A Case Study on the Scalability of a Constraint Solving Algorithm: Polymorphic Usage Analysis with Subtyping

Tobias Gedell

September 29, 2003

Abstract

It is commonly believed that polymorphic program analyses with subtyping are too computationally expensive to be feasible. Recent work has shown that a class of analyses having polymorphism and subtyping can be implemented in worst case cubic time. It is however unclear how often the worst case behaviour is exhibited in practice and it might be the case that they in practice are cheap enough to be feasible.

The aim of this thesis is to evaluate how a recently proposed constraint solving algorithm behaves in practice on real world programs. In order to do so a type based polymorphic usage analysis with subtyping is designed for a real world functional language. Previous usage analyses have either been designed for toy languages or have not been equipped with both subtyping and full-blown polymorphism. The analysis handles the complete language Core which is the intermediate language of the Glasgow Haskell Compiler. The analysis has been implemented and used to analyse a number of programs taken from the *nofib* suite which is a standard test suite consisting of real world programs. The constraints generated from these programs have been used to measure the scalability of the constraint solving algorithm. The results of the measurements are promising but not conclusive, the constraint solving algorithm is shown to scale up for all but two tested programs.

Foreword

This report is submitted as a Master's thesis in the Computer Science and Engineering programme at Chalmers University of Technology. The work has been carried out during the autumn of 2002 and spring of 2003 at the Computer Science Department.

I would like to thank my supervisor Jörgen Gustavsson for taking me under his wings. He has been a huge source of inspiration and has always taken the time to discuss things with me when I have needed to. I would also want to thank Josef Svenningsson who has worked as my second supervisor. It has been a pleasure to work together with Jörgen and Josef!

Finally I would like to thank John Hughes. He was the person that during my first year as an undergraduate student made me interested in functional programming and later encouraged me when I felt that I wanted to dig deeper into it. It was during this period that I began to feel that I wanted to be a graduate student. Without John this might not have happened, thank you John!

Contents

List of Figures

Chapter 1

Introduction

It is often difficult to make program analyses scale up in both precision and speed. This is especially true for analyses working on functional languages where context-insensitive analyses often are too crude due to the high degree of code reuse. One example of this is the monomorphic usage analysis of Wansbrough and Peyton Jones [WPJ99], which when implemented was shown to be almost useless in practice [WPJ00].

The difference between a context-insensitive and context-sensitive analysis is that the latter takes the context into account. As an example consider an analysis analysing function calls where a context-sensitive analysis would treat multiple calls to a function independently taking the local context into account. A context-insensitive analysis would instead create a single approximation of all multiple calls. This approximation may or may not have a big impact on the precision depending on how many multiple calls there are. For a functional language it would however most probably lead to a dramatic decrease of precision. The reason is that one in functional languages tend to write many functions which are called from many places within a program. It has been said that a function to a functional programmer is what a macro is to a C-programmer.

For the type of analyses which this thesis is about one way of doing them context-sensitive is by adding subtyping and polymorphism. What it means for a usage analysis to be polymorphic is not that the underlying type system is polymorphic but instead that the usage information is polymorphic. This allows us to create a unique instantiation of the usage information for each context where it is used and thus avoiding a mixup of different contexts leading to decreased precision.

When implementing analyses based on subtyping one often divides the analyses into two parts. The first part analyses a program and generates constraints which express the program's properties. The second part solves the generated constraints yielding a solution which is the result of the analysis. Solving the constraints generated from analyses having both subtyping and polymorphism has been considered being too computationally expensive. Solving this kind of constraints has recently been addressed by Rehof and Fähndrich [Reh01] and Gustavsson and Svenningsson [GS01a]. They show that a class of polymorphic program analyses based on inequality constraints can be implemented in worst case cubic time but it is believed that the worst case behaviour is rarely exhibited in practice.

The work resulting in this thesis focus on evaluating a constraint solving algorithm proposed by Gustavsson and Svenningsson. The algorithm is described in the unpublished paper [GSG03]. In order to evaluate the algorithm we design a polymorphic usage analysis with subtyping for a real world language chosen to be Core, the intermediate language of the Haskell compiler GHC. Previous analyses, such as [LGH$^+$92, TWM95, Gus98, WPJ99, WPJ00, GS01a], have been designed for toy languages or have not been equipped with both subtyping and full-blown polymorphism. The usage analysis is used to analyse a collection of real world programs. The generated constraints are then used to measure the scalability of the constraint solving algorithm. The results presented in Chapter 4 are promising, the constraint solving algorithm scales up for all but two tested programs.

The three steps are here summarised:

1. Design a usage analysis for the language Core.
2. Implement the usage analysis.
3. Measure how the constraint solver scales up for real-world programs.

How the precision of the usage analysis scales up in precision goes beyond the work presented in this thesis. It is however a very interesting topic and is left as future work.

1.1 Usage analysis

In this section we describe what usage analysis is. We start by giving an introduction to lazy evaluation and why it is sometimes unnecessary slow. We then present an analysis which solves this problem and argue for why we believe that it is important that such an analysis has both polymorphism and subtyping.

1.1.1 Lazy evaluation

The key idea behind lazy evaluation is that computations should not be evaluated before they are needed (this is why it is called *lazy* evaluation) and that a computation should be shared by all its successive uses. Consider the following example.

$$\begin{array}{ll}\textbf{let} & x = 1 + 2 \\ & y = 3 + 4 \\ \textbf{in} & x + x\end{array}$$

When starting to evaluate this program the computations for x and y will be stored in the program memory, called the *heap*, but they will not yet be evaluated. When reaching the expression $x + x$ we see that we need the value of x. We then get the computation for x from the heap and evaluate it, yielding the value 3. We now update x with this value. When we later need the value of x the second time we do not need to perform any calculation since x now consists of the value 3.

The exact evaluation of the example above goes as follows:

1. The computation for x is stored on the heap
2. The computation for y is stored on the heap
3. The computation for the left occurrence of x is taken from the heap
4. The computation for x is evaluated, yielding the result 3

5. The computation for x in the heap is *updated* with the value 3
6. The value for the right occurrence of x is taken from the heap (which is now 3)
7. The expression $3 + 3$ is evaluated yielding the result

The fifth step in the evaluation makes sure that the value of x is computed only once. Just as x, all variables which are used many times need to be updated. It is important to notice that since y was never needed its value was never evaluated.

Lazy evaluation allows the programmers to focus more on what they want to compute instead of in which order it should be computed. It also allows for the use of structures such as infinite lists which are impossible to use in *strict*[1] languages like ML.

1.1.2 Unnecessary updates

One inefficiency of lazy evaluation is that updating is not always needed. If we for example have a variable which is used only once then there is no need to update it once its computation has been evaluated. Doing this would only impose an unnecessary cost. Consider the following example.

$$\begin{array}{ll}\textbf{let} & x = 1 + 2 \\ \textbf{in} & x + 5\end{array}$$

In the example we see that the value of x will only be needed once. Updating x with the value 3 would therefore be unnecessary.

How common are unnecessary updates? Measurements by Marlow [Mar93] have shown that in the Haskell implementation which he used as many as 70% of all updates are unnecessary and that these unnecessary updates stands for up to 20% of the total running time of a program.

By knowing which variables that are going to be used only once we could omit updating these variables and thus make programs run faster. But how do we know how many times a variable is going to be used? This is what usage analysis figures out.

1.1.3 A solution

In order to find out how many times each variable in a program is used we introduce *uses* which we attach to bindings and values. Consider the following example.

$$\begin{array}{ll}\textbf{let} & x = 1 + 2 \\ & y = 3 + 4 \\ \textbf{in} & x + y + y\end{array}$$

Here x would be annotated with 1 denoting that the binding is used only once. y would be annotated with ω denoting that the binding will be used many times. 1 and ω can thus be seen as upper limits of the usage when the program is executed.

Adding annotations to the example given above gives:

$$\begin{array}{ll}\textbf{let} & x \overset{1}{=} 1^1 + 2^1 \\ & y \overset{\omega}{=} 3^1 + 4^1 \\ \textbf{in} & x + y + y\end{array}$$

[1]Here *strict* refers to call-by-value semantics.

Note that even if the binding for y is used many times and annotated with ω the values 3 and 4 are used only once since the binding will be updated with the value of $3 + 4$. Therefore these two values, 3 and 4, will indeed be used only once.

One important observation about usage analysis is that it is undecidable to infer the exact usage of all variables[2]. Therefore no usage analysis can infer the exact usage information, instead an approximation is calculated.

The approximation that is calculated must be *safe*. Since we want to use the usage information to avoid unnecessary updates we must ensure that a variable used many times is never annotated as being used only once. If it is annotated as being used only once then it would never be updated when its computation is computed. This would lead to the computation being computed every time the variable is needed. The other way around, that a value used only once is marked as being used many times is, besides being unwanted, however safe. An unnecessary update would be performed but no work will be duplicated.

1.1.4 Further applications of usage analysis

Besides update avoidance usage information allows for a number of optimisations:

- **Inlining** If a binding is known to be used only once it can safely be inlined at its use site.
- **Floating** If a lambda abstraction surrounded by a binding is used at most once the binding may safely be floated inwards. What we gain by doing this is that the binding will never be allocated if the abstraction is never called and even if it is called we will delay the allocation, which could save some memory.
- **Full laziness** The full laziness transformation is the opposite of the floating transformation described above. It tries to move bindings out of lambda abstractions. If an abstraction is used many times then moving bindings out of that abstraction would allow these bindings to be shared by all successive applications, leading to reduced work.

  If a lambda abstraction is used only once this translation will not gain anything but instead introduce an extra cost. By using usage information we can avoid performing this transformation for these abstractions.

Let-floating transformations are described in more detail in [PJPS96].

1.2 Type based usage analysis

In this thesis we define our usage analysis as a type based program analysis. A type based program analysis can be seen as an extension to the underlying type system. The typing rules are extended to not only infer types but also usage information.

It is therefore natural that we do not just want to annotate bindings and values but also their types. A function taking an integer, which will be used more than once, and returning an integer, which also will be used more than once, would for example have the following annotated type[3]: $Int^{\omega} \rightarrow Int^{\omega}$, where the omegas has the meaning as described above.

[2]This can be proved by using the halting problem.

[3]Later, when defining our usage analysis, types will sometimes have two layers of usage annotations.

1.2.1 Subtyping

What it means for a type τ_0 to be a subtype of τ_1 is that we can safely consider a value of type τ_0 as being of type τ_1. An example of this is that *Int* is a subtype of *Float* in a language where an integer automatically can be converted into a floating point number. We will write "*Int* is a subtype of *Float*" as $Int \leq Float$.

We will create a subtyping relation for usage annotated types. If we have a value of type Int^{ω} we know that it is allowed to use this value many times. It is however important to understand that when we are allowed to used it many times we are also allowed to use it only once. We may therefore safely treat a value of type τ^{ω} as if it was of type τ^1 which gives that $\tau^{\omega} \leq \tau^1$. We can generalise this rule to get the following rule,

$$\text{Sub}\ \frac{\tau_0 \leq \tau_1 \quad \pi_1 \leq \pi_0}{\tau_0^{\pi_0} \leq \tau_1^{\pi_1}}$$

which says that $\tau_0^{\pi_0}$ is a subtype of $\tau_1^{\pi_1}$ if τ_0 is a subtype of τ_1 and π_1 is less than or equal to π_0. This requires an order for annotations and we define it as $1 < \omega$. Note that we use $\leq$ to denote both subtyping for types as well as less-then-or-equal for annotations.

We are now ready to look at an example where subtyping is needed. Consider the following example.

$$\begin{array}{ll}\textbf{let} & x = 1 + 2 \\ & y = sq\ x \\ \textbf{in} & [x, y]\end{array}$$

We can see that x will be annotated with ω since it is used many times. The usage of y is dependent on how the list $[x, y]$ will be used. If the list and its elements are only used once it would be safe to annotate y with 1. It is in lazy languages common to use intermediate lists whose elements are used only once and we therefore assume that this is also the case with our list. One property of lists is that all elements must be of the same type. Either all elements must be annotated with ω or all elements must be annotated with 1. This gives us a problem since x has the type Int^{ω} and we would want y to have the type Int^1. The solution to this problem is obvious when we have subtyping. By using subtyping we can, since $Int^{\omega} \leq Int^1$, subtype x's type to Int^1. The type of the list will then be $(List\ Int^1)^1$. We can from this example draw the conclusion that subtyping is often needed in places where different types are tied together and forced to be equal. If we did not have subtyping we would in the example above be forced to annotate y with ω.

The premise $\pi_1 \leq \pi_0$, found in the Sub rule above, is called a *constraint*. Since the constraint do not force the annotations π_0 and π_1 to be equal it is an *inequality constraint*.

It is important to note that we cannot implement subtyping without having inequality constraints. But as we shall see, we do also need inequality constraints for another reason.

Consider the following example (where we have omitted the annotations on the values) and assume that we only have equality constraints.

$$\begin{array}{ll}\textbf{let} & x \overset{k_0}{=} 1 + 2 \\ & f \overset{k_1}{=} \lambda z.\ x + 4 \\ \textbf{in} & sq\ x + f\ 5\end{array}$$

Here k_0 and k_1 denote not yet known annotations. We do however know that if k_1 is equal to ω then so must k_0 (since x occurs inside the lambda abstraction). If we only have equality constraints, we must introduce the following constraint:

$$k_1 = k_0$$

We can clearly see that x is used more than once and must therefore introduce the following constraint:

$$\omega = k_0$$

When trying to find the least solution for these constraints we will get the solution $k_0 = k_1 = \omega$ which is not the solution we want. Note that even if x is used more than once this does not force f to be used more than once. In the example f is indeed used only once. The problem lies within the constraints, they cannot express that k_1 should be ω if k_0 is ω but not the other way around.

We solve this problem by using inequality constraints. We can now express the desired meaning by the following constraints,

$$\begin{array}{l} k_1 \leq k_0 \\ \omega \leq k_0 \end{array}$$

which when solved gives the expected least solution $k_0 = \omega, k_1 = 1$.

1.2.2 Usage polymorphism

Usage polymorphism is needed to express interdependencies between usage annotations within a type. This is best illustrated by an example (where we once again omit the annotations on the values):

$$\begin{array}{ll}
\textbf{let} & \mathit{apply} \overset{\omega}{=} \lambda f.\ \lambda x.\ f\ x \\
 & \mathit{sq} \overset{1}{=} \lambda x.\ x * x \\
 & \mathit{id} \overset{1}{=} \lambda x.\ x \\
 & x \overset{k_0}{=} 1 + 2 \\
 & y \overset{k_1}{=} 3 + 4 \\
\textbf{in} & \mathit{apply}\ \mathit{sq}\ x + \mathit{apply}\ \mathit{id}\ y
\end{array}$$

In this example we assume that we know that *apply* is used many times and that *sq* and *id* is used only once. What we want to find out is the usages of x and y. Since we are working with a type based analysis we want to infer the types of all bindings. But what should the type of *apply* be? If the function given as the first argument, f, uses its argument, x, more than once then the type of *apply* could be $\forall\alpha.\ \forall\beta.\ (\alpha^\omega \to \beta^1)^1 \to \alpha^\omega \to \beta^1$. Which expresses that the second argument must be annotated as being used more than once.

If, on the other hand, the function f uses its argument only once we may give the *apply* function the type $\forall\alpha.\ \forall\beta.\ (\alpha^1 \to \beta^1)^1 \to \alpha^1 \to \beta^1$ where the second argument is annotated as being used only once.

Now let us assume that we do not have usage polymorphism. We have two possible types for *apply* but which one should we choose? In the example above we see that *apply* is applied to both *sq* which uses its argument twice and *id* which uses its argument once. Since we know that at least one function (the *sq* function) passed to *apply* uses its argument more than once we have to choose the type where the second argument is annotated with ω.

Now, when inferring the usage of x and y we see that k_0 must be equal to ω but what about k_1? Since we know that *id* will only use its argument once it would be safe to set k_1 to 1 but we cannot do that! The reason for why is that the type of *apply* forces the second argument to be annotated with ω regardless of what the passed function is.

What we would want is a way of expressing that the annotation on the second argument depends on what the first argument is. One way of doing that is to introduce polymorphism. By using polymorphism we can state the type of *apply* as $\forall k.\ \forall\alpha.\ \forall\beta.\ (\alpha^k \to \beta^1)^1 \to \alpha^k \to \beta^1$. In the type we have introduced a universally quantified usage variable, k. When this function is used the usage variable is instantiated to either ω or 1. We see that by instantiating it to ω we get the type used when applied to *sq* and by instantiating it to 1 we get the type used when applied to *id*. This allows us to have a single type which can handle both the case with the *sq* function and the case with the *id* function.

This is however not the exact form of usage polymorphism we will use, we will instead use *bounded* polymorphism. Bounded polymorphism is like the polymorphism described above except for the addition of constraints. The constraints are on the form $k \leq j$ which is read as "if k is ω than j must also be ω". The type of *apply* using bounded usage polymorphism becomes $\forall k, j : k \leq j.\ \forall\alpha.\ \forall\beta.\ (\alpha^k \to \beta^1)^1 \to \alpha^j \to \beta^1$.

What makes analyses with both subtyping and polymorphism computationally expensive is that every time a polymorphic binding is used its type is instantiated yielding a fresh set of constraints. This can lead to an exponential blowup in the size of the constraints if one do not treat it carefully.

1.3 Outline

In Chapter 2 we present a somewhat simplified version of the source language, the constraint language and a usage analysis for the simplified language. In Chapter 3 we extend the source language with user defined data types and present a usage analysis for the extended language. In Chapter 4 we present the results of the measurements of the constraint solver. In Chapter 5 we describe related work and finally in Chapter 6 we draw our conclusion and propose future work.

Chapter 2

Usage Analysis

In this chapter we define a type based usage analysis for Core [Tol01], the intermediate language of the Glasgow Haskell Compiler (GHC). Core is a language which is similar to the language F^ω of Girard [Gir72]. It has polymorphism as well as higher-kinded types. The differences between Core and F^ω is that Core has general recursion but also a more limited type system. In Core the only way to introduce type abstraction over types is by the use of type definitions.

We remove algebraic data types from the language to make the presentation more comprehensible. Algebraic data types demand subtle treatment and are dealt with in Chapter 3. In the spirit of making a comprehensible presentation we also omit some other constructions: notes, literals (except integers), calls to external functions and the coerce construction. These constructions do not impose any difficulties for the analysis and can therefore safely be omitted. The remaining language is like System-F with the extension that it has recursion.

Values	v	$::=$	$\lambda x : \delta . e$	abstraction
		$\mid$	n	integer literal
Expressions	e	$::=$	v	value
		$\mid$	x	variable
		$\mid$	$e\ x$	application
		$\mid$	$\Lambda \alpha . e$	generalisation
		$\mid$	$e @ \delta$	instantiation
		$\mid$	$\texttt{let}\ \vec{x} : \vec{\delta}\ =\ \vec{e}\ \texttt{in}\ e$	binding block
Programs	p	$::=$	$\texttt{main}\ e$	main expression

Figure 2.1: Source language

Types	δ	$::=$	α	type variable
		$\mid$	Int	integer
		$\mid$	$\delta \to \delta$	function type
		$\mid$	$\forall\alpha.\delta$	universal quantification

Figure 2.2: Source type language

2.1 Source language

2.1.1 Syntax

The source language is presented in Figure 2.1. The language has, besides the usual constructions, explicit type generalisation and type instantiation.

The let construction binds a group of mutually recursive bindings. Each binding in the group is explicitly typed.

We introduce a syntactic restriction on the language, application arguments must be variables. This is by now a standard restriction [Jon92, Mar93, TWM95, GS01a] and we have it because it makes the typing rules of the analysis easier. It is easy to transform an arbitrary program into a program obeying this restriction. Our analysis moves every non-variable argument into a binding block, binding the value to a variable. For example, the following application

$$f\ (1+2)$$

will be translated into

$$\texttt{let}\ x = 1+2\ \texttt{in}\ f\ x$$

where x is a fresh variable.

The type language for the source language is presented in Figure 2.2, where we let α range over type variables. There is nothing unusual with this type language. Since we do not have data types we have removed higher kinded types and type application from the type language. This will be introduced later in Chapter 3.

2.1.2 Typing rules

In Figure 2.3 we present the typing rules for the source language. Since we have removed higher order types there is no need for kinds in the rules. The typing environment, Γ, is a function from variables to their types. We let $\Gamma, x : \delta$ denote the environment Γ extended with the mapping from the variable x to the type δ.

In the rules found in this thesis we will sometimes allow ourselves to use vector notation. Instead of writing judgements like

$$\Gamma \vdash e_i\ :\ \delta_i\ \ \textbf{for all}\ 1 \leq i \leq |\vec{e}|$$

we will write

$$\Gamma \vdash \vec{e}\ :\ \vec{\delta}$$

This makes the rules more compact but requires that the reader is careful when reading the rules.

$$\text{Abs}\ \frac{\Gamma, x : \delta_0 \vdash e\ :\ \delta_1}{\Gamma \vdash (\lambda x : \delta_0.\ e)\ :\ \delta_0 \rightarrow \delta_1}$$

$$\text{Var}\ \frac{}{\Gamma \vdash x\ : \delta}\quad \Gamma(x) = \delta \qquad \text{Lit}\ \frac{}{\Gamma \vdash n\ : \mathit{Int}}$$

$$\text{App}\ \frac{\Gamma \vdash e\ :\ \delta_0 \rightarrow \delta_1}{\Gamma \vdash e\ x\ :\ \delta_1}\quad \Gamma(x) = \delta_0$$

$$\text{Gen}\ \frac{\Gamma \vdash e\ :\ \delta}{\Gamma \vdash (\Lambda\alpha.\ e)\ :\ \forall\alpha.\delta}\quad \alpha \notin \mathit{fv}(\Gamma) \qquad \text{Inst}\ \frac{\Gamma \vdash e\ :\ \forall\alpha.\delta_1}{\Gamma \vdash e@\delta_0\ :\ \delta_1[\delta_0/\alpha]}$$

$$\text{Let}\ \frac{\Gamma, \vec{x} : \vec{\delta} \vdash \vec{e}\ :\ \vec{\delta} \qquad \Gamma, \vec{x} : \vec{\delta} \vdash e\ :\ \delta}{\Gamma \vdash \texttt{let}\ \vec{x} : \vec{\delta}\ =\ \vec{e}\ \texttt{in}\ e\ :\ \delta}$$

Figure 2.3: Typing rules

We will use the property that a program is *well typed.* If a program can be derived a type by using the typing rules, we will consider it being well typed.

When performing the usage analysis we will always assume that the input program is well typed. This assumption is reasonable since the analysis can be thought of as an internal phase in the compiler taking place after type checking. Possible type errors would be caught during type checking and would never reach the usage analysis phase.

2.2 Adding usage annotations

The aim of the usage analysis is to find an approximation of the usage of all values and bindings. These usages will when the analysis is done be stored in the source program and then later used by program transformations or in the code generation to avoid unnecessary updates.

In order to store the annotations in the source program we extend the source language with usage annotations on both values and bindings. We let π range over uses. A binding or value annotated with 1 is interpreted as being used only once, while ω is interpreted as possibly being used many times. 1 and ω should be seen as upper bounds of the usage. The annotated source language is presented in Figure 2.4.

When starting to analyse a program it will be an unannotated source program. The first thing we do is that we annotate the program with annotation variables on all values and bindings. These annotation variables will later be replaced with their inferred uses.

We have two kinds of variables, type annotation variables which are used to annotate types and program annotation variables which are used to annotate the program. The reason for having these two kinds of annotation variables is that it makes the typing rules more comprehensible.

Type annotation variables	k		
Program annotation variables	j		
Uses	π	$::=$	$1 \mid \omega \mid k \mid j$

Values	v	$::=$	$\lambda x : \delta.e$	abstraction
		$\mid$	n	integer literal
Expressions	e	$::=$	v^{π}	annotated value
		$\mid$	x	variable
		$\mid$	$e\ x$	application
		$\mid$	$\Lambda\alpha.e$	generalisation
		$\mid$	$e@\delta$	instantiation
		$\mid$	$\texttt{let}\ \vec{x} : \vec{\delta} \overset{\vec{\pi}}{=} \vec{e}\ \texttt{in}\ e$	binding block
Programs	p	$::=$	$e : \delta$	main expression

Figure 2.4: Annotated source language

Value types	ρ	$::=$	α	type variable
		$\mid$	Int	integer
		$\mid$	$\sigma \to \tau$	function type
		$\mid$	$\forall\alpha.\rho$	universal quantification
Expression types	τ	$::=$	ρ^{π}	
Binding types	σ	$::=$	τ_{π}	
Type schemas	χ	$::=$	ρ	
		$\mid$	$(\forall\vec{k}.\rho \mid \Pi)$	

Figure 2.5: Annotated type language

The first stage of the analysis types the program using the usage typing rules which generates constraints for all annotation variables. If for example the usage of a variable x depends on the usage of another variable y, we say that y's usage constrains the usage of x. In order to express these constraints we need a constraint language, which is presented in Section 2.3.

When the program has been analysed we have an annotated program containing annotation variables together with a constraint term which constrains the annotation variables in the program. Solving this constraint term gives us an assignment of annotation variables to 1:s and ω:s. What we want is the least solution and in order to find it we use a constraint solver. The constraint solver that we have chosen to use is the constraint solver which scalability we want to study.

We also extend the type language with usage annotations. The annotated type language is presented in Figure 2.5. In the language we let Π range over constraints which will be introduced in section 2.3. We allow ourselves to let α, β and γ range over type variables in both the unannotated type language and the annotated type language. This will however not cause any confusion since the context will make clear what language we are referring to.

In the type language we have four different types:

- **Value types** - The types of values in head normal form: abstractions and integers.
- **Expression types** - The types of expressions. These types have an usage annotation telling the usage of the contained value. If an expression is of type Int and the value is known to be used more than once the expression type would be Int^{ω}.
- **Binding types** - The types of bindings. The binding types consist of an expression type together with an additional usage annotation telling the usage of the binding. An binding which is used more than once holding an integer value which is used more than once would have the type Int^{ω}_{ω}, while a binding used only once containing a value which is used many times would have the type Int^{ω}_{1}.
- **Type schemas** - These are value types which may contain universally quantified annotation variables together with the constraints constraining them. All bindings are given a type scheme together with two usage annotations. When a binding is used its type scheme is instantiated to a binding type.

The reason for having these levels of types is that it offers us increased precision. By assuming that a binding type only has one usage annotation we can understand that we cannot make a difference between how many times the binding is going to be used and how many times the value of the binding is going to be used. Consider the following example:

$$\begin{array}{ll} \textbf{let} & x = 1 + 2 \\ & y = id\ x \\ \textbf{in} & sq\ y \end{array}$$

In this example we see that y will be used twice and the binding must therefore be annotated with ω. But what about x? We see that the value of x will be used twice since y will contain the value of x. But the binding of x will only be used once! Why is that? The answer lies in the updating mechanism. When y is first evaluated it will evaluate $id\ x$, which evaluates to 3, and then update y with this value. The second time y is evaluated the result 3 is fetched without needing to evaluate x again. Therefore x is only evaluated once although y is evaluated twice.

If the type of x only had one usage annotation this would be forced to be the upper bound of the value's usage and the binding's usage and since the value is used more than once the type would be annotated with ω, which we know is an unnecessary imprecise annotation.

By using our levels of types we can correctly handle these cases. Remember that a binding type has two annotations where the first holds the usage of the value and the second the usage of the binding. In our type language we would simply give x the type Int^{ω}_{1} and y the type Int^{ω}_{ω}.

Value types

$$\rho \quad ::= \quad \alpha \mid Int \mid \sigma \rightarrow \tau \mid \forall \alpha . \rho$$

consist of type variables, integers, functions and universally quantified value types. Functions take variables of binding types (recall the syntactic restriction on application) and return values of expression types.

Type schemas

$$\chi \quad ::= \quad \rho \mid (\forall\vec{k}.\rho \mid \Pi)$$

can be seen as quantified value types. Each binding has a type scheme, $\chi^{\pi}_{\pi'}$, which holds the most general type. When a binding is used its type scheme is instantiated, yielding a binding type, σ. When the type scheme is instantiated fresh annotation variables, $\vec{k}'$, are used to instantiate the value type, ρ, and the constraints, Π. The instantiation rule, $\prec$, is defined as follows.

$$\begin{array}{l} \top \vdash \rho \prec \rho \\ \Pi[\vec{\pi}/\vec{k}] \vdash (\forall\vec{k}.\rho \mid \Pi) \prec \rho[\vec{\pi}/\vec{k}] \end{array}$$

2.3 Constraint language

The constraint language presented in Figure 2.6 is the language described in [GS01a]. The language contains atomic constraints definitions of constraint abstractions, calls to constraint abstractions and existential quantification.

What makes the constraint language different from other constraint languages, and what is also its strength, is the constraint abstractions. Constraint abstractions are used to solve the problem with polymorphic analyses which sometimes have to duplicate constraints. A constraint abstraction can be seen as a function from variables to constraints. Consider the following example:

$$\begin{array}{lcl} \textbf{let } f_0 & = & ...\ \textbf{in} \\ \textbf{let } f_1 & = & ...\ f_0\ ...\ f_0\ ...\ \textbf{in} \\ \textbf{let } f_2 & = & ...\ f_1\ ...\ f_1\ ...\ \textbf{in} \\ ... & & \end{array}$$

When generating the constraints for the function f_0 we might get the constraints Π_0. When generating the constraints for the function f_1 we see that it contains two calls to the function f_0. When instantiating the function f_0 we get two instantiations of the constraints Π_0. Thus Π_1 will contain two copies of the constraints Π_0. When generating the constraints for f_2 we see that it will contain two copies of Π_1. Here is how the generated constraints would look:

$$\begin{array}{lcl} \Pi_0 & = & ... \\ \Pi_1 & = & ...\ \textbf{copy of } \Pi_0\ ...\ \textbf{copy of } \Pi_0\ ... \\ \Pi_2 & = & ...\ \textbf{copy of } \Pi_1\ ...\ \textbf{copy of } \Pi_1\ ... \end{array}$$

We can by this small example conclude that the constraints can be exponential in the size of the program.

Constraint abstraction solves this problem. For each function we create a constraint abstraction containing its constraints. Now instead of instantiating all constraints at each call site of a function, we instantiate a call to the constraint abstraction containing the constraints for the function. In this way the constraints and constraint abstractions will follow the structure of the program and be linear in the size of the explicitly typed program. The constraints for the example above would now look as this:

$$\begin{array}{l} \textbf{let } l_0\ \vec{k}_0 = ...\ \textbf{in} \\ \textbf{let } l_1\ \vec{k}_1 = ...\ \textbf{call to } l_0\ ...\ \textbf{call to } l_0\ ...\ \textbf{in} \\ \textbf{let } l_2\ \vec{k}_2 = ...\ \textbf{call to } l_1\ ...\ \textbf{call to } l_1\ ...\ \textbf{in} \\ ... \end{array}$$

Constraint terms	Π	$::=$	$\pi \leq \pi$	atomic constraint
		$\mid$	$\texttt{let}\ \vec{\phi}\ \texttt{in}\ \Pi$	binding block
		$\mid$	$\exists \vec{k}.\Pi$	existential quantification
		$\mid$	$l\ \vec{\pi}$	abstraction call
		$\mid$	$\Pi \wedge \Pi$	conjunction
		$\mid$	$\top$	true
Constraint abstractions	ϕ	$::=$	$l\ \vec{k} = \Pi$	

Figure 2.6: Constraint language

$$\text{Var}\ \frac{}{\alpha \hookrightarrow \alpha} \qquad \text{Lit}\ \frac{}{\mathit{Int} \hookrightarrow \mathit{Int}}$$

$$\text{Arrow}\ \frac{\delta_0 \hookrightarrow \sigma \quad \delta_1 \hookrightarrow \tau}{\delta_0 \to \delta_1 \ \hookrightarrow\ \sigma \to \tau} \qquad \text{Forall}\ \frac{\delta \hookrightarrow \rho}{\forall \alpha.\delta \ \hookrightarrow\ \forall \alpha.\rho}$$

$$\text{Expr}\ \frac{\delta \hookrightarrow \rho}{\delta \hookrightarrow \rho^k} \qquad \text{Bind}\ \frac{\delta \hookrightarrow \tau}{\delta \hookrightarrow \tau_k}$$

Figure 2.7: Type translation rules

When a constraint term contains only atomic constraints we will sometimes regard it as being a set.

The semantics of the constraint language is described in [GS01b].

2.4 Type translation

In Figure 2.7 we define a relation, $\hookrightarrow$, between unannotated types and annotated types. Expression and binding types are annotated with type annotation variables. When translating types during the analysis we will always use fresh annotation variables. By doing this we ensure that we will get the principal annotation.

2.5 Subtyping

In Section 1.2.1 we showed the need for subtyping. Here we define a subtyping relation for annotated types. We first define an ordering for uses, $1 < \omega$. When defining the subtyping relation we have to be careful with the annotations residing on expression and binding types. We want to define a sound subtyping relation and must therefore ensure that for example Int^ω is a subtype of Int^1. The reason for why Int^ω is a subtype of Int^1 is that a value which is marked as being used many times can safely be regarded as being used only once. Therefore we conclude that types are contravariant in the usage annotations. The subtyping rules are presented in Figure 2.8.

The most important subtyping rules are those for expression and binding types:

$$\text{Expr}\ \frac{\Pi_0 \vdash \rho \ \leq\ \rho' \quad \Pi_1 \vdash \pi' \ \leq\ \pi}{\Pi_0 \wedge \Pi_1 \vdash \rho^{\pi} \ \leq\ \rho'^{\pi'}}$$

$$\text{Var}\ \frac{}{\top \vdash \alpha\ \leq\ \alpha} \qquad \text{Lit}\ \frac{}{\top \vdash \mathit{Int}\ \leq\ \mathit{Int}}$$

$$\text{Arrow}\ \frac{\Pi_0 \vdash \sigma'\ \leq\ \sigma \quad \Pi_1 \vdash \tau\ \leq\ \tau'}{\Pi_0 \wedge \Pi_1 \vdash \sigma \to \tau\ \leq\ \sigma' \to \tau'}$$

$$\text{Forall}\ \frac{\Pi \vdash \rho_0\ \leq\ \rho_1}{\Pi \vdash \forall\alpha.\rho_0\ \leq\ \forall\alpha.\rho_1}$$

$$\text{Expr}\ \frac{\Pi_0 \vdash \rho\ \leq\ \rho' \quad \Pi_1 \vdash \pi'\ \leq\ \pi}{\Pi_0 \wedge \Pi_1 \vdash \rho^{\pi}\ \leq\ \rho'^{\pi'}}$$

$$\text{Bind}\ \frac{\Pi_0 \vdash \tau\ \leq\ \tau' \quad \Pi_1 \vdash \pi'\ \leq\ \pi}{\Pi_0 \wedge \Pi_1 \vdash \tau_{\pi}\ \leq\ \tau'_{\pi'}}$$

$$\text{An}\ \frac{}{\pi_0 \leq \pi_1 \vdash \pi_0 \leq \pi_1}$$

Figure 2.8: Subtyping rules

$$\text{Bind}\ \frac{\Pi_0 \vdash \tau\ \leq\ \tau' \quad \Pi_1 \vdash \pi'\ \leq\ \pi}{\Pi_0 \wedge \Pi_1 \vdash \tau_{\pi}\ \leq\ \tau'_{\pi'}}$$

We know that the usage annotations are contravariant and therefore when subtyping ρ^{π} to $\rho'^{\pi'}$ we subtype π' to π and ρ to ρ'. The same applies when subtyping binding types.

We do not have general subtyping on type variables, we do however define a type variable to be a subtype of itself.

The subtyping rule for function types often confuses people at the first glance.

$$\text{Arrow}\ \frac{\Pi_0 \vdash \sigma'\ \leq\ \sigma \quad \Pi_1 \vdash \tau\ \leq\ \tau'}{\Pi_0 \wedge \Pi_1 \vdash \sigma \to \tau\ \leq\ \sigma' \to \tau'}$$

What is tricky is that the argument is contravariant. This is however easily understood if one thinks of what it means to be a subtype. Consider a language with types for integers and floats where an integer can be converted to a float automatically. In this language a function returning an integer can easily be subtyped to a function returning a float. This is the case since an integer can be considered as being a float. When applying this reasoning to the function argument we see that a function taking a float as argument can be subtyped to a function taking an integer since the received integer can be considered as being a float which the function wants. Therefore $\mathit{Float} \to \mathit{Int}$ is a subtype of $\mathit{Int} \to \mathit{Float}$.

Consider the rule for universally quantified types:

$$\text{Forall}\ \frac{\Pi \vdash \rho_0\ \leq\ \rho_1}{\Pi \vdash \forall\alpha.\rho_0\ \leq\ \forall\alpha.\rho_1}$$

When subtyping two quantified types we first ensure that they quantify the same type variable (if this is not the case it can easily be done by α-converting the type variables) and then subtype the inner types.

$$\text{Abs}\ \frac{\delta \hookrightarrow \rho_{\pi_0}^{\pi_1} \quad \Pi_0; \Gamma, x : \rho_{\pi_0}^{\pi_1} \vdash e \ : \ \tau}{\Pi_0 \wedge \Pi_1; \Gamma \vdash \lambda x : \delta . e \ : \ \rho_{\pi_0}^{\pi_1} \to \tau} \qquad (*)$$

$$(*)\ \Pi_1 \equiv \begin{cases} \omega \leq \pi_0 \wedge \omega \leq \pi_1 & \textbf{if } \mathit{occur}(x, e) > 1 \\ \top & \textbf{otherwise} \end{cases}$$

$$\text{Int}\ \frac{}{\top; \Gamma \vdash n \ : \ \mathit{Int}}$$

Figure 2.9: Typing rules for values

2.6 Typing rules

The typing environment, Γ, is a function from variables to their annotated type schemas. We let $\Gamma, x : \chi_{\pi_0}^{\pi_1}$ denote the environment Γ extended with the mapping from the variable x to the annotated type scheme $\chi_{\pi_0}^{\pi_1}$.

We are now ready to define the typing rules. The program typing rule

$$\text{Main}\ \frac{\Pi; \emptyset \vdash e : \tau}{\Pi \vdash \texttt{main}\ e}$$

takes an annotated program, `main` e, and yields the constraint term, Π, constraining all program annotation variables. A program consists of a main expression, which is typed in the empty typing environment $\emptyset$ by the typing rules defined in the following sections.

The typing rule for values are on the form $\Pi, \Gamma \vdash v : \rho$ taking the typing environment Γ, the value v and yielding the type ρ and the constraints Π. The form of the typing rules for expressions are almost identical except that they yield an expression type.

2.6.1 Typing rules for values

The typing rules for values are presented in Figure 2.9.

The rule for abstraction

$$\text{Abs}\ \frac{\delta \hookrightarrow \rho_{\pi_0}^{\pi_1} \quad \Pi_0; \Gamma, x : \rho_{\pi_0}^{\pi_1} \vdash e \ : \ \tau}{\Pi_0 \wedge \Pi_1; \Gamma \vdash \lambda x : \delta . e \ : \ \rho_{\pi_0}^{\pi_1} \to \tau} \qquad (*)$$

$$(*)\ \Pi_1 \equiv \begin{cases} \omega \leq \pi_0 \wedge \omega \leq \pi_1 & \textbf{if } \mathit{occur}(x, e) > 1 \\ \top & \textbf{otherwise} \end{cases}$$

is one of the rules where we let textual occurrence constrain the usage of a binding. The occur function, defined in Figure 2.10, calculates how many textual occurrences a variable has. In the rule we use the occur function to see if the bound variable, x, occurs more than once in the expression e. If it does then it may clearly be used more than once and its usage must be constrained by ω. The side condition does this by forcing the outer usage annotations of x's type, π_0 and π_1, to be constrained by ω if x occurs more than once in e.

The rule for integer literal

$$\text{Int}\ \frac{}{\top; \Gamma \vdash n \ : \ \mathit{Int}}$$

is trivial. All integer literals are of the type Int.

Values $\quad occur(x, \lambda y : \delta.e) = \begin{cases} 0 & \textbf{if } x = y \\ occur(x,e) & \textbf{otherwise} \end{cases}$

$occur(x, n) = 0$

Expressions $\quad occur(x, v^{\pi}) = occur(x, v)$

$occur(x, y) = \begin{cases} 1 & \textbf{if } x = y \\ 0 & \textbf{otherwise} \end{cases}$

$occur(x, e\ y) = occur(x, e) + occur(x, y)$

$occur(x, \Lambda\alpha.e) = occur(x, e)$

$occur(x, e@\delta) = occur(x, e)$

$occur(x, \texttt{let } \vec{x} : \vec{\delta} \overset{\vec{\pi}}{=} \vec{e} \texttt{ in } e) =$

$\begin{cases} 0 & \textbf{if } x \in \{\vec{x}\} \\ (\sum occur(x, e_i)) + occur(x, e) & \textbf{otherwise} \end{cases}$

Figure 2.10: Occur function

2.6.2 Typing rules for expressions

The typing rules for all expressions except *let* are presented in Figure 2.11.

The rule Value

$$\text{Value}\ \frac{\Pi_0; \Gamma \vdash v\ :\ \rho}{\Pi_0 \wedge \pi' \leq \pi \wedge \Pi_1; \Gamma \vdash v^{\pi}\ :\ \rho^{\pi'}}\ (*)$$

$$(*)\quad \Pi_1 \equiv \textstyle\bigwedge_{x \in fv(v)} \left(\ \pi' \leq \pi_0 \wedge \pi' \leq \pi_1 \quad \textbf{where } \Gamma(x) = \chi_{\pi_0}^{\pi_1}\ \right)$$

takes care of annotating values. If a value has the type $\rho^{\pi'}$ where π' is ω then it means that the value can possibly be used more than once. If this is the case we must also ensure that all free variables of the value can also be used more than once. The side condition ensures that the outer usage annotations of all free variables of the value are constrained by the value usage, π'. The constraint $\pi' \leq \pi$ ensures that the program annotation residing on the value is constrained by the annotation on the type.

The rule for variables

$$\text{Var}\ \frac{}{\Pi; \Gamma, x : \chi_{\pi_0}^{\pi_1} \vdash x\ :\ \rho^{\pi_1}} \quad \Pi \vdash \chi \prec \rho$$

is simpler. We instantiate the binding type but holds on to the annotation π_1. By doing this we ensure that if the instantiated value is used more than once this will propagate back to the annotation on the variable's type scheme.

The application rule

$$\text{App}\ \frac{\Pi_0; \Gamma \vdash e\ :\ (\sigma \to \tau)^{\pi}}{\Pi_0 \wedge \Pi_1 \wedge \Pi_2; \Gamma \vdash e\ x\ :\ \tau} \quad \begin{array}{l} \Gamma(x) = \chi_{\pi_0}^{\pi_1} \\ \Pi_1 \vdash \chi \prec \rho \\ \Pi_2 \vdash \rho_{\pi_0}^{\pi_1} \leq \sigma \end{array}$$

needs to both instantiate the argument type and subtype it to the argument type of the function. If the function uses its value more than once then the binding annotation in σ will be ω. This propagates to the binding annotation on χ by the subtyping in the second side condition.

$$\text{Value}\ \frac{\Pi_0; \Gamma \vdash v \ : \ \rho}{\Pi_0 \wedge \pi' \leq \pi \wedge \Pi_1; \Gamma \vdash v^{\pi} \ : \ \rho^{\pi'}}\ (*)$$

$$(*) \quad \Pi_1 \equiv \bigwedge_{x \in fv(v)} \Big(\ \pi' \leq \pi_0 \wedge \pi' \leq \pi_1 \quad \textbf{where} \ \ \Gamma(x) = \chi^{\pi_1}_{\pi_0} \ \Big)$$

$$\text{Var}\ \frac{}{\Pi; \Gamma, x : \chi^{\pi_1}_{\pi_0} \vdash x \ : \ \rho^{\pi_1}} \quad \Pi \vdash \chi \prec \rho$$

$$\text{App}\ \frac{\Pi_0; \Gamma \vdash e \ : \ (\sigma \to \tau)^{\pi}}{\Pi_0 \wedge \Pi_1 \wedge \Pi_2; \Gamma \vdash e \ x \ : \ \tau} \quad \begin{array}{l} \Gamma(x) = \chi^{\pi_1}_{\pi_0} \\ \Pi_1 \vdash \chi \prec \rho \\ \Pi_2 \vdash \rho^{\pi_1}_{\pi_0} \leq \sigma \end{array}$$

$$\text{Gen}\ \frac{\Pi; \Gamma \vdash e \ : \ \rho^{\pi}}{\Pi; \Gamma \vdash \Lambda\alpha.e \ : \ (\forall\alpha.\rho)^{\pi}} \quad \alpha \notin fv(\Gamma)$$

$$\text{Inst}\ \frac{\delta \hookrightarrow \rho_0 \quad \Pi; \Gamma \vdash e \ : \ (\forall\alpha.\rho_1)^{\pi}}{\Pi; \Gamma \vdash e@\delta \ : \ \rho_1[\alpha := \rho_0]^{\pi}}$$

Figure 2.11: Typing rules for expressions

2.6.3 Typing rule for let expressions

The typing rule for let expressions, presented in Figure 2.12, is rather complex and we will go through it step by step. When typing the expression `let` $\vec{x} : \vec{\delta} \overset{\vec{\pi}}{=} \vec{e}$ `in` e we start by translating the unannotated types $\vec{\delta}$ into the annotated types $\vec{\rho}$. We proceed by creating the typing environment Γ' which consists of the typing environment Γ extended with the type schemas of the bindings. Each type scheme is quantified over the annotation variables $\vec{k_i}$ and consists of the annotated type ρ_i and a call to the constraint abstraction l_i. This is done by the first side condition where also each binding is given the annotation variables π'_i and π''_i.

We type all the expressions $\vec{e}$ in the environment Γ' yielding the constraints $\vec{\Pi_3}$ and the types $\vec{\rho'}^{\vec{\pi}'''}$. These types are subtyped to the types $\vec{\rho}^{\ \vec{\pi}''}$ yielding the constraints $\vec{\Pi_4}$.

We may now existentially quantify annotation variables which appear in the constraints Π_{3i} and Π_{4i} to obtain $\exists\vec{k'_i}.\Pi_{3i} \wedge \Pi_{4i}$. We do however require that the variables in $\vec{k'_i}$ do not occur free elsewhere in the judgement. This is ensured by the fourth side condition. We form the type abstractions ϕ_i consisting of the constraints $\exists\vec{k'_i}.\Pi_{3i} \wedge \Pi_{4i}$ where we have bound the annotation variables $\vec{k_i}$. The third side condition ensures that the annotation variables $\vec{k_i}$ do not occur free elsewhere in the judgement.

The fifth side condition ensures that the binding annotations π'_i constrains the program annotations π_i. The sixth side condition ensures that if a binding occurs at more than one place its annotations, π'_i and π''_i, are constrained by ω.

Finally we type the expression e in the environment Γ' yielding the constraints Π_2 and the type τ.

$$\text{Let}\ \frac{\vec{\delta} \hookrightarrow \vec{\rho} \quad \vec{\Pi_3}; \Gamma' \vdash \vec{e}\ :\ \vec{\rho'}^{\vec{\pi}'''} \qquad \Pi_2; \Gamma' \vdash e\ :\ \tau}{\Pi_0 \wedge \Pi_1 \wedge \texttt{let}\ \vec{\phi}\ \texttt{in}\ \Pi_2; \Gamma \vdash \texttt{let}\ \vec{x} : \vec{\delta} \overset{\vec{\pi}}{=} \vec{e}\ \texttt{in}\ e\ :\ \tau}\ (*)$$

$$\Gamma'(y) \equiv \begin{cases} (\forall \vec{k_i}.\rho_i \mid l_i\ \vec{k_i})^{\pi_i''}_{\pi_i'} & \texttt{if}\ y = x_i \\ \Gamma(y) & \texttt{otherwise} \end{cases}$$

$$(*)\quad \begin{array}{l} \vec{k_i} \notin fav(\Gamma', \pi_i'') \\ \vec{k_i'} \notin fav(\Gamma', \rho_i^{\pi_i''}) \\ \phi_i \equiv (l_i\ \vec{k_i} = \exists \vec{k_i'}.\Pi_{3i} \wedge \Pi_{4i})\ \ \texttt{where}\ \ \Pi_{4i} \vdash {\rho'}_i^{\pi_i'''} \leq \rho_i^{\pi_i''} \end{array}$$

$$\Pi_0 \equiv \bigwedge(\pi_i' \leq \pi_i)$$

$$\Pi_1 \equiv \bigwedge \begin{cases} \omega \leq \pi_i' \wedge \omega \leq \pi_i'' & \textbf{if}\ occur(x_i, e) + \sum occur(x_i, e_i) > 1 \\ \top & \textbf{otherwise} \end{cases}$$

Figure 2.12: Typing rule for let expressions

2.7 Soundness

What it means for a usage analysis to be *sound* is that the evaluation of a well typed program do not go wrong. What this mean for our usage analysis is simply that the analysis must not annotate a binding used many times as being used only once.

In order to prove our usage analysis being sound we must define an operational semantics for the source language and carry out a rather substantial proof. We choose to omit this in this thesis and instead refer to a soundness proof done for an analysis similar to the one presented here. The soundness proof can be found in [Gus99].

Chapter 3

Adding Data Types

In this chapter we introduce user defined data types which require a rather subtle treatment. We start by giving an example of how such a data type can look. Consider the following definition.

$$\text{List } \alpha = \text{Nil} \mid \text{Cons } \alpha \text{ (List } \alpha)$$

This is the well known list data type. The type definition says that a list containing elements of type α is either empty, in which case the list consists of the constructor Nil, or it contains at least one element in which case the list consists of the constructor Cons followed by the first element and the rest of the list. The list containing the numbers 1, 2 and 3 is thus encoded as shown by the following expression.

$$\text{Cons 1 (Cons 2 (Cons 3 Nil))}$$

The type variable α in the type definition is said to be a type parameter. By having type parameters we allow data types to be polymorphic. The List type can for example be instantiated to hold integers as well as any other type of values. A list containing integers has the type (List Int) whereas a list holding characters has the type (List Char). One way to look at the type parameters is to consider the constructors being universally quantified over the type parameters. The type for Nil would then be ($\forall\alpha.\text{List } \alpha$) while the type for Cons would be ($\forall\alpha.\alpha \rightarrow \text{List } \alpha \rightarrow \text{List } \alpha$).

In our source language, constructors may not only have universally quantified types but also existentially quantified types. One example of a type having a constructor with an existentially quantified type is the following example.

$$\text{PlaceHolder } \alpha = \exists\beta.\text{Elem } \beta \ (\beta \rightarrow \alpha)$$

In this example the type PlaceHolder has the type parameter α and the constructor Elem. The constructor is existentially quantified over β. An Elem value therefore has an element of the existentially quantified type β together with a function from this type to the parameterised type α. A value of the type (PlaceHolder String) would be a place holder for a value of an unknown (existentially quantified) type and a function taking a value of this type into a string.

Types	δ	$::=$	α	type variable
		$\mid$	T	type constructor
		$\mid$	$\delta \to \delta$	function type
		$\mid$	$\forall \alpha . \delta$	universal quantification
		$\mid$	$\delta\ \delta$	type application

Figure 3.1: Source type language

3.1 Source language

We extend the type language with type constructors, which are ranged over by T, and type applications. Type application is needed to handle higher kinded types.

Having type constructors we no longer need to have a special type for integers as we had in Chapter 2. Instead we create a nullary type constructor for integers which we name *Int*. The type language extended with type constructors and type application is presented in Figure 3.1.

We proceed by introducing data types in the annotated source language, yielding the language presented in Figure 3.2. What we add is constructor values, case expressions, case alternatives and type definitions. In the source language we let C range over constructors.

Type definitions

$$td \quad ::= \quad T\ \vec{\alpha} = ts_1 \mid ... \mid ts_n$$

consist of a type constructor, T, a vector holding all type parameters, $\vec{\alpha}$, and a number of type summands.

Type summands

$$ts \quad ::= \quad \exists \vec{\beta} . C\ \vec{\delta}$$

consist of a constructor, C, a vector of *existentially* quantified type variables, $\vec{\beta}$, and the types of the constructor arguments, $\vec{\delta}$.

When dealing with constructor values we must know both the type parameters and the witnesses (the instantiations of the existentially quantified type variables). Therefore constructor values

$$C\ \vec{\delta}\ \vec{\delta}\ \vec{x}$$

consist of the constructor arguments and two type vectors, where the first vector holds the type parameters and the second vector holds the witnesses.

The same applies to constructor alternatives in case expressions, which must bind both the existentially quantified types and the constructor arguments. There is however no need to hold the universally quantified variables since they are not local for each constructor.

Case expressions have an unusual feature. The case expression

$$\texttt{case}\ e\ \texttt{of}\ (x : \delta)\ \vec{alt}$$

evaluates the expression e and binds the variable x to the value of e. The variable x can then be used by the case branches. Since x is a different kind of variable which is always bound to already evaluated values it has no usage annotation.

Values	v	$::=$	$\lambda x : \delta . e$	abstraction
		$\mid$	n	integer literal
		$\mid$	$C\ \vec{\delta}\ \vec{\delta}\ \vec{x}$	constructor value
Expressions	e	$::=$	v^{π}	annotated value
		$\mid$	x	variable
		$\mid$	$e\ x$	application
		$\mid$	$\Lambda \alpha . e$	generalisation
		$\mid$	$e @ \delta$	instantiation
		$\mid$	`let` $\vec{x} : \vec{\delta} \overset{\vec{\pi}}{=} \vec{e}$ `in` e	binding block
		$\mid$	`case` e `of` $(x : \delta)\ \vec{alt}$	case
Alternatives	alt	$::=$	$C\ \vec{\beta}\ \vec{x} \Rightarrow e$	constructor alternative
		$\mid$	$n \Rightarrow e$	literal alternative
		$\mid$	$\_ \Rightarrow e$	default alternative
Type definitions	td	$::=$	$T\ \vec{\alpha} = ts_1 \mid ... \mid ts_n$	
Type summands	ts	$::=$	$\exists \vec{\beta} . C\ \vec{\delta}$	
Programs	p	$::=$	`data` $\vec{td}$ `in` p	type definitions
		$\mid$	`main` e	main expression

Figure 3.2: Annotated source language

Value types	ρ	$::=$	α	type variable
		$\mid$	$T\ \vec{\pi}$	type constructor
		$\mid$	$\sigma \to \tau$	function type
		$\mid$	$\forall\alpha.\rho$	universal quantification
		$\mid$	$\rho\ \rho$	type application
Expression types	τ	$::=$	ρ^{π}	
Binding types	σ	$::=$	τ_{π}	
Type schemas	χ	$::=$	ρ	
		$\mid$	$(\forall\vec{k}.\rho \mid \Pi)$	

Figure 3.3: Annotated type language

In the annotated type language, presented in Figure 3.3, the type constructors have been extended with a vector of annotations. These annotations holds the usage information for the constructor arguments in the type definition. We will explain this in more detail in the following sections.

3.2 Annotating data types

Before inferring the usage information for a program we must first annotate the type definitions. An annotated type definition

Type definitons	utd	$::=$	$T\ \vec{k}\ \vec{\alpha} = uts_1 \mid ... \mid uts_n$
Summand	uts	$::=$	$C\ \vec{\beta}\ \vec{\sigma}$

has binding types for all its constructor arguments, $\vec{\sigma}$, and a vector, $\vec{k}$, containing the usage annotation variables used to annotate the constructor arguments.

Here is an example showing the usage annotated version of the List type:

$$\text{List}\ k_0\ k_1\ k_2\ k_3\ \alpha = \text{Nil} \mid \text{Cons}\ \alpha^{k_1}_{k_0}\ (List\ k_0\ k_1\ k_2\ k_3\ \alpha)^{k_3}_{k_2}$$

When annotating the type definition each constructor argument is annotated with fresh annotation variables. These variables form the $\vec{k}$ vector which is used to annotate the recursive call in the type definition. Note that for the example above $\vec{k} = k_0\ k_1\ k_2\ k_3$. One important restriction that we impose on type definitions is that there must not be any annotation variables in the right hand side that is not an element of the annotation variable vector $\vec{k}$, in the left hand side.

When annotating recursive data types we always annotate the recursive occurrences using the annotation variables found in the left hand side. Here is an example of an annotated type definition containing two recursive calls:

$$\begin{aligned}\text{Tree}\ k0\ k1\ k2\ k3\ k4\ k5\ \alpha = \text{Leaf} \mid \text{Node}\ \alpha^{k1}_{k0}\ &(\text{Tree}\ k0\ k1\ k2\ k3\ k4\ k5\ \alpha)^{k3}_{k2}\\ &(\text{Tree}\ k0\ k1\ k2\ k3\ k4\ k5\ \alpha)^{k5}_{k4}\end{aligned}$$

The same applies when annotating mutually recursive type definitions. All type definitions in the mutually recursive group is annotated using the same annotation variables in the left hand side. Here is an example of two annotated data types being mutually recursive:

$$\text{Zig } k0\ k1\ k2\ k3\ k4\ k5\ k6\ k7\ \alpha = \text{ZigStop} \mid \text{Zig } \alpha^{k1}_{k0}\ (\text{Zag } k0\ k1\ k2\ k3\ k4\ k5\ k6\ k7\ \alpha)^{k3}_{k2}$$

$$\text{Zag } k0\ k1\ k2\ k3\ k4\ k5\ k6\ k7\ \alpha = \text{ZagStop} \mid \text{Zag } \alpha^{k5}_{k4}\ (\text{Zig } k0\ k1\ k2\ k3\ k4\ k5\ k6\ k7\ \alpha)^{k7}_{k6}$$

There are different ways of annotating recursive type definitions and the strategy we use is probably not the best. One possible way of improving the annotation is to have more than one vector of annotation variables in the left hand side and use them differently depending on if the type definitions is recursive or not, where the recursion occurs and so on. In the analysis presented in this thesis we will however only use one vector of annotation variables.

3.2.1 Type translation

In Figure 3.4 we define the relation, $\hookrightarrow$, between unannotated types and annotated types. Since we have extended the type language with type constructors we must know how to annotate each type constructor. We can annotate a type constructor in two ways. We either annotate it using fresh annotation variables or annotate it using a fixed vector of annotation variables. We create the type environment, Ω, which holds information of how each type constructor should be annotated.

$$\begin{array}{llll} \text{Type environment} & \Omega & ::= & \Omega, \Omega \\ & & \mid & T \hookrightarrow T\ \vec{k} \\ & & \mid & T \hookrightarrow \forall \vec{k}.T\ \vec{k} \end{array}$$

If a type constructor, T, should be annotated using fresh annotation variables then the type environment will contain $(T \hookrightarrow \forall \vec{k}.T\ \vec{k})$, which we refer to as a *quantified* constructor translation. If the type constructor should be annotated using a fixed vector $\vec{k}$ of annotation variables then the type environment will contain $(T \hookrightarrow T\ \vec{k})$, which we will refer to as just a constructor translation. The need for these two different ways of annotating type constructors will be made clear in the next section where the formal rules for annotating data definitions are presented.

3.2.2 Annotating type definitions

The rules for annotating type definitions are presented in Figure 3.5.

The rule for annotating a group of mutually recursive type definitions

$$\text{TDefs}\ \frac{\Omega, \Omega_0 \vdash \vec{td} \hookrightarrow \vec{utd}}{\Omega \vdash \vec{td} \hookrightarrow \vec{utd} : \Omega_1}\ (*)$$

$$(*)\ \begin{array}{l} \Omega_0 \equiv \{T \hookrightarrow T\ \vec{k} \mid T \in dom(\vec{td})\} \\ \Omega_1 \equiv \{T \hookrightarrow \forall \vec{k}.T\ \vec{k} \mid T \in dom(\vec{td})\} \end{array}$$

takes the type environment Ω, containing the constructor translations for the already annotated type definitions, and the type definitions $\vec{td}$. When annotating the type definitions we

$$\text{Var}\ \frac{}{\Omega \vdash \alpha \hookrightarrow \alpha} \qquad \text{Trans}\ \frac{}{\Omega \vdash T \hookrightarrow T\ \vec{k}}\ \ (T \hookrightarrow T\ \vec{k}) \in \Omega$$

$$\text{Q-Trans}\ \frac{}{\Omega \vdash T \hookrightarrow T\ \vec{k'}}\ \ \begin{array}{l}(T \hookrightarrow \forall \vec{k}.T\ \vec{k}) \in \Omega \\ |\vec{k'}| = |\vec{k}|\end{array}$$

$$\text{Arrow}\ \frac{\Omega \vdash \delta_0 \hookrightarrow \sigma \quad \Omega \vdash \delta_1 \hookrightarrow \tau}{\Omega \vdash \delta_0 \to \delta_1\ \hookrightarrow\ \sigma \to \tau} \qquad \text{Forall}\ \frac{\Omega \vdash \delta \hookrightarrow \rho}{\Omega \vdash \forall \alpha.\delta\ \hookrightarrow\ \forall \alpha.\rho}$$

$$\text{App}\ \frac{\Omega \vdash \delta_0 \hookrightarrow \rho_0 \quad \Omega \vdash \delta_1 \hookrightarrow \rho_1}{\Omega \vdash \delta_0\ \delta_1 \hookrightarrow \rho_0\ \rho_1}$$

$$\text{Expr}\ \frac{\Omega \vdash \delta \hookrightarrow \rho}{\Omega \vdash \delta \hookrightarrow \rho^{\pi}} \qquad \text{Bind}\ \frac{\Omega \vdash \delta \hookrightarrow \tau}{\Omega \vdash \delta \hookrightarrow \tau_{\pi}}$$

Figure 3.4: Type translation rules

first create the type environment Ω_0 containing type translations for all type constructors, defined in $\vec{td}$, to the fixed vector of annotation variables $\vec{k}$. We annotate all type definitions in the type environment Ω extended with Ω_0 yielding the annotated type definitions $\vec{utd}$. It is important to remember the restriction that there must not be any annotation variable in the right hand side of a type definition that is not an element of the annotation variable vector. This restriction forces all type definitions in a group to be annotated with the same annotations variables, $\vec{k}$.

Finally we create the type environment Ω_1 where all constructor translations have been quantified and return this environment in conjunction with the annotated type definitions.

A more detailed explanation of how data type definitions are annotated can be found in the Master's thesis of Josef Svenningsson [Sve00].

3.3 Variance

In order to extend the subtyping relation with type constructors we need the notion of variance. To just naively extend the subtyping relation would not work since the source language allows for recursive types. We cannot inductively define a subtyping relation without breaking the recursion in some way. Consider the following type definition:

$$\text{DTree}\ \alpha = \text{DNode}\ \alpha\ (\text{DTree}\ (\alpha, \alpha))$$

When subtyping the type (DTree τ) to the type (DTree τ') we unwind the types and subtype the components yielding

$$\tau \leq \tau' \quad \text{DTree}\ (\tau, \tau) \leq \text{DTree}\ (\tau', \tau')$$

When continuing to unwind the types we get

$$\tau \leq \tau' \quad (\tau, \tau) \leq (\tau', \tau') \quad \text{DTree}\ ((\tau, \tau), (\tau, \tau)) \leq \text{DTree}\ ((\tau', \tau'), (\tau', \tau'))$$

in which we can unwind the tuple types to get

$$\tau \leq \tau' \quad \tau \leq \tau' \quad \tau \leq \tau' \quad \text{DTree}\ ((\tau, \tau), (\tau, \tau)) \leq \text{DTree}\ ((\tau', \tau'), (\tau', \tau'))$$

$$\text{Sum}\ \frac{\Omega \vdash \vec{\delta} \hookrightarrow \vec{\sigma}}{\Omega \vdash C\ \vec{\beta}\ \vec{\delta}\ \hookrightarrow\ C\ \vec{\beta}\ \vec{\sigma}}$$

$$\text{TDef}\ \frac{\Omega \vdash \vec{ts} \hookrightarrow \vec{uts} \qquad (T \hookrightarrow T\ \vec{k}) \in \Omega}{\Omega \vdash T\ \vec{\alpha}\ =\ ts_1 \mid ... \mid ts_n\ \hookrightarrow\ T\ \vec{k}\ \vec{\alpha}\ =\ uts_1 \mid ... \mid uts_n}$$

$$\text{TDefs}\ \frac{\Omega, \Omega_0 \vdash \vec{td} \hookrightarrow \vec{utd}}{\Omega \vdash \vec{td} \hookrightarrow \vec{utd} : \Omega_1}\ (*)$$

$$(*)\ \begin{array}{l} \Omega_0 \equiv \{T \hookrightarrow T\ \vec{k} \mid T \in dom(\vec{td})\} \\ \Omega_1 \equiv \{T \hookrightarrow \forall \vec{k}.T\ \vec{k} \mid T \in dom(\vec{td})\} \end{array}$$

Figure 3.5: Type definition annotation rules

We can see that when trying to subtype this type we will be unfolding it infinitely.

Instead of actually performing the unfolding we calculate an approximation of what subtyping constraints that would be created when unfolding the type. We can for the example above see that the only unique subtyping constraint that will be created is that τ should be a subtype of τ'. We therefore say that the type parameter of DTree has positive *variance*. If the generated subtyping constraint instead was that τ' should be a subtype of τ we would say that the type parameter had negative variance. When now subtyping two DTree types we do not need to unfold the types. Instead we look at the variance of the type parameter and use it to create the subtyping constraints.

Variance is based on the concept of positive and negative positions. We will here explain this concept by giving an example. Consider the following type.

$$(\alpha \to \beta) \to \gamma$$

We start from the right and say that γ occurs at a positive position. We continue to the left and each time we pass a function arrow we negate the current position sign. This gives that β occurs at a negative position while γ and α occur at positive positions. We extend this concept to include usage annotated types in a similar fashion. We do however note that since usage annotations on types are contravariant we must negate the position sign for usage annotations. In the following type

$$\alpha_{k_0}^{k_1} \to \beta^{k_2}$$

β, k_0 and k_1 occur at positive positions while α and k_2 occur at negative positions.

When a variable occurs at a positive position it is said to be covariant $(+)$, as opposite to a negative position which is said to be contravariant $(-)$. If a variable occurs at both a positive and negative position it is said to be bivariant $(\pm)$ while if it does not occur at all it is said to be zerovariant (0).

We define variance together with some operators in Figure 3.6 where also a partial ordering is defined, forming a lattice.

$$\begin{array}{llcll}
\text{Variance} & \gamma & ::= & 0 & \text{zerovariant} \\
 & & | & + & \text{covariant} \\
 & & | & - & \text{contravariant} \\
 & & | & \pm & \text{bivariant}
\end{array}$$

$$\begin{array}{lllll}
\text{Negation} & \bar{0} = 0 & \bar{+} = - & \bar{-} = + & \bar{\pm} = \pm \\
\text{Product} & 0 \cdot \gamma = 0 & + \cdot \gamma = \gamma & - \cdot \gamma = \bar{\gamma} & \pm \cdot \gamma = \begin{cases} 0 & ; \gamma = 0 \\ \pm \end{cases}
\end{array}$$

Lattice

$$\begin{tikzcd}
 & \pm \arrow[dl, no head] \arrow[dr, no head] & \\
+ \arrow[dr, no head] & & - \arrow[dl, no head] \\
 & 0 &
\end{tikzcd}$$

Figure 3.6: Variance and operators

3.3.1 Inferring variance

When inferring variance we need to know the variance of all type parameters in the previously annotated type definitions. Without knowing this we would not be able to handle applications of type constructors correctly. In order to do this we extend the type environment Ω to include the variance of annotation variables and type parameters of type definitions. In the type environment $T\ \vec{\gamma}\ \vec{\gamma}'$ stands for that the type constructor T has annotation variables with the variance $\vec{\gamma}$ and type parameters with the variance $\vec{\gamma}'$.

$$\begin{array}{lccl}
\text{Type environment} & \Omega & ::= & \Omega, \Omega \\
 & & | & T \hookrightarrow T\ \vec{k} \\
 & & | & T \hookrightarrow \forall \vec{k}. T\ \vec{k} \\
 & & | & T\ \vec{\gamma}\ \vec{\gamma}'
\end{array}$$

Types

In Figure 3.7 we present the rules for inferring the variance of type and annotation variables in types. The rules are on the form $\Omega; \Upsilon \vdash t\ \diamond\ \gamma$. Given the type, t, the variance, γ, and the type environment, Ω, we infer the variance of all type and annotation variables and put it in Υ.

We let Υ range over type and annotation variables together with their variance.

$$\text{Variance env.} \quad \Upsilon \quad ::= \quad \alpha^{\gamma} \mid k^{\gamma} \mid \Upsilon, \Upsilon$$

$$\text{Var}\ \frac{}{\Omega;\Upsilon,\alpha^{\gamma}\vdash\alpha\ \diamond\ \gamma'}\ \gamma'\leq\gamma \qquad \text{AVar}\ \frac{}{\Omega;\Upsilon,k^{\gamma}\vdash k\ \diamond\ \gamma'}\ \gamma'\leq\gamma$$

$$\text{Arrow}\ \frac{\Omega;\Upsilon\vdash\sigma\ \diamond\ \bar{\gamma}\quad \Omega;\Upsilon\vdash\tau\ \diamond\ \gamma}{\Omega;\Upsilon\vdash\sigma\rightarrow\tau\ \diamond\ \gamma} \qquad \text{Forall}\ \frac{\Omega;\Upsilon,\alpha^{\pm}\vdash\rho\ \diamond\ \gamma}{\Omega;\Upsilon\vdash\forall\alpha.\rho\ \diamond\ \gamma}$$

$$\text{App}\ \frac{\Omega;\Upsilon\vdash\rho\ \diamond\ \gamma\quad \Omega;\Upsilon\vdash\rho'\ \diamond\ \pm}{\Omega;\Upsilon\vdash\rho\ \rho'\ \diamond\ \gamma}$$

$$\text{Tycon}\ \frac{\Omega;\Upsilon\vdash k'_i\ \diamond\ (\gamma\cdot\gamma_i)\quad \Omega;\Upsilon\vdash\rho_j\ \diamond\ (\gamma\cdot\gamma'_j)}{\Omega;\Upsilon\vdash T\ \vec{k'}\ \vec{\rho}\ \diamond\ \gamma}\ (T\ \vec{\gamma}\ \vec{\gamma}')\in\Omega$$

$$\text{Expr}\ \frac{\Omega;\Upsilon\vdash\rho\ \diamond\ \gamma\quad \Omega;\Upsilon\vdash k\ \diamond\ \bar{\gamma}}{\Omega;\Upsilon\vdash\rho^{k}\ \diamond\ \gamma} \qquad \text{Bind}\ \frac{\Omega;\Upsilon\vdash\tau\ \diamond\ \gamma\quad \Omega;\Upsilon\vdash k\ \diamond\ \bar{\gamma}}{\Omega;\Upsilon\vdash\tau_k\ \diamond\ \gamma}$$

Figure 3.7: Variance inference rules

The rule for type variables

$$\text{Var}\ \frac{}{\Omega;\Upsilon,\alpha^{\gamma}\vdash\alpha\ \diamond\ \gamma'}\ \gamma'\leq\gamma$$

says that if we want the type variable α to have the variance γ' then α^{γ}, where γ is greater than or equal to γ', must be an element of the variance environment. The use of γ is necessary, if we would force $\alpha^{\gamma'}$ to be an element of the variance environment we would not be able to handle variables that occur at both positive and negative positions. That can now be solved by observing that $-\leq\pm$ and $+\leq\pm$. What will happen is that the variables occurring at both positive and negative positions will have their variance inferred to $\pm$.

The rule for annotation variables is almost identical to the one for annotation variables and need no further explanation.

As described in Section 3.3 we must, when passing a function arrow,

$$\text{Arrow}\ \frac{\Omega;\Upsilon\vdash\sigma\ \diamond\ \bar{\gamma}\quad \Omega;\Upsilon\vdash\tau\ \diamond\ \gamma}{\Omega;\Upsilon\vdash\sigma\rightarrow\tau\ \diamond\ \gamma}$$

negate the variance for the argument. The negated variance is denoted $\bar{\gamma}$.

When introducing a universally quantified type variable we assume that it is bivariant.

$$\text{Forall}\ \frac{\Omega;\Upsilon,\alpha^{\pm}\vdash\rho\ \diamond\ \gamma}{\Omega;\Upsilon\vdash\forall\alpha.\rho\ \diamond\ \gamma}$$

In an application where the type constructor is unknown

$$\text{App}\ \frac{\Omega;\Upsilon\vdash\rho\ \diamond\ \gamma\quad \Omega;\Upsilon\vdash\rho'\ \diamond\ \pm}{\Omega;\Upsilon\vdash\rho\ \rho'\ \diamond\ \gamma}$$

$$\text{Sum}\ \frac{\Omega; \Upsilon, \vec{\beta}^{\pm} \vdash \vec{\sigma}\ \diamond\ +}{\Omega; \Upsilon \vdash C\ \vec{\beta}\ \vec{\sigma}}$$

$$\text{TDef}\ \frac{\Omega; \vec{k}^{\vec{\gamma}}, \vec{\alpha}^{\vec{\gamma}'} \vdash \vec{ts}}{\Omega \vdash T\ \vec{k}\ \vec{\alpha}\ =\ ts_1 \mid ... \mid ts_n\ :\ T\ \vec{\gamma}\ \vec{\gamma}'}$$

Figure 3.8: Type definition variance inference rules

we do not know how the argument will be used, it could occur at both positive and negative positions. We must therefore force the argument type to be bivariant. This reduces the precision, if the argument is only used at positive positions, and therefore safely could be regarded as being covariant we would still force it to be bivariant.

An example of an application with an unknown type constructor is the following Haskell type definition.

$$\texttt{data}\ \text{App}\ (f :: * \rightarrow *)\ v = \text{App}\ (f\ v)$$

Here the variance of v would be inferred to $\pm$ since the type constructor, f, is unknown.

When applying a known type constructor, like the type constructor List, we can do better. The variance of all type and annotation variables are available in the annotated type definition. For each annotation variable we infer the product of the given variance, γ, and the variance of the variable found in the type definition, γ_i. This means that for all annotations k_i we will infer the variance $(\gamma \cdot \gamma_i)$. We do the same for the type parameters, $\vec{\rho}$.

$$\text{Tycon}\ \frac{\Omega; \Upsilon \vdash k'_i\ \diamond\ (\gamma \cdot \gamma_i) \quad \Omega; \Upsilon \vdash \rho_j\ \diamond\ (\gamma \cdot \gamma'_j)}{\Omega; \Upsilon \vdash T\ \vec{k'}\ \vec{\rho}\ \diamond\ \gamma} \quad (T\ \vec{\gamma}\ \vec{\gamma}') \in \Omega$$

It is important to notice that there is an overlap between the App and the Tycon rule. This is resolved by always choosing the Tycon rule when applying a known type constructor and otherwise use the App rule.

The rules for expression and binding types are trivial. The annotation variables residing on binding and expression types get the negated variance of their host types.

Type definitions

Now we turn to type definitions. The rules for inferring the variance of type and annotation variables in type definitions are presented in Figure 3.8.

The rule for inferring variance in a type summand is straightforward.

$$\text{Sum}\ \frac{\Omega; \Upsilon, \vec{\beta}^{\pm} \vdash \vec{\sigma}\ \diamond\ +}{\Omega; \Upsilon \vdash C\ \vec{\beta}\ \vec{\sigma}}$$

The constructor arguments occur at positive positions. Since we are not interested in the variance of the existentially quantified types we assume that they are bivariant.

When inferring variance in a type definition

$$\text{TDef}\ \frac{\Omega; \vec{k}^{\vec{\gamma}}, \vec{\alpha}^{\vec{\gamma}'} \vdash \vec{ts}}{\Omega \vdash T\ \vec{k}\ \vec{\alpha}\ =\ ts_1 \mid ... \mid ts_n\ :\ T\ \vec{\gamma}\ \vec{\gamma}'}$$

$$\text{Var}\ \frac{}{\top; \Omega \vdash \alpha \ \leq^{\gamma} \ \alpha}$$

$$\text{Tycon}\ \frac{\Pi_0; \Omega \vdash \pi_i \leq^{\gamma \cdot \gamma'_i} \pi'_i \quad \Pi_1; \Omega \vdash \rho_j \leq^{\gamma \cdot \gamma''_j} \rho'_j}{\Pi_0 \wedge \Pi_1; \Omega \vdash T\ \vec{\pi}\ \vec{\rho} \ \leq^{\gamma} \ T\ \vec{\pi'}\vec{\rho'}} \quad (T\ \vec{\gamma}'\ \vec{\gamma}'') \in \Omega$$

$$\text{App}\ \frac{\Pi_0; \Omega \vdash \rho_0 \ \leq^{\gamma} \ \rho'_0 \quad \Pi_1; \Omega \vdash \rho_1 \ \leq^{\pm} \ \rho'_1}{\Pi_0 \wedge \Pi_1; \Omega \vdash \rho_0\ \rho_1 \ \leq^{\gamma} \ \rho'_0\ \rho'_1}$$

$$\text{Arrow}\ \frac{\Pi_0; \Omega \vdash \sigma' \ \leq^{\gamma} \ \sigma \quad \Pi_1; \Omega \vdash \tau \ \leq^{\gamma} \ \tau'}{\Pi_0 \wedge \Pi_1; \Omega \vdash \sigma \to \tau \ \leq^{\gamma} \ \sigma' \to \tau'}$$

$$\text{Forall}\ \frac{\Pi; \Omega \vdash \rho_0 \ \leq^{\gamma} \ \rho_1}{\Pi; \Omega \vdash \forall \alpha.\rho_0 \ \leq^{\gamma} \ \forall \alpha.\rho_1}$$

$$\text{Expr}\ \frac{\Pi_0; \Omega \vdash \rho \ \leq^{\gamma} \ \rho' \quad \Pi_1; \Omega \vdash \pi' \ \leq^{\gamma} \ \pi}{\Pi_0 \wedge \Pi_1; \Omega \vdash \rho^{\pi} \ \leq^{\gamma} \ \rho'^{\pi'}}$$

$$\text{Bind}\ \frac{\Pi_0; \Omega \vdash \tau \ \leq^{\gamma} \ \tau' \quad \Pi_1; \Omega \vdash \pi' \ \leq^{\gamma} \ \pi}{\Pi_0 \wedge \Pi_1; \Omega \vdash \tau_{\pi} \ \leq^{\gamma} \ \tau'_{\pi'}}$$

$$\text{An}(0)\ \frac{}{\top; \Omega \vdash \pi_0 \leq^{0} \pi_1}$$

$$\text{An}(+)\ \frac{}{\pi_0 \leq \pi_1; \Omega \vdash \pi_0 \leq^{+} \pi_1}$$

$$\text{An}(-)\ \frac{}{\pi_1 \leq \pi_0; \Omega \vdash \pi_0 \leq^{-} \pi_1}$$

$$\text{An}(\pm)\ \frac{}{\pi_0 \leq \pi_1 \wedge \pi_1 \leq \pi_0; \Omega \vdash \pi_0 \leq^{\pm} \pi_1}$$

Figure 3.9: Subtyping rules

we infer the variance in all type summands yielding the vectors $\vec{k}^{\vec{\gamma}}$ and $\vec{\alpha}^{\vec{\gamma}'}$ from which the variance is returned in the term $T\ \vec{\gamma}\ \vec{\gamma}'$.

3.4 Subtyping

In Figure 3.9 we extend the subtyping relation with a rule for application of known type constructors. This leads to a similar overlap between the App rule and the Tycon rule as found in the previous section. We solve it in the same way by always choosing the Tycon rule when trying to subtype an application of a known type constructor and otherwise use the App rule.

Since subtyping depends on the variance of the annotation variables and type parameters found in the type definitions we must add the type environment, Ω, to the subtyping rules.

$$\text{Data}\ \frac{\Omega \vdash \vec{td} \hookrightarrow \vec{utd} : \Omega_0 \quad \Omega, \vec{\Omega_1} \vdash utd_i : \vec{\Omega_1} \quad \Pi, \Omega, \Omega_0, \Omega_1, \vec{utd} \vdash p}{\Pi; \Omega \vdash \texttt{data}\ \vec{td}\ \texttt{in}\ p}$$

$$\text{Main}\ \frac{\Pi; \Omega; \emptyset \vdash e : \tau}{\Pi; \Omega \vdash \texttt{main}\ e}$$

Figure 3.10: Program typing rules

The subtyping judgements are on the form $\Pi; \Omega \vdash \tau \leq^{\gamma} \tau'$ where τ and τ' are value, expression or binding types. The rules takes the type environment Ω, the types and the variance γ and yields the constraints Π. The variance γ is used to control what constraints we get. If the variance is + we get the usual constraints while if it is - we get the negated constraints. This is especially useful in the Tycon rule where we let the variance found in the type definitions control what constraints we get. An example of how it works is that $\Pi; \Omega \vdash \tau \leq^{+} \tau'$ generates the same constraints as $\Pi; \Omega \vdash \tau' \leq^{-} \tau$.

When applying a known type constructor

$$\text{Tycon}\ \frac{\Pi_0; \Omega \vdash \pi_i \leq^{\gamma \cdot \gamma'_i} \pi'_i \quad \Pi_1; \Omega \vdash \rho_j \leq^{\gamma \cdot \gamma''_j} \rho'_j}{\Pi_0 \wedge \Pi_1; \Omega \vdash T\ \vec{\pi}\ \vec{\rho}\ \leq^{\gamma}\ T\ \vec{\pi'} \vec{\rho'}} \qquad (T\ \vec{\gamma'}\ \vec{\gamma''}) \in \Omega$$

we must take the variance of the type variables and annotations into account. The annotations $\vec{\pi}$ and $\vec{\pi}'$ are subtyped using the variances, $\vec{\gamma}'$, taken from the type definition. The type parameters are subtyped using the variances, $\vec{\gamma}''$. Note that this rule works even if the type constructor is not saturated.

3.5 Typing rules

The typing rules for programs are presented in Figure 3.10.

Some typing rules need the annotated type definitions and we therefore extend the type environment Ω to also range over annotated type definitions.

When annotating a group of data type definitions $\vec{td}$ we first annotate them yielding the annotated type definitions $\vec{utd}$ and the type environment Ω_0. We then infer the variance of the annotation variables and type parameters yielding the type environment $\vec{\Omega_1}$. Note that the variance is inferred using this environment. Finally we type the program p in the environment $\Omega, \Omega_0, \vec{\Omega_1}$ extended with the annotated type definitions utd, yielding the constraints Π which is returned by the rule.

The annotation of the main expression is not affected by the introduction of data types, except that the type environment Ω has been added to the rule, and is therefore not described here.

$$\text{Abs}\ \frac{\Omega \vdash \delta \hookrightarrow \rho^{\pi_1}_{\pi_0} \quad \Pi_0;\Omega;\Gamma, x : \rho^{\pi_1}_{\pi_0} \vdash e \ : \ \tau}{\Pi_0 \wedge \Pi_1;\Omega;\Gamma \vdash \lambda x : \delta.e \ : \ \rho^{\pi_1}_{\pi_0} \to \tau} \quad (*)$$

$$(*)\ \Pi_1 \equiv \begin{cases} \omega \leq \pi_0 \wedge \omega \leq \pi_1 & \textbf{if } \mathit{occur}(x,e) > 1 \\ \top & \textbf{otherwise} \end{cases}$$

$$\text{Lit}\ \frac{}{\top;\Omega;\Gamma \vdash n \ : \ \mathit{Int}}$$

$$\text{Con}\ \frac{\Omega \vdash \vec{\delta_0} \hookrightarrow \vec{\rho_0} \quad \Omega \vdash \vec{\delta_1} \hookrightarrow \vec{\rho_1} \quad \Omega \vdash T \hookrightarrow T\ \vec{\pi}}{\Pi_0 \wedge \Pi_1;\Omega;\Gamma \vdash C\ \vec{\delta_0}\ \vec{\delta_1}\ \vec{x} \ : \ T\ \vec{\pi}\ \vec{\rho_0}} \quad (*)$$

$$(*)\quad \begin{array}{l} \Gamma(x_i) = {\chi_i}^{\pi_{1i}}_{\pi_{0i}} \\ (T\ \vec{k}\ \vec{\alpha} = ...\ |\ C\ \vec{\beta}\ \vec{\sigma}\ \ |\ ...) \in \Omega \\ \vec{\sigma'} \equiv \vec{\sigma}[\vec{\alpha} := \vec{\rho_0}, \vec{\beta} := \vec{\rho_1}, \vec{k} := \vec{\pi}] \\ \Pi_0 \equiv \bigwedge \Pi_{0i}\ \texttt{where}\ \Pi_{0i} \vdash \chi_i \prec \rho_i \\ \Pi_1 \equiv \bigwedge \Pi_{1i}\ \texttt{where}\ \Pi_{1i};\Omega \vdash {\rho_i}^{\pi_{1i}}_{\pi_{0i}} \leq^{+} \sigma'_i \end{array}$$

Figure 3.11: Typing rules for values

3.5.1 Typing rules for values

The typing rules for values are presented in Figure 3.11 where we have added a rule for constructor values.

$$\text{Con}\ \frac{\Omega \vdash \vec{\delta_0} \hookrightarrow \vec{\rho_0} \quad \Omega \vdash \vec{\delta_1} \hookrightarrow \vec{\rho_1} \quad \Omega \vdash T \hookrightarrow T\ \vec{\pi}}{\Pi_0 \wedge \Pi_1;\Omega;\Gamma \vdash C\ \vec{\delta_0}\ \vec{\delta_1}\ \vec{x} \ : \ T\ \vec{\pi}\ \vec{\rho_0}} \quad (*)$$

$$(*)\quad \begin{array}{l} \Gamma(x_i) = {\chi_i}^{\pi_{1i}}_{\pi_{0i}} \\ (T\ \vec{k}\ \vec{\alpha} = ...\ |\ C\ \vec{\beta}\ \vec{\sigma}\ \ |\ ...) \in \Omega \\ \vec{\sigma'} \equiv \vec{\sigma}[\vec{\alpha} := \vec{\rho_0}, \vec{\beta} := \vec{\rho_1}, \vec{k} := \vec{\pi}] \\ \Pi_0 \equiv \bigwedge \Pi_{0i}\ \texttt{where}\ \Pi_{0i} \vdash \chi_i \prec \rho_i \\ \Pi_1 \equiv \bigwedge \Pi_{1i}\ \texttt{where}\ \Pi_{1i};\Omega \vdash {\rho_i}^{\pi_{1i}}_{\pi_{0i}} \leq^{+} \sigma'_i \end{array}$$

When typing $C\ \vec{\delta_0}\ \vec{\delta_1}\ \vec{x}$ we know that $\vec{\delta_0}$ are the type parameters, $\vec{\delta_1}$ the witnesses and $\vec{x}$ the constructor arguments. We substitute the given types, $\vec{\rho_0}$ and $\vec{\rho_1}$, for the type parameters and existentially quantified type variables in the constructor argument types, $\vec{\sigma}$, yielding $\vec{\sigma'}$. We also substitute the annotations, $\vec{\pi}$, for the parameterised annotation variables, $\vec{k}$. We then instantiate the type schemas for the constructor arguments and subtype the instantiated types, together with their outer annotations, to the constructor argument's types $\vec{\sigma'}$.

The type of the constructor value is the type constructor, T, applied to the annotations $\vec{\pi}$ and the type parameters, $\vec{\rho_0}$.

Values

$$occur(x, \lambda y : \delta . e) = \begin{cases} 0 & \textbf{if } x = y \\ occur(x, e) & \textbf{otherwise} \end{cases}$$
$$occur(x, n) = 0$$
$$occur(x, C\ \vec{\delta_0}\ \vec{\delta_1}\ \vec{x}) = \sum occur(x, x_i)$$

Expressions

$$occur(x, v^{\pi}) = occur(x, v)$$
$$occur(x, y) = \begin{cases} 1 & \textbf{if } x = y \\ 0 & \textbf{otherwise} \end{cases}$$
$$occur(x, e\ y) = occur(x, e) + occur(x, y)$$
$$occur(x, \Lambda\alpha . e) = occur(x, e)$$
$$occur(x, e@\delta) = occur(x, e)$$
$$occur(x, \texttt{let}\ \vec{x} : \vec{\delta} \overset{\vec{\pi}}{=} \vec{e}\ \texttt{in}\ e) =$$
$$\begin{cases} 0 & \textbf{if } x \in \{\vec{x}\} \\ (\sum occur(x, e_i)) + occur(x, e) & \textbf{otherwise} \end{cases}$$
$$occur(x, \texttt{case}\ e\ \texttt{of}\ (y : \delta)\ \vec{alt}) =$$
$$\begin{cases} occur(x, e) & \textbf{if } x = y \\ occur(x, e) + \texttt{max}\ occur(x, alt_i) & \textbf{otherwise} \end{cases}$$

Alternatives

$$occur(x, C\ \vec{\beta}\ \vec{x} \Rightarrow e) = \begin{cases} 0 & \textbf{if } x \in \{\vec{x}\} \\ occur(x, e) & \textbf{otherwise} \end{cases}$$
$$occur(x, n \Rightarrow e) = occur(x, e)$$
$$occur(x, \_ \Rightarrow e) = occur(x, e)$$

Figure 3.12: Occur function

$$\text{Value}\ \frac{\Pi_0; \Gamma \vdash v \ : \ \rho}{\Pi_0 \wedge \pi' \leq \pi \wedge \Pi_1; \Gamma \vdash v^{\pi} \ : \ \rho^{\pi'}}\ (*)$$

$$(*) \quad \Pi_1 \equiv \bigwedge_{x \in fv(v)} \Big(\ \pi' \leq \pi_0 \wedge \pi' \leq \pi_1 \quad \textbf{where } \Gamma(x) = \chi^{\pi_1}_{\pi_0} \ \Big)$$

$$\text{Var}\ \frac{}{\Pi; \Omega; \Gamma, x : \chi^{\pi_1}_{\pi_0} \vdash x \ : \ \rho^{\pi_1}} \quad \Pi \vdash \chi \prec \rho$$

$$\text{App}\ \frac{\Pi_0; \Omega; \Gamma \vdash e \ : \ (\sigma \rightarrow \tau)^{\pi}}{\Pi_0 \wedge \Pi_1 \wedge \Pi_2; \Omega; \Gamma, x : \chi^{\pi_1}_{\pi_0} \vdash e \ x \ : \ \tau} \quad \begin{array}{l} \Pi_1 \vdash \chi \prec \rho \\ \Pi_2; \Omega \vdash \rho^{\pi_1}_{\pi_0} \leq^{+} \sigma \end{array}$$

$$\text{Gen}\ \frac{\Pi; \Omega; \Gamma \vdash e \ : \ \rho^{\pi}}{\Pi; \Omega; \Gamma \vdash \Lambda\alpha.e \ : \ (\forall \alpha.\rho)^{\pi}} \quad \alpha \notin fv(\Gamma)$$

$$\text{Inst}\ \frac{\Omega \vdash \delta \hookrightarrow \rho_0 \quad \Pi; \Omega; \Gamma \vdash e \ : \ (\forall \alpha.\rho_1)^{\pi}}{\Pi; \Omega; \Gamma \vdash e@\delta \ : \ \rho_1[\alpha := \rho_0]^{\pi}}$$

$$\text{Case}\ \frac{\Omega \vdash \delta \hookrightarrow \rho \quad \Pi_0; \Omega; \Gamma \vdash e : \rho'^{\pi'} \quad \Pi_1; \Omega; \Gamma, x : \rho^{\pi}_{\pi} \vdash \vec{alt} \ : \ \rho \Rightarrow \vec{\tau}}{\Pi_0 \wedge \Pi_1 \wedge \Pi_2 \wedge \Pi_3 \wedge \Pi_4; \Omega; \Gamma \vdash \texttt{case}\ e\ \texttt{of}\ (x : \delta)\ \vec{alt} \ : \ \tau}\ (*)$$

$$(*) \quad \begin{array}{l} \Pi_2 \equiv \begin{cases} \omega \leq \pi & \textbf{if } x \in fv(\vec{alt}) \\ T & \textbf{otherwise} \end{cases} \\ \Pi_3; \Omega \vdash \tau_i \ \leq^{+} \ \tau \\ \Pi_4; \Omega \vdash \rho'^{\pi'} \ \leq^{+} \ \rho^{\pi} \end{array}$$

Figure 3.13: Typing rules for expressions

$$\text{Alt-con}\ \frac{\Pi_0; \Omega; \Gamma, \Gamma' \vdash e \ : \ \tau}{\Pi_0 \wedge \Pi_1; \Omega; \Gamma \vdash C\ \vec{\beta}\ \vec{x} \Rightarrow e \ : \ T\ \vec{\pi}\ \vec{\rho_0} \Rightarrow \tau}\ (*)$$

$$(*)\ \begin{array}{l} (T\ \vec{k}\ \vec{\alpha} = ... \mid C\ \vec{\beta}\ \vec{\sigma}\ \mid ...) \in \Omega \\ \Gamma' \equiv \vec{x} : \vec{\sigma}[\vec{\alpha} := \vec{\rho_0}, \vec{k} := \vec{\pi}] \\ \Pi_2 \equiv \bigwedge_{(x:\chi^{\pi_1}_{\pi_0}) \in \Gamma'} \begin{cases} \omega \leq \pi_0 \wedge \omega \leq \pi_1 & \textbf{if } \mathit{occur}(x, e) > 1 \\ T & \textbf{otherwise} \end{cases} \end{array}$$

$$\text{Alt-lit}\ \frac{\Pi; \Omega; \Gamma \vdash e : \tau}{\Pi; \Omega; \Gamma \vdash n \Rightarrow e \ : \ \mathit{Int} \Rightarrow \tau}$$

$$\text{Alt-def}\ \frac{\Pi; \Omega; \Gamma \vdash e : \tau}{\Pi; \Omega; \Gamma \vdash \_ \Rightarrow e \ : \ \rho \Rightarrow \tau}$$

Figure 3.14: Typing rules for case alternatives

3.5.2 Typing rules for expressions

The typing rules for expressions are presented in Figure 3.13. We have added a typing rule for case expressions.

When typing a case expression

$$\text{Case}\ \frac{\Omega \vdash \delta \hookrightarrow \rho \quad \Pi_0; \Omega; \Gamma \vdash e : \rho'^{\pi'} \quad \Pi_1; \Omega; \Gamma, x : \rho^{\pi}_{\pi} \vdash \vec{alt}\ :\ \rho \Rightarrow \vec{\tau}}{\Pi_0 \wedge \Pi_1 \wedge \Pi_2 \wedge \Pi_3 \wedge \Pi_4; \Omega; \Gamma \vdash \texttt{case}\ e\ \texttt{of}\ (x : \delta)\ \vec{alt}\ :\ \tau}\ (*)$$

$$(*)\ \begin{array}{l} \Pi_2 \equiv \begin{cases} \omega \leq \pi & \textbf{if } x \in \mathit{fv}(\vec{alt}) \\ T & \textbf{otherwise} \end{cases} \\ \Pi_3; \Omega \vdash \tau_i \ \leq^+ \ \tau \\ \Pi_4; \Omega \vdash \rho'^{\pi'} \ \leq^+ \ \rho^{\pi} \end{array}$$

we type the scrutinee to the type $\rho'^{\pi'}$ which we subtype to ρ^{π}. This is the type which is used when typing the case alternatives. The rules for typing case alternatives have the form $\Pi; \Omega; \Gamma \vdash alt : \rho \Rightarrow \tau$ where ρ is the type of the scrutinee and τ is the type of the branch. We type all alternatives and subtype all types of the branches to the type τ which is returned as the type of the expression.

We must also ensure that if the variable, x, occurs in the case alternatives then its annotation variable, π, is forced to be equal to ω. This is done by the first side condition.

$$\text{Let}\ \frac{\Omega \vdash \vec{\delta} \hookrightarrow \vec{\rho} \quad \vec{\Pi_3}; \Omega; \Gamma' \vdash \vec{e}\ :\ \vec{\rho'}^{\vec{\pi}'''} \qquad \Pi_2; \Omega; \Gamma' \vdash e\ :\ \tau}{\Pi_0 \wedge \Pi_1 \wedge \texttt{let}\ \vec{\phi}\ \texttt{in}\ \Pi_2; \Omega; \Gamma \vdash \texttt{let}\ \vec{x} : \vec{\delta}\ \overset{\vec{\pi}}{=}\ \vec{e}\ \texttt{in}\ e\ :\ \tau}\ (*)$$

$$\Gamma'(y) \equiv \begin{cases} (\forall \vec{k_i}.\rho_i \mid l_i\ \vec{k_i})^{\pi''_i}_{\pi'_i} & \texttt{if}\ y = x_i \\ \Gamma(y) & \texttt{otherwise} \end{cases}$$

$$(*)\quad \begin{array}{l} \vec{k_i} \notin \mathit{fav}(\Gamma', \pi''_i) \\ \vec{k'_i} \notin \mathit{fav}(\Gamma', \rho_i^{\pi''_i}) \\ \phi_i \equiv (l_i\ \vec{k_i} = \exists \vec{k'_i}.\Pi_{3i} \wedge \Pi_{4i})\ \ \texttt{where}\ \ \Pi_{4i}; \Omega \vdash {\rho'}_i^{\pi'''_i} \leq^+ \rho_i^{\pi''_i} \end{array}$$

$$\Pi_0 \equiv \bigwedge(\pi'_i \leq \pi_i)$$

$$\Pi_1 \equiv \bigwedge \left\{ \begin{array}{ll} \omega \leq \pi'_i \wedge \omega \leq \pi''_i & \textbf{if}\ \mathit{occur}(x_i, e) + \sum \mathit{occur}(x_i, e_i) > 1 \\ T & \textbf{otherwise} \end{array} \right\}$$

Figure 3.15: Typing rule for let expressions

3.5.3 Typing rules for case alternatives

The typing rules for literal and default case alternatives, presented in Figure 3.14, are trivial. The rule for constructor alternatives is however more subtle.

$$\text{Alt-con}\ \frac{\Pi_0; \Omega; \Gamma, \Gamma' \vdash e\ :\ \tau}{\Pi_0 \wedge \Pi_1; \Omega; \Gamma \vdash C\ \vec{\beta}\ \vec{x} \Rightarrow e\ :\ T\ \vec{\pi}\ \vec{\rho_0} \Rightarrow \tau}\ (*)$$

$$(*)\quad \begin{array}{l} (T\ \vec{k}\ \vec{\alpha} = ... \mid C\ \vec{\beta}\ \vec{\sigma}\ \ \mid ...) \in \Omega \\ \Gamma' \equiv \vec{x} : \vec{\sigma}[\vec{\alpha} := \vec{\rho_0}, \vec{k} := \vec{\pi}] \\ \Pi_2 \equiv \bigwedge_{(x:\chi^{\pi_1}_{\pi_0}) \in \Gamma'} \begin{cases} \omega \leq \pi_0 \wedge \omega \leq \pi_1 & \textbf{if}\ \mathit{occur}(x, e) > 1 \\ T & \textbf{otherwise} \end{cases} \end{array}$$

We type the branch expression, e, in the environment, Γ', where all constructor arguments, $\vec{x}$, have had their types instantiated with the type parameters $\vec{\rho_0}$ and usage annotations $\vec{\pi}$. We must also, as is done by the last side condition, ensure that all bindings that occur more than once in the branch expression have their outer annotations constrained by ω.

3.5.4 Typing rule for let expressions

The typing rule for let expressions, presented in Figure 3.15, is not affected by the introduction of data types except that the type environment Ω has been added and that the subtyping is indexed with variance.

Chapter 4

Measurements

4.1 Setup

The measurements have been carried out on an Athlon XP 2000+ equipped with 1.5 GB internal memory running Gentoo Linux. The machine is besides its higher amount of memory a standard desktop PC.

All running times are the minimum of three successive runs. The time have been measured using the GNU time utility. The reported system and user times have been added to get the running time.

4.2 Programs

We ran the analysis on 33 programs. Most of the programs are taken from the *real* category in the nofib suite [Par93]. The nofib suite is a benchmark suite consisting of Haskell programs, the *real* category consists of real world programs. We have excluded two programs, HMMS and PolyGP. HMMS was excluded because it is not a single program, PolyGP because we could not get it to compile with GHC.

Although being real world programs the nofib programs are medium sized. We have therefore added five other programs. The programs we have added are: GHC - the Glasgow Haskell Compiler, which as far as we know is the largest Haskell program, SLC - an interpreter for a C like language, CoreSA - the usage analysis itself, SSC - a compiler for the lazy functional language STG and GF - a grammatical framework.

4.3 Haskell versus Core

Since the usage analysis runs on Core programs all Haskell programs must first be translated into Core. The size of the Core programs resulting from the Haskell programs is shown in Figure 4.1.

We have chosen to plot all diagrams using logarithmic scales to make them more readable. Each diagram has a dotted reference line showing the slope that a linear function would have.

In the diagram it is interesting to see that the size of the Core programs seems to be fairly linear in the size of the Haskell programs. It is known that going from an implicitly typed program into an explicitly typed program might give an exponential blowup in code size but as the diagram indicates this does usually not happen in practice.

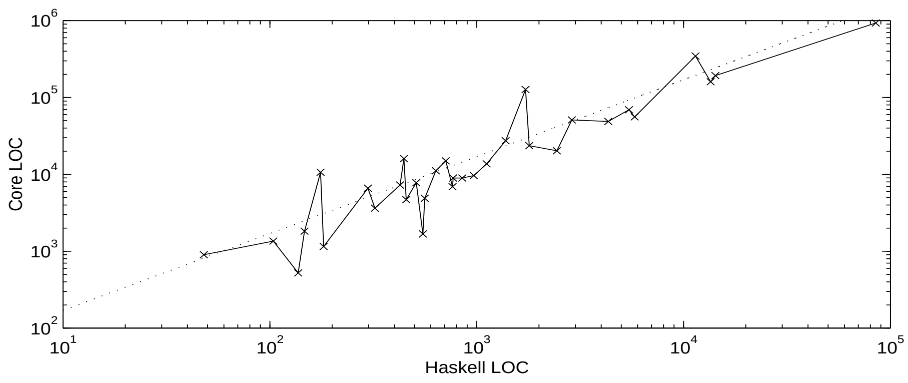

Figure 4.1: Lines of Core code versus lines of Haskell code

4.4 Analysis

The analysis is divided into two parts: the constraint generation and the constraint solving. Results for both parts are presented in the following sections.

4.4.1 Constraint generation

The implementation of the constraint generation should be considered as a prototype. The emphasis has been on evaluating the scalability of the constraint solver. Since almost no work has been done on optimising the implementation of the constraint generation there are strong reasons for believing that it could run considerably faster.

When generating the constraints we had a threshold for how many unique annotation variables that could be used to annotate a type definition. Without having a threshold the number of unique annotation variables could increase exponentially and the size of the constraints would be to large to handle. We have set to threshold to 50 unique annotation variables. If a type definition requires more annotation variables we start to reuse already used annotation variables. This will of course lead to reduced precision and different strategies of reusing the annotation variables will reduce the precision in different ways. We currently use a strategy that when the threshold is reached replaces all annotation variables of each variance with a single annotation variable. What we believe is good with this strategy is that annotation variables with different variance will never be mixed with each other. Whether this leads to an increased precision is hard to tell and more work needs to be done on evaluating different strategies.

The running times for the constraint generation are presented in figure 4.2. The columns in the table contain the number of lines of Haskell code (excluding standard libraries), the number of Core lines (excluding standard libraries) and the constraint generation time. Generation time versus lines of code is plotted in Figure 4.3 which shows that the constraint generation scales up for all tested programs. Constraint size versus lines of Core code is plotted in Figure 4.4 which shows that the constraint size seems to be almost linear in lines of Core code. It is however important to remember that lines of code is not an very exact measure since a code fragment spanning over many lines may generate few constraints while a code fragment spanning over few lines may generated many constraints.

Program	Haskell LOC	Core LOC	Time (s)
rsa	48	902	6.2
grep	104	1358	6.9
maillist	137	522	3.9
compress2	147	1825	9.5
linear	176	10659	40.8
mkhprog	182	1154	5.1
pic	298	6610	30.2
prolog	322	3630	24.4
lift	426	7256	20.1
scs	445	16021	30.6
hidden	456	4693	41.7
gamteb	510	7804	45.3
compress	549	1679	15.1
infer	561	4886	43.8
gg	635	11156	32.3
bspt	708	15001	59.5
fem	764	6922	59.0
hpg	770	8981	33.7
symalg	851	8973	34.5
reptile	969	9651	41.5
fulsom	1117	13701	43.6
ebnf2ps	1380	27426	56.8
cacheprof	1723	126898	30.7
fluid	1796	23628	71.3
parser	2442	20320	12.5
slc	2887	51074	83.9
rx	4324	48742	221.5
veritas	5452	69360	140.4
anna	5802	55850	135.6
coresa	11430	346712	336.4
ssc	13519	159845	316.3
gf	14252	192940	497.8
ghc	85033	931588	4024.54

Figure 4.2: Constraint generation times

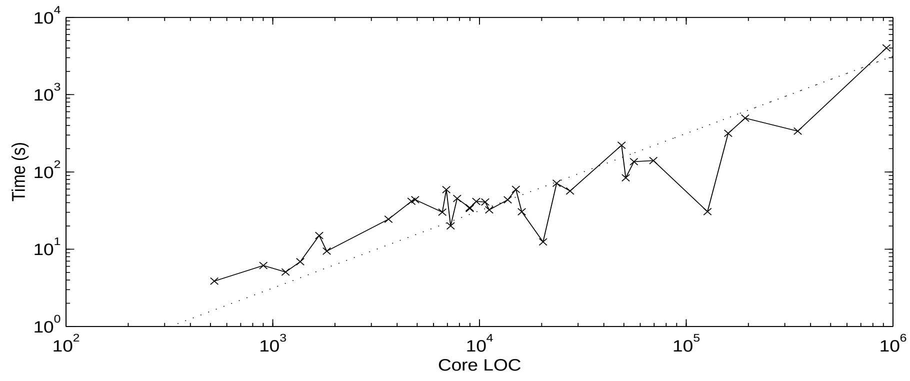

Figure 4.3: Constraint generation time versus lines of Core code

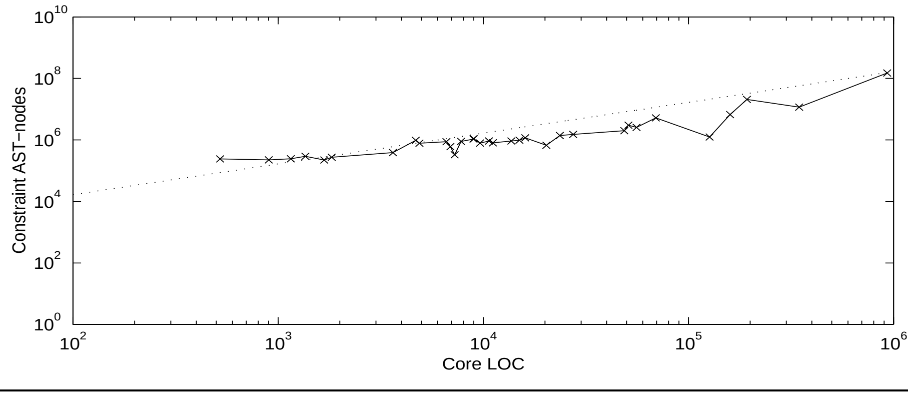

Figure 4.4: Constraint AST-nodes versus lines of Core code

4.4.2 Constraint solver

The constraint solver, whose solving algorithm is described in the unpublished paper [GSG03], is implemented in O'Caml[1].

Some work has been done on optimising the implementation but it should also be regarded as a prototype. An example of this is that up to 60% of the running time is consumed by the O'Caml garbage collector. With a tailor made memory management system this is believed to be improved.

The solving times are presented in Figure 4.5. The columns contain the number of lines of Haskell code (excluding standard libraries), the size of the generated constraints (measured as the number of abstract syntax nodes, *including* standard libraries), the running time and the amount of heap (as reported by the O'Caml memory manager) used by the constraint solver. In Figure 4.6 running time versus constraint size is plotted. The diagram shows that the constraint solver scales up for all but one program which is veritas. This is however not the whole truth since when solving the constraints for GHC the solver runs out of memory. Therefore GHC is not present in the diagrams showing solving times. For veritas the memory is sufficient but the running time is surprisingly long.

In Figure 4.7 running time versus lines of Haskell code is plotted. In Figure 4.8 running time versus lines of Core code is plotted.

We believe that the reason for why the constraint solver does not scale up for GHC and veritas is that they both contain large mutually recursive data types with contravariant recursion. A large data type leads to a high number of annotation variables and contravariant recursions leads to all annotation variables being bivariant. These two things together lead to a large amount of spurious constraints which makes the constraint solver to locally behave cubic.

[1]The O'Caml language: www.ocaml.org

Program	Haskell LOC	AST-nodes	Time (s)	Heap size (MB)
compress	549	222132	2.1	16
rsa	48	224999	1.9	16
maillist	137	242056	1.9	16
mkhprog	182	244938	2.1	16
compress2	147	272485	2.3	32
grep	104	293070	2.7	32
lift	426	330438	2.7	32
prolog	322	389839	3.0	32
fem	764	602542	4.8	32
parser	2442	670852	6.5	48
infer	561	779590	6.3	32
reptile	969	788824	5.9	32
gg	635	807724	6.6	48
pic	298	874717	6.9	32
gamteb	510	894157	6.8	48
fulsom	1117	917626	7.0	48
linear	176	918605	6.9	48
hidden	456	966645	7.2	48
bspt	708	984486	6.9	48
symalg	851	1070726	8.7	48
hpg	770	1097682	9.0	64
scs	445	1164117	10.1	64
cacheprof	1723	1246538	9.1	48
fluid	1796	1395094	10.9	64
ebnf2ps	1380	1512910	14.2	80
rx	4324	1997612	19.5	96
anna	5802	2565320	25.7	112
slc	2887	3025310	43.6	128
veritas	5452	5237168	2043.2	1120
ssc	13519	6658844	102.8	208
coresa	11430	11599811	148.2	288
gf	14252	20800100	269.9	928
ghc	85033	150694593	-	-

Figure 4.5: Solving times

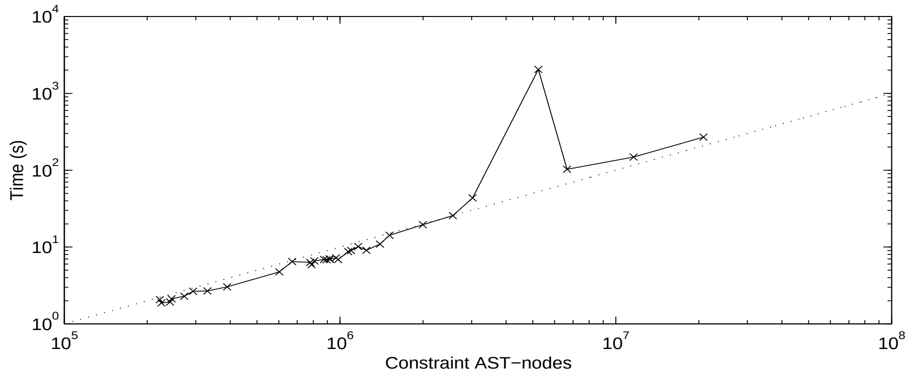

Figure 4.6: Solving time versus constraint size

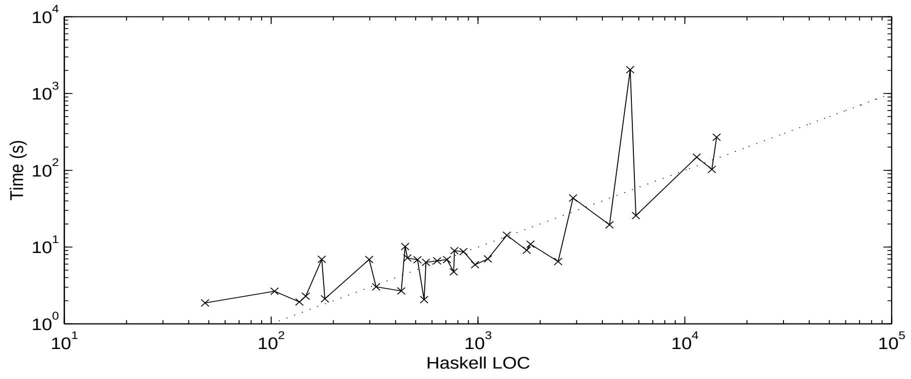

Figure 4.7: Solving time versus lines of Haskell code

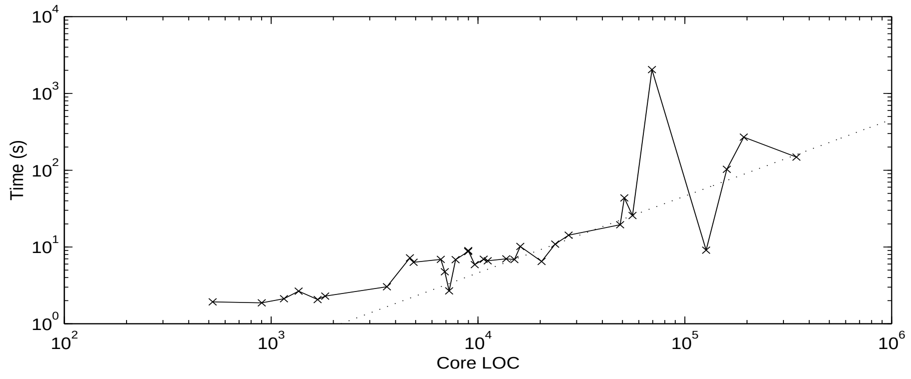

Figure 4.8: Solving time versus lines of Core code

Chapter 5

Related Work

The work presented in this thesis has been to design and implement a usage analysis which should be used to evaluate a constraint solving algorithm. Therefore most of the discussion of related work is focused on usage analysis.

5.1 Usage analysis

There is much work done on analyses which aims at avoiding updates [LGH$^+$92, Mar93, TWM95, Gus98, WPJ99, WPJ00, GS01a, Wan02]. The usage analysis presented in this thesis builds upon the analysis of Gustavsson and Svenningsson [GS01a]. Their analysis has usage polymorphism and subtyping but lacks user defined data types. The main contribution of this thesis is that user defined data types are added and that the analysis has been implemented and tested on real world programs.

Previous analyses have been presented which did have subtyping but not usage polymorphism [Gus98, WPJ99]. Analyses without polymorphism are computationally cheap but does not give very precise results. This is shown by the work of Wansbrough and Peyton Jones, when their analysis presented in [WPJ99] was implemented they found out that the analysis was "almost useless in practice" [WPJ00].

Following this the analysis presented in [Gus98] was modified and bounded polymorphism was added yielding the analysis presented in [GS01a]. Wansbrough and Peyton Jones did also modify their analysis but added what they call *simple polymorphism* instead of bounded polymorphism. Simple polymorphism is cheaper than bounded polymorphism because it approximates constraints. Whenever two quantified variables are constrained to each other they are unified and the constraint is removed. This removes the problem with copying constraints since there will be no constraints on quantified annotation variables, but it also reduces the precision of the analysis. Their analysis with simple polymorphism has been implemented and were shown to scale up computationally to large programs but did not give any dramatic results when used to avoid unnecessary updates. The analysis presented in this thesis is more precise since it do not unify constraints.

5.2 The constraint copying problem

In order to avoid an explosion of generated constraints, the usage analysis presented in this thesis makes use of *constraint abstractions* which are introduced by Gustavsson and Svenningsson [GS01b]. Work by Rehof and Fähndrich does also address this problem. In [Reh01] they present a flow analysis with polymorphic subtyping which uses *instantiation constraints*. Instantiation constraints are used to cope with the problem of copying constraints when instantiating polymorphic constraints. Although instantiation constraints and constraint abstractions look rather different they solve the same problem.

Chapter 6

Conclusion and Future Work

We have implemented a polymorphic usage analysis with subtyping and used it to study the scalability of a constraint solving algorithm. The solving algorithm is worst case cubic time but for constraints with some restrictions it is believed to be much cheaper in practice.

The measurements presented in this thesis show that the constraint solver scales up computationally to all but two tested programs. We believe that the problem with the two programs has to do with large mutually recursive data types with contravariant recursion which leads to a large amount of spurious constraints, leading to the constraint solver behaving locally cubic. We hope that a modification of how the usage analysis annotates type definitions will improve this situation. The modification is however not yet implemented and is left as future work.

We find the results of the measurements of the constraint solver promising but not conclusive. The fact that the constraint solver scaled up for all but two tested programs may indicate that subtyping in combination with full-blown polymorphism is not that expensive as earlier commonly believed.

As future work remains:

- Look at different strategies of annotating data types.
- Further investigate why the constraint solver does not scale up for veritas and GHC.
- Measuring how the usage analysis scales up in precision.
- Measuring how the constraint solver works for other analyses such as flow analysis.

Bibliography

[Gir72] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse de Doctorat d'État, Université Paris VII, 1972.

[GS01a] Jörgen Gustavsson and Josef Svenningsson. A Usage Analysis with Bounded Usage Polymorphism and Subtyping. *Lecture Notes in Computer Science*, 2011:140–??, 2001.

[GS01b] Jörgen Gustavsson and Josef Svenningsson. Constraint Abstractions. *Lecture Notes in Computer Science*, 2053:63–??, 2001.

[GSG03] Jörgen Gustavsson, Josef Svenningsson, and Tobias Gedell. A Constraint Solving Algorithm with Applications to Type Based Program Analyses with Subtyping and Polymorphism. *Unpublished*, 2003.

[Gus98] Jörgen Gustavsson. A Type Based Sharing Analysis for Update Avoidance and Optimisation. In *Proc. of ICFP'98*, pages 39–50, Baltimore, Maryland, September 1998.

[Gus99] Jörgen Gustavsson. A Type Based Sharing Analysis for Update Avoidance and Optimisation. *Licentiate thesis*, 1999.

[Jon92] Simon Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[LGH$^+$92] J. Launchbury, A. Gill, J. Hughes, S. Marlow, S. Peyton Jones, and P. Wadler. Avoiding unnecessary updates. In J. Launchbury and P. Sansom, editors, *Functional Programming, Glasgow 1992*. Springer-Verlag, 1992.

[Mar93] Simon Marlow. Update Avoidance Analysis by Abstract Interpretation. In *Proceedings of the 1993 Glasgow Workshop on Functional Programming*, Workshops in Computing, Ayr, Scotland, 1993. Springer-Verlag.

[Par93] W. Partain. The nofib Benchmark Suite of Haskell Programs, 1993.

[PJPS96] Simon Peyton Jones, W. Partain, and A. Santos. Let-floating: Moving Bindings to Give Faster Programs. In *International Conference on Functional Programming*, pages 1–12, 1996.

[Reh01] Rehof and Fähndrich. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *POPL: 28th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2001.

[Sve00] Josef Svenningsson. An Efficient Algorithm for a Sharing Analysis with Polymorphism and Subtyping. *Master's thesis*, 2000.

[Tol01] Andrew Tolmach. An External Representation for the GHC Core Language, 2001.

[TWM95] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *7'th International Conference on Functional Programming and Computer Architecture*, pages 1–11, La Jolla, California, June 1995. ACM Press.

[Wan02] Keith Wansbrough. *Simple Polymorphic Usage Analysis.* PhD thesis, Computer Laboratory, University of Cambridge, England, 28 March 2002. Revised July 2003.

[WPJ99] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 15–28, New York, NY, USA, 1999. ACM Press.

[WPJ00] Keith Wansbrough and Simon Peyton Jones. Simple Usage Polymorphism, 2000.