UTRECHT UNIVERSITY

Strictness analysis in UHC

Gerben Verburg

Master of Science Thesis

INF-3019993

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

Daily Supervisor:
dr. Atze Dijkstra
Second Supervisor:
dr. Jurriaan Hage

Abstract

This thesis will investigate how strictness analysis can be done using a type-based approach in a real life compiler. It is inspired by the work of Holdermans et al. [9]. They used relevance typing as an approach to strictness analysis. The work is done in the context of the Utrecht Haskell Compiler (UHC). For the UHC Core language, an implementation is defined for the analysis and the transformation.

In this thesis a monovariant analysis is chosen since there is no support for polyvariance in the code generation. The result is that there is no good support for higher order functions. Recursion is implemented using fixed point iteration. There is no support for datatypes yet.

Contents

Chapter 1

Introduction

Compilers can make use of static program analysis to optimize the code that is generated. For a lazily evaluated language like Haskell, strictness analysis is such an optimizing analysis. For the Utrecht Haskell Compiler(UHC), strictness analysis is not implemented yet. In this thesis, we try to tackle this problem.

1.1 Description of the problem

In static program analysis, the goal is to get information about runtime properties of a program. The derived information about these properties must be valid for all possible executions of the program. This information is necessarily approximative, because of theoretical limits to decidability of these properties

The property that we are focused on in this thesis is strictness. This is explained with the following example:

$$\begin{array}{l} \textbf{let} f = \lambda x\ y \to x \\ \textbf{in} f\ (1+2+3+4+5+6)\ 20 \end{array}$$

This expressions is lazily evaluated, so a thunk is created for each application. When looking at the first argument of f, this will be a collection of nested thunks which will consume a lot of extra memory. But when this expression is evaluated, it can safely be stated that the first argument of f needs to be evaluated too since f simply returns its first argument. With this information, it can be concluded that this first argument can be evaluated eagerly (so no thunks are created for it). The result is that less memory will be consumed and the evaluation will go quicker since no extra work has to be done for the creation and destruction of the thunks.

Strictness analysis is used to derive whether a subexpression is guaranteed to be evaluated when an expression is evaluated. We say that a function is strict in an argument when it is guaranteed that this argument is evaluated whenever the body of the function is evaluated. A function is lazy when this cannot be guaranteed. With this information, the optimization that is explained in the previous paragraph can be applied. So when an argument is supplied to a strict function, this argument can be evaluated eagerly and its value can be passed to the function instead of the thunk. This optimisation is only performed if we can be sure that an argument is needed. Otherwise, we safely choose not to change the call at all.

The setting for this thesis is the Core language of UHC. The difficulty with this is that it is a complete programming language for which an implementation is required. We are going for the low-hanging fruit, so there is good support for simple functions. For more advanced expressions like higher order functions and data types, there is less support.

1.2 Outline

The thesis is organized as follows:

Chapter	Contents
Chapter 2: Background	Background information is given. Relevant research is presented with different approaches.
Chapter 3: Approach	Information about UHC and the approach which is chosen.
Chapter 4: Relevance typing	Relevance typing is defined for the Core language. Inference rules are given for an annotated type system.
Chapter 5: Implementation	Implementation of the relevance typing. A translation is given from the type system to an actual implementation.
Chapter 6: Transformation	The actual transformation is defined using the relevance types.
Chapter 7: Conclusions	Conclusions and future work.

Chapter 2

Background

There are a number of different approaches to strictness analysis. In the following sections the literature for each of the approaches used in the past (abstract interpretation, abstract reduction, projection analysis, totality analysis and relevance typing) is discussed. For each of these approaches there is a general description which will be worked out with a few examples.

2.1 Abstract interpretation

In abstract interpretation, an abstract domain is used instead of the domain of concrete values. This domain contains values concerning the properties that are useful for this analysis. The abstract functions also have to work on the abstract domain instead of on the normal domain.

The first mentioning of strictness analysis was by Mycroft [13], which used abstract interpretation. The specification of strictness which is used is that a function is strict in an argument when the termination behavior is not changed when the argument is evaluated. Formally, for a function f with n arguments, f is strict in the ith argument iff

$$f\, x_1\, \ldots\, x_{i-1} \perp x_{i+1}\, \ldots\, x_n = \perp$$

for all possible values x$_j$ where (j $\neq$ i) and $\perp$ denotes a diverging argument. This makes sense because a function diverges when it gets a diverging argument that it actually uses. An example of how this works out is the function g. g x = x + 1 This function

g is strict in its first argument. This can be concluded because the addition diverges when it gets a diverging argument. So indeed it holds that g $\perp = \perp$.

The abstract domain D$^{\#}$ = $\{0,1\}$ is used. Here 1 stands for all possible values (top) and 0 stands for all non-terminating values (bottom or $\perp$). The ordering in this domain is $0 \sqsubset 1$. These values in the abstract domains are used as booleans so boolean operators can be used in the abstract functions.

For every function in a program, an abstract function is created which works on the abstract domain. These functions are annotated with a $^{\#}$. For example a function

IF (p, x, y) = if p then x else y

is in the abstract domain

$IF^{\#}$ (p, x, y) = $p \wedge (x \vee y)$

From this abstract function it can be conlcuded that p is guaranteed to be used and x or y is used in this function.

2.1.1 Simple functions

The conversion of the addition operator to the abstract domain becomes:

$PLUS^{\#}$ (x, y) = $x \wedge y$

With this definition, you can see that the addition operator is strict in both of its arguments since they are both needed (which is expressed by the $\wedge$). The translation to the abstract domain is done by a function called #, which is defined as follows:

$$\begin{array}{lcl} \#\,[[\ c\ ]] & = & 1 \\ \#\,[[\ x\ ]] & = & x \\ \#\,[[\ f\ E_1 \ldots E_n\ ]] & = & f^{\#}\ \#\,[[E_1]] \ldots \#\,[[\ E_n]] \end{array}$$

To determine whether a function is strict in an argument, bottom should be supplied for that argument and top for all the other arguments. This has to be done for all the arguments. When looking at the function f

f x y z = if (x=0) (y+z) (x-y) ,

the abstract function $f^{\#}$ becomes:

$$\begin{array}{ll} f^{\#}\ x\ y\ z & = \#\,[[\ \text{if } (x=0)\ (y+z)\ (x-y)]] \\ & = \#\,[[(x=0)]] \wedge (\#\,[[(y+z)]] \vee \#\,[[(x-y)]]\ ) \\ & = (\#\,[[x]] \wedge \#\,[[0]]) \wedge ((\#\,[[y]] \wedge \#[[z]]\ ) \vee (\#[[x]] \wedge \#\,[[y]])\ ) \\ & = (x \wedge 1) \wedge ((y \wedge z) \vee (x \wedge y)) \end{array}$$

Now this abstract function can be used to determine the strictness:

$$\begin{array}{l} f^{\#}\ 0\ 1\ 1 = (0 \wedge 1) \wedge ((1 \wedge 1) \vee (0 \wedge 1)) = 0 \wedge (1 \vee 0) = 0 \\ f^{\#}\ 1\ 0\ 1 = (1 \wedge 1) \wedge ((0 \wedge 1) \vee (1 \wedge 0)) = 0 \\ f^{\#}\ 1\ 1\ 0 = (1 \wedge 1) \wedge ((1 \wedge 0) \vee (1 \wedge 1)) = 1 \end{array}$$

So the function f is strict in its first two argument and lazy in its third.

2.1.2 Recursive functions

There is some added complexity when considering recursive functions. For example in the following function:

f x y = if (x=0) y (f (x-1) y)

In the abstract domain this becomes:

$f^{\#}$ x y = (x $\wedge$ 1) $\wedge$ (y $\vee$ ($f^{\#}$ (x $\wedge$ 1) y))

It can be seen from the function that it is strict in both x and y but this cannot be calculated that easily for y. Calculating the strictness of the first argument is straightforward:

$f^{\#}$ 0 1 $\rightarrow$ (0 $\wedge$ 1) $\wedge$ (y $\vee$ ($f^{\#}$ (0 $\wedge$ 1) y) $\rightarrow$ 0

So *f* is strict in its first argument. In general, recursive functions can lead to a non-terminating reduction. For example, calculating the strictness of the second argument of f:

$f^{\#}$ 1 0 $\rightarrow$ (1 $\wedge$ 1) $\wedge$ (0 $\vee$ ($f^{\#}$ (1 $\wedge$ 1) 0) $\rightarrow$ 1 $\wedge$ ($f^{\#}$ (1 $\wedge$ 1) 0) = f# 1 0 $\rightarrow$ ...

It can be approximated to the upper bound (which is 1) to stop the calculation. This is safe, but information is lost. A way of approximating it from below is by fixed point iteration. This is done in the following manner for a function *f* with n arguments:

$$\begin{aligned} f^{\#0}\ x_1 \ldots x_n &= 0 \\ f^{\#m}\ x_1 \ldots x_n &= E\,[[f^{\#} := f^{\#m-1}]] \end{aligned}$$

Here [[x:=y]] means that all occurrences of x are replaced by y. The first approximation is bottom. The next approximation is the normal abstract function, where the recursive appearances are replaced by the previous approximation. This has to be done until a fixed point is reached ($f^{\#u} = f^{\#u+1}$). The resulting abstract function becomes $f^{\#} = f^{\#u}$. The result for the previous example is:

$$\begin{aligned} f^{\#0}\ x\ y &= 0 \\ f^{\#1}\ x\ y &= (x \wedge 1) \wedge (y \vee (f^{\#0}\ (0 \wedge 1)\ y) = (x \wedge 1) \wedge (y \vee 0) = x \wedge y \\ f^{\#2}\ x\ y &= (x \wedge 1) \wedge (y \vee (f^{\#1}\ (0 \wedge 1)\ y) = (x \wedge 1) \wedge (y \vee (f^{\#1}\ x\ y)) \\ &= x \wedge (y \vee (x \wedge y)) = x \wedge y \end{aligned}$$

So $f^{\#}$ x y = x $\wedge$ y. Calculating the strictness for the second argument gives:

f# 1 0 $\rightarrow$ 0.

So f is also strict in its second argument. The real pain is in the test of equality for two functions. For this, it has to be checked whether it gives the same result for each combination of arguments. The complexity is exponential in the number of arguments, so it can be computationally expensive to calculate the fixed point.

2.1.3 Higher order functions

A way to handle higher order functions with abstract interpretation is proposed by Burn[3]. A new domain is created for functions, because the normal top and bottom of the normal abstract domain do not represent the values in the new functional abstract domain. An example of this is the function hof.

hof g a b = (g a b) + (g b b)

The first argument is a function so this is in another domain: D x D $\rightarrow$ D. For this new domain, a new top and bottom have to be defined:

$f_\perp$ :: $D^\#$ x $D^\#$ $\rightarrow$ $D^\#$
$f_\perp$ x y = 0

$f_\top$ :: $D^\#$ x $D^\#$ $\rightarrow$ $D^\#$
$f_\top$ x y =1

Now, the strictness of hof can be determined again by using the top and bottom from the function domain:

$hof^\#$ $f_\perp$ 1 1 = 0
$hof^\#$ $f_\top$ 0 1 = 1
$hof^\#$ $f_\top$ 1 0 = 1

So now it can be determined that hof is only strict in its first argument. This is a safe approximation, but a better result can be calculated. When it is given that g is strict in both of its arguments, it can be inferred that hof is also strict in its second and third argument. So, in this manner a lot of information is thrown away.

2.1.4 Data types (lists)

A way to handle data types is proposed by Wadler[19]. When looking at lists, an infinite domain will be created when an abstract value is created for each concrete value. Instead, Wadler suggests a four-point abstract domain for lists. This consists of:

- $\top_\in$ - any finite list of which no member is $\perp$ (e.g. (cons 1 (cons 2 nil)))
- $\perp_\in$ - any finite list of which some member is $\perp$ (e.g (cons 1 (cons $\perp$ nil)))
- ∞ - any infinite list except $\perp$ (e.g. (cons 1 $\perp$))
- $\perp$ - $\perp$

These are ordered like: $\bot \sqsubset \infty \sqsubset \bot_\in \sqsubset \top_\in$. Now functions using lists should use this domain. There are two constructor alternatives, so a function which takes a list as an argument has two cases:

```
f [] = E
f (h:t) = N h t
```

Here, E and N are expressions where N is a function which takes two arguments and E does not take any arguments. Then the abstract function becomes:

$$
\begin{array}{ll}
f^{\#}\ \bot & = \bot \\
f^{\#}\ \infty & = \#\ [[\ N\ ]]\ 1\ \infty \\
f^{\#}\ \bot_\in & = (\#\ [[\ N\ ]]\ 0\ \top_\in) \cup (\#\ [[\ N\ ]]\ 1\ \bot_\in) \\
f^{\#}\ \top_\in & = \#\ [[\ E\ ]] \cup (\#\ [[\ N\ ]]\ 1\ \top_\in
\end{array}
$$

Additionally, there has to be support for the construction of lists. The mapping to the abstract domain becomes:

$$
\begin{array}{ll}
\#\ [[\ []\ ]] & = \top_\in \\
\#\ [[\ E_1 : E_2\ ]] & = \mathrm{Cons}^{\#}\ \#\ [[E_1]]\ \#\ [[\ E_2]]
\end{array}
$$

The Cons$^{\#}$ is a special function defined like a mapping:

Cons$^{\#}$ x y	x = $\top$	x = $\bot$
y = $\top_\in$	$\top_\in$	$\bot_\in$
y = $\bot_\in$	$\bot_\in$	$\bot_\in$
y = ∞	∞	∞
y = $\bot$	∞	∞

In this way, it can be determined if a function is element strict (strict in constructors and elements, e.g. sum), head strict (only strict in the first constructor and not in tail or elements, e.g. isEmpty) or spine strict (only strict in the constructors and not in the elements, e.g. length).

The complexity increases when you are using more complex element types of a list since the number of values in the abstract domain increases. For example in a list of lists, the domain contains six values. When using more complicated data structures, the abstract domains becomes more complicated to construct and the complexity of the complete analysis may explode.

2.2 Abstract Reduction

Abstract reduction is proposed by Nöcker[14]. This approach also uses an abstract domain like in abstract interpretation. The difference is that it also takes into account arbitrary data structures and pattern matching which is done using infinite sets. This analysis can become an infinite calculation, but this can be prevented by using reduction path analysis in the graph.

When S is the set of concrete values, the abstract power domain is used: PS#. This is done by first taking the powerset of S and then joining every element of the set with the set Bot of all non-terminating and undefined expressions. An important part of this analysis is the abstract rewriting which is used to reduce terms. It is related to the concrete function definition and the applied reduction strategy. There is a pattern matching mechanism, which has to choose which rewriting rule should be used. In this way, an abstract value is matched against a pattern. For this matching there are four alternatives:

- Total match: the pattern is a superset of the value, so the value is contained in the pattern (e.g. $0^\#$ and $0^\#$)
- Partial match: the value is a superset of the pattern. When this is the case another rewrite rule may be applicable (e.g. Top and Cons#)
- Bottom match: the value is bottom and the pattern is a non-variable (e.g. Bot and Cons# x)
- No match: when non of the above alternatives match, it is a no match (e.g. Nil# and Cons# x)

When more than one rule matches, the union of the results is taken.

2.2.1 Simple functions

An example of an simple function is *f*

$$\begin{aligned} f\,0\,y\,z &= y+z \\ f\,x\,y\,z &= x-y \end{aligned}$$

Note that this is the same function as f x y z = if (x=0) (y+z) (x-y), where the used definition is written with pattern matching.

The abstract reduction becomes:

$f^\#$ Bot Top Top $\rightarrow$ Bot
$f^\#$ Top Bot Top $\rightarrow$ (Bot $+^\#$ Top) $\cup$ (Top $-^\#$ Bot) $\rightarrow$ ... $\rightarrow$ Bot $\cup$ Bot = Bot
$f^\#$ Top Top Bot $\rightarrow$ (Top $+^\#$ Bot) $\cup$ (Top $-^\#$ Top) $\rightarrow$ ... $\rightarrow$ Bot $\cup$ Top = Top

It can be concluded that *f* is strict in its first and second argument.

2.2.2 Recursive functions

For recursive functions, a technique called path analysis is used. The main idea is that recursive occurrences which are needed further on in the reduction can be replaced by bottom. Suppose a reduction path for a term *t*: $t \rightarrow_* E\,(t)$ in which $E\,(t)$ is an expression in which *t* occurs. When the reduction of *t* is needed for the reduction of E(t) it can be replaced by bottom, otherwise it must be replaced by top. For example in:

$$\begin{aligned} f\,0\,y &= y \\ f\,x\,y &= f\,(x-1)\,y \end{aligned}$$

The strictness analysis for the second argument becomes:

$$\begin{array}{lcl}
\text{f}^{\#}\ \text{Top Bot} & = & (\text{f}^{\#}\ 0^{\#}\ \text{Bot}) \cup (\text{f}^{\#}\ (\text{Top}\backslash 0^{\#})\ \text{Bot}) \\
 & \rightarrow & \text{Bot} \cup \text{f}^{\#}\ ((\text{Top}\backslash 0^{\#}) -^{\#} 1^{\#})\ \text{Bot} = \text{f}^{\#}\ ((\text{Top}\backslash 0^{\#}) -^{\#} 1^{\#})\ \text{Bot} \\
 & \rightarrow & \text{f}^{\#}\ \text{Top Bot} \\
 & \rightarrow & \text{Bot}
\end{array}$$

So it can be determined that f is strict in both of its arguments.

2.2.3 Data types (lists)

For data types, similar constructs as in abstract interpretation are used (e.g. Topmem, Botmem, Inf and Bot). These values are represented as subsets of the abstract powerset:

$$\begin{array}{lcl}
\text{Bot} & \equiv & \text{set of all non-terminating expressions} \\
\text{Inf} & \equiv & \text{set of all infinite list} \cup \text{Bot} \\
\text{BotMem} & \equiv & \text{set of all lists with at least one non-terminating element} \cup \text{Inf} \\
\text{TopMem} & \equiv & \text{set of all possible lists}
\end{array}$$

The recursive definitions for these values are:

$$\begin{array}{lcl}
\text{Inf} & = & \text{Cons}^{\#}\ \text{Top Inf} \\
\text{BotMem} & = & (\text{Cons}^{\#}\ \text{Top BotMem}) \cup (\text{Cons}^{\#}\ \text{Bot TopMem}) \\
\text{TopMem} & = & \text{Nil}^{\#} \cup (\text{Cons}^{\#}\ \text{Top TopMem})
\end{array}$$

In a sum function on lists this works out like this:

$$\begin{array}{lcl}
\text{sum}^{\#}\ \text{Bot} & \rightarrow & \text{Bot} \\
\text{sum}^{\#}\ \text{Inf} & = & \text{sum}^{\#}\ (\text{Cons}^{\#}\ \text{Top Inf}) \\
 & \rightarrow & \text{Top} +^{\#} (\text{sum}^{\#}\ \text{Inf}) \\
 & \rightarrow & \text{Top} +^{\#} \text{Bot} \\
 & \rightarrow & \text{Bot} \\
\text{sum}^{\#}\ \text{BotMem} & = & \text{sum}^{\#}\ ((\text{Cons}^{\#}\ \text{Top BotMem}) \cup (\text{Cons}^{\#}\ \text{Bot TopMem})) \\
 & \rightarrow & (\text{Top} +^{\#} (\text{sum}^{\#}\ \text{BotMem})) \cup (\text{Bot} +^{\#} (\text{sum}^{\#}\ \text{TopMem})) \\
 & \rightarrow & (\text{Top} +^{\#} (\text{sum}^{\#}\ \text{BotMem})) \cup \text{Bot} \\
 & \rightarrow & \text{Top} +^{\#} (\text{sum}^{\#}\ \text{BotMem}) \\
 & \rightarrow & \text{Top} +^{\#} \text{Bot} \\
 & \rightarrow & \text{Bot} \\
\text{sum}^{\#}\ \text{TopMem} & = & \text{sum}^{\#}\ (\text{Nil}^{\#} \cup (\text{Cons}^{\#}\ \text{Top TopMem})) \\
 & \rightarrow & 0^{\#} \cup (\text{sum}^{\#}\ (\text{Cons}^{\#}\ \text{Top TopMem})) \\
 & \rightarrow & 0^{\#} \cup (\text{Top} +^{\#} (\text{sum}^{\#}\ \text{TopMem})) \\
 & \rightarrow & 0^{\#} \cup (\text{Top} +^{\#} \text{Top}) \\
 & \rightarrow & 0^{\#} \cup \text{Top} \\
 & \rightarrow & \text{Top}
\end{array}$$

It can be concluded that sum is element strict because sum$^{\#}$ only returns Top when TopMem is the input.

2.3 Projection

Projection for strictness analysis was used first by Wadler and Hughes[18]. An improved version of this is published by Davis and Wadler[5] which uses a more general and powerful technique. The concept of the projection is borrowed from domain theory. A projection is an idempotent function on a domain that removes information from its argument (like abstract interpretation), but does not not change its type.

Formally, a continuous function α is a projection when

$$\begin{aligned} \alpha\,\mathrm{u} &\sqsubseteq \mathrm{u} \\ \alpha(\alpha\,\mathrm{u}\,) &= \alpha\,\mathrm{u} \end{aligned}$$

The first rule says that a projection should only remove information. The second rule says that all the information should be removed at once.

The projections form a lattice under the ordering of $\sqsubseteq$. The identity function ID is the greatest element of the lattice and BOT (which is defined as BOT u = $\perp$ for all u) is the least element. An example of such a projection on pairs is:

F (u,v) = (u,$\perp$)
S (u,v) = ($\perp$,v)

A projection can be seen as specification of the minimum degree of definedness of the argument. The unneeded part can be mapped to $\perp$, the rest can be left untouched. So it can be seen that the projection F stands for the first element of the pair and S stands for second element of the pair. An example of how this can be used is with a function f, which only needs the first element of the pair. A call to this function f may be replaced by F $\circ$ f, where the composition with F depicts the context in which f is evaluated. Consider for example the function reverse:

reverse (u,v) = (v,u)

When reverse is in context F it is safe to apply S to the argument, that is

F $\circ$ reverse = F $\circ$ reverse $\circ$ S

If it holds for certain projections α and β that $\alpha \circ$ f = $\alpha \circ$ f $\circ\ \beta$, we write f: $\alpha \Rightarrow \beta$. To define that a certain degree of definedness is needed (for example a value has to be more defined than bottom), a special element has to be added to the domain which is called Abort. This is the new least element in the lattice. α u = Abort means that α needs a value more defined than u. All functions are strict in Abort, so f Abort = Abort for all f. Strictness is defined with the projection STR, which returns Abort when the argument is $\perp$ and acts as the identity function for all the other input values. A function f is strict when

$$f : \mathrm{STR} \Rightarrow \mathrm{STR}$$

Another useful projection is ABS, which stands for absent, which returns $\perp$ when given any value except Abort and acts as the identity function on Abort. When a function f does not use its argument, then f: STR $\Rightarrow$ ABS. The least projection is FAIL. It always returns $\perp$ independent of the argument given.

The order in the lattice is FAIL $\sqsubseteq$ STR, FAIL $\sqsubseteq$ ABS, ABS $\sqsubseteq$ ID and STR $\sqsubseteq$ ID.

To handle data types, new projections are added for each constructor. For example for lists, the NIL projection and the CONS projection generator are used. NIL only accepts an empty list, while CONS takes two projection arguments next to the normal argument and only accepts a non-empty list (so created with the (:) constructor). It also applies the two argument projections to the two values of the constructor. For example the projection CONS STR ID specifies that it is strict in the head of the list and nothing is know about the strictness of the tail.

2.3.1 Projection transformers

For a function f with n arguments, a transformer f^i is defined for every i from 1 to n. This transformer takes a projection which is applied to the result of f and transforms it into a projection which may safely be applied to the ith argument. According to the safety requirement, it must hold that (when $\beta_i = f^i\ \alpha$):

$$\alpha(f\, u_1...u_i...u_n) \sqsubseteq f\, u_1...(\beta_i\ u_i)...u_n$$

When the safety requirement holds for $f^1...f^n$, it holds that:

$$\alpha(f\, u_1...u_i...u_n) \sqsubseteq f\ (\beta_1\ u_1)...(\beta_n\ u_n)$$

for all $u_1,...,u_n$, where $\beta_i = f^i \alpha$ for each i from 1 to n.
Like for functions, for each expression e and each variable x, a transformer e^x is defined that takes a projection which is applied to e and transforms it into a projection which may safely be applied to each instance of x in e. So the safety requirement must hold again (when $\beta = e^x \alpha$):

$$\alpha\ e \sqsubseteq e\ [(\beta\ x)/x]$$

for all values of the variables in e.
The definition of these transformers becomes (when α is strict and $\alpha \neq$ FAIL):

$$\begin{array}{ll}
x^x\ \alpha & = \alpha \\
y^x\ \alpha & = \text{ABS} \\
(f\ e_1...e_n)^x\ \alpha & = e_1^x\ (f^1\ \alpha)\ \&\ \cdots\ \&\ e_n^x\ (f^n\ \alpha) \\
(\text{case } e_0 \text{ of } [] \Rightarrow e1 \mid y : ys \Rightarrow e_2)^x\ \alpha & = (e_0^x\ \text{NIL}\ \&\ e_1^x\ \alpha) \sqcup (e_0^x\ (\text{CONS}\ (e_2^y\ \alpha)\ (e_2^{ys}\ \alpha))\ \&\ e_2^x\ \alpha)
\end{array}$$

More generally, when x does not appear in e, the ABS projection can be taken. Otherwise the projection α is used. The case expression is handled in more detail in Subsection 2.3.3.

2.3.2 Simple functions

A simple example of how a projection transformers for a function works is the K combinator. The definition of this function is

```
K x y = x
```

When doing the analysis, the result becomes:

$K^1\ \alpha = x^x\ \alpha = \alpha$
$K^2\ \alpha = x^y\ \alpha = \text{ABS}$

where K^n stands for calculating the projection of the n^{th} argument. In other words, evaluating K in context α causes the first argument to be evaluated in context α and the second argument is ignored.

2.3.3 Data types (Lists)

As stated previously, the rule for the case statement is:

$$\begin{aligned}&(\text{case } e_0 \text{ of } [] => e1 \mid y : ys => e_2)^x\ \alpha \\ &\quad = (e_0^x\ \text{NIL}\ \&\ e_1^x\ \alpha) \sqcup (e_0^x\ (\text{CONS}\ (e_2^y\ \alpha)\ (e_2^{ys}\ \alpha))\ \&\ e_2^x\ \alpha)\end{aligned}$$

So in words, if the case expression is evaluated strictly, e_0 must evaluate to the nil or cons constructor. When e_0 evaluates to the nil constructor, x is evaluated under the NIL projection in e_0 and under α in e_1. If e_0 evaluates to the cons constructor, then the head is evaluated as much evaluated as y is in e_2 under α. The tail of the list is as much evaluated as ys is in e_2 under α. In e_2, x is evaluated under α.

An example is given with the head function:

$$\begin{aligned}&\mathit{head}\ \mathit{xs} = \textbf{case}\ \mathit{xs}\ \textbf{of} \\ &\qquad\qquad y : ys \rightarrow y\end{aligned}$$

The result of the projection analysis becomes:

$$\begin{aligned}\text{head}\ \alpha\ &= (\text{xs}^{xs}\ (\text{CONS}\ (y^y\alpha)\ (y^{ys}\ \alpha))\ \&\ y^{xs}\ \alpha) \\ &= (\text{CONS}\ \alpha\ \text{ABS})\end{aligned}$$

As expected, it is determined that the head function is strict in the head of the argument list and that the tail of the argument list is not used.

2.3.4 Recursive functions

To handle recursive functions, fixed point iteration is needed. This is done similar to abstract interpretation. The first iteration is represented by FAIL, and in the next iterations the recursive positions is filled with the answer of the previous iteration. Some useful projections on list are FIN and INF and are defined as:

$\text{FIN}\ \alpha = \text{NIL} \sqcup \text{CONS}\ \alpha\ (\text{FIN}\ \alpha)$
$\text{INF}\ \alpha = \text{NIL} \sqcup \text{CONS}\ \alpha\ (\text{ABS} \sqcup \text{INF}\ \alpha)$

These projections can be read as finite and infinite and can be used in the next example. The function length is used here to show how such a fixed point iteration is used:

$$\begin{array}{ll} \textit{length}\ \textit{xs} = \textbf{case}\ \textit{xs}\ \textbf{of} & \\ \quad [\,] & \Rightarrow 0 \\ \quad (y:ys) & \Rightarrow 1 + \textit{length}\ ys \end{array}$$

The definitions to be used for the fixed point iteration are:

$\text{length}^0\ \alpha \quad = \text{FAIL}$
$\text{length}^{i+1}\ \alpha \quad = \text{NIL} \sqcup \text{CONS ABS } (\text{length}^i\ \alpha)$

The fixed point iteration becomes:

length^0 STR = FAIL
length^1 STR = NIL $\sqcup$ CONS ABS FAIL
= FIN FAIL
length^2 STR = NIL $\sqcup$ CONS ABS (FIN FAIL)
= FIN ABS
length^3 STR = NIL $\sqcup$ CONS ABS (FIN ABS)
= FIN ABS

As expected, it is determined that length needs a finite list, but it ignores the elements of the list.

2.4 Totality analysis

Totality analysis is a little different from the usual approach to strictness analysis. With totality analysis, it is the purpose to detect values which are guaranteed to terminate, so an expression can be evaluated before supplying it as an argument to a function. In this way, the termination behavior is not changed. This differs from standard strictness analysis in the sense that the purpose of the standard analysis is to detect if an argument is used (evaluated to weak head normal form) by the function instead of detecting if the input is terminating. Totality analysis is often defined by non-standard type inferencing. With this technique, extra information is added to the type. In this case, this is the information about the totality of a function.

The first to be using non-standard type inferencing for strictness analysis are Kuo and Mishra[11]. They use an inference algorithm based on the approach of Hindley-Milner. The type contains the function structure in the form of arrows and a set of constraints about the type variables. The lattice that is used for the values for the type variables contains two values: ϕ for looping terms and $\Box$ for all possible values such that $\phi \subseteq \Box$. The form of a type is (C,τ), where C is the set of constraints and τ is the actual type. The resulting type for an expression can be instantiated to a concrete type. This is called an interpretation, where the constraints restrict the possible valuations of a type. For example the function twice,

twice = λ f x. f (f x)

obtains the following type

$(\{\alpha \subseteq \beta\}, (\beta \to \alpha) \to \beta \to \alpha)$

Two instantiations of the type are possible: $(\Box \to \phi) \to \Box \to \phi$ and $(\phi \to \phi) \to \phi \to \phi$. From the first instantiation, it can be concluded that twice is strict in f, since when f has a diverging result, twice also has a diverging result. From the second instantiation, it can be concluded that twice f is strict when f is strict. This is valid because when f has a strict type $(\phi \to \phi)$, the resulting type will also be strict $(\phi \to \phi)$. A set of constraints is said to be consistent when there exists a correct interpretation. In this way, the strictness analysis is split up into two parts: the inferencing and the consistency checking. Nothing is said about the handling of non-flat domains (data types).

Another instance of totality analysis is presented by Jensen[10]. His goal was to infer strictness properties in a polymorphic setting. For the representation of the strictness properties he used f and t, where the former is used for the undefined terms and the latter is used for all possible terms.

Conjunctions in the types are used to have more flexibility in the definition of the strictness types. For example an addition which is strict in both of its arguments has the type $\mathrm{f} \to \mathrm{t} \to \mathrm{f}$ to show that is strict in its first argument and $\mathrm{t} \to \mathrm{f} \to \mathrm{f}$ to show that it is strict in its second argument. This can be combined to $\mathrm{f} \to \mathrm{t} \to \mathrm{f} \wedge \mathrm{t} \to \mathrm{f} \to \mathrm{f}$ to show both strictness properties at the same time.

In addition, conditional strictness properties are used. This is useful when determining the strictness properties of a conditional construct such as an if-then-else. It is represented with a question mark. The term $\varphi?\alpha$ returns φ when $\alpha = \mathrm{t}$ and f when $\alpha = \mathrm{f}$. An example of how this can be used is in

λ x. if x then e_1 else e_2

which gets the type

$(\alpha \to (\varphi \mathbin{?} \alpha))$

This says that the result is undefined if the value of x is undefined, and φ describes the value of the complete if-then-else expression.

A complete inferencing algorithm is specified using an extended version of algorithm T of Damas. The difference with algorithm W is that it deduces a property of a term and makes assumptions on its free variables where algorithm W takes a set of properties of free variables as an input. A set of constraints is collected which restricts the valuation of a type. The handling of data types was left as future work.

A combination of strictness analysis and totality analysis is presented by Solberg[16]. To use a combination of these two, an extra value is added to lattice which is used for values which are guaranteed not to be bottom. So the possibilities for the types now become:

$t^{\mathbf{b}}$: the value has type t and is definitely bottom
$t^{\mathbf{n}}$: the value has type t and is definitely not bottom
$t^{\top}$: the value has type t and it can be any value

Also, conjunctions on the type level are used. Consider for example the function twice: λ f x. f (f x). The type becomes ($\text{Int}^{\mathbf{n}} \to \text{Int}^{\mathbf{b}}$) $\to \text{Int}^{\top} \to \text{Int}^{\mathbf{b}}$. Data types remained as future work, but they proposed a similar way as done in abstract interpretation for lists (as can be seen in Subsection 2.1.4).

Another approach working with constraints is presented by Glynn et al.[8]. It has support for higher order functions, let style polymorphism and data types. The type of a function contains a boolean constraint which can use the standard boolean functions (AND, OR and implication). For example, for an if-then-else construct,

f x y z b = if b then x + y else z

the boolean constraint becomes

b $\wedge$ ((x$\wedge$ y) $\vee$ z)

These types are represented as a combination of a type and a constraint. For the previous function this becomes:

f:$(b \wedge ((x \wedge y) \vee z) \leftrightarrow r \Rightarrow x \mapsto y \mapsto z \mapsto b \mapsto r$

Here $\to$ is used for boolean implication, $\mapsto$ is used for function types and δ ranges over annotations. This can be easily extended to higher order functions by matching the arguments and joining the sets of constraints. For example the functions twice (λ f x. f (f x)) and id (λ x. x) have the following types:

twice	: $\delta_3 \to \delta_1 \wedge \delta_2 \to \delta_1 \wedge \delta_2 \to \delta_4 \Rightarrow (\delta_1 \mapsto \delta_2) \mapsto \delta_3 \mapsto \delta_4$
id	: $\delta_1 \to \delta_2 \Rightarrow \delta_1 \mapsto \delta_2$
twice id	: $(\delta_1 \to \delta_2 \wedge \delta_3 \to \delta_1 \wedge \delta_2 \to \delta_1 \wedge \delta_2 \to \delta_4 \Rightarrow \delta_3 \mapsto \delta_4) = (\delta_3 \to \delta_4 \Rightarrow \delta_3 \mapsto \delta_4)$

Also, a way of handling data types is presented. They use an annotation on the type name, an annotation on each type variable and an annotation for each constant type in the constructors. This is easily understood with an example. The data type Maybe

$$\textbf{data}\ \mathit{Maybe\ a = Just\ a}\ |\ \mathit{Error\ Int}$$

translates into the following strictness data type

$$\mathit{Maybe}^{\delta_1}\ \beta\ \delta_2 = \mathit{Just}\ \beta\ |\ \mathit{Error}\ \delta_2$$

Only non-recursive data types are supported. Recursive occurrences are simply discarded. For example for the list data type, the type becomes: $[\beta]^{\delta}$ where δ describes the strictness of the topmost constructor and β describes the strictness of the first element of the list. There is no annotation for the tail because this is a recursive occurence.

2.5 Relevance typing

Relevance typing is also an example of non-standard type inferencing. This is more like strictness analysis since its goal is to determine if a parameter of a function is relevant to the body of a function (i.e. guaranteed to be used).

A way of using relevance typing in strictness analysis is presented by Wright[21]. The definition of strictness which is used here is the same as the definition of head-needed. An expression is said to be head-needed when it is needed during beta reduction.

For the type inferencing, the arrows for the functional types are changed such that different arrows are used in a type instead of one single sort. When a function is strict in an argument, $\Rightarrow$ is used. When it is not strict, $\not\rightarrow$ is used. For example for the identity function λ x .x, the type is $\alpha \Rightarrow \alpha$ since it is strict in its argument. The type of the K combinator λ x y. x, is $\alpha \Rightarrow \beta \not\rightarrow \alpha$ since it strict in its first argument and lazy in its second.

Variables on arrows are introduced for higher order functions. For example the function λ f x . f x becomes $(\alpha \rightarrow_1 \beta) \Rightarrow \beta$. Here the $\rightarrow_n$ describes a variable for an arrow. This is like a normal variable and can be instantiated to $\not\rightarrow$ and $\Rightarrow$.

Boolean algebra can be used on the type level, for example conjunction. Here $\Rightarrow$ has the role of 1 and $\not\rightarrow$ has the role of 0. Conjunctions on the type level are introduced for example in λ x. f (g x) where g : $\alpha \rightarrow_1 \beta$ and f : $\beta \rightarrow_2 \gamma$. The type becomes $\alpha\ (\rightarrow_1 \wedge \rightarrow_2)\ \gamma$

Additionally, constraints are collected. For example in λ x y. f (g x) (g y) where (f : $\gamma \rightarrow_1 \gamma \rightarrow_2 \beta$) and (g : $\alpha \rightarrow_3 \gamma$) The resulting type is $\alpha'\ (\rightarrow_1 \wedge \rightarrow_3')\ \alpha''\ (\rightarrow_2 \wedge \rightarrow_3'')\ \beta$. In addition, a set of constraints is needed that is generated for the applications. The constraints are:

$$\alpha \rightarrow_3 \gamma \leqslant \alpha' \rightarrow_3' \gamma' \text{ and}$$
$$\alpha \rightarrow_3 \gamma \leqslant \alpha'' \rightarrow_3'' \gamma''$$

This states that any head-neadedness properties from the function g (which uses $\rightarrow_3$), must be propagated to the applications of g which are g x ($\rightarrow_3'$) and g y ($\rightarrow_3''$).

Another approach is taken by Amtoft[1]. A distinction is made between call-by-value evaluation and call-by-name evaluation. The former is transformed in the latter by creating thunks to simulate lazy evaluation. For creating these thunks, the translation T is defined:

- An abstraction λx.e translates into λx.T (e)
- An application e_1 e_2 translates into T(e_1) (λx.T(e_2)) (where x is a fresh variable), so the computation is suspended (”thunkified”)
- A variable x translates into (x d) (where d is a dummy argument), so x is evaluated (”dethunkified”)

The idea is, that by finding that functions are strict, less ”thunkification” and ”dethunkification” is needed. The strictness types used here are $\rightarrow_0$ for strict functions and $\rightarrow_1$

for general functions (for which it is not known if the function is strict). For example the function

$$rec\, f \lambda x\; y\; z \rightarrow \textbf{if}\; (z = 0)\; (x + y)\; (f\; y\; x\; (z - 1))$$

has strictness type

$$\text{Int} \rightarrow_0 \text{Int} \rightarrow_0 \text{Int} \rightarrow_0 \text{Int}$$

since it is strict in all of its three arguments. An inference system is presented for the lambda calculus with constants. In addition, an algorithm for the thunkification is given where the strictness information is used to reduce the thunkification.

Relevance typing is also used by Holdermans and Hage[9]. This is recent work on strictness analysis and it also handles strictness annotations like *seq* in Haskell. They present an inference system and a transformation in one. The strictness annotation they use is a strict application (\$!) which can be used instead of normal (lazy) application. For the strictness information in the type, annotations on the arrows are used ($\xrightarrow{\varphi}$). These are variables which take on the value S for strict or L for lazy. An example is in the function $\lambda x\; y \rightarrow x$ in the setting of an application:

$$(\lambda x\; y \rightarrow y)\; \mathit{true}\; \mathit{false}$$

The type of the function is

$$(\text{bool} \xrightarrow{S} \text{bool} \xrightarrow{L} \text{bool})^S$$

Because this function is strict in its first argument, the program can be transformed into

$$(\lambda x\; y \rightarrow x)\; \$!\; \mathit{true}\; \mathit{false}$$

When including user defined strictness annotations into the language, a conservative approach where all the information about strict applications is ignored results in poor precision. An example is $\lambda x \rightarrow (\lambda y \rightarrow (\lambda z \rightarrow y))\; \mathit{true}\; \$!\, x$. It cannot be inferred that z is relevant because it is not used in the body of the function, but it is evaluated because of the strict application.

A more ambitious attempt where the information from user defined strictness annotations is used, can change the termination behavior. For example $\lambda x \rightarrow (\lambda y \rightarrow 0)\; \$!\; (\lambda z \rightarrow x)$. This evaluates to 0 independent of the argument. This approach infers that the function is strict so when applied to an argument, this can be done strictly. Especially

$$\lambda x \rightarrow (\lambda y \rightarrow 0)\; \$!\; (\lambda z \rightarrow x)\; \$!\; \bot$$

diverges and the termination behavior is changed.

Another approach has to be taken, which tracks information why an abstraction is relevant. This can be achieved by adding applicativity analysis which discovers which function will be applied. As a result, the former lambda abstraction is seen to be strict and the latter is seen to be lazy.

Chapter 3

Approach

After providing some information about UHC and the language that is used in the analysis, we discuss the approach that is taken for the analysis, the implementation that is used and the transformation.

3.1 Design of UHC

We implement our analysis in the Utrecht Haskell Compiler (UHC). This compiler is an implementation of Haskell from the Utrecht University, where the target is to keep the compiler implementation modular. To do this, some different intermediate representation (or languages) are used in the compiler. The actual pipeline of these representations in UHC is shown in Fig. 3.1.

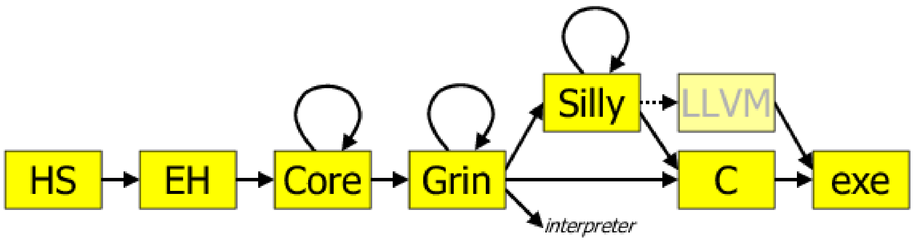

Figure 3.1: UHC pipeline

The parts of the pipeline that are relevant for this thesis are:

Core Core is a representation of an untyped λ-calculus.

Grin Graph Reduction Intermediate Notation is a representation proposed by Boquist[2] in which local definitions have been made sequential and the need for evaluation has been made explicit.

Between these two languages, the conversion is made from the lazy evaluated functional setting of the Core language to the eager evaluated setting of GRIN. The analysis is done at the Core level. In this way the information from the strictness analysis is used to control the evaluation in GRIN.

At the Core language, a number of transformations has already been performed. More information about these transformations is provided by Dijkstra et al. [6]. We have chosen to perform the analysis at the end of the transformation pipeline, because in this situation the other transformations will not change the code that is generated. Also, a number of assumptions can be made about the input: it is lambda lifted and it is in administrative normal form (ANF). That the code is lambda lifted is an invariant which also has to hold at the end of the transformation. That the code is in ANF means that all arguments to a function must be trivial. This notion was introduced by Flanagan et al. [7].

3.2 Core language of UHC

The language which is used for the analysis is the Core language of UHC. It is a representation of the untyped lambda calculus with the ability to represent data types and calls to the Foreign Function Interface.

We assume the following syntactic categories, comprising the set of variables, the set of characters, the set of integers and the set of data type constructors:

$$\begin{array}{rcll} z & \in & \mathbb{Z} & \text{integers} \\ c & \in & \textbf{Unicode} & \text{characters} \\ x & \in & \mathit{Var} & \text{variables} \\ \mathit{tag} & \in & \mathit{Tag} & \text{datatype constructors} \end{array}$$

The syntax of the language is defined by the following abstract syntax:

$$\begin{array}{rcl} e & ::= & z \\ & | & c \\ & | & x \\ & | & \lambda x \to e \\ & | & e_1\ e_2 \\ & | & \textbf{let}\ \mathit{bind}\ \textbf{in}\ e \\ & | & \textbf{letrec}\ [\mathit{bind}]\ \textbf{in}\ e \\ & | & \textbf{let!}\ \mathit{bind}\ \textbf{in}\ e \\ & | & \textbf{case}\ e\ \textbf{of}\ [\mathit{alt}] \\ & | & \mathit{tup}\ \mathit{tag} \\ & | & \mathit{FFI}\ x \\ \mathit{bind} & ::= & x = e \\ \mathit{alt} & ::= & \mathit{pat} \to e \\ \mathit{pat} & ::= & x \\ & | & \mathit{tup}\ \mathit{tag}\ [x] \\ & | & z \\ & | & c \end{array}$$

A Core module is a representation of a normal Haskell module. A Core module consists of nothing more than a name and one expression. This is a let binding where the body has a call to the *main* function when it is the main module and an empty expression (the integer 0) when it is a library module.

The used expressions are normal variables, constants, function abstractions (lambdas) and applications. Data types aren't available in λ-calculus, so a data type constructor is encoded by a *tup*. This must be used as a part of an application to create a data type alternative or a tuple. The *tag* is used to determine the arity, the data type and the constructor. In a case expression, the *tup* is also used in the patterns, where the destruction of data types take place. Another special construction is for the foreign function interface, which is the *FFI* alternative. Its argument identifies the function to be called. A let binding has three alternatives: a recursive binding (**letrec**), a strict binding (**let**!) and a plain binding (**let**). The **let**! is used for evaluation in the Core language, and *seq* is defined in terms of it. The language has a standard lazy semantics, with exception of the **let**!. For the **let**! holds that when the body of the strict let expression must be evaluated, the right hand side of the binding will also be evaluated.

This is a simplified version of the actual abstract syntax that is used in UHC. The actual implementation is made in the UUAGC system (Utrecht University Attribute Grammar Compiler) [17].

3.3 Analysis

The basis for the strictness analysis in this thesis is the paper written by Holdermans and Hage[9]. They use a type based approach to strictness analysis which is based on relevance typing.

A type system will be defined for the UHC Core language. Type rules that are given are based on the Hindley-Milner type system[4]. Annotations on the types are used to capture the relevance information. A new notation is introduced which is more refined. It keeps track of the saturation of an expression instead of the applicativeness. For the saturation, two counters are added to the types in the type environment and to the typing judgements. These counters are used to keep track of the number of arguments that are needed and the number of arguments that are supplied. With this notation, it is easier to keep track of the relevance information in partial applications.

There are some differences between the language that is used in [9] and the UHC Core language. These differences are

- UHC Core contains both plain and recursive let bindings
- Strict let bindings are used instead of strict applications
- Case expressions (which has branching) have been added
- Tup expressions are used to construct data types
- Expressions for calls to the foreign function interface have been added

The type system should therefore be extended to cover these new language constructs.

Also, a choice can be made between polyvariant analysis and monovariant analysis. In this thesis we have chosen to do a monovariant analysis since this will not need an extension in the pipeline in UHC. The influence of polyvariance on the code generation is described in more detail in the master thesis of Lokhorst[12].

Further details of the analysis can be found in Chapter 4.

3.4 Implementation

For the implementation, two options are available. These are the syntax-driven approach and the constraint-based approach.

3.4.1 Constraint-based

In a constraint-based implementation, constraints are used to express the properties of an expression. For relevance typing, these constraints would look like $\varphi_1 \sqsubseteq \varphi_2$. The solving of these constraints can be done using a worklist algorithm. Since a monovariant approach is taken, constraints can be solved globally. When a full language is used, some scoping mechanism is needed. This complicates things, because now for every (variable, scope) combination a relevance value has to be stored. This is needed because the relevance value of a variable can be different in different scopes. For example in a case expression, a variable can have different relevance values in the different case arms. When the constraint language and the solving have to be extended to accommodate all this, things get rather complicated. Also, all results of the solving have to distributed over the abstract syntax tree (ast) again which makes it more complex. The advantage of using constraints is that it is easier to get better precision for recursion and let bindings can be handled in a more natural way.

3.4.2 Syntax-driven

In a syntax-driven approach, the expression influences the results directly. This leads to a simpler implementation since no separate solver is needed. Also, the results are in place directly so no distribution is needed. This implementation will also more closely resemble the type rules. The advantage of using a syntax-driven approach is that no scoping mechanism is needed since this is evident from the abstract syntax tree. Annotated type environments are used to encode the relevance values of variables. For example in the case expression, an annotated type environment is calculated for every case arm. A disadvantage of this approach is that good support for recursion is harder to implement.

After considering all advantages and disadvantages of both approaches, we have chosen to take the syntax-driven approach.

Further details of the implementation can be found in Chapter 5.

3.5 Transformation

The target of the transformation is to introduce extra **let**!'s and to put them as high as possible in the tree. For example in the following expression:

$$\begin{array}{l}\textbf{let}\, ten = 5 + 5\ \textbf{in} \\ \textbf{let}\, f = \lambda x \to x + 5 \\ \textbf{in}\, f\ ten\end{array}$$

It can be determined that f is strict in its first argument, so after inspecting body of the let binding it can be concluded that *ten* is also strict. Now *ten* can be evaluated at the place where this is concluded (i.e. in the body of the let binding), so the body of the let binding becomes f $(\textbf{let}\,!\,ten' = ten\ \textbf{in}\ ten')$. This is not the optimal point since this information can be propagated to the definition in the first let binding, where it can be evaluated. The resulting expression is:

$$\begin{array}{l}\textbf{let}!\ ten = 5+5\ \textbf{in}\\ \textbf{let}\ f = \lambda x \rightarrow x+5\\ \textbf{in}\, f\ ten\end{array}$$

To get these results, it needs to be calculated what variables must be evaluated and what relevance information can be propagated further up the tree.

Since a monovariant approach is taken, something extra must be done for higher order functions. For each function, a wrapper must be generated which evaluates the arguments in which the function is strict. This is like the worker/wrapper transformation which was first used by Peyton Jones and Launchbury [15]. The wrapper is needed when a function is passed as a higher-order argument to another function. Consider the following expression:

$$\begin{array}{l}\textbf{let}\ addone = \lambda x \rightarrow x+1\ \textbf{in}\\ \textbf{let}\ app = \lambda f\ x \rightarrow f\ x\\ \textbf{in}\ app\ addone\ 5\end{array}$$

The *addone* function is used as an argument to *app*. Because a monovariant approach is taken, the type cannot contain variables so a safe estimation has to be made for the functional argument of *app*. This safe estimation is a lazy function. The consequence is that a lazy function has to be given. In the case of *addone*, a wrapper has to be generated to create a lazy function. The resulting expression is:

$$\begin{array}{l}\textbf{let}\ addone = \lambda x \rightarrow x+1\ \textbf{in}\\ \textbf{let}\ addone_{wrap} = \lambda x \rightarrow \textbf{let}\,!\,x' = x\ \textbf{in}\ addone\ x'\ \textbf{in}\\ \textbf{let}\ app = \lambda f\ x \rightarrow f\ x\\ \textbf{in}\ app\ addone_{wrap}\ 5\end{array}$$

Further details of the transformation can be found in Chapter 6.

Chapter 4

Relevance typing

This chapter introduces the inference system for the UHC Core language. First the syntax of the types and the typing rules will be explained. Subsequently, the inference rules will be presented. We illustrate the rules with examples.

4.1 Annotated types

The target of relevance typing is to find relevant abstractions. An abstraction $\lambda x \to e$ is said to be relevant when the argument x is relevant to the body e. To keep track of this relevance information of abstractions, annotated types ($\widehat{\tau}$) are used. These types are defined by:

$$\widehat{\tau} \quad ::= \quad () \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2.$$

So a type will be a unit type, an empty tuple, or a functional type with an annotation φ. This annotation is defined by:

$$\varphi \quad ::= \quad \mathsf{S} \mid \mathsf{L}.$$

Here S is used to annotate relevant abstractions, and L is used when this is not known to be the case. Relevance implies strictness, because when it is known that a variable x is relevant to an expression e, then e is strict in x. The order in the lattice of the strictness values is $\mathsf{S} \sqsubset \mathsf{L}$. The joins and meets for the values in this lattice are:

$$\begin{array}{lcl} \mathsf{S} \sqcup \varphi = \varphi & \text{and} & \mathsf{S} \sqcap \varphi = \mathsf{S} \\ \mathsf{L} \sqcup \varphi = \mathsf{L} & & \mathsf{L} \sqcap \varphi = \varphi. \end{array}$$

Note that in the case of the Core language, no concrete information about the types is needed since it has already been type checked. As a result, the annotated types contain the shape of the types and the relevance information, but not the actual types like *Int* or *Char*.

4.2 Type rules

When designing the type rules, some problems were encountered with partial applications. To illustrate, consider

$$\begin{array}{l}\textbf{let}!\,f = (\lambda x\, y \rightarrow x + 10)\ (5+5)\\ \textbf{in}\ 0\end{array}$$

Here the evaluation of f by a strict let binding causes that the first argument that is supplied to the lambda will be deemed relevant. But the expression $(5+5)$ cannot be concluded to be relevant before the function f gets another argument in a strict context. This is because the right hand side of the binding is a partial application. To keep better track of how many arguments are already given and how many are needed, two counters are introduced: sat_l and sat_r. Here sat_l keeps track of how many arguments are needed until the function is saturated and sat_r counts the number of arguments that are passed to an expression in a strict setting.

As a result, the typing rules will feature judgements of the form

$$\widehat{\Gamma} \vdash e :: \widehat{\tau}^{(\varphi, sat_l, sat_r)},$$

expressing that from the annotated type environment $\widehat{\Gamma}$, it can be derived that the expression e has the type $\widehat{\tau}$ and the relevance annotation φ. The expression e needs at least sat_l arguments before it is saturated, and it is guaranteed to get sat_r arguments in a relevant context.

The annotated type environments map variables x to $(\widehat{\tau}, \varphi, sat_l, sat_r)$ consisting of an annotated type $\widehat{\tau}$, a relevance annotation φ, a sat_r counter and a sat_l counter. We write $[\,]$ for the empty environment, $[x \mapsto (\widehat{\tau}, \varphi, sat_l, sat_r)]$ for the singleton environment that maps x to $(\widehat{\tau}, \varphi, sat_l, sat_r)$, and $\widehat{\Gamma}_1\ [x \mapsto (\widehat{\tau}, \varphi, sat_l, sat_r)]$ for the environment that is obtained by extending $\widehat{\Gamma}_1$ with a binding from x to $(\widehat{\tau}, \varphi, sat_l, sat_r)$. For merging two $\widehat{\Gamma}$'s with a disjoint domain, $\widehat{\Gamma}_1 +\!\!+ \widehat{\Gamma}_2$ is used.

Our inference system makes use of the substructural type discipline (see Walker[20]), which can be seen from the careful treatment of type environments throughout the typing rules. The invariant that is maintained is that any S-annotated variable in an annotated type environment must appear in an S-context at least once. The rules of the type deduction system are given in the following subsections.

4.2.1 Constants and variables

The typing rules for constants are [*r-int*] and [*r-char*]; they can be found in Fig. 4.1. The constants that are used in the UHC Core language are *Char*s and *Int*s. Constants cannot have parameters so sat_l is zero. Also, they never get a parameter in a strict setting (when the program is type correct), so sat_r can be safely set to zero as well. The type of a constant is always $()$. The constants can be handled in both a strict and lazy setting.

Also in Fig. 4.1, the rule [*r-var*] for variables is present. From this rule it can be determined that the assigned type and relevance information for a variable should match the type and information that is available for the variable in the type environment.

Relevance typing $\boxed{\widehat{\Gamma} \vdash e :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}}$

$$\frac{}{[\,] \vdash z :: ()^{(\varphi,0,0)}}\,[\textit{r-int}] \qquad \frac{}{[\,] \vdash c :: ()^{(\varphi,0,0)}}\,[\textit{r-char}]$$

$$\frac{}{[x \mapsto (\widehat{\tau}, \varphi, sat_l, sat_r)] \vdash x :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}}\,[\textit{r-var}]$$

Figure 4.1: Relevance typing for constants and variables.

4.2.2 Abstractions and applications

The typing rule [*r-abs*] in Fig. 4.2 for a function abstraction is crucial since this is where we detect the relevant abstractions. When looking at the typing rule, it can be seen that the typing of the abstraction $\lambda x \to e_1$ depends on the typing of the body e_1 in a type environment that is extended with parameter x. To detect whether the abstraction is relevant, the relevance of the body is set to S and sat_r is set to *maxInt*. When x is relevant in the resulting annotated type environment of the body, it can be concluded that the abstraction is relevant. When setting the relevance of the body to S and the sat_r to *maxInt*, this reset also propagates to free variables that are in the body. To prevent this from happening, we require that the none of the bindings in the resulting type environment carries an annotation that is smaller (or stricter) than the input relevance φ. Also, sat_r must be set to zero when the input relevance is L, since the number of supplied arguments in a strict context is zero. This is captured in the containment restriction $\blacktriangleright$. Containment is defined as follows:

Containment $\boxed{\varphi \blacktriangleright \widehat{\Gamma}}$

$$\frac{}{\varphi \blacktriangleright [\,]}\,[\textit{c-nil}] \qquad \frac{\mathsf{S} \blacktriangleright \widehat{\Gamma}_1}{\mathsf{S} \blacktriangleright \widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \varphi, sat_l, sat_r)]}\,[\textit{c-cons-s}]$$

$$\frac{\mathsf{L} \blacktriangleright \widehat{\Gamma}_1}{\mathsf{L} \blacktriangleright \widehat{\Gamma}_1[x \mapsto (\widehat{\tau}, \mathsf{L}, sat_l, 0)]}\,[\textit{c-cons-l}]$$

A helper function is used to determine whether all the information should be propagated or contained. This helper function will determine if enough arguments are supplied to this function to lift the containment restriction. It is defined as follows:

$$\begin{aligned} \textit{prop}\ (l, r, \varphi) &\mid l > r &&= \mathsf{L} \\ &\mid \textit{otherwise} &&= \varphi \end{aligned}$$

As an input it takes the sat_l, the sat_r and the φ of the function. It checks if the sat_r is higher than the sat_l. When sat_l is higher, L will be returned. Otherwise φ is returned. Also, the counters for the saturation must match the expression. The sat_l of the lambda abstraction is the sat_l of the body increased with one because it needs one argument more than the body. Also the sat_r for the body is the sat_r of the lambda expresssion decreased by one.

The typing rule for the application [*r-app*] is rather straightforward. The derived relevance of the function is propagated to the argument only when enough arguments are supplied (which is also determined by the *prop* function). The counters for the saturation will also have to be updated. This means that the sat_r of the expression is increased by one to get the sat_r of the function. For the resulting type environment $\widehat{\Gamma}$, the meet is taken over the two resulting type environments $\widehat{\Gamma}_1$ and $\widehat{\Gamma}_2$ of *e1* and *e2*. This is a partial function which calculates new values for the type environment pointwise. The meet is taken over the relevance values and the maximimum is taken over the sat_r. This is defined as:

$$\begin{array}{ll} [\,] & \sqcap [\,] \qquad\qquad = [\,] \\ \widehat{\Gamma}_1\ [x \mapsto (\widehat{\tau}, \varphi_1, sat_l, sat_{r1})] \sqcap \widehat{\Gamma}_2\ [x \mapsto (\widehat{\tau}, \varphi_2, sat_l, sat_{r2})] = & \\ \qquad (\widehat{\Gamma}_1 \sqcap \widehat{\Gamma}_2)\ [x \mapsto (\widehat{\tau}, \varphi_1 \sqcap \varphi_2, sat_l, max\ sat_{r1}\ sat_{r2})] & \end{array}$$

This corresponds to the intuition that a variable is relevant when its relevance can be established in at least one of the two type environments.

Relevance typing $\boxed{\widehat{\Gamma} \vdash e :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}}$

$$\frac{prop\ (sat_{l2}+1, sat_r, \varphi) \blacktriangleright \widehat{\Gamma} \quad \widehat{\Gamma}[x \mapsto (\widehat{\tau}_1, \varphi_1, sat_{l1}, sat_{r1})] \vdash e_1 :: \widehat{\tau}_2^{(\mathsf{S}, sat_{l2}, maxInt)}}{\widehat{\Gamma} \vdash \lambda x \to e_1 :: (\widehat{\tau}_1 \xrightarrow{\varphi_1} \widehat{\tau}_2)^{(\varphi, sat_{l2}+1, sat_r)}}\ [r\text{-}abs]$$

$$\frac{\widehat{\Gamma}_1 \vdash e_1 :: (\widehat{\tau}_2 \xrightarrow{\varphi_1} \widehat{\tau})^{(\varphi, sat_l, sat_r+1)} \quad \widehat{\Gamma}_2 \vdash e_2 :: \widehat{\tau}_2^{(prop\ (sat_l-1, sat_r, \varphi_1), sat_{l1}, sat_{r1})}}{\widehat{\Gamma}_1 \sqcap \widehat{\Gamma}_2 \vdash e_1\ e_2 :: \widehat{\tau}^{(\varphi, sat_l-1, sat_r)}}\ [r\text{-}app]$$

Figure 4.2: Relevance typing for lambdas and applications.

4.2.3 Let bindings

There are three different kinds of let bindings: plain let bindings (**let**), recursive let bindings (**letrec**) and strict let bindings (**let**!).

When looking at the typing rule for the plain let binding [*r-let*] in Fig. 4.3, it can be seen that the body of the binding has the same relevance type as the binding itself. The relevance information of the right hand side of the binding e_1 should be equal to the relevance information for the identifier x in the type environment of the body. The meet over type environments is used again to combine the type environments of the body and the right hand side.

The typing rule for the strict let binding [*r-let!*] in Fig. 4.3 is similar to the typing rule for the plain let binding. The only difference is that the relevance value φ of the right hand side is the same as that of the whole expression to express that it is a strict binding.

For the recursive let binding, the typing rule [*r-letrec*] in Fig. 4.3 is again almost the same as the plain let binding. The difference is that the relevance information of the right hand side is enforced to be the same as the relevance information that is added to the type environment $\widehat{\Gamma}_i$ (i = 0, ..., n) and $\widehat{\Gamma}$. Also, the typing rule deals with a list of bindings here so mutually recursive functions can be defined.

Relevance typing $\boxed{\widehat{\Gamma} \vdash e :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}}$

$$\frac{\widehat{\Gamma}_1 \vdash e_1 :: \widehat{\tau}_1^{(\varphi_1, sat_{l1}, sat_{r1})} \quad \widehat{\Gamma}_2\,[x \mapsto (\widehat{\tau}_1, \varphi_1, sat_{l1}, sat_{r1})] \vdash e_2 :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}}{\widehat{\Gamma}_1 \sqcap \widehat{\Gamma}_2 \vdash \mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2 :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}}\ [\textit{r-let}]$$

$$\frac{\widehat{\Gamma}_1 \vdash e_1 :: \widehat{\tau}_1^{(\varphi, sat_{l1}, sat_{r1})} \quad \widehat{\Gamma}_2\,[x \mapsto (\widehat{\tau}_1, \varphi, sat_{l1}, sat_{r1})] \vdash e_2 :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}}{\widehat{\Gamma}_1 \sqcap \widehat{\Gamma}_2 \vdash \mathbf{let!}\, x = e_1 \,\mathbf{in}\, e_2 :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}}\ [\textit{r-let!}]$$

$$\frac{\begin{array}{c}\forall i. 0 \leqslant i \leqslant n : \widehat{\Gamma}_i\,[x_0 \mapsto (\widehat{\tau}_0, \varphi_0, sat_{l0}, sat_{r0}), ..., x_n \mapsto (\widehat{\tau}_n, \varphi_n, sat_{ln}, sat_{rn})] \vdash e_i :: \widehat{\tau}_i^{(\varphi_i, sat_{li}, sat_{ri})} \\ \widehat{\Gamma}\,[x_0 \mapsto (\widehat{\tau}_0, \varphi_0, sat_{l0}, sat_{r0}), ..., x_n \mapsto (\widehat{\tau}_n, \varphi_n, sat_{ln}, sat_{rn})] \vdash e :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}\end{array}}{(\widehat{\Gamma}_0 \sqcap ... \sqcap \widehat{\Gamma}_n) \sqcap \widehat{\Gamma} \vdash \mathbf{letrec}\,[x_0 = e_0, ..., x_n = e_n]\,\mathbf{in}\, e :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}}\ [\textit{r-letrec}]$$

Figure 4.3: Relevance typing for the plain, strict and recursive let bindings.

4.2.4 Case expressions

The case expression is special since it has branching in it. When looking at the typing rule [*r-case*] in Fig. 4.4, the relevance information ($\widehat{\tau}, \varphi, sat_l$ and sat_r) should be the same for each expression e_i (where i = 0, ..., n) in the right hand side of a case alternative. This relevance information must also be the same as the information of the case expression. The relevance of the scrutinee must be the same as that of the complete expression. Also, a containment restriction is used for the case arms. The relevance is set to S and sat_r to *maxInt* to get all the information from the case arms. When it is in a lazy setting or not enough arguments are supplied, the containment restricts the annotated type environments of all the case arms.

For the patterns, a function is used to create the type environment for the variables in the pattern. This function *getVarTys* gets all the variables with the corresponding types from a pattern and returns this information in the form of an environment. It can be assumed that the information about tags is already available in the input $\widehat{\Gamma}$ of the whole module.

$$\begin{array}{ll}
\mathit{getVarTys}\ (\mathit{pat}, \widehat{\tau}, \widehat{\Gamma}) = \widehat{\Gamma} & \\
\mathit{getVarTys}\ (x, \widehat{\tau}, \widehat{\Gamma}) & = [x \mapsto (\widehat{\tau}, \varphi, 0, 0)] \\
\mathit{getVarTys}\ (\mathit{tup\ tag\ xs}, \widehat{\tau}, \widehat{\Gamma}) & = \mathbf{let}\ (\widehat{\tau}_t, \_, \_, \_) = \widehat{\Gamma}\ [\mathit{tag}] \\
 & \quad \mathbf{in}\ \mathit{match\ xs}\ \widehat{\tau}_t \\
\mathit{getVarTys}\ (z, \widehat{\tau}, \widehat{\Gamma}) & = [\,] \\
\mathit{getVarTys}\ (c, \widehat{\tau}, \widehat{\Gamma}) & = [\,] \\
\mathit{match}\ ([\mathit{Var}], \widehat{\tau}) = \widehat{\Gamma} & \\
\mathit{match}\ ((x : xs), \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2) = (\mathit{match\ xs}\ \widehat{\tau}_2) &
\end{array}$$

For combining the information from the type environments from the different case branches, a point wise join is used. This is justified by the fact that at least one of these alternatives is taken. The partial function is defined as:

$$\begin{aligned}
&[\,] \sqcup [\,] = [\,] \\
&\widehat{\Gamma}_1\ [x \mapsto (\widehat{\tau}, \varphi_1, sat_l, sat_{r1})] \sqcup \widehat{\Gamma}_2\ [x \mapsto (\widehat{\tau}, \varphi_2, sat_l, sat_{r2})] = \\
&\qquad (\widehat{\Gamma}_1 \sqcup \widehat{\Gamma}_2)\ [x \mapsto (\widehat{\tau}, \varphi_1 \sqcup \varphi_2, sat_l, min\ sat_{r1}\ sat_{r2})]
\end{aligned}$$

Relevance typing $\boxed{\widehat{\Gamma} \vdash e :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}}$

$$\frac{\begin{array}{c}\widehat{\Gamma} \vdash e :: \widehat{\tau}_1^{(\varphi, sat_{l1}, sat_{r1})} \\ \forall i. 0 \leqslant i \leqslant n : prop\ (sat_l, sat_r, \varphi) \blacktriangleright \widehat{\Gamma}_i \quad \widehat{\Gamma}_i + \!\! + (getVarTys\ (p_i, \widehat{\tau}_1, \widehat{\Gamma}_i)) \vdash e_i :: \widehat{\tau}^{(\mathsf{S}, sat_l, maxInt)}\end{array}}{\widehat{\Gamma} \sqcap (\widehat{\Gamma}_1 \sqcup \ldots \sqcup \widehat{\Gamma}_n) \vdash \textbf{case}\ e\ \textbf{of}\ [p_0 \rightarrow e_0, \ldots, p_n \rightarrow e_n] :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}}\ [r\text{-}case]$$

Figure 4.4: Relevance typing for a case expression.

4.2.5 Tup and FFI

Tups are used for the construction of datatypes. It can be assumed that strictness information about the tags is already available in the type environment (where a tag is treated just like a variable). In this way, the typing rule for the tup looks similar to the typing rule for variables which can be seen in [*r-tup*].

FFI's are used to make calls via the foreign function interface. Just like tups, it can be assumed that the strictness information is already present in the type environment (where the identifier of the FFI is treated as a normal variable). When looking at the typing rule [*r-ffi*], this also looks similar to the variable typing rule.

Relevance typing $\boxed{\widehat{\Gamma} \vdash e :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}}$

$$\frac{}{[tag \mapsto (\widehat{\tau}, \varphi, sat_l, sat_r)] \vdash tup\ tag :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}}\ [r\text{-}tup]$$

$$\frac{}{[x \mapsto (\widehat{\tau}, \varphi, sat_l, sat_r)] \vdash FFI\ x :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}}\ [r\text{-}ffi]$$

Figure 4.5: Relevance typing for a tup and an ffi.

4.2.6 Subeffecting and weakening

The rule [*r-sub*] is used for subeffecting. This states that derivations can selectively forget about the relevance of an expression. With this rule, more programs are considered well-typed. Informally, it states that any derivation that can be made from the assumption that a term is not relevant is still valid if it is actually relevant.

The rule [*r-weak*] is used for weakening. This expresses that any type that can be derived for an annotated type environment $\widehat{\Gamma}$ that does not contain a mapping for the variable x can also be derived for an annotated type environment that maps x to the L annotation.

Relevance typing $\boxed{\widehat{\Gamma} \vdash e :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}}$

$$\frac{\widehat{\Gamma} \vdash e :: \widehat{\tau}^{(\mathsf{L}, sat_l, 0)}}{\widehat{\Gamma} \vdash e :: \widehat{\tau}^{(\mathsf{S}, sat_l, sat_r)}} \; [\textit{r-sub}]$$

$$\frac{x \notin \mathit{dom}\,(\widehat{\Gamma}) \quad \widehat{\Gamma} \vdash e :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}}{\widehat{\Gamma}\,[x \mapsto (\widehat{\tau}_0, \mathsf{L}, sat_{lx}, 0)] \vdash e :: \widehat{\tau}^{(\varphi, sat_l, sat_r)}} \; [\textit{r-weak}]$$

Figure 4.6: Relevance typing for subeffecting and weakening.

4.3 Example

An example is presented. This gives a feeling what kind of outcomes can be derived for the expressions and what a derivation looks like.

Consider a simple example:

$$(\lambda x \to \lambda y \to x)\ 1\ 2$$

The result of the derivation is:

$$(\lambda x \to \lambda y \to x)\ 1\ 2 :: ()^{(\mathsf{S},0,0)}$$

This expression is saturated, so sat_l and sat_r are zero and the type is unit. When looking at the derivation in Figure 4.7, it can be seen that the function $\lambda x \to \lambda y \to x$ gets the type $(() \xrightarrow{\mathsf{S}} () \xrightarrow{\mathsf{L}} ())^{(\mathsf{S},2,2)}$. The type implies that it is strict in its first argument and lazy in its second. The sat_l of two shows that it needs two arguments before it is saturated, while the sat_r of two shows that two arguments are supplied in a strict context. These sat_l and sat_r values are also used as an input for the *prop* function. As a result, the type environment does not have to be contained because we know the function body will be evaluated.

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{\mathit{prop}\ (2,2,\mathsf{S}) \blacktriangleright [\,]}
      \quad
      \dfrac{
        \dfrac{}{\mathit{prop}\ (1,1,\mathsf{S}) \blacktriangleright [x \mapsto ()^{(\mathsf{S},0,0)}]}
        \quad
        \dfrac{
          \dfrac{}{[x \mapsto ((),\mathsf{S},0,0)] \vdash x :: ()^{(\mathsf{S},0,0)}}
        }{[x \mapsto ((),\mathsf{S},0,0), y \mapsto ((),\mathsf{L},0,0)] \vdash x :: ()^{(\mathsf{S},0,0)}}
      }{[x \mapsto ((),\mathsf{S},0,0)] \vdash \lambda y \rightarrow x :: (() \xrightarrow{\mathsf{L}} ())^{(\mathsf{S},1,1)}}
    }{[\,] \vdash \lambda x \rightarrow \lambda y \rightarrow x :: (() \xrightarrow{\mathsf{S}} () \xrightarrow{\mathsf{L}} ())^{(\mathsf{S},2,2)}}
    \quad
    \dfrac{}{[\,] \vdash 1 :: ()^{(\mathsf{S},0,0)}}
  }{[\,] \vdash (\lambda x \rightarrow \lambda y \rightarrow x)\ 1 :: (() \xrightarrow{\mathsf{L}} ())^{(\mathsf{S},1,1)}}
  \quad
  \dfrac{}{[\,] \vdash 2 :: ()^{(\mathsf{L},0,0)}}
}{[\,] \vdash (\lambda x \rightarrow \lambda y \rightarrow x)\ 1\ 2 :: ()^{(\mathsf{S},0,0)}}
$$

Figure 4.7: A derivation for the expression $(\lambda x \rightarrow \lambda y \rightarrow x)\ 1\ 2$.

Chapter 5

Implementation

The implementation of the annotated type system is done in the context of the UHC system. The starting point for the syntax-driven approach is algorithm W [4]. Just like in the typing rules, the relevance information is added to the type environments and the types. The counters for the saturation are incorporated as in- and output of the inferencing function, where sat_r is the input and sat_l is the output. Since no concrete types are used, the part of the inference system concerning polymorphism is not used. This system is based on the typing rules of Chapter 4. For ease of reference, the complete set of typing rules is repeated in Fig. 5.1.

Relevance typing $\boxed{\widehat{\Gamma} \vdash e :: \widehat{\tau}^{(\varphi,sat_l,sat_r)}}$

$$\frac{}{[\,] \vdash z :: ()^{(\varphi,0,0)}}\ [\textit{r-int}] \qquad \frac{}{[\,] \vdash c :: ()^{(\varphi,0,0)}}\ [\textit{r-char}]$$

$$\frac{}{[x \mapsto (\widehat{\tau}, \varphi, sat_l, sat_r)] \vdash x :: \widehat{\tau}^{(\varphi,sat_l,sat_r)}}\ [\textit{r-var}]$$

$$\frac{\textit{prop}\ (sat_{l2}+1, sat_r, \varphi) \blacktriangleright \widehat{\Gamma} \quad \widehat{\Gamma}[x \mapsto (\widehat{\tau}_1, \varphi_1, sat_{l1}, sat_{r1})] \vdash e_1 :: \widehat{\tau}_2^{(\mathsf{S},sat_{l2},\textit{maxInt})}}{\widehat{\Gamma} \vdash \lambda x \to e_1 :: (\widehat{\tau}_1 \xrightarrow{\varphi_1} \widehat{\tau}_2)^{(\varphi,sat_{l2}+1,sat_r)}}\ [\textit{r-abs}]$$

$$\frac{\widehat{\Gamma}_1 \vdash e_1 :: (\widehat{\tau}_2 \xrightarrow{\varphi_1} \widehat{\tau})^{(\varphi,sat_l,sat_r+1)} \quad \widehat{\Gamma}_2 \vdash e_2 :: \widehat{\tau}_2^{(\textit{prop}\ (sat_l-1,sat_r,\varphi_1),sat_{l1},sat_{r1})}}{\widehat{\Gamma}_1 \sqcap \widehat{\Gamma}_2 \vdash e_1\ e_2 :: \widehat{\tau}^{(\varphi,sat_l-1,sat_r)}}\ [\textit{r-app}]$$

$$\frac{\widehat{\Gamma}_1 \vdash e_1 :: \widehat{\tau}_1^{(\varphi_1,sat_{l1},sat_{r1})} \quad \widehat{\Gamma}_2\ [x \mapsto (\widehat{\tau}_1, \varphi_1, sat_{l1}, sat_{r1})] \vdash e_2 :: \widehat{\tau}^{(\varphi,sat_l,sat_r)}}{\widehat{\Gamma}_1 \sqcap \widehat{\Gamma}_2 \vdash \textbf{let}\, x = e_1\ \textbf{in}\ e_2 :: \widehat{\tau}^{(\varphi,sat_l,sat_r)}}\ [\textit{r-let}]$$

$$\frac{\widehat{\Gamma}_1 \vdash e_1 :: \widehat{\tau}_1^{(\varphi,sat_{l1},sat_{r1})} \quad \widehat{\Gamma}_2\ [x \mapsto (\widehat{\tau}_1, \varphi, sat_{l1}, sat_{r1})] \vdash e_2 :: \widehat{\tau}^{(\varphi,sat_l,sat_r)}}{\widehat{\Gamma}_1 \sqcap \widehat{\Gamma}_2 \vdash \textbf{let!}\, x = e_1\ \textbf{in}\ e_2 :: \widehat{\tau}^{(\varphi,sat_l,sat_r)}}\ [\textit{r-let!}]$$

$$\frac{\begin{array}{c}\forall i.0 \leqslant i \leqslant n : \widehat{\Gamma}_i\ [x_0 \mapsto (\widehat{\tau}_0, \varphi_0, sat_{l0}, sat_{r0}), ..., x_n \mapsto (\widehat{\tau}_n, \varphi_n, sat_{ln}, sat_{rn})] \vdash e_i :: \widehat{\tau}_i^{(\varphi_i,sat_{li},sat_{ri})} \\ \widehat{\Gamma}\ [x_0 \mapsto (\widehat{\tau}_0, \varphi_0, sat_{l0}, sat_{r0}), ..., x_n \mapsto (\widehat{\tau}_n, \varphi_n, sat_{ln}, sat_{rn})] \vdash e :: \widehat{\tau}^{(\varphi,sat_l,sat_r)}\end{array}}{(\widehat{\Gamma}_0 \sqcap ... \sqcap \widehat{\Gamma}_n) \sqcap \widehat{\Gamma} \vdash \textbf{letrec}\ [x_0 = e_0, ..., x_n = e_n]\ \textbf{in}\ e :: \widehat{\tau}^{(\varphi,sat_l,sat_r)}}\ [\textit{r-letrec}]$$

$$\frac{\begin{array}{c}\widehat{\Gamma} \vdash e :: \widehat{\tau}_1^{(\varphi,sat_{l1},sat_{r1})} \\ \forall i.0 \leqslant i \leqslant n : \textit{prop}\ (sat_l, sat_r, \varphi) \blacktriangleright \widehat{\Gamma}_i \quad \widehat{\Gamma}_i +\!\!+ (\textit{getVarTys}\ (p_i, \widehat{\tau}_1, \widehat{\Gamma}_i)) \vdash e_i :: \widehat{\tau}^{(\mathsf{S},sat_l,\textit{maxInt})}\end{array}}{\widehat{\Gamma} \sqcap (\widehat{\Gamma}_1 \sqcup ... \sqcup \widehat{\Gamma}_n) \vdash \textbf{case}\ e\ \textbf{of}\ [p_0 \to e_0, ..., p_n \to e_n] :: \widehat{\tau}^{(\varphi,sat_l,sat_r)}}\ [\textit{r-case}]$$

$$\frac{}{[\textit{tag} \mapsto (\widehat{\tau}, \varphi, sat_l, sat_r)] \vdash \textit{tup tag} :: \widehat{\tau}^{(\varphi,sat_l,sat_r)}}\ [\textit{r-tup}]$$

$$\frac{}{[x \mapsto (\widehat{\tau}, \varphi, sat_l, sat_r)] \vdash \textit{FFI}\ x :: \widehat{\tau}^{(\varphi,sat_l,sat_r)}}\ [\textit{r-ffi}]$$

$$\frac{\widehat{\Gamma} \vdash e :: \widehat{\tau}^{(\mathsf{L},sat_l,0)}}{\widehat{\Gamma} \vdash e :: \widehat{\tau}^{(\mathsf{S},sat_l,sat_r)}}\ [\textit{r-sub}] \qquad \frac{x \notin \textit{dom}\ (\widehat{\Gamma}) \quad \widehat{\Gamma} \vdash e :: \widehat{\tau}^{(\varphi,sat_l,sat_r)}}{\widehat{\Gamma}\ [x \mapsto (\widehat{\tau}_0, \mathsf{L}, sat_{lx}, 0)] \vdash e :: \widehat{\tau}^{(\varphi,sat_l,sat_r)}}\ [\textit{r-weak}]$$

Figure 5.1: Relevance typing including saturation.

5.1 Inference algorithm

Guided by the formal specification of the relevance typing defined in Chapter 4, an inference algorithm $\mathscr{W}_{rel}$ is defined which reconstructs the types and determines the relevance values and saturation. This type inference algorithm for relevance typing analysis is based on the classical algorithm W developed for the Hindley/Milner polymorphic type system [4].

The differences of this implementation with algorithm W lies in how equalities on types are derived. An often used implementation of algorithm W uses unification and substitution. In this inference algorithm, the substitutions are implemented by returning an updated type environment. Unification uses the join on types. The result is a little less precise but resembles the substructural typing discipline that is used in the formal specification more closely.

The inference algorithm is defined according to the following scheme:

$$\mathscr{W}_{rel}\ (\widehat{\Gamma}, e, \varphi, sat_r) = (\widehat{\Gamma}, \widehat{\tau}, sat_l)$$

The inference algorithm $\mathscr{W}_{rel}$ takes four arguments: an annotated type environment $\widehat{\Gamma}$, an expression e, the relevance value φ and the counter sat_r. Here the $\widehat{\Gamma}$ maps variables to relevance information which consists of the $\widehat{\tau}$, φ, sat_l and sat_r. The relevance value φ denotes the relevance of the expression and the counter sat_r gives information on how many arguments are supplied to this expression. The result of the inference algorithm $\mathscr{W}_{rel}$ consists of three parts: an annotated type environment $\widehat{\Gamma}$, an annotated type $\widehat{\tau}$ and the counter sat_l. The output $\widehat{\Gamma}$ reflects the changes in the relevance information for variables after analysing the expression. The $\widehat{\tau}$ gives the annotated type of the expression, and the counter sat_l gives how many arguments are needed before the expression is saturated. The starting point for the inference of a top-level expression e is a $\widehat{\Gamma}$ containing the information about the tag's and FFI's that are in scope and the sat_r set to zero.

The inference algorithm $\mathscr{W}_{rel}$ is defined in Fig. 5.2.

```
Type Inference Algorithm                    $\boxed{\mathscr{W}_{rel}\ (\widehat{\Gamma}, e, \varphi, sat_r) = (\widehat{\Gamma}, \widehat{\tau}, sat_l)}$

$\mathscr{W}_{rel}\ (\widehat{\Gamma}, z, \varphi, sat_r) = (\widehat{\Gamma}, (), 0)$
$\mathscr{W}_{rel}\ (\widehat{\Gamma}, c, \varphi, sat_r) = (\widehat{\Gamma}, (), 0)$
$\mathscr{W}_{rel}\ (\widehat{\Gamma}, x, \varphi, sat_r) =$
    $\mathbf{let}\ (\widehat{\tau}, \varphi_1, sat_l, sat_{rx}) = \widehat{\Gamma}\ (x)$
        $\widehat{\Gamma}_1 = \widehat{\Gamma}\ [x \mapsto (\widehat{\tau}, \varphi \sqcap \varphi_1, sat_l, \mathbf{if}\ \varphi \equiv \mathsf{S}\ \mathbf{then}\ max\ (sat_r, sat_{rx})$
                                                  $\mathbf{else}\ \ sat_{rx})]$
    $\mathbf{in}\ (\widehat{\Gamma}_1, \widehat{\tau}, sat_l)$
$\mathscr{W}_{rel}\ (\widehat{\Gamma}, tup\ tag, \varphi, sat_r) =$
    $\mathbf{let}\ (\widehat{\tau}, \varphi_1, sat_l, sat_{rt}) = \widehat{\Gamma}\ (tag)$
        $\widehat{\Gamma}_1 = \widehat{\Gamma}\ [tag \mapsto (\widehat{\tau}, \varphi \sqcap \varphi_1, sat_l, \mathbf{if}\ \varphi \equiv \mathsf{S}\ \mathbf{then}\ max\ (sat_r, sat_{rt})$
                                                  $\mathbf{else}\ \ sat_{rt})]$
    $\mathbf{in}\ (\widehat{\Gamma}_1, \widehat{\tau}, sat_l)$
$\mathscr{W}_{rel}\ (\widehat{\Gamma}, FFI\ x, \varphi, sat_r) =$
    $\mathbf{let}\ (\widehat{\tau}, \varphi_1, sat_l, sat_{rf}) = \widehat{\Gamma}\ (x)$
        $\widehat{\Gamma}_1 = \widehat{\Gamma}\ [x \mapsto (\widehat{\tau}, \varphi \sqcap \varphi_1, sat_l, \mathbf{if}\ \varphi \equiv \mathsf{S}\ \mathbf{then}\ max\ (sat_r, sat_{rf})$
                                                  $\mathbf{else}\ \ sat_{rf})]$
    $\mathbf{in}\ (\widehat{\Gamma}_1, \widehat{\tau}, sat_l)$
$\mathscr{W}_{rel}\ (\widehat{\Gamma}, \lambda x \to e, \varphi, sat_r) =$
    $\mathbf{let}\ \widehat{\Gamma}_1 = \widehat{\Gamma}\ [x \mapsto ((), \mathsf{L}, 0, 0)]$
        $(\widehat{\Gamma}_2, \widehat{\tau}_2, sat_l) = \mathscr{W}_{rel}\ (\widehat{\Gamma}_1, e, \mathsf{S}, maxInt)$
        $(\widehat{\tau}_x, \varphi_x, sat_{lx}, sat_{rx}) = \widehat{\Gamma}_2\ (x)$
        $\widehat{\Gamma}_{res} = (\mathbf{if}\ (sat_l + 1) > sat_r \wedge \varphi \equiv \mathsf{S}\ \mathbf{then}\ \widehat{\Gamma}_2\ \mathbf{else}\ \widehat{\Gamma}) \setminus \{x\}$
    $\mathbf{in}\ (\widehat{\Gamma}_{res}, \widehat{\tau}_x \xrightarrow{\varphi_x} \widehat{\tau}_2, sat_l + 1)$
$\mathscr{W}_{rel}\ (\widehat{\Gamma}, e_1\ e_2, \varphi, sat_r) =$
    $\mathbf{let}\ (\widehat{\Gamma}_1, \widehat{\tau}_1 \xrightarrow{\varphi_1} \widehat{\tau}, sat_{l1}) = \mathscr{W}_{rel}\ (\widehat{\Gamma}, e_1, \varphi, \mathbf{if}\ \varphi \equiv \mathsf{L}\ \mathbf{then}\ 0\ \mathbf{else}\ sat_r + 1)$
        $(\widehat{\Gamma}_2, \widehat{\tau}_2, sat_{l2}) = \mathscr{W}_{rel}\ (\widehat{\Gamma}_1, e_2, prop\ (sat_{l1}, sat_r + 1, \varphi_1), 0)$
    $\mathbf{in}\ (\widehat{\Gamma}_2, \widehat{\tau}, sat_{l1} - 1)$
$\mathscr{W}_{rel}\ (\widehat{\Gamma}, \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2, \varphi, sat_r) =$
    $\mathbf{let}\ (\widehat{\Gamma}_1, \widehat{\tau}_1, sat_{l1}) = \mathscr{W}_{rel}\ (\widehat{\Gamma}, e_1, \mathsf{S}, maxInt)$
        $(\widehat{\Gamma}_2, \widehat{\tau}_2, sat_{l2}) = \mathscr{W}_{rel}\ (\widehat{\Gamma}\ [x \mapsto (\widehat{\tau}_1, \mathsf{L}, sat_{l1}, 0)], e_2, \varphi, sat_r)$
        $(\widehat{\tau}_x, \varphi_x, sat_{lx}, sat_{rx}) = \widehat{\Gamma}_2\ (x)$
        $\widehat{\Gamma}_{res} = \mathbf{if}\ \varphi_x \equiv \mathsf{S} \wedge sat_{lx} \leqslant sat_{rx}\ \mathbf{then}\ \widehat{\Gamma}_1 \sqcap \widehat{\Gamma}_2\ \mathbf{else}\ \widehat{\Gamma}_2$
    $\mathbf{in}\ (\widehat{\Gamma}_{res} \setminus \{x\}, \widehat{\tau}_2, sat_{l2})$
```

Figure 5.2: Inference Algorithm for Monovariant Relevance Typing

```
$\mathscr{W}_{rel}\ (\widehat{\Gamma}, \textbf{let!}\ x = e_1\ \textbf{in}\ e_2, \varphi, sat_r) =$
  $\textbf{let}\ (\widehat{\Gamma}_1, \widehat{\tau}_1, sat_{l1}) = \mathscr{W}_{rel}\ (\widehat{\Gamma}, e_1, \mathsf{S}, maxInt)$
      $(\widehat{\Gamma}_2, \widehat{\tau}_2, sat_{l2}) = \mathscr{W}_{rel}\ (\widehat{\Gamma}\ [x \mapsto (\widehat{\tau}_1, \varphi, sat_{l1}, 0)], e_2, \varphi, sat_r)$
      $(\widehat{\tau}_x, \varphi_x, sat_{lx}, sat_{rx}) = \widehat{\Gamma}_2\ (x)$
      $\widehat{\Gamma}_{res} = \textbf{if}\ \varphi_x \equiv \mathsf{S} \wedge sat_{lx} \leqslant sat_{rx}\ \textbf{then}\ \widehat{\Gamma}_1 \sqcap \widehat{\Gamma}_2\ \textbf{else}\ \widehat{\Gamma}_2$
  $\textbf{in}\ (\widehat{\Gamma}_{res} \setminus \{x\}, \widehat{\tau}_2, sat_{l2})$
$\mathscr{W}_{rel}\ (\widehat{\Gamma}, \textbf{letrec}\ [x_0 = e_0, ..., x_n = e_n]\ \textbf{in}\ e, \varphi, sat_r) =$
  $\textbf{let}\ \forall i. 0 \leqslant i \leqslant n$
          $\widehat{\Gamma} = \widehat{\Gamma} +\!\!+ [x_i \mapsto ((), \mathsf{L}, 0, 0)]$
      $\widehat{\Gamma}_{-1} = \widehat{\Gamma}$
      $\forall i. 0 \leqslant i \leqslant n$
          $(\widehat{\Gamma}_i, \widehat{\tau}_i, sat_{li}) = \mathscr{W}_{rel}\ (\widehat{\Gamma}_{(i-1)}, e_i, \mathsf{S}, maxInt)$
          $\widehat{\Gamma}$                 $= \widehat{\Gamma} +\!\!+ [x_i \mapsto (\widehat{\tau}_i, \mathsf{L}, sat_{li}, 0)]$
      $(\widehat{\Gamma}_b, \widehat{\tau}_b, sat_{lb}) = \mathscr{W}_{rel}\ (\widehat{\Gamma}, e, \varphi, sat_r)$
      $\forall i. 0 \leqslant i \leqslant n$
          $(\widehat{\tau}_i, \varphi_i, sat_{li}, sat_{ri}) = \widehat{\Gamma}_b\ (x_i)$
          $\widehat{\Gamma}_b = \textbf{if}\ \varphi_i \equiv \mathsf{S} \wedge sat_{li} \leqslant sat_{ri}\ \textbf{then}\ \widehat{\Gamma}_i \sqcap \widehat{\Gamma}_b\ \textbf{else}\ \widehat{\Gamma}_b$
  $\textbf{in}\ (\widehat{\Gamma}_b \setminus \{x_0, ..., x_n\}, \widehat{\tau}_b, sat_{lb})$
$\mathscr{W}_{rel}\ (\widehat{\Gamma}, \textbf{case}\ e\ \textbf{of}\ [p_0 \to e_0, ..., p_n \to e_n], \varphi, sat_r) =$
  $\textbf{let}\ (\widehat{\Gamma}_e, \widehat{\tau}_e, sat_{le}) = \mathscr{W}_{rel}\ (\widehat{\Gamma}, e, \varphi, 0)$
      $\forall i. 0 \leqslant i \leqslant n$
          $(\widehat{\Gamma}_i, \widehat{\tau}_i, sat_{li}) = \mathscr{W}_{rel}\ (\widehat{\Gamma}_e +\!\!+ (getVarTys\ (p_i, \widehat{\tau}_e, \mathsf{L}, \widehat{\Gamma})), e_1, \mathsf{S}, maxInt)$
          $\widehat{\Gamma}_i = \widehat{\Gamma}_i \setminus (getVars\ (p_i))$
      $\widehat{\Gamma}_{iter} = \widehat{\Gamma}_e$
      $sat_{lres} = 0$
      $\forall i. 0 \leqslant i \leqslant n$
          $sat_{lres} = max\ sat_{lres}\ sat_{li}$
          $\widehat{\Gamma}_{iter}\ \ = \widehat{\Gamma}_{iter} \sqcup \widehat{\Gamma}_i$
      $\widehat{\Gamma}_{res} = \textbf{if}\ \varphi \equiv \mathsf{S} \wedge sat_{lres} \leqslant sat_r$
              $\textbf{then}\ \widehat{\Gamma}_{iter}$
              $\textbf{else}\ \ \widehat{\Gamma}_e$
      $\widehat{\tau}_{res} = \widehat{\tau}_0$
      $\forall i. 1 \leqslant i \leqslant n$
          $\widehat{\tau}_{res} = \widehat{\tau}_{res} \sqcup \widehat{\tau}_i$
  $\textbf{in}\ (\widehat{\Gamma}_{res}, \widehat{\tau}_{res}, sat_{lres})$
```

Figure 5.3: Inference Algorithm for Monovariant Relevance Typing (Continued)

Two auxiliary functions are used. The $(\setminus)$ function takes two arguments, a $\widehat{\Gamma}$ and a set of variables. The returned $\widehat{\Gamma}$ will be the input $\widehat{\Gamma}$ from which the set of variables is removed. In addition, the function *getVars* is used which retrieves all the variables in a pattern and returns it as a set.

In the actual implementation, the algorithm is specified in the attribute grammar system. The input is defined by inherited attributes and the output is defined by synthesized attributes.

The inference algorithm is described in the following Subsections.

5.1.1 Constants, variables, FFI's and tags

The inference for constants is straightforward. The input sat_r and φ are ignored. A constant always gets a unit type and sat_l is zero since it does not require any arguments. Also the annotated type environment is returned unchanged.

For variables, the output $\widehat{\Gamma}$ is updated for the variable that is analysed. The context information (the input of the function) sat_r and φ is used. A lookup for the variable in the input type environment is required first to retrieve the relevance information. The returned $\widehat{\Gamma}$ is the input $\widehat{\Gamma}$ where the variable maps to updated relevance information. The φ and sat_r are only updated with the new versions when this is an "improvement" of the information. With an improvement, a higher sat_r or a φ that is more strict is meant. The $\widehat{\tau}$ and sat_l remain unchanged. The resulting sat_l of this expression is looked up in the input annotated type environment.

An *FFI* construct, (for the foreign function interface) is handled the same as the variables. The annotated type environment is updated for the variable of the FFI when this is necessary. The φ and sat_r of the context are used for this.

For the *tup tag* combination, the same is done as for the *FFI* expression. The relevance information for the *tag* is updated in the annotated type environment when this is necessary.

5.1.2 Abstractions and applications

For function abstractions, a context reset is done for the body. This means that the φ is set to S and the sat_r is set to *maxInt*. The input $\widehat{\Gamma}$ of the lambda is extended with a mapping for a parameter before it is propagated to the body. The $\widehat{\Gamma}$ that is returned for the lambda depends on the containment. When the input relevance is S and enough arguments are guaranteed to be supplied, the containment restriction is lifted and the annotated type environment from the body is propagated. When the relevance is L or not enough arguments are supplied in an S context, the input $\widehat{\Gamma}$ of the lambda is returned. This is justified by the fact that the body is not allowed to change the output $\widehat{\Gamma}$ of the lambda, because it will not be guaranteed to be evaluated. For the annotated type, the type from the body is used to which an extra arrow is added at the front. On top of this arrow comes the relevance value that can be established for the parameter in the body. The sat_l of the expression is the sat_l of the body increased by one since it needs an additional argument before it is saturated.

For applications, two separate analyses are done for the function and the argument. For the analysis of the function, the relevance φ of the expression is used as the input. The sat_r that is used for the function is the sat_r of the expression increased by one (since it gets one additional argument). The sat_r also depends on the value of φ since it only counts the arguments supplied in a strict context. When φ is L, the sat_r is set to zero. The input $\widehat{\Gamma}$ is propagated directly to the function. The output of the analysis of the function influences the input of the analysis of the argument. When a strict type is found for the function, the argument is also analysed in a strict context if enough arguments are guaranteed to be supplied. The resulting type environment is calculated by taking the meet over the two type environments obtained for the function and the argument. So when it can be derived that a variable is relevant in one of the two sub-analyses, it can be concluded to be relevant in the resulting type environment. The sat_l is determined by decreasing the sat_l of the function by one since it needs one argument less before it is saturated. The resulting $\widehat{\tau}$ is constructed by removing the first argument of the $\widehat{\tau}$ of the function. This is because one argument is applied to the function.

5.1.3 Let bindings

When using a syntax-driven approach, it is rather hard to make an efficient implementation for the let binding. This is because there is a mutual dependency between the body and the binding. It originates in the fact that in the body of the let binding the relevance types of the identifiers should be known. This is because when the rhs is a function and the identifier gets applied in the body, the relevance of the function can be propagated to the arguments. On the other hand, the sat_r and the relevance of the identifier should be known for the input of the analysis of the right hand side. Consider, for example:

$$\begin{array}{l}\textbf{let} f = \lambda x\ y \rightarrow x + y \\ \textbf{in} f\ 5\ 6\end{array}$$

For the analysis of the right hand side of the let binding, a φ and sat_r is needed as an input. The φ and sat_r for f are not available because the body is not analysed yet. But if we analyse the body first, there is not a type for f derived yet and the arguments of f cannot get the correct input relevance.

We have chosen to use the idea of the containment restriction to overcome this problem. The bindings are analysed first, where the input relevance is set to S and the sat_r to *maxInt*. In this way, all the possible relevance information can be found from the right hand side of the binding. The $\widehat{\Gamma}$ that is supplied to the analysis of the binding is the same as the $\widehat{\Gamma}$ of the whole expression. The input $\widehat{\Gamma}$ of the body is the input $\widehat{\Gamma}$ of the let expression where a mapping is added for the bound variable with the $\widehat{\tau}$ and sat_l that were determined by the analysis. For this variable, the φ is set to L and the sat_r is set to zero.

The input sat_r and φ for the body come directly from the expression itself. For the calculation of the output $\widehat{\Gamma}$ of the let expression, it has to be determined if the bound variable is used saturated in a strict context in the body of the binding. When this is the case, the meet over the output environment of the rhs and the output environment of the body is returned. Otherwise, the output environment of the body is returned as the output environment of the let expression. The resulting $\widehat{\tau}$ and sat_l comes directly from the body.

The definition of $\mathscr{W}_{rel}$ for the strict let bindings is quite similar to that of the plain let binding. The only difference is that when the relevance information is added to the $\widehat{\Gamma}$, the φ is set to S instead of L.

For the recursive let bindings, the difference in the inference algorithm is that a mapping is added to the $\widehat{\Gamma}$ for the variable of each binding. In this way, the recursive usages can be analysed. Additionally, a list of bindings is used instead of a single binding. As a result, all the identifiers have to be updated after the right hand sides are analysed. An improvement in precision in the analysis of recursive let binding is presented in Section 5.2.

5.1.4 Case expressions

For the inference of the case expression, the scrutinized expression and the expressions in the case alternatives are analysed first. The input φ as well as the $\widehat{\Gamma}$ for the scrutinized expression comes from the case expression. The input sat_r is set to zero. The output $\widehat{\Gamma}$ of the scrutinee is used for the input $\widehat{\Gamma}$ of the expressions in the case alternatives. The $\widehat{\Gamma}$ that is passed to a case alternative is the input $\widehat{\Gamma}$ extended with a mapping for the variables in the patterns. The sat_r and φ of the case expression are *maxInt* and sat_r to resemble the resetting of the context. The output sat_l is determined by taking the maximimum of the sat_l's of the case alternatives. The relevance type is calculated by taking the join over the relevance types of the alternatives. This uses the join over annotated types which is defined as:

$$\begin{array}{lll}(\widehat{\tau}_{11} \xrightarrow{\varphi_1} \widehat{\tau}_{12}) & \sqcup (\widehat{\tau}_{21} \xrightarrow{\varphi_2} \widehat{\tau}_{22}) & = (\widehat{\tau}_{11} \sqcup \widehat{\tau}_{21}) \xrightarrow{\varphi_1 \sqcup \varphi_2} (\widehat{\tau}_{12} \sqcup \widehat{\tau}_{22}) \\ () & \sqcup () & = ()\end{array}$$

The resulting type environment $\widehat{\Gamma}$ combines the information that is gathered from the case alternatives. The $\widehat{\Gamma}$'s are combined using the join over type environments. For the environments, the containment restriction will be active when not enough arguments are supplied or when the expression is not used in a strict context.

5.2 Recursion

In Section 5.1, nothing really special was done for recursive let bindings. The only difference between a normal and recursive let binding is that in the latter the names of all the bindings were added to the environment before the right hand sides of the bindings were inspected.

This gives a relatively good result for common recursive functions. Consider, for example the following function for calculating a fibonacci number.

```
letrec fib = λx → let! x′ = x
                  in case x′ of
                       0 → 0
                       1 → 0
                       n → let l = fib (x − 1)
                               r = fib (x − 2)
                           in l + r
in 0
```

It can be determined that the function is strict in its first argument x since it is evaluated by a **let**!. The resulting type for the function is $() \xrightarrow{\mathsf{S}} ()$ which is the optimal result.

More information can be derived when using fixed point iteration. This can be useful since the pessimistic assumption made for the recursive calls can influence the resulting relevance type. This is the case when an argument of the function is used as a parameter for the recursive call. An example of this is the following function f:

```
$\textbf{letrec}\, f = \lambda x\, y \to \textbf{case}\, x \equiv 0\, \textbf{of}$
                        $\mathit{True} \;\to y$
                        $\mathit{False} \to f\ (x - 1)\ y$
$\textbf{in}\, 0$
```

When this is inspected in the previously defined way, the function f will obtain type $() \xrightarrow{\mathsf{S}} () \xrightarrow{\mathsf{L}} ()$. The first argument is relevant because it is relevant to the scrutenee of the case expression. For the second argument it can never be established that it is relevant, even though it can be easily seen that this is the case. A solution for this is to use fixed point iteration. The starting point of the iteration is that every function is strict in all of its arguments. Then all the bindings are inspected. The corresponding type is updated in the type environment when it is changed. This iterates until a fixed point is reached.

The fixed point iteration is defined as a worklist algorithm in Fig. 5.4.

Fixed point iteration $\boxed{\mathit{fixRec}\ (\mathit{binds}, \widehat{\Gamma}) = \widehat{\Gamma}}$

```
$\mathit{fixRec}(\mathit{binds}, \widehat{\Gamma}) =$ do
    worklist     $:= \{\,\}$
    dependencies $:= [\,]$
    codemap      $:= [\,]$
    $\forall\ (x = e)$ in $\mathit{binds}$ do
        $\widehat{\Gamma}\ [x] := (\mathit{createStrictTy}\ (\mathit{satl}\ (e)), \mathsf{L}, 0, 0)$
        worklist $:=$ worklist $\cup \{x\}$
        codemap $[x] := e$
        $\forall\ x'$ in $\mathit{fv}(e)$ do
            dependencies $[x'] :=$ dependencies $[x'] \cup \{x\}$
    while worklist $\neq \{\,\}$ do
        let $V_1 \uplus \{x\} =$ worklist
            $(\_, \widehat{\tau}_x, \_) = \mathscr{W}_{rel}\ (\widehat{\Gamma}, \text{codemap}\ [x], \mathsf{S}, \mathit{maxInt})$
        in do worklist $:= V_1$
              if $\widehat{\tau}_x \neq \widehat{\Gamma}\ [x]$
              then do $\widehat{\Gamma}\ [x] := (\widehat{\tau}_x, \mathsf{L}, 0, 0)$
                      worklist $:=$ worklist $\cup$ dependencies$[x]$
    return $\widehat{\Gamma}$
$\mathit{createStrictTy}\ (\mathit{Int}) = \widehat{\tau}$
$\mathit{createStrictTy}\ (n)\quad = \textbf{if}\ n < 1\ \textbf{then}\ ()\ \textbf{else}\ () \xrightarrow{\mathsf{S}} (\mathit{createStrictTy}\ (n - 1))$
```

Figure 5.4: Fixed point iteration for recursive bindings.

This algorithm employs a worklist, a dependency mapping and a code environment. The worklist keeps track of the bindings which have to be analysed. The dependency mapping keeps track of dependencies between bindings. So it contains the bindings that should be analysed when the relevance type of a certain binding has changed. The building of this dependency mapping uses the *fv* function which returns the free variables of an expression. The code environment is a mapping from variables to a piece of code. Also an annotated type environment is used to keep track of annotated types of the bindings. It is assumed that the function *satl* is given which determines the sat_l of an expression. The function *createStrictTy* is used to generate strict types of a certain length.

With this all in place, fixed point iteration can be performed. In an iteration, a binding name is taken from the worklist and is looked up in the code environment. This code is supplied to the function $\mathscr{W}_{rel}$ to determine the $\widehat{\tau}$ of the code. When this has changed in comparison with the $\widehat{\tau}$ from the annotated type environment, the $\widehat{\tau}$ of the binding is updated in the $\widehat{\Gamma}$ and the dependencies are added to the worklist. This proceeds until the worklist is empty, and a fixed point has been reached. Then the current $\widehat{\Gamma}$ is returned.

The inference algorithm for the recursive let bindings is changed to use this optimization. The defintion is given in Fig. 5.5.

```
$W\ (\widehat{\Gamma}, \textbf{letrec}\ [x_0 = e_0, ..., x_n = e_n]\ \textbf{in}\ e, \varphi, sat_r) =$
  $\textbf{let}\ \widehat{\Gamma}_{-1} = \mathit{fixRec}\ ([x_0 = e_0, ..., x_n = e_n], \widehat{\Gamma})$
      $\forall i. 0 \leqslant i \leqslant n$
          $(\widehat{\Gamma}_i, \widehat{\tau}_i, sat_{li}) = W\ (\widehat{\Gamma}_{(i-1)}, e_i, \mathsf{S}, \mathit{maxInt})$
          $\widehat{\Gamma} \qquad\qquad = \widehat{\Gamma} +\!\!+ [x_i \mapsto (\widehat{\tau}_i, \mathsf{L}, sat_{li}, 0)]$
      $(\widehat{\Gamma}_b, \widehat{\tau}_b, sat_{lb}) = W\ (\widehat{\Gamma}, e, \varphi, sat_r)$
      $\forall j. 0 \leqslant j \leqslant n$
          $(\widehat{\tau}_i, \varphi_i, sat_{li}, sat_{ri}) = \widehat{\Gamma}_b\ [x_i]$
          $\widehat{\Gamma}_b = \textbf{if}\ \varphi_i \equiv \mathsf{S}\ \textbf{then}\ \widehat{\Gamma}_i \sqcup \widehat{\Gamma}_b\ \textbf{else}\ \widehat{\Gamma}_b$
  $\textbf{in}\ (\widehat{\Gamma}_b, \widehat{\tau}_b, sat_{lb})$
```

Figure 5.5: New definition of inference algorithm for letrec.

Note that the resulting type environment of the fixed point iteration is used as the input for the analysis of the bindings (and not as input of the analysis for the body). This is done to get the correct type environment of the bindings in place, which is useful for the transformation, as explained in Chapter 6.

We apply the algorithm to the expression

```
$\textbf{letrec}\ f = \lambda x\ y \to \textbf{case}\ x \equiv 0\ \textbf{of}$
                        $\mathit{True}\ \ \to y$
                        $\mathit{False} \to f\ (x - 1)\ y$
$\textbf{in}\ 0$
```

which was given earlier.

The starting point for the first iteration is:

$$\begin{array}{ll} \mathsf{worklist} & = \{f\} \\ \mathsf{dependencies} & = [f \mapsto f] \\ \widehat{\Gamma} & = [f \mapsto (() \overset{\mathsf{S}}{\to} () \overset{\mathsf{S}}{\to} (), \mathsf{S}, 0, 0)] \\ \mathsf{codemap} & = [f \mapsto f = \lambda x\ y \to \textbf{case}\ x \equiv 0\ \textbf{of} \\ & \qquad\qquad\qquad\qquad \mathit{True} \ \to y \\ & \qquad\qquad\qquad\qquad \mathit{False} \to f\ (x-1)\ y] \end{array}$$

After the first iteration, the worklist becomes empty. Since the annotated type also has not changed, the fixed point is reached immediately. The resulting $\widehat{\Gamma}$ contains the mapping for f to the relevance type $() \overset{\mathsf{S}}{\to} () \overset{\mathsf{S}}{\to} ()$. So it can be concluded that the function is strict in both of its arguments.

5.3 Higher order functions

When using a monovariant approach, there is not much support for higher order functions. This becomes clear when looking at the following example that defines the higher order function *hof*

$$\begin{array}{l} \textbf{let}\ \mathit{hof} = \lambda f\ x\ y \to f\ x\ y\ \textbf{in} \\ \textbf{let}\ \mathit{fst} = \lambda x\ y \to x\ \textbf{in} \\ \textbf{let}\ \mathit{snd} = \lambda x\ y \to y \\ \textbf{in}\ (\mathit{hof}\ \mathit{fst}\ 1\ 2) + (\mathit{hof}\ \mathit{snd}\ 1\ 2) \end{array}$$

Since there is monovariance, there are no variables in the type. The consequence is that one general type has to be used for the type of the parameter. This type must be a safe estimation, which is a function that is lazy in all of its arguments. As a result, the type for *hof* becomes $(() \overset{\mathsf{L}}{\to} () \overset{\mathsf{L}}{\to} ()) \overset{\mathsf{S}}{\to} () \overset{\mathsf{L}}{\to} () \overset{\mathsf{L}}{\to} ()$.

The fact that the first parameter is a function does not have any result for the further transformation. That is why a unit type is used for a functional argument and the type for the function *hof* is $() \overset{\mathsf{S}}{\to} () \overset{\mathsf{L}}{\to} () \overset{\mathsf{L}}{\to} ()$.

5.4 Data types

Data types occur in two places: the construction and the destruction. At both sites, strictness annotations can be added in Haskell. For the construction, annotations can be used on the parameters of a constructor. Consider for example the data type definition **data** *StrictMaybe a* = *StrictJust* ! *a* | *StrictNothing*. Here the data type constructor *StrictJust* is strict in its first argument. When looking at the destruction of a data type value in a case statement, annotations can also be used in patterns to introduce strictness. For example in Haskell syntax, $f\ !x = \mathit{True}$. In this example f is strict in its argument because of the bang pattern.

Bang patterns are transformed into let!'s in the Core langauge in a previous stage, so nothing extra has to be done. The data type annotations can be used both at the construction and destruction. At construction, the strictness can be enforced for the parameters that are given. This information is available in an environment for each *tag*, so a relevance type can be generated for a data type constructor. For example for the constructor *StrictJust*, the relevance type is $() \xrightarrow{\mathsf{S}} ()$ which represents the presence of the data type annotation on the first field.

At destruction, information can be used that parts of the patterns are already evaluated when the data type was constructed. This is useful for the transformation, since the variables from the patterns which are relevant in the expression will not need to be evaluated since they already are. Consider for example:

```
case x of
  StrictJust x → x + 10
  Nothing      → 5
```

For the first case arm, it can be concluded that x is relevant and should be evaluated. When taking the strictness annotations into account, it can be concluded that x is already evaluated at the construction. As a result, no **let**! has to be introduced for the evaluation of x.

Chapter 6

Transformation

The transformation consists of three parts. The first part is the insertion of the let!'s at the correct places. The second part is the usage of wrappers which handle the evaluation of the strict arguments. At certain places, these wrappers have to be called instead of the original functions. The third part is a postprocessing step, where unnecessary let!'s and let's are removed. We elaborate on the three parts in the following sections.

6.1 Introduction of let!'s

In order to know when a **let**! has to be introduced, first it should be calculated where exactly a variable can be evaluated. This information is exactly what is provided by the annotated type environments. An additional data structure, an evaluation list, is used to encode what variables should be evaluated.

6.1.1 Evaluation lists

An evaluation list is nothing more than a list of variables. The evaluation list can be seen as an annotation for an expression, where it contains the variables that should be evaluated at that expression. To calculate an evaluation list, a boundary has to be found where the relevance value of a variable goes from S to L. Such a boundary is between an expression and a subexpression. Consider the following expression:

$$\begin{array}{l} \textbf{let}\, f = 5 + 5 \\ \textbf{in}\, \lambda x \rightarrow \textbf{case}\ x\ \textbf{of} \\ \qquad\qquad \mathit{True} \ \ \rightarrow f \\ \qquad\qquad \mathit{False} \rightarrow 4 \end{array}$$

Here it is clear that f is relevant to the first case arms, but not to the whole case expression. The boundary that is determined is the first case arm, because the relevance of f goes from S in the first case arm to L in the case expression. As a result, f can be evaluated when entering the first case arm. To determine such a boundary, the differences between two type environments are calculated. In this case, the difference is calculated between the type environment of the first case arm and type environment of the complete case expression. Because the code is in administrative normal form, the evaluation lists will only have to contain variables and not complete expressions.

The function *tyEnvDiff* is used to determine the differences between two $\widehat{\Gamma}$'s:

$$\mathit{tyEnvDiff}\ (\widehat{\Gamma}_1, \widehat{\Gamma}_2) = \mathit{lvars}\ (\widehat{\Gamma}_1) \cap \mathit{svars}\ (\widehat{\Gamma}_2)$$

$$\begin{aligned}
&\mathit{lvars}\ \widehat{\Gamma} = \mathit{toVarsSet}\ (\mathit{filter}\ (\lambda(x \mapsto (\_, \varphi, \mathit{sat}_l, \_)) \to \varphi \equiv \mathsf{L} \wedge \mathit{sat}_l \equiv 0)\ \widehat{\Gamma}) \\
&\mathit{svars}\ \widehat{\Gamma} = \mathit{toVarsSet}\ (\mathit{filter}\ (\lambda(x \mapsto (\_, \varphi, \mathit{sat}_l, \_)) \to \varphi \equiv \mathsf{S} \wedge \mathit{sat}_l \equiv 0)\ \widehat{\Gamma})
\end{aligned}$$

In this function, the intersection is taken over two sets of variables. The first set contains all the variables which have a φ of L in $\widehat{\Gamma}_1$. The second set contains all the variables which have a φ of S in $\widehat{\Gamma}_2$. Additionally, the sat_l of the variables must be zero because only saturated expressions must be evaluated. Note that the order in which the arguments are supplied is important. First the $\widehat{\Gamma}$ of the whole expression shoud be supplied and secondly the $\widehat{\Gamma}$ of the subexpression.

When this approach is applied to a let binding, the difference should be calculated between the $\widehat{\Gamma}$ of the right hand side of the binding and the $\widehat{\Gamma}$ of the body of the binding, when the corresponding variable is not relevant to the body (because then the $\widehat{\Gamma}$ of the rhs must be contained). An example of this is the following expression:

$$\begin{aligned}
&\mathbf{let}\ g = 5 + 5\ \mathbf{in} \\
&\mathbf{let} f = \lambda x \to x + g \\
&\mathbf{in}\ 0
\end{aligned}$$

Looking at this expression, g is relevant to the body of the lambda in the right hand side of f. Because the right hand side of a binding is inspected in an S context and the sat_r is set to *maxInt*, the relevance of the body of the lambda will also be S. The containment restriction is lifted and the relevance of g propagates through the lambda. Since f is not relevant to the body of the binding, the evaluation list is calculated for the $\widehat{\Gamma}$ of the rhs and the $\widehat{\Gamma}$ of the body. From this it can be concluded that g should be evaluated in the rhs of f. The result of the transformation is:

$$\begin{aligned}
&\mathbf{let}\ g = 5 + 5\ \mathbf{in} \\
&\mathbf{let} f = \mathbf{let}\,!\, g' = g\ \mathbf{in} \lambda x \to x + g' \\
&\mathbf{in}\ 0
\end{aligned}$$

However, this is not entirely correct. The variable g is not relevant to the lambda itself but only to the body of the lambda. The correct transformed expression must be:

$$\begin{aligned}
&\mathbf{let}\ g = 5 + 5\ \mathbf{in} \\
&\mathbf{let} f = \lambda x \to \mathbf{let}\,!\, g' = g\ \mathbf{in}\ x + g' \\
&\mathbf{in}\ 0
\end{aligned}$$

This is a problem since the resetting of the relevance for the right hand side of the let binding causes the lambda to propagate too much information. The evaluaton list that is determined at the let should be pushed back into the right hand side of the binding. This is done using discrepancy lists which is explained in the following section.

6.1.2 Discrepancy list

The idea of the discrepancy list is similar to that of the evaluation list. For every variable that is in scope, it is detected when the relevance has changed between the input and output environment. Such a change in relevance is called a discrepancy. The difference is that the discrepancy list can also be propagated to a subexpression when this is needed. This is useful for let expressions, since now the calculated discrepancy list can be pushed deeper into the right hand side of a binding when this is needed. The general idea is to check whether an expression is saturated or not. When an expression is saturated, the discrepancy list becomes the evaluation list because the variables can be evaluated. Otherwise, the discrepancy list is propagated into the expression to determine the right place for the evaluation.

In the following definition of *discTrans*, the evaluation lists are expressed by adding a **let**! for every variable in the evaluation list. The function *discrTrans* takes an expression and a discrepancy list as input. The result is the transformed expression.

The definition of the *discTrans* is given in Fig. 6.1.

$\boxed{\mathit{discrTrans}\ (e,[x]) = e}$

```
$\mathit{discrTrans}\ (c, ds) = c$
$\mathit{discrTrans}\ (n, ds) = n$
$\mathit{discrTrans}\ (x, ds) = \mathbf{if}\ x \in ds\ \mathbf{then\ let}\ !\, x' = x\ \mathbf{in}\ x'\ \mathbf{else}\ x$
$\mathit{discrTrans}\ (\mathit{tup\ tag}, ds) = \mathit{tup\ tag}$
$\mathit{discrTrans}\ (\mathit{FFI}\ x, ds) = \mathit{FFI}\ x$
$\mathit{discrTrans}\ (\lambda x \to e, ds) = \lambda x \to (\mathit{discrTrans}\ (e, (\mathit{tyEnvDiff}\ \widehat{\Gamma}_{lam}\ \widehat{\Gamma}_{e}) +\!\!+\ ds))$
$\mathit{discrTrans}\ (e_1\ e_2, ds) =$
  $\mathbf{if}\ \mathit{satl}\ e_1 < 2$
  $\mathbf{then}\ (\mathit{discrTrans}\ (e_1, \mathit{filter}\ (\lambda x \to (\mathit{relval}\ \widehat{\Gamma}_{e1}\ [x]) \equiv \mathsf{S})\ ds))$
        $(\mathit{discrTrans}\ (e_2, \mathit{filter}\ (\lambda x \to (\mathit{relval}\ \widehat{\Gamma}_{e2}\ [x]) \equiv \mathsf{S})\ ds))$
  $\mathbf{else}\ \ \mathit{addEval}\ (ds, e_1\ e_2)$
```

Figure 6.1: Transformation where discrepancy lists are used.

```
$discrTrans\ (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2, ds)\ =$
  $\mathbf{let}\ (\widehat{\tau}_x, \varphi_x, sat_{lx}, sat_{rx}) = \widehat{\Gamma}_{e2}\ [x]$
     $(e_1', isRel) = \mathbf{if}\ \varphi_x \equiv \mathsf{S} \land sat_{rx} \geqslant sat_{lx} \land satl\ e_2 < 1$
                 $\mathbf{then}\ (discrTrans\ (e_1, [\,]), True)$
                 $\mathbf{else}\ \ (discrTrans\ (e_1, tyEnvDiff\ \widehat{\Gamma}_{let}\ \widehat{\Gamma}_{e1}), False)$
     $e_2' = \mathbf{if}\ satl\ e_2 < 1$
          $\mathbf{then}\ discrTrans\ (e_2, [\,])$
          $\mathbf{else}\ discrTrans\ (e_2, ds)$
     $e_{let} =\ \mathbf{if}\ isRel$
             $\mathbf{then}\ \mathbf{let!}\ x = e_1'\ \mathbf{in}\ e_2'$
             $\mathbf{else}\ \ \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2'$
  $\mathbf{in}\ \mathbf{if}\ satl\ e_2 < 1$
     $\mathbf{then}\ addEval\ (ds, e_{let})$
     $\mathbf{else}\ \ e_{let}$
$discrTrans\ (\mathbf{let!}\ x = e_1\ \mathbf{in}\ e_2, ds) =$
  $\mathbf{let}\ (\widehat{\tau}_x, \varphi_x, sat_{lx}, sat_{rx}) = \widehat{\Gamma}_{e2}\ [x]$
     $(e_1', isRel) = \mathbf{if}\ \varphi_x \equiv \mathsf{S} \land sat_{rx} \geqslant sat_{lx} \land satl\ e_2 < 1$
                 $\mathbf{then}\ (discrTrans\ (e_1, [\,]), True)$
                 $\mathbf{else}\ \ (discrTrans\ (e_1, tyEnvDiff\ \widehat{\Gamma}_{let}\ \widehat{\Gamma}_{e1}), False)$
     $e_2' = \mathbf{if}\ satl\ e_2 < 1$
          $\mathbf{then}\ discrTrans\ (e_2, [\,])$
          $\mathbf{else}\ \ discrTrans\ (e_2, ds)$
     $e_{let} = \mathbf{let!}\ x = e_1'\ \mathbf{in}\ e_2'$
  $\mathbf{in}\ \mathbf{if}\ satl\ e_2 < 1$
     $\mathbf{then}\ addEval\ (ds, e_{let})$
     $\mathbf{else}\ \ e_{let}$
$discrTrans\ (\mathbf{letrec}\ [x_0 = e_0, ..., x_n = e_n]\ \mathbf{in}\ e, ds) =$
  $\mathbf{let}\ \forall i. 0 \leqslant i \leqslant n$
     $(\widehat{\tau}_{xi}, \varphi_{xi}, sat_{rxi}, sat_{lxi}) = \widehat{\Gamma}_e\ [x_i]$
     $e_i' =\quad \mathbf{if}\ \varphi_{xi} \equiv \mathsf{S} \land sat_{lxi} \geqslant sat_{rxi} \land satl\ e < 1$
             $\mathbf{then}\ discrTrans\ (e_i, [\,])$
             $\mathbf{else}\ \ discrTrans\ (e_i, tyEnvDiff\ \widehat{\Gamma}_{let}\ \widehat{\Gamma}_{ei})$
     $e_2' = \mathbf{if}\ satl\ e < 1$
          $\mathbf{then}\ discrTrans\ (e_2, [\,])$
          $\mathbf{else}\ \ discrTrans\ (e_2, ds)$
     $e_{let} = \mathbf{letrec}\ [x_0 = e_0', ..., x_n = e_n']\ \mathbf{in}\ e_2'$
  $\mathbf{in}\ \mathbf{if}\ satl\ e < 1$
     $\mathbf{then}\ addEval\ (ds, e_{let})$
     $\mathbf{else}\ \ e_{let}$
$discrTrans\ (\mathbf{case}\ e\ \mathbf{of}\ [p_0 \to e_0, ..., p_n \to e_n], ds) =$
  $\mathbf{let}\ for\ 0 \leqslant i \leqslant n$
     $ds_i = tyEnvDiff\ \widehat{\Gamma}_{case}\ \widehat{\Gamma}_{ei}$
  $\mathbf{in}\ \mathbf{if}\ max\ [sat_l\ p_0, ..., sat_l\ p_n] > 0$
     $\mathbf{then}\ \mathbf{case}\ e\ \mathbf{of}\ [p_0 \to discrTrans\ (e_0, ds_0 +\!\!+ ds), ..., p_n \to discrTrans\ (e_n, ds_n +\!\!+ ds)]$
     $\mathbf{else}\ \mathbf{let}\ e' = \mathbf{case}\ e\ \mathbf{of}\ [p_0 \to discrTrans\ (e_0, ds_0), ..., p_n \to discrTrans\ (e_n, ds_n)]$
         $\mathbf{in}\ addEval\ (ds, e')$
```

Figure 6.2: Transformation where discrepancy lists are used (Continued).

For the defintion of *discrTrans*, it is assumed that the function *satl* is given which determines the sat_l of an expression. This does the same as the calculation of the sat_l in the definition of $\mathscr{W}_{rel}$. Also the $\widehat{\Gamma}$ is used in combination with expression names in subscript (for example $\widehat{\Gamma}_{case}$). This is just the output $\widehat{\Gamma}$ from the inference algorithm for the correspondig expression. The function *addEval* is used which takes an expression and a list of variables which have to be evaluated by a let!. The result is an expression where these evaluations have been added including substitutions.

$$\begin{aligned} &addEval\,(e,(d:ds)) = \textbf{let}\,!\,d' = d\,\textbf{in}\,[d \mapsto d']\,(addEval\,(e,ds)) \\ &addEval\,(e,[\,]) \qquad\; = e \end{aligned}$$

In the definiction of *addEval*, substitutions are applied to the variables in an expression. The used syntax is $[x_1 \mapsto x_2]e$. Here the variable x_1 is replaced by the variable x_2 in the expression e.

The definition of *discrTrans* is intertwined with $\mathscr{W}_{rel}$ (especially the *satl* and $\widehat{\Gamma}$), so the actual implementation in AG is done together. In the AG, the expressions is annotated with evaluation lists and the evaluation is done in a separate step of the transformation.

The function *discrTrans* is explained for all the expressions in the following sections.

Constants, variables, FFI's and tags

For the transformation of constants, nothing special needs to be done. Constants are always strict since they cannot be evaluated any further. That is why no extra evaluation needs to be added.

For variables, the incoming discrepancy list should be checked if the variable is an element of the list. When it is present in the list, the variable can be evaluated at this place. For example for a variable x where the incoming discrepancy list is $[x]$, x has to be evaluated at this place. The resulting code is $\textbf{let}!\,x' = x\,\textbf{in}\,x'$.

When the variable is not in the incoming discrepancy list, nothing has to be done. So when a variable x occurs and the incoming discrepancy is empty, the expression x is the result.

For *FFI*'s and *tags*, nothing has to be done. These expressions can be treated as constants because no extra evaluation can be done in these expressions.

Lambdas and applications

When the input discrepancy list for a lamba is non-empty, the discrepancy is caused by the body of the lambda. As a result, all the discrepancies are propagated into the body of the lambda. Consider, for example the simple lambda function $\lambda x \to ten$, where ten is in scope and the input discrepancy list is $[ten]$. This discrepancy is propagated into the body of the lambda since that is the only place where it could have came from. In the body, the variable ten is encountered right away. As a result, it can be evaluated at that place. The resulting function becomes $\lambda x \to \textbf{let}!\,ten' = ten\,\textbf{in}\,ten'$.

For an application, it has to be determined whether it is saturated. If it is saturated, an actual calculation takes place. This is why the elements of the discrepancy list can be evaluated at this place. Consider, for example the application $ten - 5$ where *ten* is in scope and the input discrepancy list is $[ten]$. This is a saturated application since the subtraction operator gets both its arguments. As a result, the evaluation of the variables in the discrepancy list can be done right away. The transformed code is $\mathbf{let!}\ ten' = ten\ \mathbf{in}\ ten' - 5$.

With non-saturated applications, the source of the discrepancy has to be found. This is done by using the output $\widehat{\Gamma}$'s of the function and the argument. For each of the two, the variables from the incoming discrepancy list are looked up in the type environment. Only when it is relevant in a $\widehat{\Gamma}$ of a subexpression, the discrepancy is propagated to the discrepancy list of the corresponding subexpression. Consider the following example: $(\lambda x\ y \to x + 5)\ ten$, where *ten* is in scope and the discrepancy list is $[ten]$. Now it has to be calculated from which part of the application (the function or the argument), the discrepancy arises. When looking at the $\widehat{\Gamma}$ of the function, it can be determined that *ten* is not relevant to the function, so the variable is not propagated into the discrepancy list of the function. After performing a lookup for *ten* on the $\widehat{\Gamma}$ of the argument, it can be seen that it is relevant to the argument so it must be propagated into the discrepancy list of the argument. There it is immediately concluded that it can be evaluated. The transformed code is $(\lambda x\ y \to x + 5)\ (\mathbf{let!}\ ten' = ten\ \mathbf{in}\ ten')$.

Let bindings

For a let binding, the bound variable is not guaranteed to be used in the body. Here a discrepancy will be created for all the variables that are relevant in the right hand side of the binding when the bound variable is not used in a relevant and saturated manner in the body. This discrepancy list is propagated to the right hand side to be resolved.

The let binding also has an input discrepancy list. The elements of this list can be evaluated when the body has a sat_l of zero. If this is not the case (so the body is not saturated), the discrepancy must be propagated to the body of the let binding.

Consider the example:

$$\begin{array}{l}\mathbf{let}\ g = 5 + 5\ \mathbf{in}\\ \mathbf{let}\ f = \lambda x \to x + g\\ \mathbf{in}\ 0\end{array}$$

The calculation of the discrepancy list for the right hand side of f results in $[g]$. This is then propagated into the right hand side of f to be resolved. The result is:

$$\begin{array}{l}\mathbf{let}\ g = 5 + 5\ \mathbf{in}\\ \mathbf{let}\ f = \lambda x \to \mathbf{let!}\ g' = g\ \mathbf{in}\ x + g'\\ \mathbf{in}\ 0\end{array}$$

Also, when the bound variable is used saturated in a relevant context, a plain let binding can be transformed into a strict let binding. For example in the following expression:

$$\begin{array}{l}\mathbf{let}\ ten = 5 + 5\ \mathbf{in}\\ \mathbf{let}\ f = \lambda x\ y \to x\\ \mathbf{in}\ f\ ten\ 5\end{array}$$

The variable *ten* is used in a relevant context because the function f is strict in its first argument, and the function gets all of its arguments. Now it can be concluded that *ten* can be evaluated when it is defined. The resulting expression is:

$$\begin{array}{l}\textbf{let}!\ \mathit{ten} = 5+5\ \textbf{in}\\ \textbf{let}\ f = \lambda x\ y \to x\\ \textbf{in}\ f\ \mathit{ten}\ 5\end{array}$$

Note that f is also in a relevant context in the body, but evaluation has no consequences since it is a lambda.

Strict let bindings are similar to plain let bindings. The only difference is that it does not have to be transformed into a strict binding when it is found to be relevant and saturated in the body since it is already known to be strict.

Recursive let bindings are also handled similarly. The difference with respect to the plain let binding is that it has a list of bindings. So for each binding in the list, the same has to be done as is done for a binding in a plain let binding. Also no transformation into a strict binding has to be done as it cannot be strict and recursive at the same time.

Cases

For a case expression, first the discrepancy lists for the case alternatives have to be calculated. Here the $\widehat{\Gamma}$ of each of the arms is compared against the $\widehat{\Gamma}$ of the complete case expression. The outcoming discrepancy lists are propagated to the corresponding case arms.

Second, the input discrepancy lists of the case arms depend on the saturatedness of the expressions in the case alternatives. If they are saturated, the variables in the incoming discrepancy list can be evaluated immediately. For example in the following expression:

$$\begin{array}{lll}\textbf{let}\ \mathit{ten} & = 5+5\ \textbf{in} & \\ \textbf{let}\ \mathit{five} & = 2+3\ \textbf{in} & \\ \textbf{let}\ v & = \textbf{case}\ \mathit{True}\ \textbf{of} & \\ & \quad\quad \mathit{True} \to \mathit{ten} - \mathit{five} & \\ & \quad\quad \mathit{False} \to \mathit{ten} & \\ \textbf{in}\ 0 & & \end{array}$$

the discrepancy list of v contains *ten*. Since the arms of the case expression are saturated, *ten* can be evaluated before the case expression. When calculating the difference between $\widehat{\Gamma}$ of the first case alternative and the $\widehat{\Gamma}$ of the case expression, it can be concluded that *five* needs to be added to discrepancy list of the first case alternative. This is a saturated application, so *five* is evaluated immediately in the first case arm. So the transformed expression is:

$$\begin{array}{ll}\textbf{let}\ \mathit{ten} & = 5+5\ \textbf{in}\\ \textbf{let}\ \mathit{five} & = 2+3\ \textbf{in}\\ \textbf{let}\ v & = \textbf{let}\,!\,\mathit{ten} = \mathit{ten}'\\ & \quad \textbf{in case}\ \mathit{True}\ \textbf{of}\\ & \quad\quad \mathit{True} \to \textbf{let}\,!\mathit{five}' = \mathit{five}\ \textbf{in}\ \mathit{ten}' - \mathit{five}'\\ & \quad\quad \mathit{False} \to \mathit{ten}'\\ \textbf{in}\ 0 & \end{array}$$

When the case arms are not saturated, the discrepancy list is propagated to each of the case arms. This is done since in the analysis it is already determined that all the variables in the discrepancy list are relevant in all of the case arms. Now, the correct places for the evaluation is found when the expressions in the case arms are given the discrepancy list. For example, in the expression:

$$\begin{array}{l} \textbf{let}\, ten \;\;= 5+5\ \textbf{in} \\ \textbf{let}\, plus = \lambda x\, y \to x+y\ \textbf{in} \\ \textbf{let}\, min \;= \lambda x\, y \to x-y\ \textbf{in} \\ \textbf{let}\, f \quad\;\; = \textbf{case}\ True\ \textbf{of} \\ \qquad\qquad\quad True \;\;\to plus\ ten \\ \qquad\qquad\quad False \to min\ ten \\ \textbf{in}\ 0 \end{array}$$

the discrepancy list of *f* contains *ten*. This list is propagated downwards since the case arms are not saturated. This discrepancy list is propagated to the variable *ten* in both case arms, where it is evaluated. The result becomes:

$$\begin{array}{l} \textbf{let}\, ten \;\;= 5+5\ \textbf{in} \\ \textbf{let}\, plus = \lambda x\, y \to x+y\ \textbf{in} \\ \textbf{let}\, min \;= \lambda x\, y \to x-y\ \textbf{in} \\ \textbf{let}\, f \quad\;\; = \textbf{case}\ True\ \textbf{of} \\ \qquad\qquad\quad True \;\;\to plus\ (\textbf{let}\,!\, ten' = ten\ \textbf{in}\ ten') \\ \qquad\qquad\quad False \to min\ (\textbf{let}\,!\, ten' = ten\ \textbf{in}\ ten') \\ \textbf{in}\ 0 \end{array}$$

Another thing that is important for the case expressions are the variables from the patterns. Since the pattern variables are still in $\widehat{\Gamma}$, they are found when calculating the difference between the two $\widehat{\Gamma}$'s. In this way, the variables are added to the discrepancy list of the expression of the same case arm when this is necessary. For example, consider:

$$\begin{array}{l} \textbf{let}\, f = \textbf{case}\ Just\ 5\ \textbf{of} \\ \qquad\qquad Nothing \to 20 \\ \qquad\qquad Just\ x \quad\; \to x+5 \\ \textbf{in}\ 0 \end{array}$$

For the second case arm, it is determined that *x* is relevant. Now, the variable *x* is added to the discrepancy list of this case alternative. There is a saturated application, so it is evaluated at that place. The result of the transformation is:

$$\begin{array}{l} \textbf{let}\, f = \textbf{case}\ Just\ 5\ \textbf{of} \\ \qquad\qquad Nothing \to 20 \\ \qquad\qquad Just\ x \quad\; \to \textbf{let}!\, x' = x\ \textbf{in}\ x'+5 \\ \textbf{in}\ 0 \end{array}$$

The evaluation of these pattern variables can be optimized by taking the strictness annotations on constructor fields into account. The variables that correspond to strict data fields do not have to be evaluated when they are present in a discrepancy list. This is because they are already evaluated at the construction of the data type value. This optimization can be performed when the unnecessary let!'s are removed.

6.2 Generate and add calls to wrappers

The generation of wrappers and adding the calls to these wrappers is the second step in the transformation. As discussed in Section 5.3, something has to be done for higher order functions. This is done with a worker/wrapper transformation[15]. The workers are the original functions and wrappers take care of the evaluation.

6.2.1 Generate wrappers

For higher order functions, a different version of the function must be generated. When a function is strict, it expects an evaluated argument. This cannot be guaranteed when it is used higher order. That is why a wrapper is generated which does the evaluation of the lazy arguments which are expected to be strict in the normal function. Consider for example the following piece of code:

$$\begin{aligned}&\textbf{let}\ id\ \ = \lambda x \to x\ \textbf{in}\\ &\textbf{let}\ app = \lambda f\ x \to f\ x\\ &\textbf{in}\ app\ id\ (5+5)\end{aligned}$$

The inferred type for id is $\widehat{\tau} \overset{\mathsf{S}}{\to} \widehat{\tau}$, while the type of app is $(\widehat{\tau} \overset{\mathsf{L}}{\to} \widehat{\tau}) \overset{\mathsf{S}}{\to} \widehat{\tau} \overset{\mathsf{L}}{\to} \widehat{\tau}$. From the type of *app* it can be seen that a safe estimation is made because the input function is lazy. But when a strict function is used as the first argument of *app* (like id), this function may get an unevaluated argument. To prevent this from happening, a wrapper is generated which is lazy but evaluates the thunks that can be passed as strict arguments to the normal function which is referred to as the worker. A wrapper is generated for each binding with a lambda on the right hand side. This is shown in the transformed version of the example which contain the functions id_{wrap} and app_{wrap}:.

$$\begin{aligned}&\textbf{let}\ id &&= \lambda x \to x\ \textbf{in}\\ &\textbf{let}\ id_{wrap} &&= \lambda x \to \textbf{let}\,!\,x' = x\ \textbf{in}\ id\ x'\ \textbf{in}\\ &\textbf{let}\ app &&= \lambda f\ x \to f\ x\\ &\textbf{let}\ app_{wrap} &&= \lambda f\ x \to app\,f\ x\\ &\textbf{in}\ app\ id\ (5+5)\end{aligned}$$

For partial applications on the right hand side of a let binding, wrappers can also be generated. When the wrappers for partial applications are generated, lambdas are introduced to do the evaluation. These lambdas are not guaranteed to be on the top-level, because partial applications can also occur in a local definition. So these new lambdas will break the invariant that the code is lambda lifted. By making only wrappers for lambdas, the code is still lambda lifted because the wrappers are also on the top-level.

A possible solution for this is to do lambda lifting after the generation of the wrappers. The wrapper and the function must be lifted to the same level to keep them together. Because the arity of the functions can become higher after lambda lifting, the functionality of the wrapper can break because nothing is known about the relevance of the argument that is added to the function.

To keep things easy, we have chosen to replace calls to lambdas in partial applications with the calls to the top-level wrappers. In this way, the partial applications can be safely used as a higher order argument and no wrappers have to be generated for these. The adding of the calls to the wrappers is explained further in Section 6.2.2.

The function *genWrap* adds all the wrappers for the lambdas. It takes an expression and returns the same expression where the wrappers are added.

The function *genWrap* is defined in Fig. 6.3.

$\boxed{genWrap\ (e) = e}$

```
$genWrap\ (z) = z$
$genWrap\ (c) = c$
$genWrap\ (x) = x$
$genWrap\ (tup\ tag) = tup\ tag$
$genWrap\ (FFI\ x) = FFI\ x$
$genWrap\ (\lambda x \rightarrow e) = \lambda x \rightarrow (genWrap\ e)$
$genWrap\ (e_1\ e_2) = (genWrap\ e_1)\ (genWrap\ e_2)$
$genWrap\ (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) =$
  $\mathbf{if}\ isLam\ e_1$
  $\mathbf{then\ let}\ x = e_1\ \mathbf{in\ let}\ x_{wrap} = wrap\ (relTy\ e_1, x)\ \mathbf{in}\ (genWrap\ e_2)$
  $\mathbf{else\ \ let}\ x = e_1\ \mathbf{in}\ (genWrap\ e_2)$
$genWrap\ (\mathbf{let!}\ x = e_1\ \mathbf{in}\ e_2) =$
  $\mathbf{if}\ isLam\ e_1$
  $\mathbf{then\ let!}\ x = e_1\ \mathbf{in\ let}\ x_{wrap} = wrap\ (relTy\ e_1, x)\ \mathbf{in}\ (genWrap\ e_2)$
  $\mathbf{else\ \ let!}\ x = e_1\ \mathbf{in}\ (genWrap\ e_2)$
$genWrap\ (\mathbf{letrec}\ [x_1 = e_1, ..., x_n = e_n]\ \mathbf{in}\ e) =$
  $\mathbf{let}\ bs = [\,]$
    $foreach\ x_i = e_i\ \mathbf{in}\ [x_1 = e_1, ..., x_n = e_n]$
      $bs = \mathbf{if}\ isLam\ e_i$
          $\mathbf{then}\ [x_i = e_i, x_{wrap} = wrap\ (relTy\ e_i, x)]$
          $\mathbf{else}\ \ [x_i = e_i]$
  $\mathbf{in\ letrec}\ bs\ \mathbf{in}\ (genWrap\ e)$
$genWrap\ (\mathbf{case}\ e\ \mathbf{of}\ [p_1 \rightarrow e_1, ..., p_n \rightarrow e_n]) =$
  $\mathbf{case}\ e\ \mathbf{of}\ [p_1 \rightarrow (genWrap\ e_1), ..., p_n \rightarrow (genWrap\ e_n)]$
```

Figure 6.3: Generating the wrapper functions.

Some auxiliary functions are used. The function *isLam* returns *True* when the input expression is a lambda and *False* for any other expression. Also, the function *relTy* is used, which returns the $\widehat{\tau}$ of the input expression. Additionally, the function *wrap* is used to generate the actual wrapper, which takes a $\widehat{\tau}$ and a function name as an input. For example for a function f with a $\widehat{\tau}$ of $() \xrightarrow{\mathsf{S}} () \xrightarrow{\mathsf{L}} () \xrightarrow{\mathsf{S}} ()$, the generated wrapper is: $\lambda x\, y\, z \rightarrow \mathbf{let}\,!x' = x\ \mathbf{in}\ \mathbf{let}\,!z' = z\ \mathbf{in}\, f\ x'\ y\ z'$.

When looking at the definition of *genWrap*, it can be seen that the only work is done when any kind of let binding is found where the right hand side is a lambda abstraction. The $\widehat{\tau}$ of the right hand side is calculated with which the wrapper is generated. This wrapper is added to the expression.

6.2.2 Add calls to wrappers

For adding the calls to the wrappers, first the names of the variables which have a wrapper must be collected. When such a variable is used as an argument in an application, the call to the normal variable must be changed into a call to the wrapper. Additionally, calls to the wrappers are added in the right hand side of a binding when this rhs is a partial application.

Consider the example from Section 6.2.1:

$$\begin{array}{ll}
\textbf{let}\ id & = \lambda x \to x\ \textbf{in} \\
\textbf{let}\ id_{wrap} & = \lambda x \to \textbf{let}!\ x' = x\ \textbf{in}\ id\ x'\ \textbf{in} \\
\textbf{let}\ app & = \lambda f\ x \to f\ x \\
\textbf{let}\ app_{wrap} & = \lambda f\ x \to app\ f\ x \\
\textbf{in}\ app\ id\ (5+5) &
\end{array}$$

Here, *id* is used as the first argument of *app*. As a result, *id* is replaced by a call to the wrapper of *id*, id_{wrap}. The transformed expression is:

$$\begin{array}{ll}
\textbf{let}\ id & = \lambda x \to x\ \textbf{in} \\
\textbf{let}\ id_{wrap} & = \lambda x \to \textbf{let}!\ x' = x\ \textbf{in}\ id\ x'\ \textbf{in} \\
\textbf{let}\ app & = \lambda f\ x \to f\ x \\
\textbf{let}\ app_{wrap} & = \lambda f\ x \to app\ f\ x \\
\textbf{in}\ app\ id_{wrap}\ (5+5) &
\end{array}$$

Also for partial applications in right hand sides, calls to wrappers have to be added as was discussed in Section 6.2.1. Consider the following example where the wrappers are already present:

$$\begin{array}{ll}
\textbf{let}\ ten & = 5+5 \\
\textbf{let}\ f & = \lambda x\ y \to x+y\ \textbf{in} \\
\textbf{let}\ f_{wrap} & = \lambda x\ y \to \textbf{let}!\ x' = x\ \textbf{in}\ \textbf{let}!\ y' = y\ \textbf{in}\ f\ x'\ y'\ \textbf{in} \\
\textbf{let}\ g & = f\ ten \\
\textbf{in}\ 0 &
\end{array}$$

Here, the right hand side of *g* contains a partial application. As a result, the calls to the lambdas have to be replaced by the calls to wrappers of the lambdas. The resulting expression is:

$$\begin{array}{ll}
\textbf{let}\ ten & = 5+5 \\
\textbf{let}\ f & = \lambda x\ y \to x+y\ \textbf{in} \\
\textbf{let}\ f_{wrap} & = \lambda x\ y \to \textbf{let}!\ x' = x\ \textbf{in}\ \textbf{let}!\ y' = y\ \textbf{in}\ f\ x'\ y'\ \textbf{in} \\
\textbf{let}\ g & = f_{wrap}\ ten \\
\textbf{in}\ 0 &
\end{array}$$

Adding the calls to the wrappers is performed by the function *addWrapCalls*. It takes an expression, a list of variables which have a wrapper and a boolean. This boolean determines if the normal calls should be replaced by calls to the wrapper for the input expression.

The function *addWrapCalls* is defined in Fig. 6.4

```
                                        $\boxed{addWrapCalls\ (e,[x],Bool) = e}$

$addWrapCalls\ (z,ws,b) = z$
$addWrapCalls\ (c,ws,b) = c$
$addWrapCalls\ (x,ws,b) = \textbf{if}\ b \wedge x \in ws\ \textbf{then}\ x_{wrap}\ \textbf{else}\ x$
$addWrapCalls\ (tup\ tag,ws,b) = tup\ tag$
$addWrapCalls\ (FFI\ x,ws,b) = FFI\ x$
$addWrapCalls\ (\lambda x \to e,ws,b) = \lambda x \to (addWrappCalls\ (e,ws,b))$
$addWrapCalls\ (e_1\ e_2,ws,b) = (addWrapCalls\ (e_1,ws,b))$
                                 $(addWrapCalls\ (e_2,ws,True))$
$addWrapCalls\ (\textbf{let}\ x = e_1\ \textbf{in}\ e_2,ws,b) =$
  $\textbf{let}\ e'_1\ = \textbf{if}\ \neg\ (isWrap\ x) \wedge \neg\ (isLam\ e_1) \wedge (satl\ e_1 > 0)$
           $\textbf{then}\ (addWrapCalls\ (e_1,w,True))$
           $\textbf{else}\ \ e_1$
      $ws' = \textbf{if}\ isLam\ e_1 \wedge \neg\ (isWrap\ e_1)$
           $\textbf{then}\ x : ws$
           $\textbf{else}\ \ ws$
  $\textbf{in let}\ x = e'_1\ \textbf{in}\ (addWrapCalls\ (e_2,ws',b))$
$addWrapCalls\ (\textbf{let!}\ x = e_1\ \textbf{in}\ e_2,ws,b) =$
  $\textbf{let}\ e'_1\ = \textbf{if}\ \neg\ (isWrap\ x) \wedge \neg\ (isLam\ e_1) \wedge (satl\ e_1 > 0)$
           $\textbf{then}\ (addWrapCalls\ (e_1,ws +\!\!+ ws',True))$
           $\textbf{else}\ \ e_1$
      $ws' = \textbf{if}\ isLam\ e_1 \wedge \neg\ (isWrap\ e_1)$
           $\textbf{then}\ x : ws$
           $\textbf{else}\ \ ws$
  $\textbf{in let}\ x = e'_1\ \textbf{in}\ (addWrapCalls\ (e_2,ws',b))$
$addWrapCalls\ (\textbf{letrec}\ [x_0 = e_0,...,x_n = e_n]\ \textbf{in}\ e,ws,b) =$
  $\textbf{let}\ ws' = [\,]$
      $\forall i.0 \leqslant i \leqslant n$
        $e'_i\ \ = \textbf{if}\ \neg\ (isWrap\ x_i) \wedge \neg\ (isLam\ e_i) \wedge (satl\ e_i > 0)$
             $\textbf{then}\ (addWrapCalls\ (e_i,ws +\!\!+ ws',True))$
             $\textbf{else}\ \ e_i$
        $ws' = \textbf{if}\ isLam\ e_i \wedge \neg\ (isWrap\ x_i)$
             $\textbf{then}\ x_i : ws'$
             $\textbf{else}\ \ ws'$
  $\textbf{in letrec}\ [x_0 = e'_0,...,x_n = e'_n]\ \textbf{in}\ (addWrapCalls\ (e,ws +\!\!+ ws',b))$
$addWrapCalls\ (\textbf{case}\ e\ \textbf{of}\ [p_0 \to e_0,...,p_n \to e_n],ws,b) =$
  $\textbf{case}\ (addWrapCalls\ (e,ws,b))\ \textbf{of}$
    $[p_0 \to (addWrapCalls\ (e_0,ws,b)),...,p_n \to (addWrapCalls\ (e_n,ws,b))]$
```

Figure 6.4: Adding the calls to the wrappers.

For this definition, some auxiliary functions are used. The function *isWrap* gets a variable and determines if this variable corresponds to a wrapper. The function *isLam* takes an expression. It returns *True* if the expression is a lambda and *False* for any other expression. In addition, the function *satl* is used to determine the sat_l of an expression.

The function *addWrapCalls* only adds calls to the wrapper when the input boolean is *True*. This can be seen at the definition of variables, where a variable is only changed into a wrapper variable when the input boolean is *True* and the name of the variable is in the list of wrapper names. For the handling of lambdas, nothing special is done. All the input (boolean and wrapper list) is propagated to the body of the lambda since no variable can be changed. For the application, the boolean for the argument is always set to *True*. This is because the higher order arguments must always be transformed into a call to the wrapper. The function part of the application gets the same input as the application itself. The let bindings is where the list of the wrapper names is filled. This is done with the list ws' which gets the name of the binding added to it when the right hand side is a lambda, but not a wrapper. In this way, the list of wrappers contains the original names and the lookup of the variable in the list of wrapper names can be done directly. Additionally, the boolean of the right hand side of the binding is set to *True* when this is a partial application. For the case expression, the input boolean and wrapper list is just propagated to the scrutinee and all the case alternatives.

6.3 Removing unnecessary let!'s

After the let!'s are introduced, it can be the case that a variable that is evaluated by a let! (so a variable on the right hand side of the let!) is already evaluated previously. That a variable is already evaluated can be derived from a previous let!, a parameter of a lambda which is relevant or from variables in a pattern which match strictness annotated constructor fields.

When a let! is encountered and right hand side only contains a variable, it should be determined if it is already evaluated. Consider the following expression:

$$\begin{array}{l}\mathbf{let}!\,\mathit{five} = 3+3\\ \mathbf{in}\;\mathbf{let}!\,\mathit{five}' = \mathit{five}\;\mathbf{in}\;\mathit{five}'+3\end{array}$$

Here it can be seen that *five* is evaluated when it is introduced, because a let! is used. But after that, *five* is evaluated again in the second binding. Here it can be determined that *five* is already evaluated by the first binding, so the second let! can be removed. The transformed code is:

$$\begin{array}{l}\mathbf{let}!\,\mathit{five} = 3+3\\ \mathbf{in}\;\mathbf{let}\,\mathit{five}' = \mathit{five}\;\mathbf{in}\;\mathit{five}'+3\end{array}$$

For lambda abstractions, a relevance type is determined during the analysis. When a function is strict in an argument, it can be safely assumed that the argument that is supplied is already evaluated. This information can also be used to determine if a let! is unnecessary. For example in the following function:

$$\lambda x\; y \to \mathbf{let}!\; x' = x\;\mathbf{in}\; y$$

This function obtains the type $() \xrightarrow{\mathsf{S}} () \xrightarrow{\mathsf{S}} ()$. So it can be concluded that x and y are already evaluated in the body. As a result, the strict let binding for x can be transformed into a plain binding. The resulting function is:

$$\lambda x\; y \to \mathbf{let}\; x' = x\;\mathbf{in}\; y$$

As is previously stated, some evaluation of the variables in the patterns can be removed. When a variable in a pattern matches with a constructor field which has a strictness annotation on it, it can be concluded that the variable is already evaluated. Consider the data type *StrictMaybe*:

$$\textbf{data}\ \mathit{StrictMaybe}\ a = \mathit{StrictJust}\ !\, a \mid \mathit{StrictNothing}$$

When a variable is matched against the first field of *StrictJust*, it can be concluded that it is already evaluated. For example in the following case expression:

$$\begin{array}{l}\textbf{case}\ x\ \textbf{of}\\ \quad \mathit{StrictJust}\ y \quad \rightarrow \textbf{let}\ !\, y' = y\ \textbf{in}\ y' + 10\\ \quad \mathit{StrictNothing} \rightarrow 5\end{array}$$

For the variable y it can be concluded that it is already evaluated because of the strictness annotation on the constructor field. That is why the let! can be transformed into a normal let. The resulting expression is:

$$\begin{array}{l}\textbf{case}\ x\ \textbf{of}\\ \quad \mathit{StrictJust}\ x \quad \rightarrow \textbf{let}\ x' = x\ \textbf{in}\ x' + 10\\ \quad \mathit{StrictNothing} \rightarrow 5\end{array}$$

All these parts are incorporated into the function *removeLetBang*. This function takes a list of variables and an expression. The idea is that the list contains the variables that are already evaluated. When such a variable is encountered in the right hand side of let!, the let! is transformed into a plain let binding. The function returns the transformed version of the input expression.

The function *removeLetBang* is defined in Fig. 6.5.

$$\boxed{\mathit{removeLetBang}\ (e, [x]) = e}$$

$$\begin{array}{ll}
\mathit{removeLetBang}\ (z, vs) & = z\\
\mathit{removeLetBang}\ (c, vs) & = c\\
\mathit{removeLetBang}\ (x, vs) & = x\\
\mathit{removeLetBang}\ (\mathit{tup}\ \mathit{tag}, vs) & = \mathit{tup}\ \mathit{tag}\\
\mathit{removeLetBang}\ (\mathit{FFI}\ x, vs) & = \mathit{FFI}\ x\\
\mathit{removeLetBang}\ (\lambda x \rightarrow e, vs) = & \\
\quad \textbf{let}\ (\widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2) = \mathit{relTy}\ (\lambda x \rightarrow e) & \\
\qquad e' = \textbf{if}\ \varphi \equiv \mathsf{S} & \\
\qquad\qquad \textbf{then}\ \mathit{removeLetBang}\ (e, (x : vs)) & \\
\qquad\qquad \textbf{else}\ \mathit{removeLetBang}\ (e, vs) & \\
\quad \textbf{in}\ (\lambda x \rightarrow e') & \\
\mathit{removeLetBang}\ (e_1\ e_2, vs) = & \\
\quad (\mathit{removeLetBang}\ (e_1, vs))\ (\mathit{removeLetBang}\ (e_2, vs)) &
\end{array}$$

Figure 6.5: Removing the let!'s that are unnecessary.

```
$removeLetBang\ (\mathbf{let}\, x = e_1\ \mathbf{in}\ e_2, vs)\ =$
  $\mathbf{let}\ e'_1 = removeLetBang\ (e_1, vs)$
     $e'_2 = removeLetBang\ (e_2, vs)$
  $\mathbf{in\ let}\, x = e'_1\ \mathbf{in}\ e'_2$
$removeLetBang\ (\mathbf{let!}\, x = e_1\ \mathbf{in}\ e_2, vs) =$
  $\mathbf{let}\ e'_1 = removeLetBang\ (e_1, vs)$
     $e'_2 = removeLetBang\ (e_2, (x : vs))$
  $\mathbf{in\ if}\ isVar\ e_1$
     $\mathbf{then\ if}\ (getVar\ e_1) \in vs$
          $\mathbf{then}\ (\mathbf{let}\, x = e'_1\ \mathbf{in}\ e'_2)$
          $\mathbf{else}\ \ (\mathbf{let!}\, x = e'_1\ \mathbf{in}\ e'_2)$
     $\mathbf{else\ let!}\, x = e'_1\ \mathbf{in}\ e'_2$
$removeLetBang\ (\mathbf{letrec}\ [x_0 = e_0, ..., x_n = e_n]\ \mathbf{in}\ e, vs) =$
  $\mathbf{let}\ \forall i.0 \leqslant i \leqslant n$
        $e'_i = removeLetBang\ (e_i, vs)$
     $e' = removeLetBang\ (e, vs)$
  $\mathbf{in}\ (\mathbf{letrec}\ [x_0 = e'_0, ..., x_n = e'_n]\ \mathbf{in}\ e')$
$removeLetBang\ (\mathbf{case}\ e\ \mathbf{of}\ [p_0 \to e_0, ..., p_n \to e_n], vs) =$
  $\mathbf{let}\ e' = removeLetBang\ (e, vs)$
     $\forall i.0 \leqslant i \leqslant n$
        $v_i = strictVarsPattern\ (p_i)$
        $e'_i = removeLetBang\ (e_i, vs + v_i)$
  $\mathbf{in}\ (\mathbf{case}\ e'\ \mathbf{of}\ [p_0 \to e'_0, ..., p_n \to e'_n])$
```

Figure 6.6: Removing the let!'s that are unnecessary (Continued).

Some auxiliary functions are used in this definition. The function *relTy* is used to determine the relevance type of an expression. To determine whether an expression is a variable, the function *isVar* is used. To extract a variable from an expression, the function *getVar* is used. This is only used when *isVar* has returned true for that expression. For the patterns, the function *strictVarsPattern* is used to determine which variables are strict. For a constructor pattern, the tag is used to look up which fields are strict. When a pattern is a variable, this is also added to the strict variables since it is evaluated before the case expression.

6.4 Remove unnecessary let's

After removing the strictness on let bindings, normal let bindings remain. These plain binding can be simple renamings, so an extra transformation can be done to remove these. These two steps could be done together, but the second step is already available in UHC on the Core leve. In this way, the code is separated and easier to maintain.

Chapter 7

Conclusions

This last chapter presents the conclusions of this thesis and includes pointers for future work.

7.1 Conclusion

This thesis investigated how strictness analysis can be implemented in the context of a real Haskell compiler (in this case UHC). The UHC Core language was chosen for the language under analysis since this was the last lazy functional language in the pipeline of UHC. Inspired by the work of Holdermans and Hage [9] which used a type based approach to strictness analysis (relevance typing), a more refined type system was defined using saturation. The saturation in the typing rules was needed to handle both partial applications and user defined strictness annotations. For the saturation, two counters were introduced to keep track of the number of arguments that are needed by an expression and the number of arguments that are already passed to an expression.

The type system was used as a basis for a syntax directed implementation. For the actual implementation, this definition is transformed into AG code. In addition, a definition is given to handle recursion. Fixed point iteration is used to get more precise types for recursive bindings.

Finally, the transformation was presented which determines which variables can be evaluated at what places. For this, the annotated type environments from the analysis are used. In addition, wrappers are added which take care of the evaluation. These wrappers are used when a function is an argument to a higher order function.

7.2 Future work

In Chapter 5, a definition is given to get more precise results for determining the relevance types of recursive let bindings. The implementation of this in UHC is left as future work.

More strictness information can already be gained for higher order functions without using a polyvariant approach. When a higher order function is not exported, the join over all the call sites can be calculated to get the minimal strictness of the higher order function. So if all functions are strict in a certain parameter, it can be safely established that this also holds for the higher order function and this can be propagated into the type of the higher order function. For example consider that *hof* is not exported in the following expression:

$$\begin{array}{l}\textbf{let}\ \mathit{hof}\ \ = \lambda f\ x\ y \to f\ x\ y\ \textbf{in}\\ \textbf{let}\ \mathit{plus} = \lambda x\ y \to x + y\ \textbf{in}\\ \textbf{let}\ \mathit{snd}\ \ = \lambda x\ y \to x\\ \textbf{in}\ \mathit{hof}\ \mathit{plus}\ 1\ 2 + \mathit{hof}\ \mathit{snd}\ 1\ 2\end{array}$$

Here *plus* and *snd* obtain the types $() \xrightarrow{\mathsf{S}} () \xrightarrow{\mathsf{S}} ()$ and $() \xrightarrow{\mathsf{S}} () \xrightarrow{\mathsf{L}} ()$. Taking the join over these types, the result is $() \xrightarrow{\mathsf{S}} () \xrightarrow{\mathsf{L}} ()$. The type for *hof* can be: $(() \xrightarrow{\mathsf{S}} () \xrightarrow{\mathsf{L}} ()) \xrightarrow{\mathsf{S}} () \xrightarrow{\mathsf{S}} () \xrightarrow{\mathsf{L}} ()$, because both *plus* and *snd* are strict in their first argument.

For data types, extra annotations can be added. Instead of only one annotation which describes whether it can be evaluated to WHNF, annotations can be added for constructor fields. In this way, something can be said about the strictness of a function in certain fields of a data type.

Another possibility to improve the precision for higher order functions is to add polyvariance. By allowing variables in the relevance types, the functions that are used higher order can have a variable type. Constraints can be put in the type to link the variables to each other. Consider, for example the expression:

$$\begin{array}{l}\textbf{let}\ \mathit{fst} = \lambda x\ y \to x\ \textbf{in}\\ \textbf{let}\ \mathit{snd} = \lambda x\ y \to y\ \textbf{in}\\ \textbf{let}\ \mathit{app} = \lambda f\ x\ y \to f\ x\ y\\ \textbf{in}\ \mathit{app}\ \mathit{fst}\ 1\ 2 + \mathit{app}\ \mathit{snd}\ 1\ 2\end{array}$$

The type for *app* is:

$$\{\varphi_1 \sqsubseteq \varphi_4, \varphi_2 \sqsubseteq \varphi_5, \varphi_3 \sqsubseteq \varphi_6\} \Rightarrow (() \xrightarrow{\varphi_1} () \xrightarrow{\varphi_2} ()) \xrightarrow{\varphi_3} () \xrightarrow{\varphi_4} () \xrightarrow{\varphi_5} ()^{\varphi_6}$$

The input function determines the concrete type since *hof fst* gets the type $\{\varphi_4 \sqsubseteq \varphi_6\} \Rightarrow () \xrightarrow{\varphi_4} () \xrightarrow{\varphi_5} ()^{\varphi_6}$ since *fst* is strict in its first argument. On the other hand, *hof snd* gets the type $\{\varphi_5 \sqsubseteq \varphi_6\} \Rightarrow () \xrightarrow{\varphi_4} () \xrightarrow{\varphi_5} ()^{\varphi_6}$, because *snd* is strict in its second argument.

An important issue here is code generation. This becomes a lot harder when using polyvariant types since for every higher order function, multiple versions have to be generated. For example for *hof*, different code has to be generated for each combination of relevance values on the input functions. So four different versions have to be available for *hof*. This may lead to a considerable amount of code duplication.

Bibliography

[1] Torben Amtoft. Minimal thunkification. In Patrick Cousot, Moreno Falaschi, Gilberto Filé, and Antoine Rauzy, editors, *Static Analysis, Third International Workshop, WSA '93, Padova, Italy, September 22–24, 1993, Proceedings*, volume 724 of *Lecture Notes of Computer Science*, pages 218–229. Springer-Verlag, 1993.

[2] Urban Boquist and Thomas Johnsson. The grin project: A highly optimising back end for lazy functional languages. In *Selected Papers from the 8th International Workshop on Implementation of Functional Languages*, IFL '96, pages 58–84. Springer-Verlag, 1997.

[3] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. The theory of strictness analysis for higher order functions. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects, Proceedings of a Workshop, Copenhagen, Denmark, October 17–19, 1985*, volume 217 of *Lecture Notes in Computer Science*, pages 42–62. Springer-Verlag, 1986.

[4] Luis Damas and Robin Milner. Principal type-schemes for functional prorgams. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212. ACM, 1982.

[5] Kei Davis and Philip Wadler. Backwards strictness analysis: Proved and improved. In Kei Davis and John Hughes, editors, *Functional Programming, Proceedings of the 1989 Glasgow Workshop, 21–23 August 1989, Fraserburgh, Scotland, UK*, Workshops in Computing, pages 12–30. Springer-Verlag, 1990.

[6] 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, pages 93–104. ACM, 2009.

[7] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 237–247. ACM, 1993.

[8] Kevin Glynn, Peter J. Stuckey, and Martin Sulzmann. Effective strictness analysis with HORN constraints. In Patrick Cousot, editor, *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16–18, 2001, Proceedings*, volume 2126 of *Lecture Notes in Computer Science*, pages 73–92. Springer-Verlag, 2001.

[9] 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, pages 121–130. ACM, 2010.

[10] Thomas P. Jensen. Inference of polymorphic and conditional strictness properties. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 19–21, 1998, San Diego, CA, USA*, pages 209–221. ACM Press, 1998.

[11] Tsung-Min Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In *FPCA '89, Conference on Functional Programming Languages and Computer Architecture, Imperial College, London, England, 11–13 September 1989*, pages 260–272. ACM Press, 1989.

[12] Tom Lokhorst. Strictness optimization in a typed intermediate language. Master's thesis, Utrecht University, 2010.

[13] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In Bernard Robinet, editor, *International Symposium on Programming, Proceedings of the Fourth 'Colloque International sur la Programmation', Paris, France, 22–24 April 1980*, volume 83 of *Lecture Notes in Computer Science*, pages 269–281. Springer-Verlag, 1980.

[14] Eric Nöcker. Strictness analysis using abstract reduction. In *FPCA '93 Conference on Functional Programming Languages and Computer Architecture. Copenhagen, Denmark, 9–11 June 1993*, pages 255–265. ACM Press, 1993.

[15] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 636–666. Springer-Verlag, 1991.

[16] Kirsten Lackner Solberg Gasser, Hanne Riis Nielson, and Flemming Nielson. Strictness and totality analysis. *Science of Computer Programming*, 31(1):113–145, 1998.

[17] UUAGC. Utrecht haskell compiler structure structure documentation. `http://www.cs.uu.nl/wiki/bin/view/Ehc/EhcStructureDocumentation`.

[18] Phil Wadler and John Hughes. Projections for strictness analysis. In *Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture*, pages 385–407. Springer-Verlag, 1987.

[19] Philip Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 266–275. Ellis Horwood, Chichester, 1987.

[20] David Walker. Substructural type systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2005.

[21] David A. Wright. A new technique for strictness analysis. In Samson Abramsky and Tom Maibaum, editors, *TAPSOFT'91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Brighton, UK, April 8–12, 1991, Volume 2: Advances in Distributed Computing (ADC) and*

Colloquium on Combining Paradigms for Software Development (CCPSD), volume 494 of *Lecture Notes in Computer Science*, pages 235–258. Springer-Verlag, 1991.