}} \widehat{\tau}_1)^{\text{S}}}$$

The rules also do not contain a transformation rule for applications which are already strict. An obvious typing rule would be [r-sapp], which ignores φ_0 as it does not matter if the argument is relevant, as the application is already strict. However, adding this rule allows for unsound transformations as well.

$$\dfrac{\widehat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\widehat{\tau}_2 \xrightarrow{\varphi_0} \widehat{\tau})^{\varphi} \quad \widehat{\Gamma}_2 \vdash t_2 \rhd t_2' : \widehat{\tau}_2^{\varphi}}{\widehat{\Gamma}_1 \diamond \widehat{\Gamma}_2 \vdash t_1 \bullet t_2 \rhd t_1' \bullet t_2' : \widehat{\tau}^{\varphi}} \text{ [r-sapp]}$$

For the function $(\lambda x.((\lambda y.0) \bullet (\lambda z.x)))$, we can derive the type $(\widehat{\tau} \xrightarrow{\text{S}} \text{nat})$ with this rule, which means any application of this function can be transformed to a strict application. However, the function $\lambda z.x$ is evaluated in the strict application, resulting in the same function, and then passed to $\lambda y.0$ where it is never used. Hence the argument is never evaluated. Transforming to a strict application would mean the argument is evaluated, resulting in a failure if the argument is error. A fix would be to take the join of φ and φ_0 as context in the applicant. While this is sound, it is unable to use the extra information in case φ_0 is L, as the context will be set to L, making it impossible to derive strictness from there.

3.2 Relevance and applicativeness typing

The relevance typing discussed in the previous section provided an implementation for strictness analysis. However, there were two problems: it does not support applications which are already strict and it is unsound on partial applications. Holdermans and Hage combat these issues by adding applicativeness annotations (ψ) in addition to relevance annotations [10]. A term is applicative if it is guaranteed to be applied to an argument at least once. This was not the case in the two previous examples, as partial application does not apply arguments and strict application does not guarantee application if the argument is a function. They use the same lattice as the relevance annotation, with S describing a term is guaranteed to be applied to an argument, and L describing the term might not be applied to an argument. Instead of one annotation on the function arrow, there are now three: an applicativeness and relevance annotation for the argument, and an applicativeness annotation for the remainder of the function [1].

$$\widehat{\tau} ::= \text{bool} \mid \text{nat} \mid \widehat{\tau}_1 \xrightarrow{(\psi_1,\varphi,\psi_2)} \widehat{\tau}_2$$

The transformation rules are of the form $\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi,\psi)}$, with an applicativeness context ψ added next to the relevance context. The type environment $\widehat{\Gamma} \in \textbf{TEnv} = \textbf{Var} \to_{\text{fin}} \widehat{\textbf{Type}} \times \textbf{Ann} \times \textbf{Ann}$ stores both the relevance and applicativeness annotation per variable. The extended transformation rules are described in figure 3.2.

[1] Holdermans and Hage place the applicativeness annotations on the terms instead of the function arrow. For convenience in functions with multiple arguments, all three annotations will be placed on the arrow in the remainder of this thesis.

$$\frac{}{[] \vdash \text{false} \rhd \text{false} : \text{bool}^{(\varphi,\text{L})}}\ [\text{r-false}] \qquad \frac{}{[] \vdash \text{true} \rhd \text{true} : \text{bool}^{(\varphi,\text{L})}}\ [\text{r-true}]$$

$$\frac{}{[] \vdash 0 \rhd 0 : \text{nat}^{(\varphi,\text{L})}}\ [\text{r-zero}] \qquad \frac{}{[x \mapsto (\hat{\tau},\varphi,\psi)] \vdash x \rhd x : \hat{\tau}^{(\varphi,\psi)}}\ [\text{r-var}]$$

$$\frac{\psi \blacktriangleright \hat{\Gamma} \quad \hat{\Gamma}[x \mapsto (\hat{\tau}_1,\varphi_1,\psi_1)] \vdash t_1 \rhd t_1' : \hat{\tau}_2^{(\text{S},\psi_2)}}{\hat{\Gamma} \vdash \lambda x.t_1 \rhd \lambda x.t_1' : (\hat{\tau}_1 \xrightarrow{(\psi_1,\varphi_1,\psi_2)} \hat{\tau}_2)^{(\varphi,\psi)}}\ [\text{r-abs}]$$

$$\frac{\hat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\hat{\tau}_2 \xrightarrow{(\psi_2,\text{S},\psi)} \hat{\tau})^{(\varphi,\varphi)} \quad \hat{\Gamma}_2 \vdash t_2 \rhd t_2' : \hat{\tau}_2^{(\varphi,\varphi\sqcup\psi_2)}}{\hat{\Gamma}_1 \diamond \hat{\Gamma}_2 \vdash t_1 t_2 \rhd t_1' \bullet t_2' : \hat{\tau}^{(\varphi,\psi)}}\ [\text{r-app1}]$$

$$\frac{\hat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\hat{\tau}_2 \xrightarrow{(\psi_2,\text{L},\psi)} \hat{\tau})^{(\varphi,\varphi)} \quad \hat{\Gamma}_2 \vdash t_2 \rhd t_2' : \hat{\tau}_2^{(\text{L},\varphi\sqcup\psi_2)}}{\hat{\Gamma}_1 \diamond \hat{\Gamma}_2 \vdash t_1 t_2 \rhd t_1' t_2' : \hat{\tau}^{(\varphi,\psi)}}\ [\text{r-app2}]$$

$$\frac{\hat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\hat{\tau}_2 \xrightarrow{(\psi_2,\varphi_0,\psi)} \hat{\tau})^{(\varphi,\varphi)} \quad \hat{\Gamma}_2 \vdash t_2 \rhd t_2' : \hat{\tau}_2^{(\varphi,\varphi\sqcup\psi_2)}}{\hat{\Gamma}_1 \diamond \hat{\Gamma}_2 \vdash t_1 \bullet t_2 \rhd t_1' \bullet t_2' : \hat{\tau}^{(\varphi,\psi)}}\ [\text{r-sapp}]$$

$$\frac{\hat{\Gamma}_1 \vdash t_1 \rhd t_1' : \text{bool}^{(\varphi,\text{L})} \quad \hat{\Gamma}_2 \vdash t_2 \rhd t_2' : \hat{\tau}^{(\text{L},\psi)} \quad \hat{\Gamma}_2 \vdash t_3 \rhd t_3' : \hat{\tau}^{(\text{L},\psi)}}{\hat{\Gamma}_1 \diamond \hat{\Gamma}_2 \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rhd \text{if } t_1' \text{ then } t_2' \text{ else } t_3' : \hat{\tau}^{(\varphi,\psi)}}\ [\text{r-if}]$$

$$\frac{\hat{\Gamma} \vdash t_1 \rhd t_1' : \text{nat}^{(\text{L},\text{L})}}{\hat{\Gamma} \vdash \text{succ } t_1 \rhd \text{succ } t_1' : \text{nat}^{(\varphi,\text{L})}}\ [\text{r-succ}] \qquad \frac{\hat{\Gamma} \vdash t_1 \rhd t_1' : \text{nat}^{(\varphi,\text{L})}}{\hat{\Gamma} \vdash \text{pred } t_1 \rhd \text{pred } t_1' : \text{nat}^{(\varphi,\text{L})}}\ [\text{r-pred}]$$

$$\frac{\hat{\Gamma} \vdash t_1 \rhd t_1' : \text{bool}^{(\varphi,\text{L})}}{\hat{\Gamma} \vdash \text{iszero } t_1 \rhd \text{iszero } t_1' : \text{nat}^{(\varphi,\text{L})}}\ [\text{r-iszero}] \qquad \frac{}{[] \vdash \text{error} \rhd \text{error} : \hat{\tau}^{(\varphi,\psi)}}\ [\text{r-error}]$$

$$\frac{\hat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{(\text{L},\text{L})}}{\hat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{(\text{S},\psi)}}\ [\text{r-sub}] \qquad \frac{\hat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{(\varphi,\psi)}}{\hat{\Gamma}[x \mapsto (\hat{\tau}_0,\text{L},\text{L})] \vdash t \rhd t' : \hat{\tau}^{(\varphi,\psi)}}\ [\text{r-weak}]$$

$$\frac{}{\varphi \blacktriangleright []}\ [\text{c-nil}] \qquad \frac{\text{S} \blacktriangleright \hat{\Gamma}_1}{\text{S} \blacktriangleright \hat{\Gamma}_1[x \mapsto (\hat{\tau},\varphi_0,\psi_0)]}\ [\text{c-cons-s}] \qquad \frac{\text{L} \blacktriangleright \hat{\Gamma}_1}{\text{L} \blacktriangleright \hat{\Gamma}_1[x \mapsto (\hat{\tau},\text{L},\text{L})]}\ [\text{c-cons-l}]$$

FIGURE 3.2: *Relevance and applicativeness typing, call-by-value transformations and containment. Adapted from [10]*

Booleans and natural numbers cannot be applied to arguments, so rules [r-false], [r-true] and [r-zero] always need to have an L-context. For variables ([r-var]), the applicativeness context is now stored alongside the relevance annotation. Weakening ([r-weak]) is extended such that a variable can only be dropped if both its relevance and applicativeness are L. Abstraction ([r-abs]) now contains over the applicativeness context instead of the relevance context. The containment rules [c-nil], [c-cons-s] and [c-cons-l] are updated to reflect that containing on an L-annotation also means the applicativeness annotations have to become L. Its right applicativeness annotation is used as context for the body of the abstraction.

The application rules [r-app1] and [r-app2] have the applicativeness context for the function set to the relevance annotation, and the context for the argument is the join of the relevance context and the applicativeness of the function. Furthermore, the applicativeness of the argument is equal to the applicativeness context, which might have to be achieved via subeffecting. The use of applicativeness annotations makes it possible to add a rule for applications which are already strict ([r-sapp]). We ignore the relevance of the function, and propagate the relevance of the entire expression directly into the argument.

The conditional in [r-if] cannot be applicative because we can only match on booleans which cannot

be applied. The adaptations for [r-succ], [r-pred], [r-iszero] and [r-error] are trivial. The subeffecting rule [r-sub] automatically weakens the applicativeness to L if the relevance is subeffected, as the invariant $\varphi \sqsubseteq \psi$ might not be respected otherwise.

The definition for context split is also updated to reflect the addition of applicativeness annotations in the environment.

$$[] \diamond [] = []$$
$$\widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \varphi_1, \psi_1)] \diamond \widehat{\Gamma}_2[x \mapsto (\widehat{\tau}, \varphi_2, \psi_2)] = (\widehat{\Gamma}_1 \diamond \widehat{\Gamma}_2)[x \mapsto (\widehat{\tau}, \varphi_1 \sqcap \varphi_2, \psi_1 \sqcap \psi_2)]$$

Recall that *const* could be derived as $\widehat{\tau}_1 \xrightarrow{\text{S}} \widehat{\tau}_2 \xrightarrow{\text{L}} \widehat{\tau}_1$ with respect to the original rules. Under the new rules, leaving the applicativeness annotations uninstantiated, we instead derive the type

$$\widehat{\tau}_1 \xrightarrow{(\psi_1 \sqcup \psi_2, \psi_1, \psi_1)} \widehat{\tau}_2 \xrightarrow{(\text{L},\text{L},\psi_2)} \widehat{\tau}_1$$

The second argument is still not relevant regardless of how the function is applied, but the first argument only becomes relevant when the function is guaranteed to be fully applied, which is when ψ_1 and ψ_2 become S. This means we can transform $((\lambda x.\lambda y.x)$ error false$)$ to $((\lambda x.\lambda y.x) \bullet$ error false$)$, but we cannot transform $((\lambda x.\lambda y.x)$ error$)$ to $((\lambda x.\lambda y.x) \bullet$ error$)$.

To prove that the partial application of *const* cannot be transformed to a strict application, we attempt to transform the function by having an S-annotation on the function arrow. Since the entire term is not guaranteed to be applied, we have to start the inference in an L-context for applicativeness. This means the right applicativeness annotation on the function arrow becomes L, which is the annotation used for containment. Since we contain on L, all annotations in the environment have to be set to L, which is impossible if the function is relevant. Therefore, this transformation is impossible, meaning we can only annotate the first argument with an L-annotation, which means rule [r-app1] cannot be used. Thus we have to use rule [r-app2] which cannot perform a transformation.

$$\dfrac{\dfrac{\text{S} \blacktriangleright [] \quad \dfrac{\text{L} \blacktriangleright [x \mapsto (\widehat{\tau}, \text{S}, \psi)] \quad [x \mapsto (\widehat{\tau}, \text{S}, \psi), y \mapsto (\text{bool}, \text{L}, \text{L})] \vdash x : \widehat{\tau}^{(\text{S},\psi)}}{[x \mapsto (\widehat{\tau}, \text{S}, \psi)] \vdash \lambda y.x : (\text{bool} \xrightarrow{(\text{L},\text{L},\psi)} \widehat{\tau})^{(\text{S},\text{L})}}}{[] \vdash \lambda x.\lambda y.x : (\widehat{\tau} \xrightarrow{(\psi,\text{S},\text{L})} \text{bool} \xrightarrow{(\text{L},\text{L},\psi)} \widehat{\tau})^{(\text{S},\text{S})}} \quad [] \vdash \text{error} : \widehat{\tau}^{(\text{S},\text{L})}}{[] \vdash (\lambda x.\lambda y.x)\ \text{error} : (\text{bool} \xrightarrow{(\text{L},\text{L},\psi)} \widehat{\tau})^{(\text{S},\text{L})}}$$

4

HELIUM

This chapter discusses the design philosophy of the Helium compiler (4.1), its architecture (4.2) and the Core language (4.3).

4.1 DESIGN PHILOSOPHY

Helium is a compiler for Haskell, developed at Utrecht University [9]. The target audience of Helium is students learning Haskell, and is not intended for industry use. The design philosophy of the compiler is to give high-quality (type) error message. Consider the function *(sic)*, which is a syntax error as the closing bracket should be replaced by a parenthesis. GHC, the de facto standard Haskell compiler [7], gives an error which only points out the point of the parse error (4.1). If this function is executed in GHCi, only the parse error message appears. The red arrow pointing to the syntax error is not present, leaving the user to figure out where exactly the error occurs.

```
test.hs:3:7: error: parse error on input `]'
  |
3 | (4 + 1]
  |       ^
```

FIGURE 4.1: *Message from GHC for syntax errors*

Helium is more helpful by pointing out the cause of the syntax error and giving a hint to fix the problem 4.2.

```
(3,7), (3,1): Unexpected close bracket ']'
  Hint: Expecting a close bracket for '('
Compilation failed with 1 error
```

FIGURE 4.2: *Message from Helium for syntax errors*

Another area where the Helium compiler aims to be more helpful is type errors. Consider the function

$f :: [Bool]$
$f = map\ [1,2,3]\ even$

The error here is that the arguments to *map* are switched, as the first argument should be the function and the second argument should be the list. GHC throws two error messages, one for each argument (4.3). The first one claims that it expects a function but instead receives a list of something, which is a bit confusing given that the actual types of the arguments are not used. The second error expects a list instead of a function and gives the same message, but then assesses that the probable cause of this error is the function *even* being applied to too few arguments, pointing the user in the wrong direction as the actual fix is switching the arguments.

```
test.hs:4:9: error:
    * Couldn't match expected type `a0 -> Bool' with actual type `[a2]'
    * In the first argument of `map', namely `[1, 2, 3]'
      In the expression: map [1, 2, 3] even
      In an equation for `f': f = map [1, 2, 3] even
  |
4 | f = map [1, 2, 3] even
  |         ^^^^^^^^^

test.hs:4:19: error:
    * Couldn't match expected type `[a0]' with actual type `a1 -> Bool'
    * Probable cause: `even' is applied to too few arguments
      In the second argument of `map', namely `even'
      In the expression: map [1, 2, 3] even
      In an equation for `f': f = map [1, 2, 3] even
  |
4 | f = map [1, 2, 3] even
  |                   ^^^^
```

FIGURE 4.3: *Message from GHC for type errors*

Helium provides one error message with the full type signature of the function, including the instantiated types Bool and Int, and gives the user the probable fix of switching the arguments (4.4).

```
(4,5): Type error in application
 expression       : map [1, 2, 3] even
 term             : map
   type           : (a -> b) -> [a]            -> [b    ]
   does not match : [Int]     -> (Int -> Bool) -> [Bool]
 probable fix     : re-order arguments

Compilation failed with 1 error
```

FIGURE 4.4: *Message from Helium for type errors*

Recent work in the Helium compiler includes the addition of heap recycling analysis [22], normalizing the core representation [13], higher-ranked region inference for compile-time garbage collection [4], and type error diagnosis for OutsideIn(X) [3].

4.2 Architecture

The Helium compiler has eleven phases, of which the first eight are:

Lexing converts the input into a sequence of tokens, removing whitespace.

Parsing builds a representation of the program according to the Haskell Language Report [14].

Importing handles the importing of Prelude and other modules specified in a Haskell file.

Resolving operators resolves operators based on their infix specification in lists of expressions.

Static checks performs static checks on the program such checking for undefined variables and scope errors, but also for warnings like missing type signatures;

Kind inferencing infers the kinds of types. This phase is optional and turned off by default.

Type inference directives loads typing strategies which help the type inferencer to give better error messages.

Type inferencing and checking infers the types of expression and checks whether they match the type signatures. The generated constraints are solved using Top [22].

The next two phases handle desugaring to intermediate representations: Core and Iridium. The abstract syntax tree from Helium is desugared to Core (see 4.3). The resulting Core representation is then desugared to Iridium. Iridium is a strict, imperative language which handles memory allocation and laziness. One of the functions defined in Iridium is *seq*. The two optimizations which are performed in this phase are dead code removal and tail recursion [4]. Finally, Iridium is desugared to the LLVM back-end. LLVM defines primitive operations such as addition and comparison of integers [11].

4.3 Core

Core is the first intermediate representation used in the Helium compiler. It is an explicitly-typed variant of System F. Inside Core, various optimizations are performed, such as let-inlining and strictness analysis. As the new strictness analysis developed in this thesis is set to replace the old one, it will also be performed on Core. This section contains information on the module system (4.3.1), type system (4.3.2), expression representation (4.3.3), optimization passes (4.3.4) and the existing strictness analysis (4.3.5).

4.3.1 Module

A module in Core consists of the following five fields: name, major version, minor version, imports and a list of declarations. A declaration can be either a **DeclValue** (functions), **DeclAbstract** (imported functions), **DeclCon** (constructors), **DeclExtern** (foreign functions), **DeclTypeSynonym** (type synonyms and newtypes) or **DeclCustom** (datatypes, infix, ...). All declarations have a name, a flag whether they are exported or not, a type (except DeclCustom) and a list of customs which contain any additional information.

4.3.2 Type system

The type system of Core is defined in figure 4.5:

TCon represents type constants. They can be regular datatypes, tuples and typeclasses. The function arrow is also a constant, but needs to be used within an application.

```
data Type         = TCon    TypeConstant
                  | TVar    TypeVar
                  | TAp     Type Type
                  | TForall Quantor Kind Type
                  | TStrict Type
data TypeConstant = TConDataType             Id
                  | TConTuple                Int
                  | TConTypeClassDictionary  Id
                  | TConFun
newtype Quantor   = Quantor (Maybe String)
data Kind         = KFun Kind Kind
                  | KStar
```

FIGURE 4.5: *Type datatype in Core*

TVar represents type variables, which are represented as De Bruijn indices, meaning TypeVar is just a synonym for Int.

TAp is used for application to datatypes. It is also used to represent functions, with the function $\tau_1 \rightarrow \tau_2$ represeted as (TAp (TAp (TCon TConFun) τ_1) τ_2).

TStrict is used to communicate strict fields in constructors or strictness information from imported functions. It should not occur in any other place.

TForall allows for quantification over types. Quantors can have an optional name. Kind is the type of a type constructor. For example, Int has kind * (KStar) while Maybe has kind * -> * (KFun KStar KStar) as it has one argument.

4.3.3 EXPRESSION

The expression syntax of Core is defined in figure 4.6.

Con represents data constructors and tuples.

Lit represents literals, such as integers and strings.

Var represents variables. Unlike type variables, these are referenced by identifier.

Let represents let bindings. A let binding has a variable with an expression bound to it, and a body. The bindings can be strict, in which case the binding is evaluated before the body, or recursive.

Lam represents lambda functions, which have a flag whether it is strict or not, a variable and a body.

Ap represents function or constructor application.

Match represents pattern matching on a variable, including if-then-else statements. As the variable needs to be evaluated before the pattern match, it is always preceded by a strict let or lambda which evaluates the value to be matched on. Patterns which can be matched on are constructors (including tuples) and literals. A default pattern can be specified for the user, or will be placed there by the parser if it is possible for the pattern match to fail.

Forall represents quantification. If the type of the expression is quantified, the expression should be preceded by foralls as well.

ApType represents type instantiation. If the type of the expression is quantified, all occurrences of the type variable will be replaced by the given type.

```
data Expr     = Con     Con
              | Lit     Literal
              | Var     Id
              | Let     Binds Expr
              | Lam     Bool Variable Expr
              | Ap      Expr Expr
              | Match   Id Alts
              | Forall  Quantor Kind Expr
              | ApType  Expr Type
data Variable = Variable {variableName :: Id, variableType :: Type}
data Binds    = Rec [Bind]
              | Strict  Bind
              | NonRec  Bind
data Bind     = Bind    Variable Expr
type Alts     = [Alt]
data Alt      = Alt Pat Expr
data Pat      = PatCon Con [Type] [Id]
              | PatLit  Literal
              | PatDefault
data Literal  = LitInt     Int IntType
              | LitDouble  Double
              | LitBytes   Bytes
data Con      = ConId      Id
              | ConTuple   Int
```

FIGURE 4.6: *Expr datatype in Core*

4.3.4 PASSES

Core contains the following optimization passes:

Rename makes all identifiers globally unique.

Saturate saturates all calls to constructors.

LetSort sorts all let bindings such that variables only occur after definition, or are actually recursive.

LetInline1 inlines let bindings if the definition is used at most once.

LetInline2 performs let-inlining again. This pass is done twice consecutively because it attempts to reach a fixpoint. There is no guarantee of optimality after these passes, but the result after two iterations is considered good enough compared to adding extra passes until a fixpoint is reached.

Normalize moves all non-trivial subexpressions to let bindings. This is useful for the strictness pass because every application only contains variables as arguments, and those can be made strict in the corresponding let binding.

Strictness1 makes all expressions which are guaranteed to be used strict (see 4.3.5).

Strictness2 performs the strictness analysis again. Like LetInline, the strictness pass also has another iteration to improve the solution.

RemoveAliases removes let bindings which only give a new name for a variable, with every occurrence of the variable renamed to its original identifiers.

ReduceThunks transforms cheap expressions such as literal and constructors to strict bindings, as this transformation does not change the semantics of the program.

Lift lifts lambdas and non-strict lets to top level functions.

Strictness3 runs the strictness analysis again as Lift introduces new top-level functions. Another strictness pass can use this extra information to reach an even better solution.

4.3.5 STRICTNESS ANALYSIS

Strictness analysis is already included in the current optimization passes. It is executed three times, twice before lifting and once after lifting. Because it is a fixed point algorithm with only two iterations, there is no guarantee of optimality. The existing analysis will serve as benchmark to compare against the replacement strictness analysis, which is based on a type and effect system and only needs one pass to reach its solution. To give an idea of what the strengths and weaknesses of the current analysis are, a short description of the current pass is given.

The goal of strictness analysis is to turn as many non-recursive let bindings into strict bindings and non-strict lambdas into strict lambdas, without changing the semantics of the program. On encountering a variable in the expression, the arity of the variable is compared against the number of arguments given to see if it is fully applied, which means it is safe to derive information. For variables which do not define functions, this is always the case. For variables which define functions, this depends on the number of applications beforehand. If the arity matches the number of arguments, the applied variables are added to the set of variables which can be made strict, otherwise they are omitted. On a case expressions, the sets from all cases are intersected, as only variables which are strict in all cases can be made strict. In applications, the union of the set from the function and the set from the argument is taken, as being strict in either branch is enough. On let and lambdas, all variable identifiers which are in the set obtained from the body can be transformed from non-strict to strict. If this is the case, the remaining set can be propagated as the let or lambda is now guaranteed to be executed.

There were concerns over the soundness of the current analysis, which is one of the reasons a replacement is wanted. Upon inspection, the following example resulted in an illegal transformation:

$$
\begin{array}{l}
f :: Bool : export\ f \\
\quad = \textbf{let}\ 1 : Bool = True; \\
\quad\ \textbf{in let}\ 2 : () \to () = \textbf{let}\ 3 : () = \bot\ \{()\}; \\
\qquad\qquad\qquad\qquad\qquad \textbf{in}\ seq\ \{()\}\ \{()\}\ 3; \\
\qquad \textbf{in}\ seq\ \{() \to ()\}\ \{Bool\}\ 2\ 1;
\end{array}
$$

Because *seq* is strict in both arguments when fully applied, identifiers 1 and 2 were made strict, which is correct. However, it also made identifier 3 strict, which is not allowed as it was supplied to the partially applied *seq*. As a result, the program now returns $\bot$ instead of *True*, and $\bot$ diverges. The cause of this issue was found in the handling of variable. If a variable representing a function is fully applied, it returns all information regarding its arguments, as it should. However, the exact same information was returned in the partially applied case, resulting in the unsoundness. This bug has been fixed to make the comparison against the new analyses fairer. The results of the experimentation have not hinted at any other case of unsoundness.

Another disadvantage to the current analysis is its lack of information across modules. Functions store a list of booleans as strictness information, one per argument. Within its own module, these flags can be used to determine the strictness of arguments. However, this information is not stored after compilation of the module, meaning it is not available for any module which imports it. This means functions defined in Prelude do not offer strictness information, with the only alternative being redefinition of every function which defeats the purpose of a modular system.

Related work

The related work describes analyses which are related to strictness analysis (5.1), strictness analysis in UHC (5.2), and recent work in Helium (5.3).

5.1 Related analyses

5.1.1 Relevance typing

The goal of strictness analysis is to identify which functions can be evaluated in a strict manner, as opposed to lazy [10]. Changing these functions does not change the semantics of the program but generally improves the performance cost. However, strictness analysis is undecidable and often too conservative in its estimation.

Alternatively, one could consider relevance typing. A variable x is relevant to a term τ if x is guaranteed to be evaluated whenever τ is evaluated. In the analysis described by Holdermans and Hage, the annotation S is used for terms which provide relevant abstractions, and L for those who do not. Note that S and L can be interpreted as strict and lazy, but also as small and large, as a partial order $(\{\mathbf{S}, \mathbf{L}\}, \sqsubseteq)$ can be imposed, characterized by $S \sqsubseteq L$. Also note that strictness and relevance analysis do not necessarily agree. For instance, λx.error is strict but not relevant.

To optimize the program by introducing as many strict applications as possible, we have to show that for a non-strict application $\tau_1 \tau_2$, successfully evaluating τ_1 will always result in a relevant abstraction. The optimization is justified by the observation that τ_1's relevance means the evaluation of τ_2 is required. Because it is required to be evaluated, it is better to evaluate it strictly rather than non-strict. Holdermans and Hage demonstrated that a relevance type system for a lazy language can be used to make programs more strict.

Relevance typing cannot simply handle manual strictness in programs, meaning it is unsuitable for real-world languages like Haskell. The system can be refined to include annotations of applicativeness as well. These use the same annotations and partial order as before, but are annotated on the arguments instead of the function arrows. A term is applicative if it guaranteed to be applied to an argument at least once. Applicativeness implies relevance: if an expression is guaranteed to be applied to an argument, it is also guaranteed to be evaluated [10].

5.1.2 Usage analysis

Usage analysis is a combination of sharing analysis and uniqueness typing. Sharing analysis and uniqueness typing both try to determine if a part of a program is only used once [6]. Sharing analysis uses this information to avoid unnecessary closure updates, as the value is only used once it is not necessary to write the result back to the thunk. Uniqueness typing is concerned with the referential transparency of objects such as file handles which are meant to only be used once.

Hage, Holdermans and Middelkoop provide a generic usage analysis which combines the previous two analyses. In their paper, they show a subsumption relation between the analyses, an explicitly typed calculus including type polymorphism, effect polymorphism and subeffecting. Because subeffecting is incorporated into a framework of qualified types, many tools and techniques can be reused [6].

Walker describes substructal type systems, which limit the number of uses on operations and data structures [25]. The information from this type system can be used for compiler optimizations. If an operation is only allowed to be used once, it is beneficial to immediately evaluate it when it is used.

In his PhD thesis, Wansbrough provides a polymorphic usage analysis implemented in the Glasgow Haskell Compiler [26]. In the appendix, he observes that strictness is a property of usage. An argument is strict if it is used at least once. Wansbrough provides an extended annotation domain which incorporates strictness and absence analysis in addition to usage annotations.

5.1.3 Cardinality analysis

Cardinality analysis answers the following three questions:

- How often is a lambda expression called?
- Which components of a data structure are never evaluated?
- How many times is a syntactic thunk evaluated?

There is a difference between called and evaluated. Under normal circumstances, any argument which is called is also evaluated. However, Haskell has the seq function which forces evaluation of its first argument. If this argument is a function, it is evaluated to a lambda, but this does not guarantee that this lambda is used elsewhere. Forcing evaluation on a function which will never be called should be avoided.

Lambda expressions which are only called once are called one-shot lambdas. These are fairly common in functional programming, which makes them a target for optimization. For instance, an optimizing compiler would perform short-cut deforestation to fuse two maps into one instead of needing an intermediary list. However, currying complicates this analysis. The analysis needs to distinguish between how often an argument is called and how often it is used. It beneficial for a function to be called many times but used only once instead of being called once but used many times. In the first case, inlining functions results in an improvement of the program as it reduces the number of thunks. In the second case, inlining should be avoided.

Arguments which are never used are a waste of memory. However, optimizing unused parts of data structures is not trivial. Consider a tuple of two integers, of which only the first is used. If this tuple is referred to by a variable, it does not know yet that the second value will not be used, thus passing around unused arguments which could have been optimized.

Lazy languages use thunks to memoize parts of the program which have not been evaluated yet. When a thunk is evaluated, the result is written back to the thunk so the evaluation does not have to take place again whenever the thunk is used again. If a thunk is only evaluated once, there is no need to write back the result as it will never be used again. Preventing single-entry thunks from updating themselves after evaluation is one of the optimizations cardinality analysis sets out to achieve [20].

5.1.4 Counting analysis

Counting analysis is a combination of three different analyses: absence, strictness and usage (which consists of sharing analysis and uniqueness typing). All analyses have in common that they determines the number of times an expression is evaluated. Absence analysis determines if an expression is never used and thus can safely be removed. Strictness analysis determines if an expression is used at least once, which means it can be evaluated strictly rather than lazily. Sharing analysis determines if an expression is used at most once and hence does not require to write its result back to its closure. Uniqueness typing determines if an annotated expression is used at most once. Absence, strictness and sharing analysis are all optimizing analyses, while uniqueness typing is concerned with the correctness of the program.

Sergey et al combined absence, sharing and strictness into cardinality analysis (see 5.1.3) [20]. Verstoep combined the four individual analysis into one analysis in his thesis [24]. Bremer implemented counting analysis in the Utrecht Haskell Compiler, a different Haskell compiler originating from Utrecht University [2].

5.1.5 Higher-ranked polyvariance

A monovariant analysis associates a single abstract value per expression in the program. Polyvariance allows for a most general value to be associated with a let-bound expression, which can have different instantiations per call site. Going another step further, we can generalize over properties of lambda-bound identifiers. This is called higher-ranked polyvariance. Thorand and Hage introduced a higher-ranked polyvariant type system for dependency analysis. Dependency analysis is a generic analysis which could serve as blueprint for other analyses [21].

5.2 Strictness analysis in UHC

Strictness analysis has already been implemented in the Utrecht Haskell Compiler [5]. The Utrecht Haskell Compiler is a different compiler developed at Utrecht University, and while it has some things in common with Helium, they have largely grown their separate ways over the years. The theoretical background is based on the work of Lokhorst [12], with the first implementation done by Verburg [23]. Passalaque Martins later extended the analysis to include polyvariance [19].

Lokhorst defined strictness optimization in an typed intermediate language [12]. He implemented optimizations for first-order functions, which can be extended to include other analyses. Furthermore, he discussed extensions to include strictness optimizations for higher-order functions.

Verburg later implemented a monovariant strictness analysis in UHC [23]. The reason for implementing a monovariant but not a polyvariant analysis was that the latter also required major work in the compiler itself. The implementation was based on the relevance typing approach by Holdermans and Hage [10].

Passalaque Martins developed a polyvariant strictness analysis for UHC. His research consists of a polyvariant system with support for higher-order functions and user-annotated strictness [19].

The work from Verburg and Passalaque Martins cannot directly be copied into this research due to the differences in compiler architecture. However, certain backgrounds and analysis frameworks can be utilized in this research.

5.3 Recent work in Helium

5.3.1 Heap recycling analysis

Heap recycling analysis is the most recent addition to Helium. The thesis by Van Klei provides an implementation of heap recycling analysis in Helium [22]. When updating the value of an expression, fresh

heap space is allocated for the result. The original allocation is now redundant and could safely be removed, but this requires garbage collection. It would be much easier if the thunk could be reused with the result and avoid allocating extra memory. Van Klei's implementation in Helium provided considerable improvements in terms of memory usage for simple programs.

5.3.2 Higher-ranked inference for compile-time garbage collection

Originally, Helium used the LVM back end. However, this back end could no longer support 64-bit machines which are increasingly common nowadays. Hence, a switch was made to LLVM [11]. This introduced a new intermediate language into the compiling pipeline: Iridium.

In his Master's thesis, De Wolff investigated higher-ranked inference for compile-time garbage collection. Tied within this research was a simple strictness analysis, which only operates on first-order parts of the program. Strictness analysis is performed before the region analysis, so that the complexity of the region analysis is reduced. After evaluating a thunk, the garbage collector can remove that thunk, which is much harder to do using region inference. The strictness analysis is performed on the Core language [4].

Research questions and setup

The chapter describes the main research question and three subquestions (6.1) and the setup of the experiment (6.2).

6.1 Research questions

[MQ] What are the trade-offs between precision and analysis cost of strictness analysis in a real-world compiler?
This is the main research question for this thesis. A simple strictness analysis can give some simple improvements, but will probably not be able to handle more complicated cases. The analysis can be made more precise, at the cost of a more complex implementation. There are trade-offs between the precision of the analysis and the cost of the analysis in terms of memory usage and speed. The two different versions of the analysis scheduled to be implemented are a monovariant analysis and a polyvariant analysis.

[SQ1] Can relevance and applicativeness typing analysis be implemented in the Helium compiler?
Helium is a compiler for Haskell, satisfying the "real-world compiler" part of the main question. Implementing the analysis defined in chapter 3 in Helium will be the first part of this thesis and the basis for the other subquestions.

[SQ2] Can a representative benchmark be constructed to compare trade-offs?
To compare the different versions, a benchmark has to be constructed. The benchmarks consist of multiple programs which can or cannot be handled by certain analyses. The benchmarks should also include measurements for time and space usage to measure the analysis cost.

[SQ3] What are the trade-offs against the current strictness analysis in Helium?
Helium already contains a strictness analysis in the Core pipeline, which already provides some performance improvement. However, as described in section 4.3.5, there are many cases in which the strictness analysis does not infer strictness while it definitely should be possible. There are also suspicions that the

analysis is unsound. Furthermore, it currently requires multiple passes which still results in a suboptimal solution. This analysis can be used to compare the improved strictness analysis constructed in this thesis.

6.2 SETUP

There are three different analyses: a monovariant analysis, a polyvariant analysis, and the original analysis. They are compared on two measures: precision and analysis cost. The expectation is that the polyvariant analysis is more precise than the monovariant analysis but has a worse analysis cost. Both analyses are more advanced than the original analysis and should be much more precise, while limiting the difference in analysis cost.

Precision measures the number of let and lambdas which are made strict. A number of these bindings are already strict from the parser, mainly let bindings which precede pattern matches. The precision is measured after the second strictness pass, since ReduceThunks also introduces strictness, which might not give an accurate measure of the precision of the strictness analysis anymore. Note that the mono- and polyvariant analysis only need one pass of the strictness, and the second pass is not executed. After this pass, the Core source file is examined and the number of exclamation marks is counted. It is important that all of these transformations are in fact sound.

Analysis cost is measured by the time it takes to compile the code. As it is difficult to measure the effect of just the strictness pass, and the result of strictness analysis influences passes which come after it, the time it takes to perform the entire compilation is measured. The time is measured using RTS with profiling enabled.

The source file which is used to perform the experiment is *Test.hs*, displayed in figure 6.1. The analysis cost is measured over the entire compilation of this module, which includes importing Prelude. Prelude contains roughly 1000 lines of Haskell code, which is desugared to roughly 5000 lines of Core. Prelude defines a lot of functions but also contains some complex and diverse functions such as IO functions and type classes. The precision is measured on both the Prelude and Test module. While Prelude defines a lot of functions, it does not use all of them in other functions. For instance, ($) is defined but never used internally. The Test module uses some of the functions from Prelude and applies them to arguments to see if they can be made strict.

```haskell
module Test where
rconst :: Bool → Bool → Bool
rconst x y = y
const3 :: Bool → Bool → Bool → Bool
const3 x y z = x
nstrict :: Bool → Bool → Bool
nstrict x y = True
uconst, urconst, uconst3 :: Bool
uconst = const True False
urconst = rconst True False
uconst3 = const3 True False True
uconstp, urconstp, uconst3p :: Bool → Bool
uconstp = const True
urconstp = rconst True
uconst3p = const3 True False
uconst3p' :: Bool → Bool → Bool
uconst3p' = const3 True
fold1, fold2, fold3, fold4, fold5, fold6, fold7, fold8 :: Bool
fold1 = foldr seq     True [False, True]
fold2 = foldr const   True [False, True]
fold3 = foldr rconst  True [False, True]
fold4 = foldr nstrict True [False, True]
fold5 = foldl seq     True [False, True]
fold6 = foldl const   True [False, True]
fold7 = foldl rconst  True [False, True]
fold8 = foldl nstrict True [False, True]
applys, applyl :: Bool
applys = id $ True
applyl = (const False) $ True
applyp :: Bool → Bool
applyp = const $ True
constx3, constx25 :: Bool
constx3 = const (const True False) (const True False)
constx25 = const (const True False) (const True)
constl, constlf, constlp :: Bool
constl = let f = λx y → x in f True False
constlf = let f = λx y → x in f (f True False) (f True False)
constlp = let f = λx y → x in f (f True False) (f True)
caseall, case1a, case1b, casenone :: Bool → Bool → Bool
caseall  x y = if x then y else y
case1a   x y = if x then y else True
case1b   x y = if x then True else y
casenone x y = if x then True else False
seq2 :: Bool
seq2 = (seq False) `seq` True
```

FIGURE 6.1: *Test.hs*

IMPLEMENTATION

The implementation section contains the implementation of strictness analysis (chapter 3) in Helium (chapter 4). This includes updated typing rules (7.1), an algorithmic implementation (7.2), and the adaptation to a polyvariant analysis (7.3).

7.1 TRANSFORMATION RULES

The type system, terms and transformation rules described in section 3.2 only support a small portion of the Helium system. Therefore, we need to extend each of them to be closer to the real system. We assume variables, constructors, literals, datatypes and type variables are sets ranged over x, c, l, d and α.

$$x \in \textbf{Var}$$
$$c \in \textbf{Con}$$
$$l \in \textbf{Lit}$$
$$d \in \textbf{Data}$$
$$\alpha \in \textbf{TypeVar}$$

The definition for the type environment remains unchanged, mapping from variables to a type, relevance annotation and applicativeness annotation.

$$\widehat{\Gamma} \in \textbf{TEnv} = \textbf{Var} \to_{\text{fin}} \widehat{\textbf{Type}} \times \textbf{Ann} \times \textbf{Ann}$$

7.1.1 TERMS

Figure 7.1 describes the updated terms for the new transformation rules, with the constructor on the right hand side describing their equivalent constructors in the Helium type system.

Compared to the original system, true and false have been combined with all other data constructors in a single term for constructors. Natural numbers are now part of the literals, which also includes floats, characters and strings. Variables remain the same as they were. Abstraction is split into two terms, reflecting the ability to make lambdas strict. They also store the annotated type of the variable, which is

$t ::= c$	Con
$\mid l$	Lit
$\mid x$	Var
$\mid \lambda x\{\hat{\tau}\}.t \mid \lambda!x\{\hat{\tau}\}.t$	Lam
$\mid t_1 t_2$	Ap
$\mid \textbf{let}\ x\{\hat{\tau}\} = t_1\ \textbf{in}\ t_2 \mid \textbf{let!}\ x\{\hat{\tau}\} = t_1\ \textbf{in}\ t_2 \mid \textbf{letrec}\ \overline{x_n\{\hat{\tau}_n\} = t_n}\ \textbf{in}\ t$	Let
$\mid \textbf{case}\ x\ \textbf{of}\ \overline{a_n}$	Case
$\mid \textbf{forall}\ \alpha.t$	Forall
$\mid t\{\hat{\tau}\}$	ApType
$a ::= c\overline{x_n\{\hat{\tau}_n\}} \rightarrow t$	PatCon
$\mid l \rightarrow t$	PatLit
$\mid _ \rightarrow t$	PatDefault

FIGURE 7.1: *Terms and patterns*

helpful in the algorithm. Application (both function and constructor) no longer has its strict counterpart, reflecting the change in introducing strictness. Let bindings did not exist in the original system, and have been added in three flavors: non-strict, strict and recursive. Like abstractions, they also store the type of the variable. If-then-else is generalized to a case system, which can pattern match on any datatype and have any number of cases. Like in Helium, there needs to be a strict let or lambda defining the variable to be matched upon. A pattern a is either a data constructor with possible identifiers for constructor fields, a literal or a default pattern which matches all cases. Finally, the system allows for type generalization and instantiation.

7.1.2 ANNOTATED TYPES

Figure 7.2 described the annotated types, which see a smaller change compared to the terms.

$\hat{\tau} ::= d\,\overline{\hat{\tau}}$	TCon, TAp TCon, TAp (TAp TCon) etc.
$\mid \hat{\tau} \xrightarrow{(\psi,\varphi,\psi)} \hat{\tau}$	TAp (TAp (TCon ConFun)
$\mid \alpha$	TVar
$\mid \forall\alpha.\hat{\tau}$	TForall
$\mid !\hat{\tau}$	TStrict
$\varphi, \psi ::= S \mid L$	

FIGURE 7.2: *Annotated types*

Booleans and natural numbers are grouped together, and all other possible datatypes (even those of kinds other than *) are combined into one rule for datatypes. Tuples are assumed to be part of the

datatypes for the purpose of this chapter, with the implementation of these delegated to Appendix A. Function arrows remain annotated with three annotations, one relevance and two applicativeness annotations. Furthermore, type variables, the ability to quantify over them, and a strictness type are added. The latter should only be used for datatypes and for functions defined in Iridium.

7.1.3 Transformation rules

With the extended terms and types, the transformation rules described in 3.2 also have to be extended. The transformation no longer occurs on applications but instead focuses on variables in lets and lambdas. The transformation rules are still of the form $\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi,\psi)}$, where $\widehat{\Gamma}$ represents the environment which maps variables to a triplet containing the annotated type, a relevance and applicativeness annotation, t represents the original term, t' represents the transformed term, $\widehat{\tau}$ represents the annotated type of the term, and φ and ψ represent the relevance and applicativeness context respectively.

Constructors, literals and variables

The transformation rules for constructors, literals and variables are described in figure 7.3.

$$\frac{}{[] \vdash c \rhd c : \widehat{\tau}^{(\varphi,\psi)}} \text{[r-con]}$$

$$\frac{}{[] \vdash l \rhd l : \widehat{\tau}^{(\varphi,\mathrm{L})}} \text{[r-lit]}$$

$$\frac{}{[x \mapsto (\widehat{\tau},\varphi,\psi)] \vdash x \rhd x : \widehat{\tau}^{(\varphi,\psi)}} \text{[r-var]}$$

Figure 7.3: *Transformation rules for constructors, literals and variables*

Rule [r-con] handles constructors, and is a generalization of the [r-true] and [r-false] rules in the original inference rules. The original system was limited to booleans, whereas the actual system can handle any (user-defined) datatype. This also allows for constructors with arguments, and thus the applicativeness context is no longer mandatory to be L. The spirit of the rule remains the same, as the constructor can only be used on an empty environment, meaning all variables in it have to be mapped to L by the weakening rule.

Rule [r-lit] handles literals, which can be integers, doubles, characters or strings. The rule has the same format as [r-con], except literals cannot be applied to arguments, which means the applicativeness context has to be L here.

Rule [r-var] is the only syntax-directed rule which remains exactly the same. If a variable occurs in the expression, we set its corresponding values in the environment to the current contexts. All other variables occurring in the environment have to be weakened.

Application

The transformation rule for applications is described in figure 7.4.

$$\frac{\widehat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\widehat{\tau}_2 \xrightarrow{(\psi_2,\varphi_0,\psi)} \widehat{\tau})^{(\varphi,\varphi)} \quad \widehat{\Gamma}_2 \vdash t_2 \rhd t_2' : \widehat{\tau}_2^{(\varphi\sqcup\varphi_0,\varphi\sqcup\psi_2)}}{\widehat{\Gamma}_1 \diamond^{\sqcap} \widehat{\Gamma}_2 \vdash t_1 t_2 \rhd t_1' t_2' : \widehat{\tau}^{(\varphi,\psi)}} \text{[r-app]}$$

Figure 7.4: *Transformation rule for applications*

The original system included three different rules for application: one to turn a normal application into a strict application ([r-app1]), one which keeps the normal application ([r-app2]) and a rule which

handles strict applications ([r-sapp]). In Helium, strictness can be introduced this way as well, though it requires inserting the $! operator everywhere. Instead, Helium records strictness at the definition of variables in lambda- or let bindings.

For the application rule, it means one rule suffices in the system. Rules [r-app1] and [r-app2] can be combined into one rule, while [r-sapp] becomes obsolete as the syntax is no longer present. The relevance context of the applicant is set to the join of the incoming relevance context φ and relevance annotation φ_0 on the function arrow in the type of the function. This is an accurate representation according to the original rules, as rule [r-app1] would join φ with S, which results in φ, and [r-app2] would join φ with L, which results in L. The diamond operator for context splitting has changed appearance to receive the meet operator as input, but this is equivalent to the original definition of $\diamond$. The extended definition of this operator is explained later on.

Abstraction

The transformation rules for abstractions are described in figure 7.5.

$$\frac{\psi \blacktriangleright \hat{\Gamma} \quad \hat{\Gamma}[x \mapsto (\hat{\tau}, \mathrm{S}, \psi_1)] \vdash t_1 \rhd t_1' : \hat{\tau}_0^{(\mathrm{S},\psi_2)}}{\hat{\Gamma} \vdash \lambda x\{\hat{\tau}\}.t_1 \rhd \lambda ! x\{\hat{\tau}\}.t_1' : (\hat{\tau} \xrightarrow{(\psi_1,\mathrm{S},\psi_2)} \hat{\tau}_0)^{(\varphi,\psi)}} \text{ [r-abs1]}$$

$$\frac{\psi \blacktriangleright \hat{\Gamma} \quad \hat{\Gamma}[x \mapsto (\hat{\tau}, \mathrm{L}, \psi_1)] \vdash t_1 \rhd t_1' : \hat{\tau}_0^{(\mathrm{S},\psi_2)}}{\hat{\Gamma} \vdash \lambda x\{\hat{\tau}\}.t_1 \rhd \lambda x\{\hat{\tau}\}.t_1' : (\hat{\tau} \xrightarrow{(\psi_1,\mathrm{L},\psi_2)} \hat{\tau}_0)^{(\varphi,\psi)}} \text{ [r-abs2]}$$

$$\frac{\psi \blacktriangleright \hat{\Gamma} \quad \hat{\Gamma}[x \mapsto (\hat{\tau}, \mathrm{S}, \psi_1)] \vdash t_1 \rhd t_1' : \hat{\tau}_0^{(\mathrm{S},\psi_2)}}{\hat{\Gamma} \vdash \lambda ! x\{\hat{\tau}\}.t_1 \rhd \lambda ! x\{\hat{\tau}\}.t_1' : (\hat{\tau} \xrightarrow{(\psi_1,\varphi_1,\psi_2)} \hat{\tau}_0)^{(\varphi,\psi)}} \text{ [r-sabs]}$$

FIGURE 7.5: *Transformation rules for abstractions*

As described in the previous section, Helium uses a different way of introducing strictness. Bang patterns are introduced on the variables in abstractions instead of the applications. This means the abstraction rule has to be split into three different rules, similar to how the application rule was split into three rules in the original system. Furthermore, each variable now has its type information stored as well, though this is the same type as $\hat{\tau}_1$ in the original rule.

Rule [r-abs1] corresponds to rule [r-app1] in the original system, where a non-strict abstraction is turned into a strict abstraction. A variable can only be made strict if it is both relevant, meaning the middle annotation on the arrow has to be S in order for the transformation to be valid. The containment remains unchanged, and the relevance context of the term inside the abstraction is still set to S to measure the relevance of this particular variable within its own scope.

Rule [r-abs2] corresponds to rule [r-app2] in the original system, which does not turn a regular abstraction into a strict abstraction. The new rule therefore does not introduce a bang on the variable. This is the case when the relevance annotation is equal to L.

Finally, rule [r-sabs] corresponds to rule [r-sapp], which handles abstractions which are already declared as strict. The relevance annotation on the function arrow is not important for the transformation rules, as the lambda is strict regardless of whether it is relevant. We can add the variable to the environment with its relevance annotation set to S.

Let bindings

The transformation rules for let bindings are described in figure 7.6.

Let bindings do not exist in the original type inference rules, but play an important role in Helium. The let bindings are one of two ways (the other being lambda-abstractions) to introduce strictness. As

$$\frac{\varphi \blacktriangleright \widehat{\Gamma}_1 \quad \widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \mathrm{S}, \psi_1)] \vdash t_2 \rhd t_2' : \widehat{\tau}_0^{(\mathrm{S},\psi)} \quad \widehat{\Gamma}_2 \vdash t_1 \rhd t_1' : \widehat{\tau}^{(\varphi,\varphi)}}{\widehat{\Gamma}_1 \diamond^{\sqcap} \widehat{\Gamma}_2 \vdash \mathbf{let}\, x\{\widehat{\tau}\} = t_1 \,\mathbf{in}\, t_2 \rhd \mathbf{let!}\, x\{\widehat{\tau}\} = t_1' \,\mathbf{in}\, t_2' : \widehat{\tau}_0^{(\varphi,\psi)}} \text{[r-let1]}$$

$$\frac{\varphi \blacktriangleright \widehat{\Gamma}_1 \quad \widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \mathrm{L}, \psi_1)] \vdash t_2 \rhd t_2' : \widehat{\tau}_0^{(\mathrm{S},\psi)} \quad \widehat{\Gamma}_2 \vdash t_1 \rhd t_1' : \widehat{\tau}^{(\varphi \sqcup \varphi_1,\mathrm{L})}}{\widehat{\Gamma}_1 \diamond^{\sqcap} \widehat{\Gamma}_2 \vdash \mathbf{let}\, x\{\widehat{\tau}\} = t_1 \,\mathbf{in}\, t_2 \rhd \mathbf{let}\, x\{\widehat{\tau}\} = t_1' \,\mathbf{in}\, t_2' : \widehat{\tau}_0^{(\varphi,\psi)}} \text{[r-let2]}$$

$$\frac{\varphi \blacktriangleright \widehat{\Gamma}_1 \quad \widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \mathrm{S}, \psi_1)] \vdash t_2 \rhd t_2' : \widehat{\tau}_0^{(\mathrm{S},\psi)} \quad \widehat{\Gamma}_2 \vdash t_1 \rhd t_1' : \widehat{\tau}^{(\varphi \sqcup \varphi_1, \varphi \sqcup \psi_1)}}{\widehat{\Gamma}_1 \diamond^{\sqcap} \widehat{\Gamma}_2 \vdash \mathbf{let!}\, x\{\widehat{\tau}\} = t_1 \,\mathbf{in}\, t_2 \rhd \mathbf{let!}\, x\{\widehat{\tau}\} = t_1' \,\mathbf{in}\, t_2' : \widehat{\tau}_0^{(\varphi,\psi)}} \text{[r-let!]}$$

$$\frac{\begin{array}{c}\forall i. 1 \le i \le n : \widehat{\Gamma}_2[x_1 \mapsto (\widehat{\tau}_1, \varphi_1, \psi_1), \dots, x_n \mapsto (\widehat{\tau}_n, \varphi_n, \psi_n)] \vdash t_i \rhd t_i' : \widehat{\tau}_i^{(\varphi \sqcup \varphi_i, \varphi \sqcup \psi_i)} \\ \varphi \blacktriangleright \widehat{\Gamma}_1 \quad \widehat{\Gamma}_1[x_1 \mapsto (\widehat{\tau}_1, \varphi_1, \psi_1), \dots, x_n \mapsto (\widehat{\tau}_n, \varphi_n, \psi_n)] \vdash t \rhd t' : \widehat{\tau}_0^{(\mathrm{S},\psi)}\end{array}}{\widehat{\Gamma}_1 \diamond^{\sqcap} \widehat{\Gamma}_2 \vdash \mathbf{letrec}\, \overline{x_n\{\widehat{\tau}_n\} = t_n} \,\mathbf{in}\, t \rhd \mathbf{letrec}\, \overline{x_n\{\widehat{\tau}_n\} = t_n'} \,\mathbf{in}\, t' : \widehat{\tau}_0^{(\varphi,\psi)}} \text{[r-letrec]}$$

FIGURE 7.6: *Transformation rules for let bindings*

with the abstractions, there are three different rules depending on the strictness of the variables. Rule [r-let1] turns a non-strict let into a strict let, rule [r-let2] leaves the binding as normal, and rule [r-let!] is used for bang-patterns.

The original system can give an indication of how a rule for let bindings should look, as a non-recursive let binding **let** $x = t_1$ **in** t_2 can be rewritten to $(\lambda x.t_2)t_1$. Running the inference rules on this expression results in the following derivation tree for rule [r-let1], using [r-app] and [r-abs1]:

$$\frac{\dfrac{\dfrac{\varphi \blacktriangleright \widehat{\Gamma}_1 \quad \widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \mathrm{S}, \psi_1)] \vdash t_2 \rhd t_2' : \widehat{\tau}_0^{(\mathrm{S},\psi)}}{\widehat{\Gamma}_1 \vdash \lambda x\{\widehat{\tau}\}.t_2 \rhd \lambda! x\{\widehat{\tau}\}.t_2' : (\widehat{\tau} \xrightarrow{(\psi_1,\mathrm{S},\psi)} \widehat{\tau}_0)^{(\varphi,\varphi)}} \text{[r-abs1]} \qquad \widehat{\Gamma}_2 \vdash t_1 \rhd t_1' : \widehat{\tau}^{(\varphi,\varphi)}}{\widehat{\Gamma}_1 \diamond^{\sqcap} \widehat{\Gamma}_2 \vdash (\lambda x\{\widehat{\tau}\}.t_2)t_1 \rhd (\lambda! x\{\widehat{\tau}\}.t_2')t_1' : \widehat{\tau}_0^{(\varphi,\psi)}} \text{[r-app]}}{\widehat{\Gamma}_1 \diamond^{\sqcap} \widehat{\Gamma}_2 \vdash \mathbf{let}\, x\{\widehat{\tau}\} = t_1 \,\mathbf{in}\, t_2 \rhd \mathbf{let!}\, x\{\widehat{\tau}\} = t_1' \,\mathbf{in}\, t_2' : \widehat{\tau}_0^{(\varphi,\psi)}} \text{[r-let1]}$$

The derivations for [r-let2] (using [r-abs2]) and [r-let!] (using [r-sabs]) are analogous.

A difference between lambdas and lets is that the latter can be recursive, in which case the derivation above is not sound as the variables can occur in t_1. Rule [r-letrec] combats this by extending $\widehat{\Gamma}_2$ with the variables defined in recursive let bindings, such that they can be used in t_1. Unfortunately, no strictness transformation exists for these bindings as Helium is not designed to handle this. However, information obtained from the bindings and the body can be used for other, non-recursive let bindings. If a recursive binding is guaranteed to be used in the body, then any variable guaranteed to be used in the binding itself can be made strict.

CASE

The transformation rules for pattern matches are described in figure 7.7.

If-statements were present in the original system, but pattern matching on arbitrary datatypes was not possible. Not only is the [r-if] rule insufficient in the actual system, it is also very conservative in its transformation. Both branches are inferred in an L-context for relevance, which makes it impossible to infer strictness in the case a variable is guaranteed to be used in both branches. The problem can be fixed by extending the definition of the context split operation. In all other instances, context splitting used the meet operator, as being relevant and/or applicative in either part of the expression is enough to be relevant and/or applicative. This is not correct for case-expression, as a variable has to be relevant and applicative in all branches. Hence, we should use the join operator instead of the meet.

$$\frac{\widehat{\Gamma}_0 \vdash x : \widehat{\tau}^{(\mathrm{S},\psi)} \quad \forall i.1 \leq i \leq n : \widehat{\Gamma}_i \vdash a_i \rhd a_i' : \widehat{\tau}^{(\varphi,\psi)}}{\widehat{\Gamma}_0 \diamond^{\sqcap} (\widehat{\Gamma}_1 \diamond^{\sqcup} \dots \diamond^{\sqcup} \widehat{\Gamma}_n) \vdash \textbf{case } x \textbf{ of } \overline{a_n} \rhd \textbf{case } x \textbf{ of } \overline{a_n'} : \widehat{\tau}^{(\varphi,\psi)}} \text{ [r-case]}$$

$$\frac{\widehat{\Gamma}[x_1 \mapsto (\widehat{\tau}_1, \mathrm{L}, \mathrm{L}), \dots, x_n \mapsto (\widehat{\tau}_n, \mathrm{L}, \mathrm{L})] \vdash t \rhd t' : \widehat{\tau}^{(\varphi,\psi)}}{\widehat{\Gamma} \vdash c\overline{x_n\{\widehat{\tau}_n\}} \rightarrow t \rhd c\overline{x_n\{\widehat{\tau}_n\}} \rightarrow t' : \widehat{\tau}^{(\varphi,\psi)}} \text{ [a-con]}$$

$$\frac{\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi,\psi)}}{\widehat{\Gamma} \vdash l \rightarrow t \rhd l \rightarrow t' : \widehat{\tau}^{(\varphi,\psi)}} \text{ [a-lit]}$$

$$\frac{\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi,\psi)}}{\widehat{\Gamma} \vdash _ \rightarrow t \rhd _ \rightarrow t' : \widehat{\tau}^{(\varphi,\psi)}} \text{ [a-def]}$$

FIGURE 7.7: *Transformation rules for pattern matching*

The adjusted diamond operator receives an input $\square \in \{\sqcap, \sqcup\}$ to specify which operator needs to be used for this specific context split. The new definition is as follows:

$$[] \diamond^{\square} [] = []$$
$$\widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \varphi_1, \psi_1)] \diamond^{\square} \widehat{\Gamma}_2[x \mapsto (\widehat{\tau}, \varphi_2, \psi_2)] = (\widehat{\Gamma}_1 \diamond^{\square} \widehat{\Gamma}_2)[x \mapsto (\widehat{\tau}, \varphi_1 \square \varphi_2, \psi_1 \square \psi_2)]$$

Using the new diamond operator, we can define the inference rule [r-case], which can pattern match on any datatype or literal and can have an arbitrary number of cases. We split the environment into $n + 1$ subenvironments, one for each alternative and one for the variable. $\widehat{\Gamma}_0$ is used for the variable, which is defined in a strict let or lambda preceding the pattern match. $\widehat{\Gamma}_1$ to $\widehat{\Gamma}_n$ are the environments for the alternatives, and are split using the join operator. The resulting environment is split with $\widehat{\Gamma}_0$ using the meet operator. All branches can copy the input contexts without having to weaken them. Only if a variable is guaranteed to be used in all cases, it will be set to S in the environment before the split.

Rules [a-con], [a-lit] and [a-def] are created to handle the different kind of patterns to match on. The case for constructors, [a-con], needs to add all variables used for the constructor fields to the environment. Since we currently have no way of measuring their relevance or applicativeness, we can only set these annotations to L. Rules [a-lit] and [a-def] do not add anything to the environment, and since no information can be gained from the literal or wildcard itself, we are only concerned about the information from the term.

GENERALIZATION AND INSTANTIATION

The transformation rules for generalization and instantiation are described in figure 7.8.

$$\frac{\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi,\psi)}}{\widehat{\Gamma} \vdash \textbf{forall } \alpha.t \rhd \textbf{forall } \alpha.t' : \forall \alpha.\widehat{\tau}^{(\varphi,\psi)}} \text{ [r-forall]}$$

$$\frac{\widehat{\Gamma} \vdash t \rhd t' : \forall \alpha.\widehat{\tau}^{(\varphi,\psi)}}{\widehat{\Gamma} \vdash t\{\widehat{\tau}_1\} \rhd t'\{\widehat{\tau}_1\} : [\alpha \mapsto \{\widehat{\tau}_1\}]\widehat{\tau}^{(\varphi,\psi)}} \text{ [r-aptype]}$$

FIGURE 7.8: *Transformation rules for generalization and instantiation*

Generalization and instantiation are also part of the Helium syntax via the *Forall* and *ApType* expressions. Rule [r-forall] describes generalization, while rule [r-aptype] describes instantiation. The former quantifies over a (free) type variable, while the latter replaces the occurrence of a type variable with the given type.

SUBEFFECTING, WEAKENING AND CONTAINMENT

Subeffecting, weakening and containment are described in figure 7.9.

$$\frac{\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\mathrm{L},\mathrm{L})}}{\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\mathrm{S},\psi)}} \text{ [r-sub]}$$

$$\frac{\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi,\psi)}}{\widehat{\Gamma}[x \mapsto (\widehat{\tau}_0, \mathrm{L}, \mathrm{L})] \vdash t \rhd t' : \widehat{\tau}^{(\varphi,\psi)}} \text{ [r-weak]}$$

$$\frac{}{\varphi \blacktriangleright []} \text{ [c-nil]}$$

$$\frac{\mathrm{S} \blacktriangleright \widehat{\Gamma}_1}{\mathrm{S} \blacktriangleright \widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \varphi_0, \psi_0)]} \text{ [c-cons-s]}$$

$$\frac{\mathrm{L} \blacktriangleright \widehat{\Gamma}_1}{\mathrm{L} \blacktriangleright \widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \mathrm{L}, \mathrm{L})]} \text{ [c-cons-l]}$$

FIGURE 7.9: *Subeffecting, weakening and containment*

Rules [r-sub], [r-weak], [c-nil], [c-cons-s] and [c-cons-l] remain the same as they were in the initial system, as subeffecting, weakening and containment are not related to the syntax.

Aside from rules mentioned in previous sections, rules [r-false], [r-true], [r-zero], [r-pred], [r-succ], [r-iszero] and [r-error] have become obsolete.

OVERVIEW

All transformation rules defined in the previous sections are combined into figure 7.10 for convenience.

$$\frac{}{[] \vdash c \rhd c : \hat{\tau}^{(\varphi,\psi)}}\ [\text{r-con}] \qquad \frac{}{[] \vdash l \rhd l : \hat{\tau}^{(\varphi,\mathrm{L})}}\ [\text{r-lit}] \qquad \frac{}{[x \mapsto (\hat{\tau},\varphi,\psi)] \vdash x \rhd x : \hat{\tau}^{(\varphi,\psi)}}\ [\text{r-var}]$$

$$\frac{\psi \blacktriangleright \hat{\Gamma} \quad \hat{\Gamma}[x \mapsto (\hat{\tau},\mathrm{S},\psi_1)] \vdash t_1 \rhd t_1' : \hat{\tau}_0^{(\mathrm{S},\psi_2)}}{\hat{\Gamma} \vdash \lambda x\{\hat{\tau}\}.t_1 \rhd \lambda! x\{\hat{\tau}\}.t_1' : (\hat{\tau} \xrightarrow{(\psi_1,\mathrm{S},\psi_2)} \hat{\tau}_0)^{(\varphi,\psi)}}\ [\text{r-abs1}]$$

$$\frac{\psi \blacktriangleright \hat{\Gamma} \quad \hat{\Gamma}[x \mapsto (\hat{\tau},\mathrm{L},\psi_1)] \vdash t_1 \rhd t_1' : \hat{\tau}_0^{(\mathrm{S},\psi_2)}}{\hat{\Gamma} \vdash \lambda x\{\hat{\tau}\}.t_1 \rhd \lambda x\{\hat{\tau}\}.t_1' : (\hat{\tau} \xrightarrow{(\psi_1,\mathrm{L},\psi_2)} \hat{\tau}_0)^{(\varphi,\psi)}}\ [\text{r-abs2}]$$

$$\frac{\psi \blacktriangleright \hat{\Gamma} \quad \hat{\Gamma}[x \mapsto (\hat{\tau},\mathrm{S},\psi_1)] \vdash t_1 \rhd t_1' : \hat{\tau}_0^{(\mathrm{S},\psi_2)}}{\hat{\Gamma} \vdash \lambda! x\{\hat{\tau}\}.t_1 \rhd \lambda! x\{\hat{\tau}\}.t_1' : (\hat{\tau} \xrightarrow{(\psi_1,\varphi_1,\psi_2)} \hat{\tau}_0)^{(\varphi,\psi)}}\ [\text{r-sabs}]$$

$$\frac{\hat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\hat{\tau}_2 \xrightarrow{(\psi_2,\varphi_0,\psi)} \hat{\tau})^{(\varphi,\varphi)} \quad \hat{\Gamma}_2 \vdash t_2 \rhd t_2' : \hat{\tau}_2^{(\varphi\sqcup\varphi_0,\varphi\sqcup\psi_2)}}{\hat{\Gamma}_1 \diamond^{\sqcap} \hat{\Gamma}_2 \vdash t_1 t_2 \rhd t_1' t_2' : \hat{\tau}^{(\varphi,\psi)}}\ [\text{r-app}]$$

$$\frac{\varphi \blacktriangleright \hat{\Gamma}_1 \quad \hat{\Gamma}_1[x \mapsto (\hat{\tau},\mathrm{S},\psi_1)] \vdash t_2 \rhd t_2' : \hat{\tau}_0^{(\mathrm{S},\psi)} \quad \hat{\Gamma}_2 \vdash t_1 \rhd t_1' : \hat{\tau}^{(\varphi,\varphi)}}{\hat{\Gamma}_1 \diamond^{\sqcap} \hat{\Gamma}_2 \vdash \textbf{let}\ x\{\hat{\tau}\} = t_1\ \textbf{in}\ t_2 \rhd \textbf{let!}\ x\{\hat{\tau}\} = t_1'\ \textbf{in}\ t_2' : \hat{\tau}_0^{(\varphi,\psi)}}\ [\text{r-let1}]$$

$$\frac{\varphi \blacktriangleright \hat{\Gamma}_1 \quad \hat{\Gamma}_1[x \mapsto (\hat{\tau},\mathrm{L},\psi_1)] \vdash t_2 \rhd t_2' : \hat{\tau}_0^{(\mathrm{S},\psi)} \quad \hat{\Gamma}_2 \vdash t_1 \rhd t_1' : \hat{\tau}^{(\varphi\sqcup\varphi_1,\mathrm{L})}}{\hat{\Gamma}_1 \diamond^{\sqcap} \hat{\Gamma}_2 \vdash \textbf{let}\ x\{\hat{\tau}\} = t_1\ \textbf{in}\ t_2 \rhd \textbf{let}\ x\{\hat{\tau}\} = t_1'\ \textbf{in}\ t_2' : \hat{\tau}_0^{(\varphi,\psi)}}\ [\text{r-let2}]$$

$$\frac{\varphi \blacktriangleright \hat{\Gamma}_1 \quad \hat{\Gamma}_1[x \mapsto (\hat{\tau},\mathrm{S},\psi_1)] \vdash t_2 \rhd t_2' : \hat{\tau}_0^{(\mathrm{S},\psi)} \quad \hat{\Gamma}_2 \vdash t_1 \rhd t_1' : \hat{\tau}^{(\varphi\sqcup\varphi_1,\varphi\sqcup\psi_1)}}{\hat{\Gamma}_1 \diamond^{\sqcap} \hat{\Gamma}_2 \vdash \textbf{let!}\ x\{\hat{\tau}\} = t_1\ \textbf{in}\ t_2 \rhd \textbf{let!}\ x\{\hat{\tau}\} = t_1'\ \textbf{in}\ t_2' : \hat{\tau}_0^{(\varphi,\psi)}}\ [\text{r-let!}]$$

$$\frac{\begin{array}{c}\forall i.1 \leq i \leq n : \hat{\Gamma}_2[x_1 \mapsto (\hat{\tau}_1,\varphi_1,\psi_1),\ldots,x_n \mapsto (\hat{\tau}_n,\varphi_n,\psi_n)] \vdash t_i \rhd t_i' : \hat{\tau}_i^{(\varphi\sqcup\varphi_i,\varphi\sqcup\psi_i)} \\ \varphi \blacktriangleright \hat{\Gamma}_1 \quad \hat{\Gamma}_1[x_1 \mapsto (\hat{\tau}_1,\varphi_1,\psi_1),\ldots,x_n \mapsto (\hat{\tau}_n,\varphi_n,\psi_n)] \vdash t \rhd t' : \hat{\tau}_0^{(\mathrm{S},\psi)}\end{array}}{\hat{\Gamma}_1 \diamond^{\sqcap} \hat{\Gamma}_2 \vdash \textbf{letrec}\ \overline{x_n\{\hat{\tau}_n\} = t_n}\ \textbf{in}\ t \rhd \textbf{letrec}\ \overline{x_n\{\hat{\tau}_n\} = t_n'}\ \textbf{in}\ t' : \hat{\tau}_0^{(\varphi,\psi)}}\ [\text{r-letrec}]$$

$$\frac{\hat{\Gamma}_0 \vdash x : \hat{\tau}^{(\mathrm{S},\psi)} \quad \forall i.1 \leq i \leq n : \hat{\Gamma}_i \vdash a_i \rhd a_i' : \hat{\tau}^{(\varphi,\psi)}}{\hat{\Gamma}_0 \diamond^{\sqcap} (\hat{\Gamma}_1 \diamond^{\sqcup} \ldots \diamond^{\sqcup} \hat{\Gamma}_n) \vdash \textbf{case}\ x\ \textbf{of}\ \overline{a_n} \rhd \textbf{case}\ x\ \textbf{of}\ \overline{a_n'} : \hat{\tau}^{(\varphi,\psi)}}\ [\text{r-case}]$$

$$\frac{\hat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{(\varphi,\psi)}}{\hat{\Gamma} \vdash \textbf{forall}\ \alpha.t \rhd \textbf{forall}\ \alpha.t' : \forall\alpha.\hat{\tau}^{(\varphi,\psi)}}\ [\text{r-forall}] \qquad \frac{\hat{\Gamma} \vdash t \rhd t' : \forall\alpha.\hat{\tau}^{(\varphi,\psi)}}{\hat{\Gamma} \vdash t\{\hat{\tau}_1\} \rhd t'\{\hat{\tau}_1\} : [\alpha \mapsto \hat{\tau}_1]\hat{\tau}^{(\varphi,\psi)}}\ [\text{r-aptype}]$$

$$\frac{\hat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{(\mathrm{L},\mathrm{L})}}{\hat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{(\mathrm{S},\psi)}}\ [\text{r-sub}] \qquad \frac{\hat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{(\varphi,\psi)}}{\hat{\Gamma}[x \mapsto (\hat{\tau}_0,\mathrm{L},\mathrm{L})] \vdash t \rhd t' : \hat{\tau}^{(\varphi,\psi)}}\ [\text{r-weak}]$$

$$\frac{\hat{\Gamma}[x_1 \mapsto (\hat{\tau}_1,\mathrm{L},\mathrm{L}),\ldots,x_n \mapsto (\hat{\tau}_n,\mathrm{L},\mathrm{L})] \vdash t \rhd t' : \hat{\tau}^{(\varphi,\psi)}}{\hat{\Gamma} \vdash c\overline{x_n\{\hat{\tau}_n\}} \to t \rhd c\overline{x_n\{\hat{\tau}_n\}} \to t' : \hat{\tau}^{(\varphi,\psi)}}\ [\text{a-con}]$$

$$\frac{\hat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{(\varphi,\psi)}}{\hat{\Gamma} \vdash l \to t \rhd l \to t' : \hat{\tau}^{(\varphi,\psi)}}\ [\text{a-lit}] \qquad \frac{\hat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{(\varphi,\psi)}}{\hat{\Gamma} \vdash _ \to t \rhd _ \to t' : \hat{\tau}^{(\varphi,\psi)}}\ [\text{a-def}]$$

$$\frac{}{\varphi \blacktriangleright []}\ [\text{c-nil}] \qquad \frac{\mathrm{S} \blacktriangleright \hat{\Gamma}_1}{\mathrm{S} \blacktriangleright \hat{\Gamma}_1[x \mapsto (\hat{\tau},\varphi_0,\psi_0)]}\ [\text{c-cons-s}] \qquad \frac{\mathrm{L} \blacktriangleright \hat{\Gamma}_1}{\mathrm{L} \blacktriangleright \hat{\Gamma}_1[x \mapsto (\hat{\tau},\mathrm{L},\mathrm{L})]}\ [\text{c-cons-l}]$$

FIGURE 7.10: *Adjusted relevance and applicativeness typing, call-by-value transformations and containment*

7.2 ALGORITHM

7.2.1 PREPARATIONS

Before the strictness analysis, binding group analysis is performed to order and group functions. If a function calls another function, the other function should preferably be analyzed beforehand. However, some functions are recursive or mutually recursive, which means they have to be analyzed together. A complication for binding group analysis in Core is the structure of the declarations (see sections 4.3.1). All declarations which are not DeclValue are put into a single list, while this analysis only concerns functions and value declarations.

The analysis returns a list of binding groups, topologically sorted such that functions are defined before use. A binding group is either:

- **BindingNonFunction**, which collects all declarations which are not a DeclValue, and puts them at the head of the list;
- **BindingNonRecursive**, a single declaration which is a DeclValue;
- **BindingRecursive**, a list of declarations (DeclValues) which are recursive. If the list contains one item, it is a function recursive in itself, if the list contains multiple items they are mutually recursive (and possibly self-recursive).

The type system of Core has to be extended to include annotations, as the TStrict datatype by itself is not enough. The extension are described in figure 7.11. We extend the type system to include annotations (TAnn), which are placed on other types by using TAp. An annotation SAnn is either a constant L or S, a variable, or a join or meet of two annotations. The SAnn datatype is only meant to be used within the strictness analysis and is not part of the underlying type system of Core. Furthermore, polyvariance requires us to quantify over annotations. To differentiate between quantification over type variables and strictness variables, we add KAnn to the kind system. Technically, an annotation is not a kind, but this is the easiest solution to include strictness quantification.

```
data Type = TCon    TypeConstant
          | TVar    TypeVar
          | TAp     Type Type
          | TForall Quantor Kind Type
          | TStrict Type
          | TAnn    SAnn
data SAnn = S
          | L
          | AnnVar Id
          | Join    SAnn SAnn
          | Meet    SAnn SAnn
data Kind = KFun Kind Kind
          | KStar
          | KAnn
```

FIGURE 7.11: *Extended Type datatype in Core*

The expression syntax from figure 4.6 remains unchanged. As applications contain both function and constructor applications, and an application expects the function to be annotated, we have to annotate the constructors as well. Constructors fields are assumed to be annotated with L, unless the field is

marked as strict, in which case it is annotated as S. The saturate pass makes sure all constructors are fully applied, thus we do not have to worry about partial applications, meaning any field which is relevant is also applicative, and fields which are not relevant are not applicative.

Abstracts are functions imported from other modules. These functions export their strictness signature as well as their regular type signature, so the former can be stored in the type environment instead. However, this is not the case for functions defined in Iridium. As a solution, the regular type signatures from Iridium are annotated using the TStrict constructor for strict arguments. Unlike the data constructors, they can be partially applied, so we manually have to set the applicativeness annotations to be the join over all applicativeness annotations of later arguments.

The types stored with the variables in lambdas and lets are already annotated in the transformation rules. This is not the case in the actual system, which means we have to annotate them during the analysis. This is also the case for type instantiations. Algorithm A, described in figure 7.12), places three fresh annotations per function arrow. Type synonyms are replaced by their definition to avoid all occurrences of a type synonym requiring the same annotations.

$$
\begin{aligned}
A(d\,\overline{\tau_n}) &= d\,\overline{A(\tau_n)} \\
A(\tau_1 \to \tau_2) &= \textbf{let } (\psi_1, \varphi, \psi_2) \textbf{ be fresh in } A(\tau_1) \xrightarrow{(\psi_1,\varphi,\psi_2)} A(\tau_2) \\
A(\alpha) &= \alpha \\
A(\forall\alpha.\tau) &= \forall\alpha.A(\tau) \\
A(!\tau) &= !(A(\tau))
\end{aligned}
$$

FIGURE 7.12: *Annotation algorithm A*

7.2.2 ANALYSIS

The analysis function *W* is an implementation of the transformation rules discussed in the previous section. The input of the function is an environment with an annotated type, relevance annotation and applicativeness annotation per variable, a relevance and applicativeness context and the current expression to analyze. The reason the constraints are split between an environment and a constraint set is for containment. The annotations which belong to a variable in the expression, and thus have to be contained in abstractions, are handled via the annotation environment. Constraints such as those for subeffecting which are not affected by containment are handled via the constraint set. The output is the annotated type of the expression, an annotation environment which maps annotation variables to annotations, a set of constraints, and a map of variables which could be transformed, which maps from variables to annotation variables. The only transformation the analysis does is converting unannotated types to annotated types, and unpacking type synonyms. The actual transforming of lets and lambdas to strict ones is done after the analysis, which is done on the original expressions such that all other annotations are removed automatically. It is important that no trace of the strictness annotations is left behind at the end of the pass, as other passes do not expect annotations to occur. It is impractical to adapt every other part of the program to be able to handle annotations. The only information which has to be kept, aside from the transformed expression, is a type signature with annotations which can be used when this function is imported.

The starting environment contains all constructors and abstract functions. The remaining functions are analyzed and solved binding group by binding group. In a non-recursive declaration, the type does not have to be added to the environment in the analysis of the function. It does have to be added afterwards as the next binding group might use that function. For recursive declarations, the declarations are added

to the type environment beforehand. As they have to be added in annotated form, we annotate them with algorithm A. This will lead to constraints of the form $a \sqsubseteq a$, which we could then solve with a = S.

Constructors, literals and variables

The algorithm for constructors, literals and variables is described in figure 7.13.

$$
\begin{aligned}
W(\hat\Gamma, \varphi, \psi, c) &= \textbf{let } (\hat\tau, \varphi', \psi') = \hat\Gamma(c) \\
&\quad\ \textbf{in } (\hat\tau, \emptyset, f(\hat\Gamma, \text{L}), []) \\
W(\hat\Gamma, \varphi, \psi, l) &= (t(l), \emptyset, f(\hat\Gamma, \text{L}), []) \\
W(\hat\Gamma, \varphi, \psi, x) &= \textbf{let } (\hat\tau, \varphi', \psi') = \hat\Gamma(x) \\
&\quad\ \textbf{in } (\hat\tau, \emptyset, f(\hat\Gamma \backslash x, \text{L})[\varphi' \mapsto \varphi, \psi' \mapsto \psi], []) \\
\\
f([], \chi) &= [] \\
f(\hat\Gamma[x \mapsto (\hat\tau, \varphi, \psi)], \chi) &= f(\hat\Gamma)[\varphi \mapsto \chi, \psi \mapsto \chi]
\end{aligned}
$$

FIGURE 7.13: *Algorithm W for constructors, literals and variables*

For constructors and literals, all annotations which are in the environment are set to L. This is done using the auxiliary function f. For variables, all annotations except those associated with the variable are set to L, with the annotations of the variable set to the contexts. The type of constructors and variables can be taken from the environment. Literals are not in the environment because there are, in theory, infinitely many of them, so we assume there exists a function t which can give the type of a literal.

Application

The algorithm for applications is described in figure 7.14.

$$
\begin{aligned}
W(\hat\Gamma, \varphi, \psi, t_1 t_2) &= \textbf{let } (\hat\tau_1 \xrightarrow{(\psi_1, \varphi', \psi_2)} \hat\tau_2, cs1, ae1, r1) = W(\hat\Gamma, \varphi, \varphi, t_1) \\
&\quad\ (\hat\tau_1', cs2, ae2, r2) = W(\hat\Gamma, \varphi \sqcup \varphi', \varphi \sqcup \psi_1, t_2) \\
&\quad\ cs = \{\psi \sqsubseteq \psi_2\} \cup cs1 \cup cs2 \cup U(\hat\tau_1', \hat\tau_1) \\
&\quad\ \textbf{in } (\hat\tau_2, cs, ae1 \diamond^{\sqcap} ae2, r1 \cup r2) \\
\\
U(d\, \overline{\hat\tau_n}, d\, \overline{\hat\tau_n'}) &= \bigcup \overline{U(\hat\tau_n, \hat\tau_n')} \\
U(\hat\tau_1 \xrightarrow{(\psi_1, \varphi, \psi_2)} \hat\tau_2, \hat\tau_1' \xrightarrow{(\psi_1', \varphi', \psi_2')} \hat\tau_2') &= \{\psi_1 \sqsubseteq \psi_1', \varphi \sqsubseteq \varphi', \psi_2 \sqsubseteq \psi_2'\} \cup U(\hat\tau_1, \hat\tau_1') \cup U(\hat\tau_2, \hat\tau_2') \\
U(\alpha, \alpha) &= \emptyset \\
U(\forall\alpha.\hat\tau, \forall\alpha.\hat\tau') &= U(\hat\tau, \hat\tau') \\
U(!\hat\tau, !\hat\tau') &= U(\hat\tau, \hat\tau')
\end{aligned}
$$

FIGURE 7.14: *Algorithm W for applications*

In application, we get the type of the function from analyzing the function, which includes the annotations for the argument. Information from the function and the argument is merged using the meet. The constraints are a union of four sets: the constraint set from the function, the constraint set of the argument, a single constraint which constrains the right applicativeness annotation to the current context, and the instantiation constraints. The instantiation constraints are taken from function U, which compares two types and returns the pairwise constraints.

Abstraction

The algorithm for abstractions is described in figure 7.15.

$$
\begin{aligned}
W(\hat{\Gamma}, \varphi, \psi, \lambda x\{\tau\}.t) = {} & \textbf{let } \psi_1, \varphi, \psi_2 \textbf{ be fresh} \\
& (\hat{\tau}_2, cs, ae, r) = W(\hat{\Gamma}[x \mapsto (A(\tau), \varphi, \psi_1)], \mathrm{S}, \psi_2, t) \\
& \textbf{in } (\hat{\tau} \xrightarrow{(\psi_1, \varphi, \psi_2)} \hat{\tau}_2, cs, f(\hat{\Gamma}, \psi) \diamond^{\sqcup} ae, r[x \mapsto \varphi]) \\
W(\hat{\Gamma}, \varphi, \psi, \lambda !x\{\tau\}.t) = {} & \textbf{let } \psi_1, \psi_2 \textbf{ be fresh} \\
& (\hat{\tau}_2, cs, ae, r) = W(\hat{\Gamma}[x \mapsto (A(\tau), \mathrm{S}, \psi_1)], \mathrm{S}, \psi_2, t) \\
& \textbf{in } (\hat{\tau} \xrightarrow{(\psi_1, \mathrm{S}, \psi_2)} \hat{\tau}_2, cs, f(\hat{\Gamma}, \psi) \diamond^{\sqcup} ae, r)
\end{aligned}
$$

FIGURE 7.15: *Algorithm W for abstractions*

In the lambda rules, we analyze the body under a strict relevance and fresh applicativeness context. The type associated with the lambda starts out unannotated, so we add these annotations manually using A. The containment is implemented by a join of the applicativeness context over all variables which are in the input environment, which excludes the current variable. In the non-strict lambda, the variable is added to the map of lambdas which could be made strict. This is not necessary in the strict case as there is nothing to transform.

Let bindings

The algorithm for let bindings is described in figure 7.16.

Let bindings also have a containment relation on the information coming from the body, but this is performed on the relevance context instead of the applicativeness context. Only non-strict let bindings are added to the map of bindings able to transform. Recursive let bindings cannot be added to this set because they cannot be transformed to strict bindings even if they were guaranteed to be used. All these variables are annotated beforehand and added to the environment, unlike the type of variables in strict and non-strict bindings because their annotated type can be determined from W itself. Every binding is then analyzed on its own, and the information is merged using the meet.

Case

The algorithm for pattern matches is described in figure 7.17.

The information from case expression is merged with the join, unlike other merges where this is done with the meet. The variable to be matched upon also returns an environment, which sets all other variables to L and itself to the context. Algorithm W_a operates on the different alternatives, though the only interesting case is pattern matching on a constructor, in which case it adds variables to the environment. Otherwise, it just calls W on the body again.

$$
\begin{aligned}
W(\hat{\Gamma}, \varphi, \psi, \textbf{let } x\{\tau\} = t_1 \textbf{ in } t_2) = {} & \textbf{let } \varphi', \psi' \textbf{ be fresh} \\
& (\hat{\tau}', cs1, ae1, r1) = W(\hat{\Gamma}, \varphi \cup \varphi', \varphi \cup \psi', t_1) \\
& (\hat{\tau}_2, cs2, ae2, r2) = W(\hat{\Gamma}[x \mapsto (\hat{\tau}', \varphi', \psi')], \mathrm{S}, \psi, t_2) \\
& ae2' = f(\hat{\Gamma}, \varphi) \diamond^{\sqcup} ae2 \\
& \textbf{in } (\hat{\tau}_2, cs1 \cup cs2, ae1 \diamond^{\sqcap} ae2', (r1 \cup r2)[x \mapsto \varphi']) \\
W(\hat{\Gamma}, \varphi, \psi, \textbf{let! } x\{\tau\} = t_1 \textbf{ in } t_2) = {} & \textbf{let } \psi' \textbf{ be fresh} \\
& (\hat{\tau}', cs1, ae1, r1) = W(\hat{\Gamma}, \varphi, \varphi \cup \psi', t_1) \\
& (\hat{\tau}_2, cs2, ae2, r2) = W(\hat{\Gamma}[x \mapsto (\hat{\tau}', \mathrm{S}, \psi')], \mathrm{S}, \psi, t_2) \\
& ae2' = f(\hat{\Gamma}, \varphi) \diamond^{\sqcup} ae2 \\
& \textbf{in } (\hat{\tau}_2, cs1 \cup cs2, ae1 \diamond^{\sqcap} ae2', r1 \cup r2) \\
W(\hat{\Gamma}, \varphi, \psi, \textbf{letrec } \overline{x_n\{\tau_n\} = t_n} \textbf{ in } t) = {} & \textbf{let } \varphi_1, \psi_1, \ldots, \varphi_n, \psi_n \textbf{ be fresh} \\
& \hat{\Gamma}' = \hat{\Gamma}[x_1 \mapsto (A(\tau_1), \varphi_1, \psi_1), \ldots, x_n \mapsto (A(\tau_n), \varphi_n, \psi_n)] \\
& \overline{(_, cs_n, ae_n, r_n)} = \overline{W(\hat{\Gamma}', \varphi \cup \varphi_n, \varphi \cup \psi_n, x_n\{\tau_n\} = t_n)} \\
& (\hat{\tau}', cs, ae, r) = W(\hat{\Gamma}', \mathrm{S}, \psi, t) \\
& ae' = f(\hat{\Gamma}, \varphi) \diamond^{\sqcup} ae \\
& \textbf{in } (\hat{\tau}', cs \cup (\bigcup cs_n), ae' \diamond^{\sqcap} (\diamond^{\sqcap}_{i=1\ldots n} ae_i), r \cup (\bigcup r_n))
\end{aligned}
$$

FIGURE 7.16: *Algorithm W for let bindings*

GENERALIZATION AND INSTANTIATION

The algorithm for generalization and instantiation is described in figure 7.18.

7.2.3 CONSTRAINT SOLVING AND TRANSFORMATION

After analyzing each function or group of function in a binding group, we receive an annotation environment and a set of constraints. We solve the variables per binding group. The constraints are of the form $\varphi_1 \sqsubseteq \varphi_2$, and assuming φ_2 is a variable, the entry for φ_2 in the environment is updated with the join of its original value and φ_1. Solving the annotations in the environment requires them to be handled in a specific order. We build a graph with every variable as node, and we draw an edge between φ_1 and φ_2 if the solution of φ_1 depends on φ_2. We then find the strongly connected components of the graph and order them in reverse topological sort such that φ_2 is handled before φ_1, or they are in the same strongly connected component.

We then process all annotations in order, where annotations which have their own strongly connected component only need to look for the annotations which are already solved. For strongly connected components with two or more nodes, we have to be a bit more careful, as this means φ_1 depends on φ_2 and φ_2 depends on φ_1. We take one of the annotations and replace its occurence in all other annotations in the group. This means that an annotation could now depend on itself, which can be solved by replacing the self-depedency by S. After all annotations are processed, there should no longer be any recursion and the annotations can be solved.

In the monovariant analysis, any variable which is not solved to L or S is set to L. This has to be done because of the applicativeness annotations, as we cannot know if the function is fully applied ev-

$$\begin{aligned}
W(\hat{\Gamma}, \varphi, \psi, \textbf{case } x \textbf{ of } \overline{a_n}) = \textbf{let } & (\hat{\tau}, \varphi', \psi') = \hat{\Gamma}(x) \\
& ae1 = f(\hat{\Gamma} \backslash x)[\varphi' \mapsto \varphi, \psi' \mapsto \psi] \\
& \overline{(\hat{\tau}'_n, cs_n, ae_n, r_n)} = \overline{W_a(\hat{\Gamma}, \varphi, \psi, a_n)} \\
\textbf{in } & (\hat{\tau}'_1, \bigcup cs_n, ae1 \diamond^{\sqcap} (\diamond^{\sqcup}_{i=1\dots n} ae_i), \bigcup r_n)
\end{aligned}$$

$$\begin{aligned}
W_a(\hat{\Gamma}, \varphi, \psi, c\overline{x_n\{\tau_n\}} \rightarrow t) &= W(\hat{\Gamma}[x_1 \mapsto (A(\tau_1), \mathrm{L}), \dots, x_n \mapsto (A(\tau_n), \mathrm{L})], \varphi, \psi, t) \\
W_a(\hat{\Gamma}, \varphi, \psi, l \rightarrow t) &= W(\hat{\Gamma}, \varphi, \psi, t) \\
W_a(\hat{\Gamma}, \varphi, \psi, _ \rightarrow t) &= W(\hat{\Gamma}, \varphi, \psi, t)
\end{aligned}$$

FIGURE 7.17: *Algorithm W for pattern matches*

$$\begin{aligned}
W(\hat{\Gamma}, \varphi, \psi, \textbf{forall } \alpha.t) &= \textbf{let } (\hat{\tau}, cs, ae, r) = W(\hat{\Gamma}, \varphi, \psi, t) \\
&\quad \textbf{in } (\forall \alpha.\hat{\tau}, cs, ae, r) \\
W(\hat{\Gamma}, \varphi, \psi, t\{\tau_1\}) &= \textbf{let } (\forall \alpha.\hat{\tau}, cs, ae, r) = W(\hat{\Gamma}, \varphi, \psi, t) \\
&\quad \textbf{in } (\hat{\tau}[\alpha \mapsto A(\tau_1)], cs, ae, r)
\end{aligned}$$

FIGURE 7.18: *Algorithm W for generalization and instantiation*

erywhere, including outside its own module if the function is exported. Setting these annotations to S would mean the function is always fully applied, which renders the applicativeness annotations useless and could again lead to unsound transformations. Unsolved relevance annotations are also defaulted to L, for similar reasons to the applicativeness annotations. The function ($) does not know if it is always given a strict function, and thus it is only safe to assume it will always receive a non-strict function.

Once all constraints are solved, we can apply the substitutions to the type signature and the expression. We receive the annotated type of an expression from the algorithm, and the solved constraints allow us to replace any variable by its solved annotation. Any remaining annotation variable has to be set to L, because we cannot know all of it usages. Transforming the expression relies on the secondary environment returned from the algorithm. This environment maps variable identifiers to annotations. We traverse the expression and check for every lambda and let if the variable occurs in the environment. If it does not, it cannot be transformed to a strict binding because it is already strict or recursive. If the variable does occur, we check the solved value of the annotation. If this annotation is S, we transform the let or lambda to its strict counterpart, otherwise we leave the binding as is.

7.3 ADAPTATION TO POLYVARIANCE

The algorithm described in the previous section described the monovariant analysis. With some adaptations, this analysis can be turned into a polyvariant analysis. The main difference between the analyses are generalization and instantiation on annotations, which is not supported in the monovariant analysis. Whenever a monovariant analysis had a variable in the type after solving, it had to be set to the most conservative annotation, which is L. In the polyvariant case, we can leave the annotation variables and quantify over them, instantiating them with fresh variables whenever they are used.

The first adaptation is allowing annotation variables to occur in the type (7.19). To be as precise as

possible, we also allow unsolved joins and meets to occur in the type, under the assumption that they are simplified as far as possible. We also need to add the ability to quantify over annotation variables.

$\hat{\tau} ::= d\,\overline{\hat{\tau}}$	TCon, TAp TCon, TAp (TAp TCon) etc.
$\mid \hat{\tau} \xrightarrow{(\psi,\varphi,\psi)} \hat{\tau}$	TAp (TAp (TCon ConFun)
$\mid \alpha$	TVar
$\mid \forall \alpha.\hat{\tau}$	TForall
$\mid \forall \varphi.\hat{\tau}$	TForall
$\mid !\hat{\tau}$	TStrict

$$\varphi ::= S \mid L \mid \beta \mid \varphi \sqcup \varphi \mid \varphi \sqcap \varphi$$
$$\psi ::= \varphi$$

FIGURE 7.19: *Extended types for polyvariance*

Since we add annotation polymorphism, we also need to be able to instantiate. Every variable which refers to a function might now have quantification over its annotations, and needs to have a unique instantiation to distinguish it from other calls. We instantiate them with fresh annotation variables, which we enforce by placing ApTypes around the variable. The function const will have the type signature

$$(\forall \alpha, \forall \beta, \forall a.\forall b.a \xrightarrow{(\alpha \sqcup \beta, \alpha, \alpha)} b \xrightarrow{(\text{L},\text{L},\beta)} a)$$

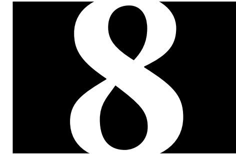
The instantiation for the type variables is already done, so we only need to add two additional ApTypes with fresh annotations.

A polyvariant analysis also extends to let bindings. If we define const locally, we would like to have the same behavior as a globally defined const. This means the type signature of the local definition needs to have the strictness quantifications. To achieve this, we have to solve part of the constraints which are gathered from within a let binding. We can only solve variables which were created within the binding, as they should not have any constraints outside their scope. This process is called simplification. Simplification works exactly the same as solving and transformation (7.2.3), except certain constrains cannot be touched.

The solving algorithm is unchanged, except annotation variables which were not solved to a constant are not defaulted to L. This allows for annotation variables, joins and meets to occur in the type. Transformation still occurs at the end of the pipeline, once all constraints have been solved. There is no intermediate transformation after simplification. Because transformation might depend on annotations which were solved during simplification, we have to push up the entire set of solved constraints such that we have all information available to perform transformations.

Polyvariance also allows us to improve transformation of lambdas. Since a function like const has a variable as relevance annotation on its first argument, it could not be converted to a strict lambda as that annotation would be defaulted to L. In polyvariance, this annotation variable remains, but since it is not S a transformation will not take place. However, const is meant to be strict in its first argument, but the applicativeness annotations prevent the annotation S from occurring as it needs to take into account that the function might be partially applied. The expressions $((\lambda x.\lambda y.x)\text{error})$ and $((\lambda !x.\lambda y.x)\text{error})$ are equivalent, as the lambda is only triggered when the function is evaluated, which does not happen in partial application. This means it is allowed to transform a lambda into a strict lambda even if the annotation is S, as long as the argument is strict when all arguments are applied. This transformation

is important, as it tells the compiler to expect an argument which is guaranteed to be used. We collect all applicativeness annotations which occur on the right hand side of the function arrow, as these are unique for every argument. If the annotation variable belonging to a lambda is solved to S when all these applicativeness annotations are set to S, the lambda can be made strict.



EXPERIMENT

The experiment tests the analyses described in the previous section (7) against the analysis which currently exists in Helium[1] (4.3.5). The setup of the experiment is described in section 6.2. The analyses will compile a test file which includes importing the Prelude. The precision of the analyses is measured by the number of strict bindings in the source file, the analysis cost is measured by time using RTS. The compiler was used with the argument –build-all enabled to force rebuilding of Prelude, and –strictness=n, where n stand for the chosen analysis (1 = existing, 2 = monovariant, 3 = polyvariant)[2].

	Time (ms)	**let!** (Prelude)	$\lambda!x$ (Prelude)	**let!** (Test)	$\lambda!x$ (Test)
Old	7172	552	190	13	11
Monovariant	6734	418	191	23	9
Polyvariant	8422	641	200	34	11

TABLE 8.1: *Results for the existing, monovariant and polyvariant analysis*

Comparing the monovariant and polyvariant implementation, it is expected that the latter has a better precision, but is also a bit slower. That expectation is realized, though the margins are much bigger than expected, especially in regards to precision.

The problem of the monovariant analysis are the applicativeness annotations. In a monovariant setting, all these annotations have to be set to either L or S. This means that it has to take into account all applications of a function. If there is one case where the function is not fully applied, the annotation becomes L and none of the usages will be able to derive strictness. Furthermore, even if all usages within the module are fully applied, exported functions have no guarantee of being fully applied in any module where they are imported, meaning they have to be conservatively set to L as well. This problem means that the analysis is only able to infer strictness in the final argument, which can simply be achieved without the applicativeness annotations. Therefore, the monovariant analysis has unacceptable precision as it is even less precise than the old analysis, even though it is a slight bit faster.

[1] This includes the patch for soundness described in 4.3.5

[2] The implementation can be found at `https://github.com/Helium4Haskell/helium/tree/strictness-analysis-deprecated`. The full command (for $n = 1$) is helium +RTS -T -s -RTS "Test.hs" –build-all –strictness=1

The polyvariant analysis does not have this problem, as all applicativeness annotations are quantified. This results in a very precise analysis, though it also slows down the analysis. Comparing the polyvariant analysis to the existing analysis, it improves the number of strict lets and lambdas considerably. All information which was found by the old analysis is also found by the new analysis, but the new analysis also manages to gain information from places which the old analysis is unable to. However, the polyvariant analysis is much slower than the existing analysis, the difference being almost one-and-a-half seconds. While this is not a terrible analysis cost, especially considering Prelude is a very large file and is typically only compiled once, it is an argument in favor of keeping the old analysis.

As for an intermediate conclusion, both systems have their positive and negative aspects. The polyvariant analysis has a better precision but worse analysis cost, while the old analysis has a better analysis cost but worse precision. Neither system has a balanced trade-off, which means the decision as to which system to implement boils down to personal preference. One might prefer the analysis to be as precise as possible at all cost, while another wants an average analysis which is very fast. As there were concerns over the soundness of the strictness analysis, though the fix described in 4.3.5 seems to have fixed the issue, the polyvariant analysis is the preferred aalysis to use.

While the polyvariant analysis is an improvement over the existing analysis in terms of precision, there is still some room for improvement on the analysis cost. The next chapter describes two attempts to make the analysis faster and create a better trade-off.

9

DIFFERENT APPROACHES

The results from the previous section provided an ambiguous result. The new analysis has a considerable improvement in precision, but also slows down the compiler. This chapter hopes to find a compromise which is both precise and fast. Section 9.1 describes an adaptation of the analysis which uses an applicativeness counter instead of applicativeness annotations. Section 9.2 describes an adaptation of the monovariant analysis to include monotypes instead of monomorphic types.

9.1 COUNTING ARGUMENTS

The applicativeness annotations seem to be slowing down the analysis. Removing these annotations alltogether leads to an unsound analysis, but this can be mitigated by preventing the transformation of partial applications. Section 9.1.1 describes the general idea, section 9.1.2 describes the new transformation rules and algorithm, and section 9.1.4 describes the results of the new analysis and compares it to the previous version and the existing analysis. Finally, a note is made on the soundness of the analysis 9.1.5.

9.1.1 GENERAL IDEA

The original analysis uses relevance and applicativeness annotations to make strict transformations. The relevance annotations are the ones which actually signal whether this transformation can take place. The applicativeness annotations are there to make the system sound, but do not actually contribute to the transformation as this is decided by a relevance annotation. The crucial observation why the analysis with only relevance annotations failed is partial applications. As their name suggest, the applicativeness annotations solved this problem making sure a relevance annotation could only become S if all arguments of a function have been applied. Nevertheless, all these annotations have to be stored on function arrows, in environments and have to be solved. If we can find a way to remove the applicativeness annotation, the time needed to solve annotations could be reduced by two thirds. However, we would have to find an alternative approach to ensure correctness.

Instead of looking at annotations to signal full application, we could look at the application rule itself. If we were to add a counter, which counts how many arguments have been given, and know the arity of a function (which can be calculated), we could prevent partial applications from being analyzed

in a S context. Instead of an applicativeness context as annotation, we have an applicativeness context as counter. The counters starts at 0 at the start of every expression or when a context reset takes place. If we encounter an application, the context for the function is incremented. By taking the arity of the function, we can see how many arguments are needed for this function, and by the incremented counter we know how many arguments have been given. If these are equal, then the function is fully applied and the relevance context of the argument is decided by the relevance of the entire application and the function. Otherwise, the function is applied to not enough arguments, and we set to relevance context to L.

As an example, consider the fully applied *const true false* and partially applied *const true*, *const* being a function of arity 2. In the full application, we start with context counter 0, and encounter the application of *const true* to *false*. The function has arity 1 because it still expects one argument, and the number of arguments given is $0 + 1 = 1$, meaning they match and we can use the relevance annotation of the function, which is L as const is not strict in its second argument. As we increment the counter, the expression *const true* has context counter 1. As *const* is of arity 2, and there have now been $1 + 1 = 2$ arguments provided, we are again allowed to use the annotation on the function arrow, which in this case is S, meaning the subexpression *true* can be made strict. If we look at *const true* as partial application, we start with context counter 0, the function *const* has arity 2 but only $0+1 = 1$ argument is being provided, thus the relevance context of the argument is L even if *const* is strict in its first argument.

The type and terms from the original implementation can be reused, except for the applicativeness annotations on the function arrows. The type environment also no longer stores an applicativeness annotation, only the annotated type and relevance annotation. We denote the arity of a type by $\|\hat{\tau}\|$:

$$\begin{aligned}
\|d\,\overline{\hat{\tau}_n}\| &= 0 \\
\|\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2\| &= 1 + \|\hat{\tau}_2\| \\
\|\alpha\| &= 0 \\
\|\forall\alpha.\hat{\tau}\| &= \|\hat{\tau}\| \\
\|\,!\hat{\tau}\| &= \|\hat{\tau}\|
\end{aligned}$$

9.1.2 Transformation rules and algorithm

The transformation are presented in a similar fashion as section 7.1. The terms remain unchanged, and the type system reverts from three annotations on the arrow to one. Instead of an applicativeness annotation, we have an applicativeness context, which is a natural number. The adaptation from transformation rules to algorithm, constraint solving and adaptations for polyvariance are identical to those sketched in chapter 7, and are omitted in this section.

Constructors, literals and variables

The transformation rules for constructors, literals and variables are described in figure 9.1.

$$\frac{}{[] \vdash c \rhd c : \hat{\tau}^{(\varphi,n)}}\ [\text{r-con}]$$

$$\frac{}{[] \vdash l \rhd l : \hat{\tau}^{(\varphi,n)}}\ [\text{r-lit}]$$

$$\frac{}{[x \mapsto (\hat{\tau},\varphi)] \vdash x \rhd x : \hat{\tau}^{(\varphi,n)}}\ [\text{r-var}]$$

Figure 9.1: *Transformation rules for constructors, literals and variables*

Rules [r-con], [r-lit] and [r-var] remain the same except for the applicativeness context being dropped. In return, the applicativeness counter is added, but they play no role in these transformation rules.

APPLICATION

The transformation rules for application are described in figure 9.2.

$$\frac{\|\hat{\tau}_2\| = n+1 \quad \hat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\hat{\tau}_2 \xrightarrow{\varphi_0} \hat{\tau})^{(\varphi,n+1)} \quad \hat{\Gamma}_2 \vdash t_2 \rhd t_2' : \hat{\tau}_2^{(\varphi \sqcup \varphi_0, 0)}}{\hat{\Gamma}_1 \diamond^{\sqcap} \hat{\Gamma}_2 \vdash t_1 t_2 \rhd t_1' t_2' : \hat{\tau}^{(\varphi,n)}} \text{[r-app]}$$

$$\frac{\|\hat{\tau}_2\| \neq n+1 \quad \hat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\hat{\tau}_2 \xrightarrow{\varphi_0} \hat{\tau})^{(\varphi,n+1)} \quad \hat{\Gamma}_2 \vdash t_2 \rhd t_2' : \hat{\tau}_2^{(\mathrm{L},0)}}{\hat{\Gamma}_1 \diamond^{\sqcap} \hat{\Gamma}_2 \vdash t_1 t_2 \rhd t_1' t_2' : \hat{\tau}^{(\varphi,n)}} \text{[r-papp]}$$

FIGURE 9.2: *Transformation rules for applications*

The rule for application, [r-app], now only handles fully applied functions. This is the case if the arity of t_1 is equal to the incremented applicativeness counter. If so, the context of the argument is the join of the annotation on the function arrow and the relevance context. The applicativeness counter in the argument is reset to 0, as there might be a new application in there. The counter for the function is incremented by one to reflect an argument has been provided.

Rule [r-papp] is added to reflect partial application. In this case, the arity is not equal to the incremented counter, and thus the argument receives an L-context. In practice, the applicativeness counter cannot be bigger than the arity, as this implies more arguments than expected have been given to the function, which should not be possible in a well-typed system. The function *const not False True* might seem to have too many arguments, but the instantiated type of *const* is

$$(\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool} \rightarrow (\text{Bool} \rightarrow \text{Bool})$$

which is equivalent to

$$(\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$$

which has an arity of three. Because it is given exactly three arguments, both *not* and *True* can be converted to strict bindings.

ABSTRACTION

The transformation rules for abstractions are described in figure 9.3.

$$\frac{\varphi \blacktriangleright \hat{\Gamma} \quad \hat{\Gamma}[x \mapsto (\hat{\tau}, \mathrm{S})] \vdash t_1 \rhd t_1' : \hat{\tau}_0^{(\mathrm{S},0)}}{\hat{\Gamma} \vdash \lambda x\{\hat{\tau}\}.t_1 \rhd \lambda !x\{\hat{\tau}\}.t_1' : (\hat{\tau} \xrightarrow{\mathrm{S}} \hat{\tau}_0)^{(\varphi,n)}} \text{[r-abs1]}$$

$$\frac{\varphi \blacktriangleright \hat{\Gamma} \quad \hat{\Gamma}[x \mapsto (\hat{\tau}, \mathrm{L})] \vdash t_1 \rhd t_1' : \hat{\tau}_0^{(\mathrm{S},0)}}{\hat{\Gamma} \vdash \lambda x\{\hat{\tau}\}.t_1 \rhd \lambda x\{\hat{\tau}\}.t_1' : (\hat{\tau} \xrightarrow{\mathrm{L}} \hat{\tau}_0)^{(\varphi,n)}} \text{[r-abs2]}$$

$$\frac{\varphi \blacktriangleright \hat{\Gamma} \quad \hat{\Gamma}[x \mapsto (\hat{\tau}, \mathrm{S})] \vdash t_1 \rhd t_1' : \hat{\tau}_0^{(\mathrm{S},0)}}{\hat{\Gamma} \vdash \lambda !x\{\hat{\tau}\}.t_1 \rhd \lambda !x\{\hat{\tau}\}.t_1' : (\hat{\tau} \xrightarrow{\varphi_1} \hat{\tau}_0)^{(\varphi,n)}} \text{[r-sabs]}$$

FIGURE 9.3: *Transformation rules for abstractions*

The containment in the abstraction rules [r-abs1], [r-abs2] and [r-sabs] reverts to the relevance context, as the applicativeness context is no longer an annotation. The term bound to the abstractions have their contexts reset to S and 0. This is done to get the strictness information for the newly defined variable. The purpose of the containment is to avoid the information on every other variable being used if we are in an L-context.

LET BINDINGS

The transformation rules for let bindings are described in figure 9.4.

$$\frac{\varphi \blacktriangleright \widehat{\Gamma}_1 \quad \widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \mathrm{S})] \vdash t_2 \rhd t_2' : \widehat{\tau}_0^{(\mathrm{S},n)} \quad \widehat{\Gamma}_2 \vdash t_1 \rhd t_1' : \widehat{\tau}^{(\varphi,0)}}{\widehat{\Gamma}_1 \diamond^{\sqcap} \widehat{\Gamma}_2 \vdash \textbf{let}\, x\{\widehat{\tau}\} = t_1 \,\textbf{in}\, t_2 \rhd \textbf{let!}\; x\{\widehat{\tau}\} = t_1' \,\textbf{in}\, t_2' : \widehat{\tau}_0^{(\varphi,n)}} \text{ [r-let1]}$$

$$\frac{\varphi \blacktriangleright \widehat{\Gamma}_1 \quad \widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \mathrm{L})] \vdash t_2 \rhd t_2' : \widehat{\tau}_0^{(\mathrm{S},n)} \quad \widehat{\Gamma}_2 \vdash t_1 \rhd t_1' : \widehat{\tau}^{(\mathrm{L},0)}}{\widehat{\Gamma}_1 \diamond^{\sqcap} \widehat{\Gamma}_2 \vdash \textbf{let}\, x\{\widehat{\tau}\} = t_1 \,\textbf{in}\, t_2 \rhd \textbf{let}\, x\{\widehat{\tau}\} = t_1' \,\textbf{in}\, t_2' : \widehat{\tau}_0^{(\varphi,n)}} \text{ [r-let2]}$$

$$\frac{\varphi \blacktriangleright \widehat{\Gamma}_1 \quad \widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \mathrm{S})] \vdash t_2 \rhd t_2' : \widehat{\tau}_0^{(\mathrm{S},n)} \quad \widehat{\Gamma}_2 \vdash t_1 \rhd t_1' : \widehat{\tau}^{(\varphi,0)}}{\widehat{\Gamma}_1 \diamond^{\sqcap} \widehat{\Gamma}_2 \vdash \textbf{let!}\; x\{\widehat{\tau}\} = t_1 \,\textbf{in}\, t_2 \rhd \textbf{let!}\; x\{\widehat{\tau}\} = t_1' \,\textbf{in}\, t_2' : \widehat{\tau}_0^{(\varphi,n)}} \text{ [r-let!]}$$

$$\frac{\begin{array}{c}\forall i.1 \leq i \leq n : \widehat{\Gamma}_2[x_1 \mapsto (\widehat{\tau}_1, \varphi_1), \ldots, x_n \mapsto (\widehat{\tau}_n, \varphi_n)] \vdash t_i \rhd t_i' : \widehat{\tau}_i^{(\varphi \sqcup \varphi_i, 0)} \\ \varphi \blacktriangleright \widehat{\Gamma}_1 \quad \widehat{\Gamma}_1[x_1 \mapsto (\widehat{\tau}_1, \varphi_1), \ldots, x_n \mapsto (\widehat{\tau}_n, \varphi_n)] \vdash t \rhd t' : \widehat{\tau}_0^{(\mathrm{S},n)}\end{array}}{\widehat{\Gamma}_1 \diamond^{\sqcap} \widehat{\Gamma}_2 \vdash \textbf{letrec}\, \overline{x_n\{\widehat{\tau}_n\} = t_n} \,\textbf{in}\, t \rhd \textbf{letrec}\, \overline{x_n\{\widehat{\tau}_n\} = t_n'} \,\textbf{in}\, t' : \widehat{\tau}_0^{(\varphi,n)}} \text{ [r-letrec]}$$

FIGURE 9.4: *Transformation rules for let bindings*

The derivation of the let rules shown previously still holds, meaning rules [r-let1], [r-let2] and [r-let!] can be derived as combination from the previous abstraction rules and the full application rule. Rule [r-letrec] also follows the same format as these rules, with the adaptation that it contains multiple bindings and all variables have to be included in $\widehat{\Gamma}_2$.

CASE

The transformation rules for pattern matches are described in figure 9.5.

$$\frac{\forall i.1 \leq i \leq n : \widehat{\Gamma}_i \vdash a_i \rhd a_i' : \widehat{\tau}^{(\varphi,n)}}{(\widehat{\Gamma}_1 \diamond^{\sqcup} \ldots \diamond^{\sqcup} \widehat{\Gamma}_n) \vdash \textbf{case}\; x \;\textbf{of}\; \overline{a_n} \rhd \textbf{case}\; x \;\textbf{of}\; \overline{a_n'} : \widehat{\tau}^{(\varphi,n)}} \text{ [r-case]}$$

$$\frac{\widehat{\Gamma}[x_1 \mapsto (\widehat{\tau}_1, \mathrm{L}), \ldots, x_n \mapsto (\widehat{\tau}_n, \mathrm{L}, \mathrm{L})] \vdash t \rhd t' : \widehat{\tau}^{(\varphi,n)}}{\widehat{\Gamma} \vdash c\overline{x_n\{\widehat{\tau}_n\}} \to t \rhd c\overline{x_n\{\widehat{\tau}_n\}} \to t' : \widehat{\tau}^{(\varphi,n)}} \text{ [a-con]}$$

$$\frac{\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi,n)}}{\widehat{\Gamma} \vdash l \to t \rhd l \to t' : \widehat{\tau}^{(\varphi,n)}} \text{ [a-lit]}$$

$$\frac{\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi,n)}}{\widehat{\Gamma} \vdash _ \to t \rhd _ \to t' : \widehat{\tau}^{(\varphi,n)}} \text{ [a-def]}$$

FIGURE 9.5: *Transformation rules for pattern matching*

Rule [r-case] follows the same transformation as seen in previous rules, the applicativeness annotation is replaced by the applicativeness counter and boolean. Like its previous counterpart, all alternatives have the same context, the context before the case expression, and its information is merged with the join to reflect that all branches should report S for a variable to be strict. As the variable to be matched on is handled in a strict bind before the expression, and we no longer need to communicate the variable is being applied in the pattern match, we can remove the premise surrounding the variable and no longer need the initial context split to convey this information.

Rules [a-con], [a-lit] and [a-def] trade their applicativeness context for an applicativeness counter.

GENERALIZATION AND INSTANTIATION

The transformation rules for generalization and instantiation are described in figure 9.6.

$$\frac{\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi,n)}}{\widehat{\Gamma} \vdash \textbf{forall}\ \alpha.t \rhd \textbf{forall}\ \alpha.t' : \forall\alpha.\widehat{\tau}^{(\varphi,n)}}\ \text{[r-forall]}$$

$$\frac{\widehat{\Gamma} \vdash t \rhd t' : \forall\alpha.\widehat{\tau}^{(\varphi,n)}}{\widehat{\Gamma} \vdash t\{\widehat{\tau}_1\} \rhd t'\{\widehat{\tau}_1\} : [\alpha \mapsto \widehat{\tau}_1]\widehat{\tau}^{(\varphi,n)}}\ \text{[r-aptype]}$$

FIGURE 9.6: *Transformation rules for generalization and instantiation*

The transformations for [r-forall] and [r-aptype] are trivial.

SUBEFFECTING, WEAKENING AND CONTAINMENT

Subeffecting, weakening and containment are described in figure 9.7.

$$\frac{\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\mathrm{L},n)}}{\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\mathrm{S},n)}}\ \text{[r-sub]}$$

$$\frac{\widehat{\Gamma} \vdash t \rhd t' : \widehat{\tau}^{(\varphi,n)}}{\widehat{\Gamma}[x \mapsto (\widehat{\tau}_0, \mathrm{L})] \vdash t \rhd t' : \widehat{\tau}^{(\varphi,n)}}\ \text{[r-weak]}$$

$$\frac{}{\varphi \blacktriangleright []}\ \text{[c-nil]}$$

$$\frac{\mathrm{S} \blacktriangleright \widehat{\Gamma}_1}{\mathrm{S} \blacktriangleright \widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \varphi_0)]}\ \text{[c-cons-s]}$$

$$\frac{\mathrm{L} \blacktriangleright \widehat{\Gamma}_1}{\mathrm{L} \blacktriangleright \widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \mathrm{L})]}\ \text{[c-cons-l]}$$

FIGURE 9.7: *Subeffecting, weakening and containment*

The transformations for [r-sub] and [r-weak] are trivial. Containment rules [c-nil], [c-cons-s] and [c-cons-l] revert to their original definition in section 3.1.

9.1.3 OVERVIEW

All transformation rules defined in the previous sections are combined into figure 9.8 for convenience.

$$\frac{}{[] \vdash c \rhd c : \hat{\tau}^{(\varphi,n)}}\ [\text{r-con}] \qquad \frac{}{[] \vdash l \rhd l : \hat{\tau}^{(\varphi,n)}}\ [\text{r-lit}] \qquad \frac{}{[x \mapsto (\hat{\tau}, \varphi)] \vdash x \rhd x : \hat{\tau}^{(\varphi,n)}}\ [\text{r-var}]$$

$$\frac{\varphi \blacktriangleright \hat{\Gamma} \quad \hat{\Gamma}[x \mapsto (\hat{\tau}, \mathrm{S})] \vdash t_1 \rhd t_1' : \hat{\tau}_0^{(\mathrm{S},0)}}{\hat{\Gamma} \vdash \lambda x\{\hat{\tau}\}.t_1 \rhd \lambda! x\{\hat{\tau}\}.t_1' : (\hat{\tau} \xrightarrow{\mathrm{S}} \hat{\tau}_0)^{(\varphi,n)}}\ [\text{r-abs1}]$$

$$\frac{\varphi \blacktriangleright \hat{\Gamma} \quad \hat{\Gamma}[x \mapsto (\hat{\tau}, \mathrm{L})] \vdash t_1 \rhd t_1' : \hat{\tau}_0^{(\mathrm{S},0)}}{\hat{\Gamma} \vdash \lambda x\{\hat{\tau}\}.t_1 \rhd \lambda x\{\hat{\tau}\}.t_1' : (\hat{\tau} \xrightarrow{\mathrm{L}} \hat{\tau}_0)^{(\varphi,n)}}\ [\text{r-abs2}]$$

$$\frac{\varphi \blacktriangleright \hat{\Gamma} \quad \hat{\Gamma}[x \mapsto (\hat{\tau}, \mathrm{S})] \vdash t_1 \rhd t_1' : \hat{\tau}_0^{(\mathrm{S},0)}}{\hat{\Gamma} \vdash \lambda! x\{\hat{\tau}\}.t_1 \rhd \lambda! x\{\hat{\tau}\}.t_1' : (\hat{\tau} \xrightarrow{\varphi_1} \hat{\tau}_0)^{(\varphi,n)}}\ [\text{r-sabs}]$$

$$\frac{\|\hat{\tau}_2\| = n+1 \quad \hat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\hat{\tau}_2 \xrightarrow{\varphi_0} \hat{\tau})^{(\varphi,n+1)} \quad \hat{\Gamma}_2 \vdash t_2 \rhd t_2' : \hat{\tau}_2^{(\varphi \sqcup \varphi_0,0)}}{\hat{\Gamma}_1 \diamond^{\sqcap} \hat{\Gamma}_2 \vdash t_1 t_2 \rhd t_1' t_2' : \hat{\tau}^{(\varphi,n)}}\ [\text{r-app}]$$

$$\frac{\|\hat{\tau}_2\| \neq n+1 \quad \hat{\Gamma}_1 \vdash t_1 \rhd t_1' : (\hat{\tau}_2 \xrightarrow{\varphi_0} \hat{\tau})^{(\varphi,n+1)} \quad \hat{\Gamma}_2 \vdash t_2 \rhd t_2' : \hat{\tau}_2^{(\mathrm{L},0)}}{\hat{\Gamma}_1 \diamond^{\sqcap} \hat{\Gamma}_2 \vdash t_1 t_2 \rhd t_1' t_2' : \hat{\tau}^{(\varphi,n)}}\ [\text{r-papp}]$$

$$\frac{\varphi \blacktriangleright \hat{\Gamma}_1 \quad \hat{\Gamma}_1[x \mapsto (\hat{\tau}, \mathrm{S})] \vdash t_2 \rhd t_2' : \hat{\tau}_0^{(\mathrm{S},n)} \quad \hat{\Gamma}_2 \vdash t_1 \rhd t_1' : \hat{\tau}^{(\varphi,0)}}{\hat{\Gamma}_1 \diamond^{\sqcap} \hat{\Gamma}_2 \vdash \mathbf{let}\ x\{\hat{\tau}\} = t_1\ \mathbf{in}\ t_2 \rhd \mathbf{let!}\ x\{\hat{\tau}\} = t_1'\ \mathbf{in}\ t_2' : \hat{\tau}_0^{(\varphi,n)}}\ [\text{r-let1}]$$

$$\frac{\varphi \blacktriangleright \hat{\Gamma}_1 \quad \hat{\Gamma}_1[x \mapsto (\hat{\tau}, \mathrm{L})] \vdash t_2 \rhd t_2' : \hat{\tau}_0^{(\mathrm{S},n)} \quad \hat{\Gamma}_2 \vdash t_1 \rhd t_1' : \hat{\tau}^{(\mathrm{L},0)}}{\hat{\Gamma}_1 \diamond^{\sqcap} \hat{\Gamma}_2 \vdash \mathbf{let}\ x\{\hat{\tau}\} = t_1\ \mathbf{in}\ t_2 \rhd \mathbf{let}\ x\{\hat{\tau}\} = t_1'\ \mathbf{in}\ t_2' : \hat{\tau}_0^{(\varphi,n)}}\ [\text{r-let2}]$$

$$\frac{\varphi \blacktriangleright \hat{\Gamma}_1 \quad \hat{\Gamma}_1[x \mapsto (\hat{\tau}, \mathrm{S})] \vdash t_2 \rhd t_2' : \hat{\tau}_0^{(\mathrm{S},n)} \quad \hat{\Gamma}_2 \vdash t_1 \rhd t_1' : \hat{\tau}^{(\varphi,0)}}{\hat{\Gamma}_1 \diamond^{\sqcap} \hat{\Gamma}_2 \vdash \mathbf{let!}\ x\{\hat{\tau}\} = t_1\ \mathbf{in}\ t_2 \rhd \mathbf{let!}\ x\{\hat{\tau}\} = t_1'\ \mathbf{in}\ t_2' : \hat{\tau}_0^{(\varphi,n)}}\ [\text{r-let!}]$$

$$\frac{\begin{array}{c}\forall i.1 \leq i \leq n : \hat{\Gamma}_2[x_1 \mapsto (\hat{\tau}_1, \varphi_1), \ldots, x_n \mapsto (\hat{\tau}_n, \varphi_n)] \vdash t_i \rhd t_i' : \hat{\tau}_i^{(\varphi \sqcup \varphi_i,0)} \\ \varphi \blacktriangleright \hat{\Gamma}_1 \quad \hat{\Gamma}_1[x_1 \mapsto (\hat{\tau}_1, \varphi_1), \ldots, x_n \mapsto (\hat{\tau}_n, \varphi_n)] \vdash t \rhd t' : \hat{\tau}_0^{(\mathrm{S},n)}\end{array}}{\hat{\Gamma}_1 \diamond^{\sqcap} \hat{\Gamma}_2 \vdash \mathbf{letrec}\ \overline{x_n\{\hat{\tau}_n\} = t_n}\ \mathbf{in}\ t \rhd \mathbf{letrec}\ \overline{x_n\{\hat{\tau}_n\} = t_n'}\ \mathbf{in}\ t' : \hat{\tau}_0^{(\varphi,n)}}\ [\text{r-letrec}]$$

$$\frac{\forall i.1 \leq i \leq n : \hat{\Gamma}_i \vdash a_i \rhd a_i' : \hat{\tau}^{(\varphi,n)}}{(\hat{\Gamma}_1 \diamond^{\sqcup} \ldots \diamond^{\sqcup} \hat{\Gamma}_n) \vdash \mathbf{case}\ x\ \mathbf{of}\ \overline{a_n} \rhd \mathbf{case}\ x\ \mathbf{of}\ \overline{a_n'} : \hat{\tau}^{(\varphi,n)}}\ [\text{r-case}]$$

$$\frac{\hat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{(\varphi,n)}}{\hat{\Gamma} \vdash \mathbf{forall}\ \alpha.t \rhd \mathbf{forall}\ \alpha.t' : \forall \alpha.\hat{\tau}^{(\varphi,n)}}\ [\text{r-forall}] \qquad \frac{\hat{\Gamma} \vdash t \rhd t' : \forall \alpha.\hat{\tau}^{(\varphi,n)}}{\hat{\Gamma} \vdash t\{\hat{\tau}_1\} \rhd t'\{\hat{\tau}_1\} : [\alpha \mapsto \hat{\tau}_1]\hat{\tau}^{(\varphi,n)}}\ [\text{r-aptype}]$$

$$\frac{\hat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{(\mathrm{L},n)}}{\hat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{(\mathrm{S},n)}}\ [\text{r-sub}] \qquad \frac{\hat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{(\varphi,n)}}{\hat{\Gamma}[x \mapsto (\hat{\tau}_0, \mathrm{L})] \vdash t \rhd t' : \hat{\tau}^{(\varphi,n)}}\ [\text{r-weak}]$$

$$\frac{\hat{\Gamma}[x_1 \mapsto (\hat{\tau}_1, \mathrm{L}), \ldots, x_n \mapsto (\hat{\tau}_n, \mathrm{L}, \mathrm{L})] \vdash t \rhd t' : \hat{\tau}^{(\varphi,n)}}{\hat{\Gamma} \vdash c\overline{x_n\{\hat{\tau}_n\}} \to t \rhd c\overline{x_n\{\hat{\tau}_n\}} \to t' : \hat{\tau}^{(\varphi,n)}}\ [\text{a-con}]$$

$$\frac{\hat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{(\varphi,n)}}{\hat{\Gamma} \vdash l \to t \rhd l \to t' : \hat{\tau}^{(\varphi,n)}}\ [\text{a-lit}] \qquad \frac{\hat{\Gamma} \vdash t \rhd t' : \hat{\tau}^{(\varphi,n)}}{\hat{\Gamma} \vdash _ \to t \rhd _ \to t' : \hat{\tau}^{(\varphi,n)}}\ [\text{a-def}]$$

$$\frac{}{\varphi \blacktriangleright []}\ [\text{c-nil}] \qquad \frac{\mathrm{S} \blacktriangleright \hat{\Gamma}_1}{\mathrm{S} \blacktriangleright \hat{\Gamma}_1[x \mapsto (\hat{\tau}, \varphi_0)]}\ [\text{c-cons-s}] \qquad \frac{\mathrm{L} \blacktriangleright \hat{\Gamma}_1}{\mathrm{L} \blacktriangleright \hat{\Gamma}_1[x \mapsto (\hat{\tau}, \mathrm{L})]}\ [\text{c-cons-l}]$$

FIGURE 9.8: *Adjusted relevance typing, call-by-value transformations and containment*

9.1.4 RESULTS

A monovariant and polyvariant analysis based on the applicativeness counter (r) are compared against the monovariant and polyvariant analysis with applicativeness annotations (r & a). The experiment is performed using the same setup as described in 6.2, and with the commands described in 8 [1].

	Time (ms)	**let!** (Prelude)	λ! (Prelude)	**let!** (Test)	λ! (Test)
Old	7172	552	190	13	11
Monovariant (r & a)	6734	418	191	23	9
Polyvariant (r & a)	8422	641	200	34	11
Monovariant (r)	7266	626	189	35	11
Polyvariant (r)	7891	626	189	40	11

TABLE 9.1: *Results for the existing analysis, the monovariant and polyvariant analysis with applicativeness annotations, and the monovariant and polyvariant analysis without applicativeness annotations*

Comparing the new monovariant and polyvariant with each other, there is almost no difference in precision. While this might seem surprising, the equality on Prelude is easily explained. The Prelude mostly defines functions instead of using them. For instance, ($) is a function which would behave differently in a monovariant and polyvariant analysis. While Prelude defines this function, it is never used in that module, meaning the difference between the two analysis is not visible in Prelude. Only when the dollar sign is used, which is the case in Test, does the difference become apparent. The test cases in which they differ is a test case where the ($) is used with a strict and non-strict function, and folds. The monovariant analysis infers strictness on neither, while the polyvariant analysis is able to determine the argument applied to the first function can be made strict.

Their difference in time is much closer than the monovariant and polyvariant analysis with applicativeness annotations. This can be explained by the reduced number of annotations. The previous versions included three annotations per argument, while the new version only has one. This leads to a reduction in the number of annotations needed to be solved, which means the solving algorithm becomes quicker. However, the algorithm itself is still called the same number of times as in the previous versions, meaning polyvariance is much quicker than its counterpart. The monovariant analysis is slightly slower than its previous version because the overhead of checking whether a variable is used in a fully applied function is bigger than the time we gain by reducing the number of annotations, which points to the observation that calling the solving algorithm many times with a smaller set of constraints is now almost as fast as calling the algorithm once with all constraints. As the monovariant and polyvariant analysis are almost as precise as each other, and the former is considerably faster, a monovariant analysis without applicativeness annotations is preferable over a polyvariant analysis without applicativeness annotations.

Comparing to the analyses with applicativeness annotations, while the speed is a step forward, the precision is a slight step back, which is to be expected with a less complex analysis. The difference between the previous and current version remains marginal though, as the five cases for let bindings in Prelude are all related to IO functions. Since IO is stored as a type synonym for (RealWorld -> IORes), the new version is not able to infer any strictness because of the missing arguments for RealWorld. An example in the test file which is handled by the previous versions but not by the current is that of functions returning functions. Consider the example $\lambda x.$**if** x **then** not **else** id. Giving one argument to this function returns another function, *not* or *id* depending on the input. If we give undefined as input, the function crashes

[1] The implementation can be found at `https://github.com/Helium4Haskell/helium/tree/strictness-analysis-deprecated`. The options to enable these versions of the strictness analysis is –strictness=4 for the monovariant version and –strictness=5 for the polyvariant version

as it needs to evaluate x. The analysis with applicativeness annotations is able to derive that the first argument is strict when only one argument is given, while the analysis with the application counter will determines that the function is not fully applied because no argument is given to *not* or *id*, and thus the strictness information cannot be used.

Comparing the new versions to the existing analysis, the analysis cost is almost equal if we take the monovariant version. In terms of precision, the new version can infer much more. Therefore, it looks like the monovariant analysis without applicativeness annotations provides the best trade-off, even compared against the monotyped version in the previous section, if it turns out to be sound.

9.1.5 SOUNDNESS

While the simpler approach manages to avoid the problem with partial application, it instead creates another: functions as arguments. Consider the *apply* function $\lambda f.\lambda x.fx$. This example has been used to differentiate between a monovariant and polyvariant analysis, with the former unable to differentiate between a strict or non-strict function, defaulting to the latter. The polyvariant analyses inferred a type signature $\forall\varphi.((\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2) \xrightarrow{S} \hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2)$. As example, let the function apply take *const* $(\hat{\tau}_1 \xrightarrow{S} \hat{\tau}_2 \xrightarrow{L} \hat{\tau}_1)$ and *True* as arguments.

$$\frac{\|\text{Bool} \xrightarrow{S} \hat{\tau}_2 \xrightarrow{L} \text{Bool}\| \neq 1 \quad [] \vdash \text{apply const} : (\text{Bool} \xrightarrow{S} \hat{\tau}_2 \xrightarrow{L} \text{Bool})^{(S,1)} \quad [] \vdash \text{True} : \text{Bool}^{(L,0)}}{[] \vdash \text{apply const True} : (\hat{\tau}_2 \xrightarrow{L} \text{Bool})^{(S,0)}}$$

Because the arity of *apply const* is two, *True* cannot be made strict because it is not fully applied. So far, so good. Now, assume that instead of applying function f to argument x, *apply'* evaluates fx and returns *False* $(\lambda f.\lambda x.(fx)$ 'seq' False). The type of this function is $\forall\varphi.((\hat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau_2}) \xrightarrow{S} \hat{\tau}_1 \xrightarrow{\varphi} \text{Bool})$. If this function is applied with const and true, the arity of *apply const* is no longer two but one, as it returns Bool instead of the remainder of the function. As a result, the hypothetical let binding which defines true can be made strict, because the application is full according to the calculations. This means the polyvariant version of this analysis is unsound.

$$\frac{\|\text{Bool} \xrightarrow{S} \text{Bool}\| = 1 \quad [] \vdash \text{apply' const} : (\text{Bool} \xrightarrow{S} \text{Bool})^{(S,1)} \quad [] \vdash \text{True} : \text{Bool}^{(S,0)}}{[] \vdash \text{apply' const True} : \text{Bool}^{(S,0)}}$$

The problem lies in the derivation of the type for *apply'*. The relevance of the second argument should no longer depend on the relevance of the function, but be L because it is not guaranteed that the function has an arity of one. This means that for every lambda which defines a function with type variables, we can no longer use the strictness information from the body of the abstraction because we cannot know which functions are passed to *apply'*. This restriction also exists in a let binding because of its equivalence to a lambda combined with an application. The loss of precision brings the analysis to a level even below the existing analysis, meaning it no longer serves as a compromise between precision and speed.

The monovariant analysis does not experience this specific problem, because the annotation variables are defaulted to L and thus the argument to *apply* and *apply'* are always non-strict. However, this analysis is also not free of soundness issues. The applicativeness counter was explicitly designed to combat partial applications, but the transformation rules defined by Holdermans and Hage noted another problem. When an application is already strict, it can infer strictness in certainly places when functions are given as arguments. The given example[2] was $\lambda x.((\$!)\ (\lambda y.0)\ (\lambda z.x))$, in which it erroneously derived this

[2]Strict application • replaced by its function counterpart \$!. The monomorphic type of \$! is $((\hat{\tau}_1 \xrightarrow{L} \widehat{\tau_2}) \xrightarrow{S} \hat{\tau}_1 \xrightarrow{S} \hat{\tau}_2)$.

function to be strict in x while x is never evaluated. This analysis suffers the same problem[3]:

$$\dfrac{\text{S} \blacktriangleright [] \quad \dfrac{\|((\hat{\tau}_2 \xrightarrow{\text{L}} \hat{\tau}_1) \xrightarrow{\text{S}} \text{Int})\| = 1 \quad \ldots \quad \dfrac{\text{S} \blacktriangleright [x \mapsto (\hat{\tau}, \text{S})] \quad [x \mapsto (\hat{\tau}_1, \text{S}), z \mapsto (\hat{\tau}_2, \text{L})] \vdash x : \hat{\tau}_1^{(\text{S},0)}}{[x \mapsto (\hat{\tau}_1, \text{S})] \vdash (\lambda z.x) : (\hat{\tau}_2 \xrightarrow{\text{L}} \hat{\tau}_1)^{(\text{S},0)}}}{[x \mapsto (\hat{\tau}_1, \text{S})] \vdash ((\$!)\ (\lambda y.0)\ (\lambda z.x)) : \text{Int}^{(\text{S},0)}}}{[] \vdash \lambda x.((\$!)\ (\lambda y.0)\ (\lambda z.x)) : (\hat{\tau}_1 \xrightarrow{\text{S}} \text{Int})^{(\text{S},0)}}$$

In summary, by replacing the applicativeness context with an applicativeness counter, the problems regarding partial application are fixed. However, the problems regarding strict application by program input are still present, and are not easily solvable except removing all manual strictness altogether, which seems counter-intuitive for an analysis which has the purpose of introducing strictness. The applicativeness annotations do manage to solve both problems in a sound manner while remaining precise, at a slight but not major increase in analysis cost. Therefore, this entire section cannot provide an improvement on the analysis described in chapter 7, and does not even provide a significant improvement over the existing analysis.

9.2 MONOTYPES

The monovariant analysis with applicativeness annotations is fast but imprecise, whereas the polyvariant analysis with applicativeness annotations is precise but slow. If we could somehow combine the best of both analyses, we get an analysis which is precise and fast, and would be far superior over the existing analysis. This can be achieved by adapting the monovariant analysis to support monotypes instead of monomorphism. In the previous implementation, all annotation variables had to be weakened to L to be sound. The polyvariant analysis could leave these annotation to get more precision, but the slowdown might be caused by instantiation and frequent solving. If we allow the annotation variables to stay in the type, but we do not quantify over them, we could get an analysis with precision close to the polyvariant analysis at better speeds.

The monovariant algorithm in chapter 7 barely needs any change, only the provision that any remaining annotation variable has to defaulted to L is removed. There is no instantiation with a fresh variable or simplification. This means that if a function is used multiple times per binding group, it can only receive one annotation per variable, which should be the least restrictive. If the function ($) is used with a strict and non-strict function within the same binding group, the annotation variable on the argument passed to the function becomes L and the argument to neither function can be made strict. An important assumption in this analysis is the uniqueness of the strictness annotations between functions, as using the same annotation will lead to weaker results as the annotation can only receive one value per binding group.

The analysis using monotypes is compared against the previous monovariant and polyvariant analysis, as well as the existing analysis. It is performed using the same setup as described in section 6.2, and with the commands described in chapter 8[4].

The results of the analysis with monotypes look very promising, as it matches the speed of the existing analysis while being significantly more precise. Compared to the monomorphic monovariant analysis, it solves the problem of applicativeness as it is much more precise. The differences in precision compared to the polyvariant analysis occur when a function is used multiple times within a function, as polyvariance is still able to differentiate between different calls while the monotyped analysis needs to combine the two, but unlike the monovariant analysis it can make different instantiations per function instead of the

---

[3] $\ldots = [x \mapsto (\hat{\tau}_1, \text{S})] \vdash (\$!)\ (\lambda y.0) : ((\hat{\tau}_2 \xrightarrow{\text{L}} \hat{\tau}_1) \xrightarrow{\text{S}} \text{Int})^{(\text{S},1)}$

[4] The implementation can be found at `https://github.com/Helium4Haskell/helium/tree/strictness-analysis`. The option to enable the versions with monotypes is –strictness=6.

	Time (ms)	**let!** (Prelude)	$\lambda!x$ (Prelude)	**let!** (Test)	$\lambda!x$ (Test)
Old	7172	552	190	13	11
Monovariant	6734	418	191	23	9
Monotyped	7312	611	197	35	10
Polyvariant	8422	641	200	34	11

TABLE 9.2: *Results for the existing, monovariant (monomorphic), monovariant (monotypes) and polyvariant analysis*

entire module. The expectations of the analysis are met, as it is slightly slower than the monomorphic analysis and slightly less precise than the polyvariant analysis, but does provide a very good balance between the two. Unlike the analysis proposed in 9.1, there is no clear evidence of unsoundness found in the transformations on both Prelude and the test file.

The analysis with monotypes has provided the best balance out of the analyses implemented in this thesis, and is also a considerable improvement over the existing analysis. The polyvariant analysis is still a viable option if one wishes to increase the precision and the added speed is not a problem, but the analysis with monotypes can achieve similar results for less analysis cost.

Conclusion

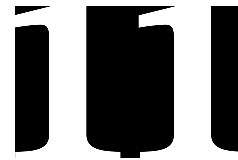
The conclusion answers the three subquestions posed in section 6.1, before answering the main research question: What are the trade-offs between precision and analysis cost of strictness analysis in a real-world compiler?

[SQ1] Can relevance and applicativeness typing analysis be implemented in the Helium compiler?
The system described by Holdermans and Hage in section 3.2 can be implemented in Helium, though they require major modifications to fit with the expression syntax in Helium. Some language constructs like let bindings and pattern matching were not handled in the original transformation rules and thus had to be defined manually. Furthermore, the method of introducing strictness proved to be impractical in Helium. Instead of strict application, Helium relies on strict let and lambda bindings, but the transformation from one method to the other did not cause many issues.

[SQ2] Can a representative benchmark be constructed to compare trade-offs?
A benchmark was constructed which measured the precision and cost on the compilation of a test file which imported the Prelude. Prelude is a very big file and thus very representative to measure the speed of the analysis. While it is suitable to measure precision improvements between different approaches to strictness analysis, it is not suited to measure the differences between a monovariant and polyvariant analysis, because Prelude is used for defining functions rather than using them. The test file provided much more variation in that regard, and provided a suitable comparison for precision. A balanced trade-off should be an analysis which is precise for relatively little analysis cost.

[SQ3] What are the trade-offs against the current strictness analysis in Helium?
The current analysis in Helium has a very low analysis cost, but its precision is left to be desired. The reason for a low precision is its lack of information carried across modules. The new analyses do provide this, which makes them more precise in that regard. The monovariant analysis matches its analysis cost, but falls behind in precision due to a conservative approach on applicativeness. The polyvariant analysis is much more precise but at an increased analysis cost. While not as bad as during development, it did provide a debatable improvement as neither analysis provided a compromise, a relatively precise analysis at relatively low analysis cost.

[MQ] What are the trade-offs between precision and analysis cost of strictness analysis in a real-world compiler?
A monovariant and polyvariant analysis were implemented in Helium. The former did not result in a suitable analysis as a monomorphic implementation of applicativeness annotations needs to be very conservative for the lack of knowledge on how the functions are applied. The polyvariant analysis did prove to be an analysis capable of being very precise, though this did come at a reduction in performance. An alternative system was proposed which left out the applicativeness annotations, and while the improvement in analysis cost seemed promising, the precision was hugely reduced if the system were to remain sound. However, if the requirement of monomorphic types in a monovariant analysis is changed to monotypes, the analysis matches the precision of the polyvariant analysis at similar analysis cost, which proves to be an almost optimal trade-off. The monotyped analysis also gives a lot more precision for little additional analysis cost compared to the analysis which is currently implemented in Helium. Therefore, the monotyped analysis is the best analysis to be implemented in Helium.



Future work

While this thesis accomplished its goal of improving the strictness analysis in Helium, it is still not perfect. Further improvements can be made in the strictness analysis itself (11.1) as well as general improvements to Helium (11.2).

11.1 Strictness analysis

Future work in strictness analysis includes improving its analysis cost (11.1.1) and precision (11.1.2). These are specifically related to the strictness analyses implemented here.

11.1.1 Analysis cost improvements

The analysis can always be made faster. This could be minor optimizations in the source code, such as using manual strictness annotations. Larger optimizations could be made in the solving algorithm, as the handling of recursive constraints could lead to a major loss of speed. As the analyses in this thesis shared a common framework, completely separating all analyses would also lead to some performance increases, though there would be no reason to keep all analyses around and thus all things unnecessary for the monotyped analysis can be removed.

If the analysis is made faster, we could also look to improve the precision again as this would remain balanced. This would mean the polyvariant analysis could come into play again. The polyvariant analysis has an improved precision compared to the monotyped analysis, but instantiation and simplification proved to be costly operations. A possible improvement to the simplification would be to maintain a set of variables which the simplification is not allowed to solve rather than calculating which variables can be solved, as the latter required an extra iteration over the expression.

11.1.2 Precision improvements

There are also a few cases where the analysis is not able to infer strictness while this should theoretically be the case. Appendix A provides a look at how to handle datatypes, which increases the precision of certain datatypes such as tuples and lists. Creating a general analysis which can handle any datatype could be complicating, given datatypes can have many shapes and supporting them all might be a real

slowdown. A naive implementation to support all datatypes was made, but this slowed down the analysis to such an extent that it took minutes to compile Prelude, which is obviously too slow. However, a few simple and often-used datatypes can benefit from extending the strictness analysis to include them at a relatively low cost in precision. A few of these cases are sketched out in the Appendix.

Another case where the analysis is still imprecise is type classes. Type classes are implemented as data constructors, with a constructor field for every function which is related to this class. Function like addition and multiplication on integers are strict by their definition. However, they are part of the Num typeclass, for which other datatypes can also have an instance. It is possible that a user-defined datatype has a Num instance which does not have strict addition. Under the current analysis, it means that all additions and multiplications are restricted in their strictness signature because it might be possible for the type to be non-strict in any argument, and thus none of the arguments provide to these functions can be made strict. A possible solution would be to replace the overarching operators with their specific instance wherever possible. This would result in any integer addition being able to infer strictness on both arguments, while the theoretical lazy instance of a user-defined datatype is only used when this datatype is used or can be used.

11.2 HELIUM

Improvements which could be made to Helium are the removal of unreachable patters (11.2.1) and the extension of strictness analysis to counting analysis (11.2.2).

11.2.1 PATTERN MATCHES

Pattern matches in Core consist of a variable and a number of alternatives. Almost every pattern match has a wildcard pattern which always matches. However, in many cases the wildcard pattern is untriggerable. Under normal circumstances this would be a slight inconvenience as it needs to generate some extra code, but for analyses it could hurt precision. An example of the problems this can give in strictness analysis is a function which takes an argument and a tuple, and returns the argument.

$$
\begin{array}{l}
f :: a \to (b, c) \to a \\
f\,x\,(y, z) = x \\
f :: \mathit{forall}\ a\ \mathit{forall}\ b\ \mathit{forall}\ c\ a \to (b, c) \to a : \mathit{export}\ f \\
\quad = \mathit{forall}\ a : *.\mathit{forall}\ b : *.\mathit{forall}\ c : *. \\
\quad\quad \lambda x : a \to \lambda u1 : (b, c) \to \\
\quad\quad\quad \textbf{let}\ !\ \mathit{var} : (b, c) = x; \\
\quad\quad\quad \textbf{in}\ \mathit{match}\ \mathit{var}\ \mathit{with}\ \{ \\
\quad\quad\quad\quad (@2)\ \{b\}\ \{c\}\ y\ z \to x; \\
\quad\quad\quad\quad _ \to \mathit{primPatternFailPacked}\ \{a\}\ \texttt{"..."};\ \};
\end{array}
$$

In the Haskell code, it is clear variable x is strict. However, after desugaring to Core, the function provides a default pattern in case the pattern match fails, which is impossible as the first case unpacks the tuple and will thus always be triggered. As the strictness analysis needs something to be strict in all branches to be strict, and the argument x is only strict in the first branch, the argument for the function is not inferred as strict.

This example is easily solvable by adding an extra analysis to remove default patterns when all constructors of a datatype are matched upon. This analysis has been included as part of the implementations in this thesis to increase the precision. Still, this analysis is not optimal and only fixes simple cases like this one, as it relies on the different pattern matches on the same argument to be consecutive. It is beneficial to perform this analysis after let inlining because the expression is more likely to be in this form, but it would also benefit from being executed before let inlining because it could remove let bindings which are no longer used because their pattern has been removed. As a compromise, the analysis was installed between the two let inlining passes, to benefit from the structure of the expression and give a chance to

remove unused bindings. The more obvious way to remove excess patterns is by not placing them in the first place, though it might be difficult to determine if a pattern match is complete during parsing.

An example of a function which still has a untriggerable case even after the added analysis is *xor*, using the following definition in Haskell and its desugared definition in Core:

```
xor :: Bool → Bool → Bool
xor True False = True
xor False True = True
xor _ _ = False
```

```
xor :: Bool → Bool → Bool : export xor
  = λu0 : Bool → λu1 : Bool →
    let nc1 : Bool = let nc2 : Bool = False;
      in let ! m1 : Bool = u0;
        in match m1 with {
          False →
            let ! m2 : Bool = u1;
            in match m2 with {
              True → True;
              _ → nc2; };
          _ → nc2; };
    in let ! m3 : Bool = u0;
      in match m3 with {
        True →
          let ! m4 : Bool = u1;
          in match m4 with {
            False → True;
            _ → nc1; };
        _ → nc1; };
```

The function should be strict in both arguments, as *xor* needs to determine that the first argument is not equal to the second argument. The first argument of *xor* will correctly be inferred as strict, but the analysis as is cannot infer the second argument to be strict. While *u1* is matched on both *True* (*m2*) and *False* (*m4*), these matches are in separate bindings. The strictness analysis cannot infer strictness because *u1* is not relevant in the wildcard match in *m1*. A possible solution would be to inline *nc1* and remove the excess pattern matches, which would lead to the following definition which would be capable of determining *xor* is strict in its second argument:

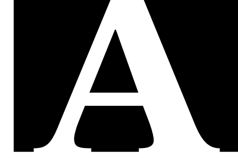
```
xor :: Bool → Bool → Bool : export xor
  = λu0 : Bool → λu1 : Bool →
    let nc1 : Bool = let nc2 : Bool = False;
    in let ! m3 : Bool = u0;
      in match m3 with {
        True →
          let ! m4 : Bool = u1;
          in match m4 with {
            False → True;
            _ → False; };
        _ →
          let ! m5 : Bool = u1;
          in match m5 with {
            True → True;
            _ → False; }; };
```

In this function, *u1* is relevant to both cases of *m3* as it is required for match *m4* and *m5*.

11.2.2 Counting analysis

Counting analysis combines strictness analysis, sharing analysis, absence analysis and uniqueness typing into one analysis. Uniqueness typing has previously been implemented by Van Klei during heap recycling analysis (5.3.1). This analysis is distinct from the other three analysis in being a verifying analysis opposed to an optimizing analysis. Absence and sharing analysis has not been implemented in Helium yet.

Verstoep provided the theoretical groundwork of combining these analyses into one analysis 5.1.4. Since they are intertwined, the structure of strictness analysis can be reused to implement the other analyses as well. One of the major facets which have to be changed are the lattice, which no longer suffices with two possible annotations. For the combined analysis, at least the annotations 0, 1 and ω (many) are necessary to define how often an expression is (guaranteed to be) used. New operators on these annotations also have to be defined, such as addition. The applicativeness annotations will also play an important role in the combined analysis as partial application also influences the other analysis in how they are allowed to use the information when a function is not guaranteed to be applied.



DATATYPES

The analyses described in this thesis did not take strictness on datatypes into account, other than was already provided by annotating the constructor fields. The appendix attempts to extend the strictness analysis to handle tuples (A.1), lists (A.2) and other datatypes (A.3). Finally, we look at the precision improvements of the extension (A.4).

A.1 TUPLES

Tuples are hard-coded into the Core type and expression system, making them suitable for a standalone implementation. For this section, we assume a tuple consists of two values. The analysis is generalizable to tuples with higher arities. The type of the constructor for tuples is $\forall a.\forall b.a \rightarrow b \rightarrow (a, b)$. As the arguments applied to the constructor are stored as applications, there needs to be a triplet of annotations on both function arrows. There is an obvious connection between the annotations on the arrows and the type variables, meaning the annotations can also be placed inside the tuples. As Helium does not allow constructors to be partially applied[1], the applicativeness annotations are obsolete and could be removed, but they are kept for now. This leads to an annotated type of

$$\forall a.\forall b.a \xrightarrow{(\psi_1,\varphi,\psi_2)} b \xrightarrow{(\psi_1',\varphi_1,\psi_2')} (a^{(\psi_1,\varphi,\psi_2)}, b^{(\psi_1',\varphi_1,\psi_2')}).$$

Consider the function *fst*:

$fst :: (a, b) \rightarrow a$
$\quad = \lambda u1 : (a, b) \rightarrow$
$\qquad$ **let** $!\, var : (a, b) = u1;$
$\qquad$ **in** *match var with* $\{$
$\qquad\quad (@2)\ \{a\}\ \{b\}\ x\ y \rightarrow x;\ \};$

The annotated type of this function should reflect that the first element of the tuple is relevant, but the second element is not. The tuple itself is strict as it is used in a pattern match. Therefore the desired type signature would be

$$\forall a.\forall b(a^{(\psi_1,\mathrm{S},\psi_1)}, b^{(\psi_2,\mathrm{L},\psi_2)}) \xrightarrow{(\psi,\mathrm{S},\psi)} a$$

[1]The saturate pass saturates all calls to constructors.

To be able to achieve this, the variables introduced by the pattern match should receive fresh annotations. This can be done by adding three fresh annotations per type instantiation. As *x* occurs in the body of the pattern match, its associated relevance annotation can be set to S, while *y* does not occur and its relevance annotation becomes L. This information will be propagated to var, then to *u1* and finally to the type signature itself. If there had been another case in the pattern match[2], then the join over all annotations is taken as the first element needs to be strict in all cases.

We can now provide information about whether a tuple is strict in its first, second or both values. This information should end up propagated to the place where a tuple is defined. Take the following function:

$f :: Bool$
$= \textbf{let } tup : (Bool, Bool) = \textbf{let } v1 : Bool = True;$
$\textbf{in let } v2 : Bool = False;$
$\textbf{in } (@2)\ \{Bool\}\ \{Bool\}\ v1\ v2;$
$\textbf{in } fst\ tup;$

To properly make use of the information, the binding for *v1* can be made strict while $v2$ should be left untouched. When creating the tuple, it has fresh annotations which correspond to the annotations used for the let bindings. Upon use, those annotations can be instantiated to the annotations provided by the function, which in this case means the annotation for the first element of the tuple, and thus the annotation for *v1*, becomes S and can be transformed to a strict binding. Note that using the tuple multiple times can provide better instantiations. If a function uses both *fst* and *snd* in a relevant context, both bindings can be strict.

Using the information relies on the information flowing back to the bindings which put the values in tuples. This implies that a polyvariant implementation for datatypes is not desired, as the information can never flow back as the type of the tuple is quantified over its annotations. Therefore, tuples (and all other datatypes) are treated as monovariant even if the analysis is polyvariant. Thus, we cannot simplify any annotation related to datatypes, and they can only be solved at function level. Strictness information from the type variables itself can also be reused. If we create a tuple of not and const, take the first element and apply it to an argument, that argument can be made strict as it is applied to a strict function.

A.2 Lists

Lists are often used in programming, thus getting strictness information about them is valuable. Unlike tuples, lists are not hard-coded into the system and are part of the regular datatype syntax. However, strings (which are lists of characters) are hard-coded as they belong to the literals. List has two constructors:

$$[] : \forall a.[a]$$
$$(:) : \forall a.a \rightarrow [a] \rightarrow [a]$$

We want to know whether every element in the list is guaranteed to be used, meaning we can evaluate every value to WHNF when creating the list.

While placing the extra annotations for tuples was straightforward, placing the annotations on lists is slightly more problematic. Placing it on the type variable *a* would be the obvious answer, but this creates a problem in type class instances of lists. The type of fmap is $\forall a.\forall b.\forall f.(a \rightarrow b) \rightarrow fa \rightarrow fb$, where f can be instantiated as []. In this instance, there would be no strictness annotation on a or b, which would now be expected. If we add annotations to *a* and *b*, every datatype needs to have these annotations, something which is not yet supported. The only solution is to place the annotation on [ ] itself. The added of bonus of placing the annotation here is that we do not have to carry around obsolete applicativeness annotations, just the relevance annotation suffices.

[2] A default pattern in case of a pattern match failure occurs in this function, but is removed due to reasons explained in 11.2.1.

Consider the following function:

$f :: [a] \to Bool$
$\quad = \lambda list : [a] \to$
$\qquad \mathbf{let}\ !\ m1 : [a] = list;$
$\qquad \mathbf{in}\ match\ m1\ with\ \{$
$\qquad\quad [\,]\ \{a\} \to True;$
$\qquad\quad ::\ \{a\}\ x\ xs \to$
$\qquad\qquad \mathbf{let}\ ys : Bool = f\ \{a\}\ xs;$
$\qquad\qquad \mathbf{in}\ seq\ \{a\}\ \{Bool\}\ x\ ys;\ \};$

This function is guaranteed to use every element in the list, and returns True. To reflect this information on the annotation, both the case for the empty list and cons have to reflect that all elements are used. The empty list trivially uses all its elements, and we can always place an annotation S there. Therefore, the type of the constructor $[\,]$ is $\forall a.[a]^{S}$. In the non-empty list case, the element at the head of the list is used in the first argument of seq and thus relevant. The tail of the list depends on the strictness of *f* itself. Because *f* is a recursive function, it will introduce a constraint $\varphi \sqsubseteq \varphi$, which can be instantiated to S if there are no other constraints for this variable. Compare this to the following function:

$g :: [a] \to Bool$
$\quad = \lambda list : [a] \to$
$\qquad \mathbf{let}\ !\ m1 : [a] = list;$
$\qquad \mathbf{in}\ match\ m1\ with\ \{$
$\qquad\quad [\,]\ \{a\} \to True;$
$\qquad\quad ::\ \{a\}\ x\ xs \to g\ xs;\ \};$

In this case, the argument at the head of the list is not used and will have an L annotations. The tail of the list will still produce the constraint $\varphi \sqsubseteq \varphi$, which might still becomes S. This implies that the relevance information from cons is the join of the relevance from the head and the tail. The annotated type of the constructor should thus become

$$\forall a.a \xrightarrow{(\psi_1 \sqcup \psi_2, \psi_1, \psi_1)} [a]^{\varphi} \xrightarrow{(\psi_2 \sqcup \varphi, \varphi, \psi_2)} [a]^{(\psi_1 \sqcup \varphi)}$$

Note that the tail can only be made strict if the annotation on the list is strict, as we cannot infer that every element in the list is guaranteed to be used if the list itself is not guaranteed to be used.

The information obtained from the functions once again needs to flow back to the let bindings which form the creation of the datatype. Like tuples, strictness information of the values stored inside lists can also be kept and used. If a list only stores strict functions, then any argument passed to a function in this list can be made strict. However, if one of the functions happens to be lazy, the strictness information of the entire list is reduced to being a list of lazy functions in order to be sound. The use of strictness information of the lists might lead to some surprising results. If we have a list only containing not (which is thus a strict function), take the tail (empty list of strict functions), take the head (a strict function) and apply it to an argument, the argument can be made strict. However, taking the head of an empty list results in an error, thus the argument hypothetically passed to the function is never used, despite the analysis determining it can be made strict. If the argument diverges, it can lead to a different error message if the argument is evaluated before head is called. This is valid according to the definition of strictness though, as this program diverges given a diverging argument, in this case diverges for any arguments as it fails on head.

User-defined datatypes do not benefit from this extension, as it is configured to look for the standard list datatype [] and its constructors [] and (:). For instance, if the user wishes to make a specialized implementation for list of integers, creating a datatype IntList with constructors Nil and Cons Int IntList will not use the analysis. However, if the user defined IntList as a type synonym of [Int], it will use the analysis as all types are unpacked from their synonyms during the analysis. As Strings are literals, their type has to be manually adjusted to $[\mathrm{Char}]^{\varphi}$, where φ is a fresh annotation.

A.3 OTHER DATATYPES

Other constructors have so far been conservatively analyzed to be non-strict in all its arguments. A first improvement can be made by allowing manual annotations on the datatypes. This is possible in the Core syntax, and reflected by an exclamation mark on type of the constructor field made strict. Using this information means we can make any argument which is passed in this specific position strict.

A simple datatype like Maybe can easily be implemented in the similar spirit as lists, except it is not a recursive datatype. This means the information from the argument to Just can be directly placed on the datatype itself. Nothing, like the empty list, always uses the argument as it does not have one. Either has two type variables, and two constructors Left and Right which both operate on a different type variable. This means that instead of one, we have to place two annotations on Either, one for each constructor. If we were to place one annotation, it could only become S if both the argument to Left and Right were guaranteed to be used, which defeats the purpose of the datatype.

As datatypes become more complex, the annotations to accurately reflect its strictness information also become more complex. Therefore, it is essential that if a user wishes to improve the performance of datatypes, it is often better to add the annotations manually than expect the strictness analysis to handle it. The complexity of datatypes can quickly diminish the performance of the analysis, as was the case with an earlier version of the analysis which naively placed an annotation on every type variable applied to a datatype, which slowed down the analysis to intolerable levels of performance for little gain.

A.4 RESULTS

The extension is implemented on the monotyped analysis discussed in section 9.2. The experiment compares this analysis without the extension against this analysis with the extension enabled. It is performed using the same setup as described in section 6.2, and with the commands described in chapter 8 [3]. The only difference is an extension of the test file to contain more function which use datatypes (figure A.1) .

	Time (ms)	**let!** (Prelude)	$\lambda!x$ (Prelude)	**let!** (Test)	$\lambda!x$ (Test)
Without datatypes	7688	611	197	58	12
With datatypes	8615	611	197	67	13

TABLE A.1: *Results for the monotyped analysis with and without the datatypes extension*

The analysis with the datatypes is obviously more precise, and does not miss any transformation which is also covered by the analysis without the extension. However, this added precision is achieved at a reduced analysis cost of a second, which makes it even slower than the polyvariant analysis. The extension worsens the trade-off between precision and analysis cost, as the added precision is not worth the added analysis cost. The annotations on data constructors do not require major changes to the analysis and are not the main cause, so those can be kept. Tuples are separate from any other datatype in the Core type and expression system, and as such can also be handled quite easily, though it does seem to have a negative effect on the analysis cost already. Lists are the major cause of the reduced speed, as it requires two annotations per element (the element itself and the tail of the list), and the analysis needs to check for the list constructors and place these annotations manually. Implementing support for more datatypes will only worsen the trade-off as most of them are not used as often and also need to be handled in special cases. Creating a general-purpose analysis extension to cover all datatypes is probably not going to get a favorable trade-off either.

[3] The implementation can be found at `https://github.com/Helium4Haskell/helium/tree/strictness-analysis`. The option to enable the versions with datatypes is –strictness=7.

```
lall, lnone, lsome :: [Bool] -> Bool
lall [] = True
lall (x : xs) = x `seq` lall xs

lnone [] = True
lnone (x : xs) = lnone xs

lsome [] = True
lsome [x] = x
lsome (x : xs) = lsome xs

l1, l2, l3 :: Bool
l1 = lall [True, False]
l2 = lnone [True, False]
l3 = lsome [True, False]

lapp :: Bool
lapp = (head [id, id]) True

tup1, tup2, tup12, tup0, tupapp :: Bool
tup1 = fst (True, False)
tup2 = snd (True, False)
tup12 = let tup = (True, False) in fst tup `seq` snd tup
tup0 = let tup = (True, False) in if True then fst tup else snd tup
tupapp = (fst (id, const)) True

data SMaybe a = SNothing | SJust !a

maybe1 :: Maybe Bool
maybe1 = Just True

maybe2 :: SMaybe Bool
maybe2 = SJust True
```

FIGURE A.1: *Extensions to Test.hs*

BIBLIOGRAPHY

[1] H.P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*, volume 103. North Holland, revised edition, 1984. http://www.cs.ru.nl/ henk/Personal Webpage.

[2] Tibor Bremer. Implementing counting analysis in UHC. Master's thesis, Utrecht University, Utrecht, Netherlands, January 2018. https://dspace.library.uu.nl/handle/1874/362949.

[3] Joris Burgers. Type error diagnosis for OutsideIn(X) in Helium. Master's thesis, Utrecht University, Utrecht, Netherlands, June 2019. https://dspace.library.uu.nl/handle/1874/382127.

[4] Ivo Gabe de Wolff. Higher ranked region inference for compile-time garbage collection. Master's thesis, Utrecht University, August 2019. https://dspace.library.uu.nl/handle/1874/383633.

[5] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell Compiler. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, page 93–104, New York, NY, USA, 2009. Association for Computing Machinery.

[6] Jurriaan Hage, Stefan Holdermans, and Arie Middelkoop. A generic usage analysis with subeffect qualifiers. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, page 235–246, New York, NY, USA, 2007. Association for Computing Machinery.

[7] Cordelia Hall, Kevin Hammond, Will Partain, Simon Peyton Jones, and Philip Wadler. The Glasgow Haskell Compiler: A retrospective. pages 62–71, 01 1992.

[8] Kevin Hammond, Simon Peyton Jones, Philip Wadler, and Cordelia Hall. Type classes in Haskell. In *ACM Transactions on Programming Languages and Systems, European Symposium on Programming (ESOP'94)*, volume 18, pages 241–256. Springer Verlag LNCS 788, April 1994.

[9] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, page 62–71, New York, NY, USA, 2003. Association for Computing Machinery.

[10] Stefan Holdermans and Jurriaan Hage. Making "stricterness" more relevant. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '10, page 121–130, New York, NY, USA, 2010. Association for Computing Machinery.

[11] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[12] Tom Lokhorst. Strictness optimization in a typed intermediate language. Master's thesis, Utrecht University, August 2010. http://www.cs.uu.nl/education/scripties/scriptie.php?SID=INF/SCR-2009-055.

[13] Reinier Maas. Normalizing the core representation in Helium. Master's thesis, Utrecht University, Utrecht, Netherlands, July 2019. https://dspace.library.uu.nl/handle/1874/383380.

[14] Simon Marlow. *Haskell 2010 Language Report*. https://www.haskell.org/onlinereport/haskell2010/.

[15] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

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

[17] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart, editors. *Real World Haskell*, chapter 25. Profiling and optimization. O'Reilly Media, Inc., November 2008.

[18] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart, editors. *Real World Haskell*, chapter 4. Functional programming. O'Reilly Media, Inc., November 2008.

[19] Augusto Passalaqua Martins. Polyvariant strictness analysis in UHC. Master's thesis, Utrecht University, Utrecht, Netherlands, August 2013. http://dspace.library.uu.nl/handle/1874/282675.

[20] Ilya Sergey, Dimitrios Vytiniotis, Simon L. Peyton Jones, and Joachim Breitner. Modular, higher order cardinality analysis in theory and practice. *Journal of Functional Programming*, 27:e11, 2017.

[21] Fabian Thorand and Jurriaan Hage. Higher-ranked annotation polymorphic dependency analysis. In Peter Müller, editor, *Programming Languages and Systems*, pages 656–683, Cham, 2020. Springer International Publishing.

[22] Mias van Klei. Heap recycling analysis in Helium. Master's thesis, Utrecht University, Utrecht, Netherlands, August 2020. https://dspace.library.uu.nl/handle/1874/398841.

[23] Gerben Verburg. Strictness analysis in UHC. Master's thesis, Utrecht University, Utrecht, Netherlands, 2012. http://dspace.library.uu.nl/handle/1874/255389.

[24] Hidde Verstoep. Counting analyses. Master's thesis, Utrecht University, Utrecht, Netherlands, July 2013. http://dspace.library.uu.nl/handle/1874/282657.

[25] David Walker. *Substructural Type Systems*, page 3–43. The MIT Press, 2002.

[26] Keith Wansbrough and Simon Peyton Jones. Simple usage polymorphism. 2000.