Counting analyses

Hidde Verstoep

MSc Thesis

July 30, 2013

Universiteit Utrecht

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

Supervisor:
prof. dr. J. Jeuring
Daily Supervisor:
dr. J. Hage

Abstract

There are a number of different analyses for functional languages that count in one way or another (e.g., strictness analysis, sharing analysis). There has been a lot of research into each of these analyses individually. The analysis described in this thesis combines a number of these counting analyses into one analysis.

Acknowledgements

I would like to express thanks to my daily supervisor: Jurriaan Hage. He had always time for me when I came by his office, unscheduled, while he had undoubtedly also other work waiting for him.

I would also like to thank everyone in rooms 681 and 682, especially Gabe & Ingo, for the often welcome distractions from my work.

Contents

List of Figures

Chapter 1

Introduction

There are number of static analyses that count in some way how often expressions are used during evaluation. Strictness analysis counts whether or not an expression is used at least once, sharing and uniqueness analysis count whether or not an expression is used at most once and absence analysis tries to determine whether or not an expression is used at all. The goal of this thesis is to combine these similar analyses into one analysis.

Why would you want to combine similar, already existing, analyses? You'd probably end up with yet another analysis that is very similar to the rest of them.

There are a number of reasons why combining analyses might be interesting:

- Implementation: maintaining an implementation of a single analysis is probably easier than maintaining the implementation of more analyses.

- Performance: running a single analysis is probably faster than running multiple analyses.

- Understanding: even if you are not going to use the single analysis, making the similarities explicit might give you the understanding necessary to reuse an optimization made in one analysis in another analysis.

In order to describe the analysis we have to start with some preliminaries (Section 2). This includes some basics about functional languages, type systems and explanations of the different analyses the new analysis is supposed to combine.

The analysis is defined using a type and effect system with constraints. The description of the analysis is split over two sections: constraint generation (Section 3) and constraint solving (Section 4).

We should prove the soundness of the static analysis with respect to some sensible operational semantics, otherwise it would be impossible to run the programs we obtain if we transform them based on information gathered during analysis. The operational semantics and some theorems on the relation between the static and operational semantics can be found in Section 5.

In Section 6 we will try to adapt the work from [13], which requires uniqueness analysis, to work with our analysis.

After these theoretical sections, we will go on to a more practical section: a simple example (Section 7). If you had any difficulties understanding the theoretical sections, this might help you understand everything more clearly.

Section 8 contains a few details about the implementation of the analysis.

Section 9 discusses papers about the relevant analyses or papers that try to combine some analyses.

And finally, in the conclusion (Section 10) we will discuss some pros and cons of the analysis and look at which things might require further exploration.

Chapter 2

Preliminaries

2.1 Overview

Programs are written in programming languages. Every programming language is defined using syntax (how to create expressions) and semantics (how to give meaning to expressions). The dynamic semantics define how to run a program. The static semantics can give an approximation at compile-time of how the program will be behave (according to the dynamic semantics) at run-time.

A lot of languages have static semantics in the form of types. The dynamic semantics cannot apply a function with an integer parameter to boolean value, that would result in a run-time error. By introducing static semantics that restricts this kind of application, we can give this information to the programmer at compile-time. So giving types to expressions can be seen as a form of verifying analysis to avoid run-time errors.

In the following sections we will explain a little about (i) functional languages (Section 2.2); (ii) different methods of analysis (Section 2.3); and (iii) the actual analyses I'm interested in (Section 2.4).

2.2 Functional languages

Most programming languages are imperative: actions that change state (memory) are executed in sequences. Functional languages on the other hand avoid mutable state and are more mathematical in nature. A few examples of functional programming languages are Haskel[21] and Clean[29]. In the following sections we will describe a few concepts which play an important role in a lot of functional languages. However, this does not imply these concepts are relevant only to functional languages.

2.2.1 Functions are values

In functional languages functions are treated just like values. It is possible to return a function as a result, or to take a function as a parameter.

This means you could write a function f which – when applied to a parameter – returns a function. In most imperative languages a function always requires all parameters at once, however, in functional languages we can apply a function to its parameters one by one, since it will just return a function which requires one less argument. This is called partial application.

Another consequence is that you can write functions which accept a function as a parameter. These functions are so-called higher-order functions.

2.2.2 Functions are pure

The result of a pure function can only depend on the values of the parameters. It cannot have side effects (operations on unrelated memory locations for example). This can lead to some very nice (automatic) optimizations: (i) since the result of a function call stays the same if the parameters stay the same, the result can be stored and reused if the call is used more than once; and (ii) if the result of a function call is not used, the call can be removed without any problems.

However, a language without side effects is not very useful. Computers are only useful if they can interact with users and that interaction (through a keyboard and screen for example) requires memory updates. That is why even pure functional languages usually incorporate (impure) side effects. There are some tricks to make sure that everything is still pure:

- Haskell uses the IO monad to distinguish between functions with and without side effects.
- Clean uses uniqueness typing, which allows for mutable state.

Both of these methods are also visible in the types (Section 2.2.4) of expressions.

2.2.3 Evaluation order

Languages that are not pure require a specific evaluation order. Suppose you have two expressions, that are used as parameters in a function, that both write to the same memory location but with different values. If you don't mandate a specific evaluation order, different compilers might generate different programs. In pure languages the choice of the evaluation order can be left open.

Call by need, or lazy evaluation, is a particular interesting way of evaluating programs. When evaluating applications, the function and its body are evaluated before the arguments (in contrast with most imperative programming languages). The arguments are only evaluated when needed in the body of the function. Every expression (function or value) can be represented by a thunk in memory. When evaluating an expression, its thunk can be updated with its result. This is possible because it won't change (there are no side effects). The advantage of this is that when the expression is used again no evaluation has to occur.

2.2.4 Types

Just like a lot of imperative languages, most functional languages have types. However, type systems in functional languages are often considered more powerful: most imperative languages either require manual type annotations, throw run-time type errors or automatically cast values from one type to another when necessary. Most functional languages on the other hand have automatic type inference using variants of the algorithms by Hindley [16] and Milner [25] (see also

Syntax

$$\begin{array}{lcllll} \mathbf{e} & ::= & x & \mid \mathbf{c} & \mid \lambda x \to \mathbf{e} & \mid \mathbf{e_1}\ \mathbf{e_2} \\ \tau & ::= & \alpha & \mid \mathit{int} & \mid \tau_1 \to \tau_2 & \end{array}$$

Static semantics $\boxed{\Gamma \vdash \mathbf{e} : \tau}$

$$\frac{\Gamma\ (x) \equiv \tau}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{}{\Gamma \vdash \mathbf{c} : \mathit{int}}\ \textsc{Lit}$$

$$\frac{\Gamma, x : \tau_1 \vdash \mathbf{e} : \tau_2}{\Gamma \vdash \lambda x \to \mathbf{e} : \tau_1 \to \tau_2}\ \textsc{Abs} \qquad \frac{\Gamma \vdash \mathbf{e_1} : \tau_1 \to \tau_2 \quad \Gamma \vdash \mathbf{e_2} : \tau_1}{\Gamma \vdash \mathbf{e_1}\ \mathbf{e_2} : \tau_2}\ \textsc{App}$$

Dynamic semantics $\boxed{\mathbf{e_1} \Downarrow \mathbf{e_2}}$

$$\frac{}{\mathbf{e} \Downarrow \mathbf{e}} \qquad \frac{\mathbf{e_1} \Downarrow (\lambda x \to \mathbf{e}) \quad \mathbf{e}\ [x := \mathbf{e_2}] \Downarrow \mathbf{e_3}}{\mathbf{e_1}\ \mathbf{e_2} \Downarrow \mathbf{e_3}}$$

Figure 2.1: Example: simple functional language

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \mathit{int}, y : \mathit{int} \vdash 3 : \mathit{int}}\ \textsc{lit}}{x : \mathit{int} \vdash \lambda y \to 3 : \mathit{int} \to \mathit{int}}\ \textsc{abs}}{\emptyset \vdash \lambda x \to (\lambda y \to 3) : \mathit{int} \to \mathit{int} \to \mathit{int}}\ \textsc{abs} \quad \dfrac{}{\emptyset \vdash 4 : \mathit{int}}\ \textsc{lit}}{\emptyset \vdash (\lambda x \to (\lambda y \to 3))\ 4 : \mathit{int} \to \mathit{int}}\ \textsc{app} \quad \dfrac{}{\emptyset \vdash 5 : \mathit{int}}\ \textsc{lit}}{\emptyset \vdash ((\lambda x \to (\lambda y \to 3))\ 4)\ 5 : \mathit{int}}\ \textsc{app}$$

Figure 2.2: Example: static derivation tree

Damas and Milner [6]). The inferred types can also be checked at compile-time, which results in an absence of run-time type errors.

A few other interesting features are:

- Algebraic data types (or even GADTs) make it easy to define complex data types [30].

- (Constrained) polymorphism makes it possible to give an expression a more generic type [34]. The expression can then behave differently, depending on how it is used.

- The type of a value shows whether or not that value contains side effects.

2.2.5 Example

A very simple functional language definition with types can be found in Figure 2.1. The compiler could give an error message whenever the top-level expression cannot be typed with *int*. When the top-level expression does have the type *int*, the compiler can generate an executable which is guaranteed to run. For some examples derivation trees, see Figure 2.2 and Figure 2.3.

$$\dfrac{\dfrac{\dfrac{}{\lambda x \to (\lambda y \to 3) \Downarrow \lambda x \to (\lambda y \to 3)} \quad \dfrac{}{(\lambda y \to 3)\ [x := 4] \Downarrow \lambda y \to 3}}{(\lambda x \to (\lambda y \to 3))\ 4 \Downarrow \lambda y \to 3} \quad \dfrac{}{3\ [y := 5] \Downarrow 3}}{((\lambda x \to (\lambda y \to 3))\ 4)\ 5 \Downarrow 3}$$

Figure 2.3: Example: dynamic derivation tree

2.3 Analysis methods

The goal of static analyses is to do one or more of the following (by changing the static and/or dynamic semantics of the language):

- Accepting and/or rejecting specific programs. Verifying analyses (like uniqueness analysis, see Section 2.4.2) reject programs that would otherwise be valid. Alternatively, replacing an existing analysis with one that is less restrictive would lead to accepting more programs.

- Deriving information for the compiler. This information might be used for program optimization, program transformation, more specific feedback and other things.

These goals also highlight some difficulties for static analyses:

- Since the static semantics are an approximation of the dynamic semantics, a change in one of the semantics must almost always be accompanied by a change in the other. A full understanding of both semantics is required to avoid subtle mistakes.

- If you start using a static analysis that can reject programs or that does not understand all language features then programs in an existing code base might be rejected, while the programs were perfectly valid before. So, rejecting too many programs or omitting certain language features in a static analysis might make a language less expressive.

There is more than one way to perform static analysis. We will explain two: type and effect systems (Section 2.3.1) and abstract interpretation (Section 2.3.2). These two methods of analysis seemed to be the most common in the papers relevant to this master thesis.

2.3.1 Type and effect systems

Type and effect systems are an extension of conventional type systems. Instead of giving every expression a simple type, the types can carry additional information. This information usually comes in the form of annotations. These annotations can be used to perform additional analyses.

Normal type systems can have polymorphism, a type can then also be a type scheme which quantifies over type variables. It is also possible to have constraints on the type variables in the type scheme (like type classes in Haskell). In some type and effect systems it is allowed to quantify over annotation variables. This is called polyvariance and can make the analysis more precise. Some analyses also introduce constraints on the annotation variables.

Subtyping and subeffecting are two other common ideas in type and effect systems. They represent a relation on types and effects (annotations), respectively. Again, the goal is to make the analysis more precise. The notation $A \sqsubseteq B$ would mean: it is safe to use the type/effect B when the type/effect A is required. An example of subtyping at work is providing a value of type *Boolean* where a value of type *Object* is required in the programming language Java.

A good and more detailed description of type and effect systems (and features like polyvariance and subtyping/subeffecting) can be found in [27].

2.3.2 Abstract interpretation

Abstract interpretation was introduced by Cousot and Cousot [5]. Dynamic or concrete semantics specify how a program behaves at run-time. Abstract interpretation requires you to introduce a new semantics for the language that is computable at compile-time and safely approximates the dynamic semantics. For example, it is possible to approximate possible integer values for possible variables in a program by only keeping the signs, or the ranges they might be in.

Abstract interpretation is implemented by interpreting a program, but mapping concrete values to possible abstract values. Often this implemented by a total, monotone function which is iterated until a fixed point is found.

This method of analysis is less important for my master thesis, for more information we refer to [19].

2.4 Analyses

In the following sections we will describe a number of analyses which are relevant to my master thesis. All the analyses expect a lazy, pure, functional programming language. We will also include examples to show how the results of different analyses can be used.

2.4.1 Sharing

Sharing analysis [22, 12, 14] is an analysis that determines whether or not an expression is used at most once.

Let's see how this information could be used, by looking at an example:

$$(\lambda x \to x + x)\ (1 + 1) \tag{2.1}$$

Since we are using lazy evaluation, the expression is evaluated in this order: (i) evaluation of the application starts with the function; (ii) evaluation of the function starts with its body; (iii) evaluation of the body starts with the $+$ function, which requires both arguments to be evaluated; (iv) the first argument x is evaluated by evaluating the thunk it points to – $(1 + 1)$ – and updating it; (v) the second argument x is evaluated by using the value from the now evaluated thunk; and lastly (vi) evaluation of $+$, the function body, the function and the application return, in that exact order. The value of $(1 + 1)$ is clearly used twice, but it is only evaluated once. Lets take a look at another example:

$$(\lambda x \to x)\ (1 + 1)$$

This expression is similar. However, the value of $(1+1)$ is only used once while evaluating. If a thunk is guaranteed to be used at most once, it is not necessary to include a thunk update like the one from step iv in expression 2.1.

In type and effect systems this analysis is often implemented using the lattice $1 \sqsubset \omega$. A 1 annotation stands for: the expression is guaranteed to be used at most once, while a ω annotation means that an expression can be used more than once. The subeffecting rule that can be used for sharing analysis is:

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

That way, when an expression is inferred to have a ω annotation it can still be used as a parameter to a function whose parameter has a 1 annotation. Since we know where we changed the ω to a 1 we can still make sure to include a thunk update when evaluating the 1 annotated expression.

2.4.2 Uniqueness

Uniqueness analysis [1, 2, 15, 24, 7, 14, 8] determines whether or not expressions are used at most once, just like sharing analysis.

Consider the function: $\mathit{writeFile} :: \mathit{String} \to \mathit{File} \to \mathit{File}$ that writes data to a file. Then

$$\lambda f \to (\mathit{writeFile}\ \texttt{"1"}\ f, \mathit{writeFile}\ \texttt{"2"}\ f) \tag{2.2}$$

could be evaluated in two ways when applied: the file could contain either "12" or "21", depending on the order in which the writeFile functions are evaluated. This is obviously not a pure function.

Uniqueness typing provides us with a safe way to use functions with side effects. With uniqueness analysis, it is possible for the *writeFile*-function to require that its file argument is unique. Since that is not the case in expression 2.2, we could reject the program. Uniqueness analysis can automatically annotate expressions with 1 and ω and will reject a program when this is not possible. A 1 annotation indicates that an expression must be unique. A ω annotation indicates that an expression is not necessarily unique. The subeffecting rule that could then be used is:

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

Note that the condition on the annotation is swapped with respect to the rule for sharing analysis. The result is that an expression which is marked unique can still be used as a parameter to a function which accepts non-unique parameters.

We could annotate the *writeFile* function: $\mathit{writeFile}::(\mathit{String}^{\omega} \to (\mathit{File}^{\mathbb{1}} \to \mathit{File}^{\mathbb{1}})^{\omega})^{\omega}$. The ω annotations mean that the first parameter, the entire function and its partial applications may be shared. However, the second parameter must be unique. The file that is produced must also be used in a unique way. That is clearly not the case in expression 2.2, a valid expression would be:

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

f is now used uniquely – in contrast with expression 2.2 – and the order in which the data is written to the file is now also clear.

The following example explains why this analysis is tricky. Consider the function $\mathit{writeFile}' :: (\mathit{File}^{\mathbb{1}} \rightarrow (\mathit{String}^{\omega} \rightarrow \mathit{File}^{\mathbb{1}})^{\mathbb{1}})^{\omega}$ which behaves exactly like *writeFile* but the parameters are swapped. A difference is that the partial application may no longer be shared. If we would allow it, expression 2.3 would be typeable and it would behave similar to expression 2.2.

$$\lambda f \rightarrow (\lambda w \rightarrow (w\ \texttt{"1"}, w\ \texttt{"2"}))\ (\mathit{writeFile}'\ f) \tag{2.3}$$

So, any partial application that has access to unique values must also be unique itself. It might seem this solves the problem, however, when the type and effect systems includes subeffecting it is possible to change the annotation on $(\mathit{writeFile}'\ f)$ from $\mathbb{1}$ to ω. Which means that expression 2.3 is still typeable. This is not a trivial problem to solve.

2.4.3 Strictness

Strictness analysis [10, 35, 4, 17] is an analysis that determines whether expressions are used at least once or not. If a parameter to a function is guaranteed to be used at least once, it could be calculated directly instead of passing an unevaluated thunk. This leads to performance benefits: passing a value on the stack is more efficient than creating a thunk – which might even refer to more thunks – on the heap.

A possible lattice to use with strictness is: $S \sqsubset L$. The meaning of the elements is: (i) an expression is used at least once, or evaluated strictly, for S; and (ii) an expression is used any number of times, or possibly evaluated lazily, for L. Even if a function has a parameter annotated S, we want to accept an argument annotated L. That is something subeffecting can express:

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

Note that is rule is the same as for sharing analysis.

2.4.4 Absence

This analysis was found in [31]. Absence analysis is an analysis that determines whether expressions are used or not. It is similar to dead or unreachable code elimination.

Lets take a look at the following standard Haskell function:

$$\mathit{const}\ x\ y = x$$

It is easy to see that the second argument is not used. This means we don't have to evaluate the second argument, since there are no possible side effects.

In code written by a person there are not a lot of functions that do not use their arguments. Even if functions like that exist, they are not frequently used. So, this analysis might not be so useful. There are however a lot of frequently used functions that use only a part of an argument. For example: *length*, *fst* and *snd*. These functions only look at the constructors of lists and pairs respectively, not at the values within. These functions are plenty – basically all container-like data structures have functions like this – and they are frequently used. So it seems it is very important to look at data types when performing an analysis like this. Both [26] and [36] discuss this issue shortly.

Other code that might benefit from this analysis is computer generated code. Computer generated code is often less “smart” and might contain a lot of dead code this analysis can detect.

2.4.5 Similarities

The question the analyses try to answer is: is an expression used (i) at most once (sharing and uniqueness analysis); (ii) at least once (strictness analysis); or (iii) at all (absence analysis)? So, the analyses are similar in that they all try to count how often an expression is used.

For sharing and uniqueness even the same notation was used: 1 and ω. These analyses differ only in subeffecting and the difficulties that arise for uniqueness with partial application.

However, there is a clear difference between uniqueness and all the other analyses. Since uniqueness analysis is a verifying analysis it can lead to the rejection of programs, while the other analyses only provide information about the program (which can then be used for optimization).

In the next chapter we will start creating a type and effect system for the combined analysis.

Chapter 3

Constraint Generation

3.1 Overview

In this chapter we will build a type and effect system to describe our analysis that should combine the four existing analyses. The emphasis in this chapter is on constraint generation, in the next chapter we will discuss how the generated constraints can be solved.

In the sections 3.2, 3.3 and 3.4 we will give definitions necessary to define the type system. The type system is defined in section 3.5. Note that the type system is based mainly on Appendix C in [35]. Being more general in some places (polymorphism/polyvariance), less so in others (annotations are finite instead of infinite). The next section (3.6) shows how the analysis can be instantiated to a specific analysis by changing a parameter and/or the lattice.

3.2 Basic definitions

An *annotation primitive* (Figure 3.1) is used to describe how often something is used: 0 means no use at all, 1 means exactly one use; and ∞ means at least used twice. We could have defined annotation primitives to encompass the entire set of natural numbers, however none of the usage analyses require this precision.

$$\pi \quad ::= \quad 0 \quad | \ 1 \quad | \ \infty$$

Figure 3.1: Annotation primitives

An *annotation value* (Figure 3.2) is a set of annotation primitives. We also define some special symbols for specific annotation values. Annotation values describe how often something is used, just like annotation primitives, however they are much more expressive. For example, we can now say that something is used at most once ($\{0, 1\}$) or at least once ($\{1, \infty\}$).

An *annotation* (Figure 3.3) is either a variable or an annotation value.

A *use annotation* (Figure 3.4) is an annotation that tells us how often a value is used.

$$\begin{array}{rcl}
\varpi & ::= & \emptyset \mid \{\pi\} \mid \varpi_1 \cup \varpi_2 \\
\bot & ::= & \emptyset \\
\mathbb{0} & ::= & \{0\} \\
\mathbb{1} & ::= & \{1\} \\
\omega & ::= & \{\infty\} \\
\top & ::= & \{0, 1, \infty\} \\
\mathcal{P}\left(\varpi\right) & ::= & \text{powerset of } \varpi
\end{array}$$

Figure 3.2: Annotation values

$$\varphi \quad ::= \quad \beta \mid \varpi$$

Figure 3.3: Annotations

$$\nu \quad ::= \quad \varphi$$

Figure 3.4: Use annotations

A *demand annotation* (Figure 3.5) is an annotation that tells us how often a variable is used. Since a variable should always have a value, a demand annotation is always accompanied by a use annotation.

$$\delta \quad ::= \quad \varphi$$

Figure 3.5: Demand annotations

The definitions for ν and δ are both simply aliases to φ. Even though the definitions are the same they describe different things ("use" and "demand" respectively) and are used differently in the type system (Section 3.5).

A *type* (Figure 3.6) can be a type variable α, a fully applied datatype or a function type. Definitions η_μ and ρ_μ are used to attach annotations to the parameter μ, which can be either τ or σ. Since a function produces a value we use η_τ to attach a use annotation to its result type. Because the argument is available to the body of the function as a variable we want to attach both use and demand annotations to the argument, so we use ρ_τ for the argument. Datatypes will be explained in Section 3.4.

$$\begin{array}{rcl}
\tau & ::= & \alpha \mid T\ \overline{\varphi_l}\ \overline{\tau_k} \mid \rho_\tau \to \eta_\tau \\
\eta_\mu & ::= & \mu^{\nu}\ (\text{where } \mu \in \{\tau, \sigma\}) \\
\rho_\mu & ::= & {\eta_\mu}^{\delta}\ (\text{where } \mu \in \{\tau, \sigma\})\ (\text{also written as: } \mu^{\nu,\delta})
\end{array}$$

Figure 3.6: Types

A *type scheme* (Figure 3.7) can be used to express polymorphic and/or polyvariant types. Constraints C will be explained in Section 3.3.

An *environment* (Figure 3.8) is a map from variable names to type schemes. However, we also store the use and demand for each variable.

$$\begin{array}{rcll} \sigma & ::= & \gamma & \mid \forall \overline{\upsilon} \,.\, C \Rightarrow \tau \\ \upsilon & ::= & \alpha & \mid \beta \end{array}$$

Figure 3.7: Type schemes

$$\begin{array}{rcll} \Gamma & ::= & \epsilon & \mid \Gamma, x : \rho_\sigma \end{array}$$

Figure 3.8: Environments

A *term* (Figure 3.9) describes a (part of a) program. The term language is annotated. Since we do not know the values of the annotations before running the analysis, we simply provide annotation variables to obtain a valid term. The terms consist of variables x, functions $\lambda^\nu\ x \to \mathbf{e}$, applications $\mathbf{e}\ x$, mutually recursive lets **let** $\overline{x_i =^{\nu,\delta} \mathbf{e_i}}$ **in** $\mathbf{e}$, the function *seq* $\mathbf{e_1}\ \mathbf{e_2}$ to force evaluation, constructors $K^\nu\ \overline{x_i}$ and case expressions **case** $\mathbf{e}$ **of** $\overline{K_i\ \overline{x_{ij}} \to \mathbf{e_i}}$. It is important to note that function and constructor application can only take variables as arguments. This makes analysis easier and it is possible to translate expressions that do not maintain this invariant to expressions that do. Constructors must always be fully applied.

$$\begin{array}{rcll} \mathbf{e} & ::= & x & \mid \lambda^\nu\ x \to \mathbf{e} \quad \mid \mathbf{e}\ x \\ & \mid & \mathbf{let}\ \overline{x_i =^{\nu,\delta} \mathbf{e_i}}\ \mathbf{in}\ \mathbf{e} & \mid \mathit{seq}\ \mathbf{e_1}\ \mathbf{e_2} \\ & \mid & K^\nu\ \overline{x_i} & \mid \mathbf{case}\ \mathbf{e}\ \mathbf{of}\ \overline{K_i\ \overline{x_{ij}} \to \mathbf{e_i}} \end{array}$$

Figure 3.9: Terms

3.3 Constraints

A lot of type systems encode operations on annotations in the type rules. Since our annotations are quite complex we start by defining some operations on annotations using operators and provide some extra notation. This simplifies the type rules significantly and keeps their intent clear.

Let's start by showing which operations we need and why:

- When we look at the addition of two numbers $\mathbf{e_1} + \mathbf{e_2}$, it is clear that both expressions are needed to calculate a result. Suppose a variable x occurs in both $\mathbf{e_1}$ and $\mathbf{e_2}$, then we need a way to combine the uses of x in both branches to find the use of x in the expression $\mathbf{e_1} + \mathbf{e_2}$. The way the uses are combined must also reflect that both expressions are used.

- Given the expression **if** $\mathbf{e}$ **then** $\mathbf{e_1}$ **else** $\mathbf{e_2}$ it is clear that either $\mathbf{e_1}$ or $\mathbf{e_2}$ will be evaluated depending on $\mathbf{e}$, but certainly not both. We need a way to combine the uses associated with variables in the expressions $\mathbf{e_1}$ and $\mathbf{e_2}$ that reflects that only one of the branches will be evaluated.

- A function $\lambda x \to \mathbf{e}$ can be applied multiple times. However, all variables occurring in $\mathbf{e}$ that are defined outside of the function are used repeatedly for every application of the function. It might even be the case that one of these variables is never used at all if the function is never applied. So, we need to be able to express repeated use.

- Consider the expression **let** $b = 0; a = b$ **in** $f\ a$. b is only used if a is used and a is used only if f uses its argument. So, we need to be able to express conditional use.

Now that we have a basic understanding of the operations we need we can define them as operators (Figure 3.10). The operator $\oplus$ expresses the combination of branches that are both used. The operator $\sqcup$ expresses the combination of branches of which only one is used. The operator $\cdot$ expresses repeated use. The notation used to define this operator is not very intuitive. The intended meaning is: for every element m in ϖ_1, for all combinations of m elements from ϖ_2, take the sum of the m elements. Example: suppose a function (which is applied twice) uses a value at most once, then the value is used: $\{\omega\} \cdot \{\mathbb{0}, \mathbb{1}\} = \{\mathbb{0}+\mathbb{0}, \mathbb{0}+\mathbb{1}, \mathbb{1}+\mathbb{0}, \mathbb{1}+\mathbb{1}\} = \{\mathbb{0}, \mathbb{1}, \omega\}$. The operator $\rhd$ expresses conditional use. The result of this operator is equal to ϖ_2 unless ϖ_1 includes 0.

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

Figure 3.10: Annotation value operators

Defining the operators for annotation values was easy since they do not include variables. However, we want to lift the operators to annotations, types, type schemes and environments and these can include annotation variables. An operator requires two annotation values to calculate a result directly. Since we might encounter annotation variables we cannot calculate a result directly. So, we will use constraints to keep track of the relations between values and variables.

Figure 3.11 contains an overview of all different kind of constraints. A constraint C can be an equality, $\oplus$, $\sqcup$, $\cdot$ or $\rhd$ constraint for either annotations, types or type schemes. Furthermore, we have included instantiation and generalization constraints.

$$\begin{array}{lllllll}
C & ::= & \varphi_1 \equiv \varphi_2 & \mid \varphi \equiv \varphi_1 \oplus \varphi_2 & \mid \varphi \equiv \varphi_1 \sqcup \varphi_2 & \mid \varphi \equiv \varphi_1 \cdot \varphi_2 & \mid \varphi \equiv \varphi_1 \rhd \varphi_2 \\
 & \mid & \tau_1 \equiv \tau_2 & \mid \tau \equiv \tau_1 \oplus \tau_2 & \mid \tau \equiv \tau_1 \sqcup \tau_2 & \mid \tau \equiv \varphi_1 \cdot \tau_2 & \mid \tau \equiv \varphi_1 \rhd \tau_2 \\
 & \mid & \sigma_1 \equiv \sigma_2 & \mid \sigma \equiv \sigma_1 \oplus \sigma_2 & \mid \sigma \equiv \sigma_1 \sqcup \sigma_2 & \mid \sigma \equiv \varphi_1 \cdot \sigma_2 & \mid \sigma \equiv \varphi_1 \rhd \sigma_2 \\
 & \mid & inst\ (\sigma) \equiv \tau & \mid gen\ (\rho_\tau, C, \Gamma) \equiv \rho_\sigma & & & \\
 & \mid & C_1 \cup C_2 & \mid \emptyset & & &
\end{array}$$

Figure 3.11: Constraints

Now that we know the operators we need to lift and have the constraints to keep track of relations between variables we can start to define how the lifting works. However, because there are a lot of operators and different things we want to lift them to we will omit certain details:

1. Only the operators $\oplus$ and $\cdot$ will be defined, the other operators can be defined in a similar way. The operator $\sqcup$ is similar to $\oplus$ and $\triangleright$ is similar to $\cdot$.

2. The flow of constraints when chaining operators will be left implicit. It does not matter which constraints are collected first, as long as you collect all of them.

3. Most equality constraints will be implicit. When two different antecedents of an inference rule contain the same variable this could be implemented as different variables with an equality constraint.

The figures 3.12, 3.13, 3.14 and 3.15 contain the definitions for the $\oplus$ and $\cdot$ operators for annotations, types, type schemes and environments respectively. The rules translate the operators into most simple constraints they can. Annotations, types and type schemes require their own kind of constraints and these are included in Figure 3.11. The application of operators on environments can be translated entirely into more simple constraints. Solving the generated constraints and looking at the different possible inhabitants for annotations, types and type schemes is the task of the constraint solver (Section 4).

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

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

Figure 3.12: Annotation operators

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

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

$$\frac{\mu_1 \oplus \mu_2 = \mu \leadsto C}{{\mu_1}^{\nu_1} \oplus {\mu_2}^{\nu_2} = \mu^{\nu} \leadsto C \cup \{\nu \equiv \nu_1 \oplus \nu_2\}}\ \eta_\mu\text{-ADD}$$

$$\frac{\varphi_1 \cdot \mu_2 = \mu \leadsto C}{\varphi_1 \cdot {\mu_2}^{\nu_2} = \mu^{\nu} \leadsto C \cup \{\nu \equiv \varphi_1 \cdot \nu_2\}}\ \eta_\mu\text{-MUL}$$

$$\frac{{\eta_1}_\mu \oplus {\eta_2}_\mu = \eta_\mu \leadsto C}{{{\eta_1}_\mu}^{\delta_1} \oplus {{\eta_2}_\mu}^{\delta_2} = {\eta_\mu}^{\delta} \leadsto C \cup \{\delta \equiv \delta_1 \oplus \delta_2\}}\ \rho_\mu\text{-ADD}$$

$$\frac{\varphi_1 \cdot {\eta_2}_\mu = \eta_\mu \leadsto C}{\varphi_1 \cdot {{\eta_2}_\mu}^{\delta_2} = {\eta_\mu}^{\delta} \leadsto C \cup \{\nu \equiv \varphi_1 \cdot \delta_2\}}\ \rho_\mu\text{-MUL}$$

Figure 3.13: Type operators

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

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

Figure 3.14: Type scheme operators

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

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

$$\Gamma\ (x) = \begin{cases} \rho_\sigma & \text{if } x : \rho_\sigma \in \Gamma \\ (\forall \{\alpha\}\ .\ \alpha)^{\mathbb{0},\mathbb{0}} & \text{otherwise} \end{cases}$$

Figure 3.15: Environment operators

For everything that has an $\sqcup$-operator, we can now define additional operators:

$$\frac{\mu_1 \sqcup \mu_2 = \mu_2 \rightsquigarrow C}{\mu_1 \sqsubseteq \mu_2 \rightsquigarrow C}$$

$$\frac{\mu_2 \sqsubseteq \mu_1 \rightsquigarrow C}{\mu_1 \sqsupseteq \mu_2 \rightsquigarrow C}$$

These operators can be used for subtyping.

3.4 Datatypes

Usually algebraic datatypes are defined like so: **data** $T\ \overline{\alpha} = \overline{K_i\ \overline{\tau_{ij}}}$, where T is the name of the datatype, $\overline{\alpha}$ refers to type parameters, $\overline{K_i}$ are the constructors and each τ_{ij} refers to the unannotated type of a field of a constructor. If we were to keep these definitions we wouldn't have a whole lot of information. We could only annotate the datatype as a whole, while it would be interesting to have information about both the values and the spine of a list (for example). Then we would be able to statically determine that the values of a list are never calculated when only the length of the list is used.

Our analysis, with datatypes, can find that: (i) the spine of xs (in Figure 3.16) is used more than once; and (ii) that the values inside xs are used exactly once. These results allow us to apply the optimizations of both sharing and strictness analysis to the values in xs. Without annotations for datatypes this would not be possible.

This means we need to use annotated types for the types of the fields of constructors. Since we don't want every use of a datatype to have the exact same annotations, this also implies that we need to use annotation variables to

$$\begin{array}{l}\textbf{let}\ l = \mathit{length}\ \mathit{xs}\\ \textbf{in}\ \mathit{map}\ (+l)\ \mathit{xs}\end{array}$$

Figure 3.16: Example with datatypes

be able to use datatypes differently in different places. So we reserve an extra spot for annotation variables in datatype definitions: **data** $T\ \overline{\beta}\ \overline{\alpha} = \overline{K_i\ \overline{\tau_{ij}}}$.

In our implementation datatypes are not annotated algorithmically, instead there are a number of predefined annotated datatypes. However, it is necessary to be able to annotate dataypes algorithmically, since we don't want to bother the user with providing annotations. The main problem of such an algorithm is that datatypes can be defined in a lot different ways, while it should be possible to annotate all of them. A simple algorithm would be: (i) for each type of a field: instantiate the top-level annotations with fresh variables and store them in the list $\overline{\beta}$ and instantiate all the other annotations with $\top$, (ii) replace every occurrence of T with $T\ \overline{\beta}$.

In Figure 3.17 we can see what this algorithm does to a List definition (**data** $\mathit{List}\ \alpha = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)$). All the values of the list share the variables ν_1 and δ_1, while the spine of the list is annotated with ν_2 and δ_2.

$$\begin{array}{l}\textbf{data}\ \mathit{List}\ [\nu_1,\delta_1,\nu_2,\delta_2]\ [\alpha]\\ \quad = \mathit{Nil}\\ \quad \mid \mathit{Cons}\ \alpha^{\nu_1,\delta_1}\ (\mathit{List}\ [\nu_1,\delta_1,\nu_2,\delta_2]\ [\alpha])^{\nu_2,\delta_2}\end{array}$$

Figure 3.17: Annotated List datatype

Even though the algorithm works for some datatypes (even simple recursive ones), it doesn't work for all. For example, we have not considered mutually recursive datatypes or datatypes where the datatype itself is used recursively as a parameter to another datatype (like rose-trees: **data** $\mathit{Rose}\ \alpha = \mathit{Leaf}\ \alpha \mid \mathit{Node}\ (\mathit{List}\ (\mathit{Rose}\ \alpha))$). Also, when the number of fields increases, the number of variables increases too, without increasing the precision very much. So, sometimes fields should share their variables to limit the number of variables.

[35], Section 5.4 contains more detailed information on annotating datatypes algorithmically and looking at all the subtleties that arise. It also describes an algorithm that uses different algorithms to annotate different kinds of datatypes and falls back on a default algorithm when none of the algorithms can annotate a datatype properly. The algorithm described above is one of the algorithms used.

We think this is a good approach since it provides very accurate annotations for common datatypes, while still being fully automatic.

3.5 Static semantics

Now that there are some basic definitions in place, we can define the static semantics (Figure 3.18). The rules under "Sequential evaluation" and "Datatypes" are optional, the type system is still valid without either of these parts. Let's take a look at each rule individually:

The Var-rule creates a fresh σ and τ using variables. The τ must be an instance of σ. Since the variable occurs exactly once, we say exactly that in the environment. It is unknown how often the value will be used so a fresh ν is created.

The Abs-rule first types the body of the function. We ensure that the type scheme of the variable is monomorphic so we can use it as the type of the argument of the function. The type of the body is used as the result type of the function. The use and demand of the other variables in the environment should be multiplied with the use of the function, for which the variable ν_2 is created.

The App-rule starts with typing both the function and its argument. The function is used exactly once. However since the function may be shared it is important to apply subeffecting here. Since the use should include one, we fix the way of subeffecting to $\sqsubseteq$. When typing the argument, which is a variable by definition, the variable will is found to be demanded exactly once. However, the function specifies (using δ_2) how often it demands an argument, so we set the demand to δ_2 when creating the result environment. This is also the only place where it is necessary to use the analysis dependent subeffecting parameter ($\diamond$, explained in Section 3.6.2).

The Let-rule is by far the most complex. The first two antecedents type the body and the definitions respectively. It is assumed that every definition occurs in every other definition and in the body. If this is not the case a fresh type variable with $\mathbb{0}$ use and demand can be used for the definitions that do not occur in another definition. The third antecedent combines the parts of the environments that do not include the variables from the let definitions. However, when a let definition is never demanded the respective environment isn't used either (as expressed by $\delta_j \triangleright \Gamma_j$). The fourth, fifth and sixth antecedents use essentially the same formula as the third. However, everything is done in a monomorphic setting and it only includes the variables from the let definitions. Also, the sixth antecedent does not contain any information about the body, while the fourth and fifth do. This is necessary to make sure we can generalize the definitions independently of how the body might use it. Now we have to think what information is necessary to generalize the monomorphic ${\tau_i}^{\nu_i,\delta_i}$ to the polymorphic ${\sigma_{0i}}^{\nu_{0i},\delta_{0i}}$. We include (i) all the constraints that can contain information about the let definitions: $C_1 \cup C_3 \cup C_4 \cup C_5$; and (ii) the environment: Γ (variables still in the environment should not be generalized over) . The result is now simply the calculated environment, the type of the body and all the leftover constraints.

At a first glance it might be surprising to see the use of the first argument of *seq* set to $\mathbb{0}$ in the Seq-rule. However, while seq will calculate it, it is never used in the result. Unless it is a shared expression which also occurs in the second argument, in which case it will also receive use information from the use of the second argument.

The Con-rule is for fully applied data constructors. We start by repeating the predefined type information for datatypes (Section 3.4). We can create a new instance using fresh variables φ_l and τ_k. Then we have to type all of the variables used. Some of the x_j might refer to the same variable, so it is important that we do not construct the environment simply by throwing them all in the environment with their respective type. Similar to the Abs-rule we reset the demand of the variables in when constructing the environment because the demand is already provided.

The Case-rule also starts with creating a fresh instance of the datatype type information, similar to the Con-rule. We make sure to type the expression we match on with use $\mathbb{1}$ and subeffecting $\sqsubseteq$, similar to the function expression in the App-rule. The result type is the same as the type of all the case-arms. The result environment is built by adding Γ_0 to the combined environment of all Γ_i. Since only one of the case-arms will be executed the combining is done using the $\sqcup$ operator.

The Sub($\backsim$)-rule is perhaps not a subeffecting rule in the traditional sense: it is only used when it is explicitly used in the rest of the typesystem and it has a parameter (the operator $\backsim$) with which it will apply the subeffecting. The alternative was hardcoding this simple transformation everywhere, this would make all the inference rules look more cluttered. The fact that this rule is only used explicitly also makes the type system syntax directed, which makes it easier to write an implementation.

In the next chapter we will provide operational semantics and relate them to the static semantics using theorems. Proving these theorems should ensure the semantics are correct.

3.6 Analysis specialization

There are two ways to specialize the analysis: by changing the lattice (Section 3.6.1) or by changing how subeffecting works (Section 3.6.2). These two methods can also be used together.

3.6.1 Lattice

Annotation values describe values of the lattice. Using the current definition of annotations values (Section 3.2) the corresponding lattice with 8 elements is: $\mathcal{P}(\{0, 1, \infty\})$. So, changing the lattice actually means changing the definition of annotation values.

Why would you want to change the lattice? There is a very clear trade-off between the precision (or size) of the lattice and the performance it would take to solve the constraints. So it makes sense to make the lattice as small as possible for the precision you require for a specific analysis.

If we only want to do sharing analysis, it is sufficient for the annotation values to describe a lattice like $\bot \sqsubset 1 \sqsubset \omega$. We would lose some precision, however, the lattice consists of only 3 elements instead of the default 8. We would also have to redefine the values $\bot$, $\mathbb{0}$, $\mathbb{1}$ and $\top$ and the meaning of the operations on annotations.

The default definition is already a specialization of the definition given by [35], Appendix C, since the lattice corresponds to $\mathcal{P}(\{0, 1, \infty\})$ instead of $\mathcal{P}(\mathit{Nat})$.

3.6.2 Subeffecting

The static semantics (Section 3.5) require a value for the parameter $\diamond$ to specify how subeffecting should be done. We have a couple of comparison operators

Basics $\boxed{\Gamma \vdash \mathbf{e} : \eta_\tau}$

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

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

$$\frac{\begin{array}{c}\Gamma_1 \vdash_{\sqsubseteq} \mathbf{e} : ({\eta_2}^{\delta_2} \to \eta_3)^{\mathbb{1}} \rightsquigarrow C_1 \\ x : {\eta_4}^{\mathbb{1}} \vdash_{\diamond} x : \eta_2 \rightsquigarrow C_2 \qquad \Gamma_1 \oplus x : {\eta_4}^{\delta_2} = \Gamma_2 \rightsquigarrow C_3\end{array}}{\Gamma_2 \vdash \mathbf{e}\ x : \eta_3 \rightsquigarrow C_1 \cup C_2 \cup C_3}\ \textsc{App}$$

$$\frac{\begin{array}{c}\Gamma_0, \overline{x_j : {\sigma_{0j}}^{\nu_{0j},\delta_{0j}}} \vdash \mathbf{e} : \eta \rightsquigarrow C_0 \\ \Gamma_i, \overline{x_j : (\forall\emptyset\,.\ \emptyset \Rightarrow \tau_{ij})^{\nu_{ij},\delta_{ij}}} \vdash \mathbf{e_i} : {\tau_i}^{\nu_i} \rightsquigarrow C_{1i} \qquad \Gamma_0 \oplus (\bigoplus_j\ (\delta_j \rhd \Gamma_j)) = \Gamma \rightsquigarrow C_2 \\ \delta_{0i} \oplus (\bigoplus_j\ (\delta_j \rhd \delta_{ij})) = \delta_i \rightsquigarrow C_{3i} \qquad \nu_{0i} \oplus (\bigoplus_j\ (\delta_j \rhd \nu_{ij})) = \nu_i \rightsquigarrow C_{4i} \\ \bigoplus_j\ (\delta_j \rhd \tau_{ij}) = \tau_i \rightsquigarrow C_{5i} \qquad C_1 = \bigcup_i C_{1i} \qquad C_3 = \bigcup_i C_{3i} \qquad C_4 = \bigcup_i C_{4i} \\ C_5 = \bigcup_i C_{5i} \qquad C_6 = \bigcup_i \{\mathit{gen}\ ({\tau_i}^{\nu_i,\delta_i}, C_1 \cup C_3 \cup C_4 \cup C_5, \Gamma) \equiv {\sigma_{0i}}^{\nu_{0i},\delta_{0i}}\}\end{array}}{\Gamma \vdash \mathbf{let}\ \overline{x_i =^{\nu_i,\delta_i} \mathbf{e_i}}\ \mathbf{in}\ \mathbf{e} : \eta \rightsquigarrow C_0 \cup C_2 \cup C_6}\ \textsc{Let}$$

Sequential evaluation $\boxed{\Gamma \vdash \mathbf{e} : \eta_\tau}$

$$\frac{\Gamma_1 \vdash_{\sqsubseteq} \mathbf{e_1} : {\tau_1}^{\mathbb{0}} \rightsquigarrow C_1 \qquad \Gamma_2 \vdash \mathbf{e_2} : \eta_2 \rightsquigarrow C_2 \qquad \Gamma_1 \oplus \Gamma_2 = \Gamma \rightsquigarrow C_3}{\Gamma \vdash \mathit{seq}\ \mathbf{e_1}\ \mathbf{e_2} : \eta_2 \rightsquigarrow C_1 \cup C_2 \cup C_3}\ \textsc{Seq}$$

Datatypes $\boxed{\Gamma \vdash \mathbf{e} : \eta_\tau}$

$$\frac{\begin{array}{c}\mathbf{data}\ T\ \overline{u_l}\ \overline{\alpha_k} = \overline{K_i\ \overline{\rho_{ij}}} \qquad {\eta_j}^{\delta_j} = \rho_{ij}\ [{}^{\overline{\varphi_l}}/_{\overline{u_l}}, {}^{\overline{\tau_k}}/_{\overline{\alpha_k}}] \\ x_j : {\eta'_j}^{\mathbb{1}} \vdash x_j : \eta_j \rightsquigarrow C_1 \qquad \bigoplus_j\ (x_j : {\eta'_j}^{\delta_j}) = \Gamma \rightsquigarrow C_2\end{array}}{\Gamma \vdash {K_i}^{\nu}\ \overline{x_j} : (T\ \overline{\varphi_l}\ \overline{\tau_k})^{\nu} \rightsquigarrow C_1 \cup C_2}\ \textsc{Con}$$

$$\frac{\begin{array}{c}\mathbf{data}\ T\ \overline{u_l}\ \overline{\alpha_k} = \overline{K_i\ \overline{\rho_{ij}}} \\ {\tau_{ij}}^{\nu_{ij},\delta_{ij}} = \rho_{ij}\ [{}^{\overline{\varphi_l}}/_{\overline{u_l}}, {}^{\overline{\tau_k}}/_{\overline{\alpha_k}}] \qquad \Gamma_0 \vdash_{\sqsubseteq} \mathbf{e} : (T\ \overline{\varphi_l}\ \overline{\tau_k})^{\mathbb{1}} \rightsquigarrow C_1 \\ \Gamma_i, \overline{x_{ij} : (\forall\emptyset\,.\ \emptyset \Rightarrow \tau_{ij})^{\nu_{ij},\delta_{ij}}} \vdash \mathbf{e_i} : \eta \rightsquigarrow C_2 \qquad \Gamma_0 \oplus (\bigsqcup_i \Gamma_i) = \Gamma \rightsquigarrow C_3\end{array}}{\Gamma \vdash \mathbf{case}\ \mathbf{e}\ \mathbf{of}\ \overline{K_i\ \overline{x_{ij}} \to \mathbf{e_i}} : \eta \rightsquigarrow C_1 \cup C_2 \cup C_3}\ \textsc{Case}$$

Subeffecting $\boxed{\Gamma \vdash_{\backsim} \mathbf{e} : \eta_\tau}$

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

Figure 3.18: Static semantics

available that we could use: $\equiv$, $\sqsubseteq$ and $\sqsupseteq$. In Section 2.4 we have seen that $\sqsubseteq$ can be used for sharing, strictness and absence, while $\sqsupseteq$ is proposed for uniqueness. We will still use $\sqsubseteq$ for sharing, strictness and absence, however, $\equiv$ will be used for uniqueness.

Section 2.4.2 shows that uniqueness with subeffecting is tricky. [7] proposes a solution that allows subeffecting with uniqueness safely, however, it introduces new annotations on the types. The extra annotations make types harder to understand and the analysis in this thesis would be less generic (the annotations would be specific to uniqueness). The solution that [8] proposes is to keep all the types as generic as possible and forget about subeffecting for uniqueness altogether. This solution fits a lot better with our analysis, so lets use $\equiv$ for uniqueness. This way the annotated types are still the same for all analyses and easy to read (or, to write if we allow the user to provide types).

Note that using $\equiv$ for sharing, strictness and absence would disable subeffecting for those analyses, however, the results of the analysis would be valid for all four of the analysis. It would not be necessary to run uniqueness analysis separately.

3.6.3 Discussion

Specializing the analysis sounds interesting. However, there is a trade-off between the runtime of the analysis and the quality of the analysis results. When looking at our own implementation, we found that:

- Using a different lattice without any other changes to the type system or definitions gives horrible results. The reason is that certain operations expect certain annotations to be present (exact zero, exact one) but have to approximate when they are not. Approximations are approximated further until finally the results are useless. The runtime of the analysis obviously improves significantly when the lattice is smaller (and there is less work for the constraint solver).

- Making changes to $\diamond$ also has a large impact on the runtime of the analysis. However, the quality of the results of the analysis barely changes. Possibly because there is still some hard-coded subeffecting in the type rules.

Chapter 4

Constraint Solving

4.1 Overview

In the previous chapter we explained how constraints can be generated. In this chapter we will describe how all of these constraints can be solved.

The different kinds of constraints can be divided into four groups: annotation constraints, type constraints, type scheme constraints and generalization & instantiation constraints. Each of these groups will be treated separately in the sections 4.3, 4.4, 4.5 and 4.6 respectively. The function *simplify* $:: C \rightarrow C$ is a function that applies all the described methods in all these sections to solve constraints as much as possible. The exact definition of this function is described in Section 4.7.

4.2 Basic definitions

For some constraint sets we want to maintain a certain property, the following definition expresses this property:

Definition 1. A constraint set C is *simple* if it only contains: (i) annotation constraints; and (ii) type constraints that contain only type variables.

Examples of constraint sets that must have this property are:

1. The output of *simplify*.

2. The constraint sets in type schemes.

Note that there may still be certain type constraints in the *simple* constraint sets. However, this is necessary if we want to be able instantiate type variables with datatypes in some polymorphic definitions.

4.3 Annotation constraints

Equality annotation constraints ($\varphi_1 \equiv \varphi_2$) can be solved using unification. For the other annotation constraints ($\varphi \equiv \varphi_1 \oplus \varphi_2$, $\varphi \equiv \varphi_1 \sqcup \varphi_2$, $\varphi \equiv \varphi_1 \cdot \varphi_2$ and $\varphi \equiv \varphi_1 \triangleright \varphi_2$), the analysis defined in Section 4.3.1 can be used. In Section 4.3.3 it is explained how to use the analysis.

$$\begin{array}{l} \mathit{valueAnalysis} :: [C] \rightarrow \mathit{State}\ (\mathit{Map}\ \beta\ (\mathit{Set}\ \varpi))\ () \\ \mathit{possibleValues} :: \beta \rightarrow \mathit{State}\ (\mathit{Map}\ \beta\ (\mathit{Set}\ \varpi))\ (\mathit{Set}\ \varpi) \end{array}$$

Figure 4.1: Value analysis function signatures

4.3.1 Annotation value analysis

The value analysis (Figure 4.1) determines for a set of annotation constraints which annotation values (ϖ) the annotation variables (β) present in the constraints can have.

The function *possibleValues* is used to query analysis results in the state and returns for an annotation variable all the possible values it can have. So, it returns the values associated with the variable if it is present in the map. Otherwise it returns a set of all annotation values except $\bot$. This function is used within the analysis itself, but it can also be used to retrieve the information after the analysis has finished.

The analysis itself is defined by the function *valueAnalysis*. It keeps iterating until it reaches the fixpoint: the state stays the same. During each iteration it loops over all the constraints. For each constraint the following is done:

- For each variable in the constraint, use *possibleValues* to determine which values it might have.
- Instantiate the variables in the constraint, using their own possible values, in all possible ways and put all these value-only constraints in a list.
- Throw away all the constraints in the list that are invalid with respect to the definition of the associated annotation value operator (Figure 3.10).
- Limit the values of each variable in the constraint to the ones that can be found in the same position in the constraints in the list.

This algorithm could possibly be implemented more efficiently using a worklist algorithm.

4.3.2 Example

The analysis might not be clear, so let's look at an example. We will use the following constraint set:

$$[1 : \mathbb{1} \sqsubseteq \beta_1, 2 : \beta_1 \sqsubseteq \mathbb{1}, 3 : \beta_2 \sqsubseteq \top, 4 : \beta_3 \sqsubseteq \beta_4, 5 : \beta_4 \sqsubseteq \beta_3, 6 : \mathbb{1} \oplus \mathbb{1} = \omega]$$

Note that they are only numbered to make it easier to reference them. Figure 4.2 shows the beginning of the *valueAnalysis* algorithm. At the beginning of the algorithm the state is empty and while processing constraints the state starts to contain more and more variables with associated information. After processing all constraints it will do so again, until the state does not change anymore. The next section describes how the information can be used to simplify the constraint set. It might turn out that not all information can be used, however, we have to collect all of it to find any relations between constraints.

Description	State
Start	$[]$
Process constraint 1	$[\beta_1 \in \{\{1\},\{0,1\},\{1,\infty\},\{0,1,\infty\}\}]$
Process constraint 2	$[\beta_1 \in \{\{1\}\}]$
Process constraint 3	$[\cdots, \beta_2 \in \mathcal{P}\left(\top\right) - \emptyset]$
Process constraint 4	$[\cdots, \beta_3 \in \mathcal{P}\left(\top\right) - \emptyset], \beta_4 \in \mathcal{P}\left(\top\right) - \emptyset]$
Process constraint 5	$[\cdots]$
Process constarint 6	$[\cdots]$

Figure 4.2: Value analysis example

4.3.3 Using the analysis

The value analysis can be used to find annotation values for annotation variables, to find relations between variables and to remove obsolete constraints. Please note that it can not solve all constraints, only some cases where the constraints uniquely determine the value of a variable.

Let's take a look at all the ways we can use the analysis. We have to start by applying the analysis:

- Given a set of annotation constraints C, apply *valueAnalysis*. The state can be initialized with the empty map.

We can now remove variables in the following ways:

- Use *possibleValues* to determine if there are any variables that only have one associated value. If so, replace each of those variables with its associated value everywhere in the constraint set.

  This rule makes sure that if we require that $\mathbb{1} \sqsubseteq \beta$ and $\beta \sqsubseteq \mathbb{1}$, we can find that $\beta = \mathbb{1}$.

- For each constraint in C that contains at least two variables, determine if the values of one the variables are always the same as the values of one of the other variables. If so, use only one variable instead of two.

  This rule makes sure that if we require that $\beta_1 \sqsubseteq \beta_2$ and $\beta_2 \sqsubseteq \beta_1$, we can find that $\beta_1 \equiv \beta_2$. These constraints can be generated, for example, in mutually recursive functions.

We can now remove constraints in the following ways:

- If a constraint in C does not contain any variables, remove the constraint.

- If a constraint in C contains one variable, and the possible values for that variable are all annotation values except $\bot$, remove the constraint.

  If the analysis has not limited the possible values further than all annotation values except $\bot$, then the variable is not constrained at all. So, this constraint does not matter. This rule removes constraints generated by, for example, $\beta \sqsubseteq \top$.

So, while we cannot solve all annotation constraints, we can simplify the set of annotation constraints a lot.

$$\frac{}{(\rho \to \eta) \equiv (\rho \to \eta) \oplus (\rho \to \eta) \rightsquigarrow \emptyset}$$

$$\frac{\overline{\varphi_{1l} \oplus \varphi_{2l} = \varphi_l} \rightsquigarrow C}{(T\ \overline{\varphi_l}\ \overline{\tau_k}) \equiv (T\ \overline{\varphi_{1l}}\ \overline{\tau_k}) \oplus (T\ \overline{\varphi_{2l}}\ \overline{\tau_k}) \rightsquigarrow C}$$

$$\frac{}{(\rho \to \eta) \equiv \varphi \cdot (\rho \to \eta) \rightsquigarrow \emptyset}$$

$$\frac{\overline{\varphi \cdot \varphi_{2l} = \varphi_l} \rightsquigarrow C}{(T\ \overline{\varphi_l}\ \overline{\tau_k}) \equiv \varphi \cdot (T\ \overline{\varphi_{2l}}\ \overline{\tau_k}) \rightsquigarrow C}$$

Figure 4.3: Type constraint rules

4.4 Type constraints

Similar to annotation constraints, the type equality constraints can be solved using unification. The monomorphic type system in [35] uses the rules in Figure 4.3 (although they are changed here such that they generate constraints). These rules only have cases for functions and constructors. But, since our typesystem is polymorphic, the type constraints can contain variables. Some of the constraints with variables can be solved in the following way:

- If one of types in the constraint is a function type, then all the other types in the constraint must be the exact same function type. This means we can substitute the function for each of the type variables we find and mark the constraint as solved or fail if we find a datatype.
- If one of types in the constraint is a datatype, then all the other types in the constraint must be the same datatype. However, the annotations may differ. Substitute each variable we find with the datatype with fresh annotation variables, generate the required annotation constraints and mark the constraint as solved. In the case we find a function, the constraint is invalid and we fail.
- If none of the types in the constraint is a function or a datatype (all the types are variables), then we have no choice but to keep the constraint.

So, solving type constraints might lead to new annotation constraints and there might be some type constraints left that contain only variables.

4.5 Type scheme constraints

As can be seen in Figure 4.4, when the variable and constraint sets are empty, type scheme constraints can be transformed into type constraints. The method from the previous section can be used to make sure the rules also work when a constraint contains at least one type scheme with empty variable and constraint sets and variables in the other positions.

The general case with variables and constraints is a lot harder. In order to solve the type scheme constraints we have to take a look at the type system (Figure 3.18) again. There are four places where type schemes are introduced

$$\frac{\tau_1 \equiv \tau_2 \rightsquigarrow C}{\forall \emptyset \,.\, \emptyset \Rightarrow \tau_1) \equiv (\forall \emptyset \,.\, \emptyset \Rightarrow \tau_2) \rightsquigarrow C}$$

$$\frac{\tau_1 \oplus \tau_2 = \tau \rightsquigarrow C}{(\forall \emptyset \,.\, \emptyset \Rightarrow \tau) \equiv (\forall \emptyset \,.\, \emptyset \Rightarrow \tau_1) \oplus (\forall \emptyset \,.\, \emptyset \Rightarrow \tau_2) \rightsquigarrow C}$$

$$\frac{\varphi \cdot \tau_2 = \tau \rightsquigarrow C}{(\forall \emptyset \,.\, \emptyset \Rightarrow \tau) \equiv \varphi \cdot (\forall \emptyset \,.\, \emptyset \Rightarrow \tau_2) \rightsquigarrow C}$$

Figure 4.4: Monomorphic type scheme constraint rules

$$\frac{[{}^{\forall \overline{v} \,.\, C \Rightarrow \tau}/_{\gamma}]}{(\forall \overline{v} \,.\, C \Rightarrow \tau) \equiv \gamma \rightsquigarrow \emptyset}$$

$$\frac{\begin{array}{cc} \mathit{fresh}\ \tau_1\ \tau_2 & \tau_1 \oplus \tau_2 = \tau \rightsquigarrow C_2 \\ C = \mathit{simplify}\ (C_1 \cup C_2) & V = \mathit{fv}\ (\forall \overline{v} \,.\, C_1 \Rightarrow \tau) \\ V_1 = \mathit{fv}\ (\forall \overline{v} \,.\, C \Rightarrow \tau_1) - V & \gamma_1 = \forall \overline{v} \cup V_1 \,.\, C \Rightarrow \tau_1 \\ V_2 = \mathit{fv}\ (\forall \overline{v} \,.\, C \Rightarrow \tau_2) - V & \gamma_2 = \forall \overline{v} \cup V_2 \,.\, C \Rightarrow \tau_2 \end{array}}{(\forall \overline{v} \,.\, C_1 \Rightarrow \tau) \equiv \gamma_1 \oplus \gamma_2 \rightsquigarrow \emptyset}$$

$$\frac{\begin{array}{cc} \mathit{fresh}\ \tau_2 & \varphi \cdot \tau_2 = \tau \rightsquigarrow C_2 \\ C = \mathit{simplify}\ (C_1 \cup C_2) & V = \mathit{fv}\ (\forall \overline{v} \,.\, C_1 \Rightarrow \tau) \\ V_2 = \mathit{fv}\ (\forall \overline{v} \,.\, C \Rightarrow \tau_2) - V & \gamma_2 = \forall \overline{v} \cup V_2 \,.\, C \Rightarrow \tau_2 \end{array}}{(\forall \overline{v} \,.\, C_1 \Rightarrow \tau) \equiv \varphi \cdot \gamma_2 \rightsquigarrow \emptyset}$$

Figure 4.5: Polymorphic type scheme constraint rules

into an environment: (i) in Abs, the argument of the function; (ii) in Case, the arguments of the constructor; (iii) in Let, the variables in the definitions (to allow for recursion); and (iv) in Let, the variables in the body. In the first three cases the variable and constraint sets are forced to be empty and can be solved the easy way. In the last case it is important to note that the result from generalization is actually a value and not a variable. Also, this type scheme is propagated only towards the leaves of the proof tree (Var), which can only use variables for type schemes. This means that we can assume that the left-hand side of a constraint is a value and the right-hand side contains variables. This proves that the rules in Figure 4.5 match all other generated type scheme constraints.

Let's take a closer look at the second rule in Figure 4.5 (the most complex). To solve the type scheme constraint we need to use a type constraint ($\tau_1 \oplus \tau_2 = \tau \rightsquigarrow C_2$). However, since τ might include variables that are quantified over in the type scheme we have to include it in the constraints of the type schemes. But, if τ is not a variable we break the second part of the *simple* property, if we include C_2 directly into the constraints of the type schemes. To fix this we first use *simplify*. This might then lead to new variables in the constraint set C. We calculate which variables are new using the sets V_1 and V_2 and include these in the type schemes.

So, solving type scheme constraints can lead to new type constraints (and thus annotation constraints). However, we can remove all type scheme constraints completely.

$$\frac{C_2 = \mathit{simplify}\ C_1 \qquad V = ((\mathit{fv}\ C_2) \cup (\mathit{fv}\ \tau)) - ((\mathit{fv}\ \Gamma) \cup \{\nu, \delta\})}{\mathit{gen}\ (\tau^{\nu,\delta}, C_1, \Gamma) \equiv (\forall V\ .\ C_2 \Rightarrow \tau)^{\nu,\delta} \rightsquigarrow \emptyset}\ \textsc{Gen}$$

$$\frac{\mathit{fresh}\ \overline{\upsilon_2} \qquad \phi = [^{\overline{\upsilon_2}}/_{\overline{\upsilon_1}}]}{\mathit{inst}\ (\forall \overline{\upsilon_1}\ .\ C \Rightarrow \tau) \equiv \tau\ [\phi] \rightsquigarrow C\ [\phi]}\ \textsc{Inst}$$

Figure 4.6: Generalization & instantiation constraint rules

4.6 Generalization & instantiation constraints

Generalization and instantiation are used to create and use expressions with polymorphic types. The rules can be found in Figure 4.6.

The GEN-rule first simplifies the constraints. There are two reasons for this: (i) making sure the constraints satisfy the *simple* property; and (ii) making sure the constraints do not need to be solved for every instantiation. Then it calculates the variables it needs to quantify over using the environment Γ. It is not necessary to quantify over ν and δ, since we need the total use and demand of a **let**-definition (even if instantiated differently).

The INST-rule is only defined for type schemes values (a type scheme can also be a variable). So, before solving instantiation constraints, make sure that all relevant generalisation constraints have already been solved. The rule simply creates fresh variables for each variable the type scheme quantifies over, and substitutes these in the type and constraints.

4.7 Combining all solving methods

In the previous sections we have explained how all the constraints can be solved individually. However, all these methods have to be applied in a very specific order.

Let us look at how the methods depend on each other:

- Instantiation constraints can only be solved if all type scheme constraints are solved completely, because it is impossible to instantiate a type scheme variable.

- Type scheme constraints can only be solved if all type & annotation constraints are solved as far as possible. Otherwise we might quantify over variables which are later found to be a function, a datatype or an annotation value.

- Type, annotation & generalization constraints have no dependencies.

Now we can define *simplify* with three steps (each using the appropriate methods defined in the previous sections):

1. Solve type, annotation & generalization constraints. This removes generalization constraints completely, leaves only certain type constraints and simplifies the annotation constraints as far as possible.

2. Solve type scheme constraints. This further removes all type scheme constraints.

3. Solve instantiation constraints. This further removes all instantiation constraints.

If solving certain constraints generates new constraints then these must be dealt with first, since other constraints might depend on them. The constraints that are left satisfy the *simple* property, since all other constraints have been removed.

If type inference where defined only by using *simplify* after constraint generation, then there would still be constraints left. This is something we want to avoid.

Therefore, the top-level expression must be monomorphic. Because there are no type variables when the expression is monomorphic, we can only have annotation constraints left in the constraint set C. To solve these, do the following:

1. Set outer ν to $\mathbb{1}$ (the expression will be used exactly once when evaluating).

2. Apply *valueAnalysis* on C.

3. Pick a possible value for any annotation variable in C.

4. Apply *simplify* on C.

5. Repeat from step 2 until $C = \emptyset$.

Step 3 should be done in a smart way: don't pick $\top$ if you can pick $\mathbb{0}$ and start with variables that have the most influence on the rest of the program.

Constraint generation (Section 3) and Constraint solving (Section 4) describe the static semantics. As explained in Section 2.1, these should safely approximate run-time behavior. In the next chapter we will introduce the operational semantics to formally define the run-time behavior.

Chapter 5

Evaluation

5.1 Overview

In this chapter we will build the operational semantics for which the analysis provides approximations.

In addition to the definitions from the previous chapters we need a few extra definitions which we will introduce in Section 5.2. These definitions make it possible to write the operational semantics in Section 5.3. And finally, we will provide some theorems in Section 5.4. It is important to note that operational semantics we use are the same as in [35] and that a lot of figures in the sections 5.2 and 5.3 are a (nearly) exact copy of parts of that work.

5.2 Basic definitions

The language used in the operational semantics is defined in Figure 5.1. It consists of different components: *atoms* A, *terms* M, *shallow evaluation contexts* R and *values* V.

A lot of terms consist of multiple parts that have to be evaluated individually and in a specific order. For an application this means: evaluate the function first and then the argument. In this system that is done using shallow evaluation contexts. The hole $[\cdot]$ denotes where the value from a previously evaluated part can be plugged in. A term $R\,[M]$ specifies that M has to be evaluated first and that the value of M can later be plugged into R.

It is useful in the operational semantics to be able to make distinctions between A, M, R and V. However, now the term languages for the static and operational semantics differ. It is easy to see that terms **e** can be translated into terms M. There exists a function that does exactly that, lets call it toM.

The annotations used in these definitions might not be necessary in an actual implementation. However, they allow us to formulate a few theorems about the operational semantics.

When a hole of a shallow evaluation context is filled with a value we need to know how often that value is used. The *use* function for shallow evaluation contexts (Figure 5.2) determines the use of a value put in a hole of a shallow evaluation context. For *seq* this means that even though the first argument is evaluated, the value isn't actually used.

$$\begin{array}{rcl}
A & ::= & x \\
M & ::= & R\,[M] \;\mid\; \mathbf{let}\ \overline{x_i =^{\nu_i,\delta_i} M_i}\ \mathbf{in}\ M \;\mid\; A \;\mid\; V^{\nu} \\
R & ::= & [\cdot]\ A \;\mid\; \mathbf{case}\ [\cdot]\ \mathbf{of}\ \overline{K_i\ \overline{x_{ij}} \to M_i} \;\mid\; \mathit{seq}\ [\cdot]\ M \\
V & ::= & K_i\ \overline{A_j} \;\mid\; \lambda x \to M
\end{array}$$

Figure 5.1: Language

$$\begin{array}{ll}
\mathit{use}\ ([\cdot]\ A) & = \mathbb{1} \\
\mathit{use}\ (\mathbf{case}\ [\cdot]\ \mathbf{of}\ \overline{K_i\ \overline{x_{ij}} \to M_i}) & = \mathbb{1} \\
\mathit{use}\ (\mathit{seq}\ [\cdot]\ M) & = \mathbb{0}
\end{array}$$

Figure 5.2: Use of shallow evaluation contexts

A *heap* (Figure 5.3) associates variable names with use, demand and a term.

$$H \quad ::= \quad \emptyset \;\mid\; H, x =^{\nu,\delta} M$$

Figure 5.3: Heaps

The *stack* (Figure 5.4) is used to delay the evaluation of a shallow evaluation context or a variable update to whenever a value is available to fill a hole or to be written to the heap respectively.

$$S \quad ::= \quad \epsilon \;\mid\; R, S \;\mid\; \#^{\nu,\delta}\ x, S$$

Figure 5.4: Stacks

The *use* function for stacks (Figure 5.5) determines how often a value is used when evaluating instructions that were put on the stack.

$$\begin{array}{ll}
\mathit{use}\ (\epsilon) & = \mathbb{0} \\
\mathit{use}\ (R, S) & = \mathit{use}\ R \\
\mathit{use}\ (\#^{\nu,\delta}\ x, S) & = \nu \oplus (\mathit{use}\ S)
\end{array}$$

Figure 5.5: Use of stacks

A *configuration* (Figure 5.6) is a collection of a heap, a term and a stack.

$$C \quad ::= \quad \langle H; M; S \rangle$$

Figure 5.6: Configurations

The *subtraction operator* $\ominus$ (Figure 5.7) is an operator that subtracts annotation values. Subtraction is a little more complex than addition ($\oplus$), because ∞ represents the infinite set “at least 2 uses” and we do not have negative numbers. We have to introduce a helper function $-$.

$$\pi_1 - \pi_2 \quad ::= \quad \begin{array}{c|ccc} \pi_2 \backslash \pi_1 & 0 & 1 & \infty \\ \hline 0 & \{0\} & \{1\} & \{\infty\} \\ 1 & \emptyset & \{0\} & \{1,\infty\} \\ \infty & \emptyset & \emptyset & \{0,1,\infty\} \end{array}$$

$$\varpi_1 \ominus \varpi_2 \quad ::= \quad \bigcap_{\pi_2 \in \varpi_2} \bigcup_{\pi_1 \in \varpi_1} \pi_1 - \pi_2$$

Figure 5.7: Subtraction operator

5.3 Operational semantics

The operational semantics can be found in Figure 5.8. First we will explain roughly how evaluation works. Then we will explain the meaning of the different groups of rules in the operational semantics.

Given a term $\mathbf{e}$, evaluation works as follows:

1. Apply type inference: $\emptyset \vdash \mathbf{let}\ main = \mathbf{e}\ \mathbf{in}\ main : \eta \rightsquigarrow \emptyset$.

2. Make sure the generalized type of *main* does not contain any demand variables. Otherwise the condition Terminate might not hold.

3. Start with configuration $\langle \emptyset; toM\ \mathbf{e}; \epsilon \rangle$.

4. Apply evaluation rules until the condition Terminate has been met.

It is important to note that all the ν and δ in $\mathbf{e}$ should take the form of annotation values (Section 3.2) after type inference, because constraint solving (Section 4) does not leave any variables. The result is that operations like $\sqsubseteq$ and $\ominus$ can be executed directly instead of generating constraints.

The rules Unwind, Reduce, Let, Lookup and Update evaluate all configurations except the ones where $M = V^{\nu}$ and $S = \epsilon$. These are matched by the condition Terminate.

The Unwind-rule evaluates a term $R\,[M]$ by first evaluating M and delaying the evaluation of R by putting it on the stack.

When a term is finally evaluated to a value, the Reduce-rule can be used to take a delayed shallow evaluation context from the stack, fill the hole, and continue evaluating the resulting term. The condition $use\ (R) \sqsubseteq \nu$ is there to make sure that the value is not used more often than the annotation permits.

The Let-rule creates fresh variables to place all the definitions on the heap, while substituting the variables everywhere necessary. If it wouldn't use fresh variables there might be naming collisions with other **let**s that use the same names. The last step is simply a matter of evaluating the body of the **let**.

The Lookup-rule is used to evaluate variables. After looking up the variable in the heap, the associated term M will be evaluated. To make sure that the term associated with this variable is only evaluated once we place a variable-update instruction on the stack. If the value of the evaluated term will be used by delayed instructions on the stack we have to lower ν by that use when placing it back on the heap. Since each lookup corresponds with a demand, δ is lowered by one. The conditions make sure this rule is only valid if it is impossible for the annotations to become invalid.

Similar to the Reduce-rule, the Update-rule requires that a term has been evaluated to a value. However, the Update-rule is used to process variable update instructions placed on the stack. The variable specified by the instruction is updated on the heap with the value of the term, along with the use and demand specified in the instruction. The resulting term is again the value. However its use has been lowered, with a condition ensuring that the annotation does not become invalid.

The rules App, Case and Seq evaluate all shallow evaluation contexts if they are provided with a value.

The App-rule expects the value to be a function, and can simply substitute the variable of the function with the argument.

The Case-rule expects the value to be a constructor, finds the matching case and returns the corresponding terms where the relevant variables have been substituted.

The Seq-rule does not care about the value and simply returns the second argument.

Operational semantics $\boxed{\langle H; M; S\rangle \rightarrowtail \langle H; M; S\rangle}$

$$\frac{}{\langle H; R\,[M]; S\rangle \rightarrowtail \langle H; M; R, S\rangle}\ \textsc{Unwind}$$

$$\frac{(\mathit{use}\ R) \sqsubseteq \nu \qquad R\,[V] \rightarrowtail M}{\langle H; V^{\nu}; R, S\rangle \rightarrowtail \langle H; M; S\rangle}\ \textsc{Reduce}$$

$$\frac{\mathit{fresh}\ \overline{y_i} \qquad \phi = [{}^{\overline{y_i}}/_{\overline{x_i}}]}{\langle H; \mathbf{let}\ \overline{x_i =^{\nu_i,\delta_i} M_i}\ \mathbf{in}\ M; S\rangle \rightarrowtail \langle H, \overline{y_i =^{\nu_i,\delta_i} M_i\,[\phi]}; M\,[\phi]; S\rangle}\ \textsc{Let}$$

$$\frac{\nu \ominus (\mathit{use}\ S) \not\equiv \bot \qquad \delta \ominus \mathbb{1} \not\equiv \bot}{\langle H, x =^{\nu,\delta} M; x; S\rangle \rightarrowtail \langle H; M; \#^{\nu \ominus \mathit{use}\ (S), \delta \ominus \mathbb{1}}\ x, S\rangle}\ \textsc{Lookup}$$

$$\frac{\nu_1 \ominus \nu_2 \not\equiv \bot}{\langle H; V^{\nu_1}; \#^{\nu_2,\delta_2}\ x, S\rangle \rightarrowtail \langle H, x =^{\nu_2,\delta_2} V^{\nu_2}; V^{\nu_1 \ominus \nu_2}; S\rangle}\ \textsc{Update}$$

Termination condition $\boxed{\langle H; M; S\rangle}$

$$\frac{\overline{\mathbb{0} \sqsubseteq \nu_i} \qquad \overline{\mathbb{0} \sqsubseteq \delta_i}}{\langle \overline{x_i =^{\nu_i,\delta_i} M_i}; V^{\nu}; \epsilon\rangle}\ \textsc{Terminate}$$

Evaluation of a shallow evaluation context $\boxed{R\,[V] \rightarrowtail M}$

$$\frac{}{(\lambda x \rightarrow M)\ A \rightarrowtail M\ [{}^{A}/_{x}]}\ \textsc{App}$$

$$\frac{}{\mathbf{case}\ K_k\ \overline{A_j}\ \mathbf{of}\ \overline{K_i\ \overline{x_{ij}} \rightarrow M_i} \rightarrowtail M_k\ [{}^{\overline{A_j}}/_{\overline{x_{kj}}}]}\ \textsc{Case}$$

$$\frac{}{\mathit{seq}\ V\ M \rightarrowtail M}\ \textsc{Seq}$$

Figure 5.8: Operational semantics

5.4 Theorems

When we stated in the last section that the rules of the operational semantics cover all possible configurations, we were not entirely truthful. While their conclusions do indeed cover all possible configurations, some of the rules have conditions which might not always be satisfied. If any of conditions are not met, a conclusion might not be reached.

In the static semantics we try to infer values for annotations. The operational semantics subtract from the statically inferred annotation values for each use or demand. If the value reaches $\bot$, the inferred value was too low and evaluation gets stuck (because of the conditions in the rules). However, a value that is too high might never be detected since it would never reach $\bot$. That is the reason for the termination condition: if any annotation does not allow zero use / demand at the end of the program, the annotation was clearly wrong and evaluation gets stuck. Note that "to allow zero" means that an annotation must include zero, not that it should be equal to zero. If we can prove that evaluation never gets stuck we know for sure that the static semantics are correct with respect to the operational semantics.

In this section we want to give two definitions and state two theorems that, when proven, show that evaluation never gets stuck. These theorems also imply soundness [38].

Definition 2. An annotation is *valid* if it is not equivalent to $\bot$.

Definition 3. A configuration C is *valid* (denoted *valid* C) if all the annotations it contains are valid.

Theorem 1 (Subject reduction). Let $C_1 = \langle H_1; toM\ \mathbf{e_1}; S_1\rangle$ and $C_2 = \langle H_2; toM\ \mathbf{e_2}; S_2\rangle$. If $C_1 \rightarrowtail C_2$ and $\emptyset \vdash \mathbf{e_1} : \eta$ and *valid* C_1 then $\emptyset \vdash \mathbf{e_2} : \eta$ and *valid* C_2.

Theorem 2 (Termination). Let $C = \langle \overline{x_i =^{\nu_i,\delta_i} M_i}; toM\ \mathbf{e}; \epsilon\rangle$. If $toM\ \mathbf{e} = V^{\nu}$ and $\emptyset \vdash \mathbf{e} : \eta$ and *valid* C then $\overline{\mathbb{0} \sqsubseteq \nu_i}$ and $\overline{\mathbb{0} \sqsubseteq \delta_i}$.

The first theorem states that the evaluation preserves both type and validity of annotations.

The second theorem states that the termination conditions hold if a configuration is reached that should terminate.

Both theorems have been formalized in Coq. However, there was not enough time to fully prove both theorems.

Chapter 6

Heap recycling

6.1 Overview

[13] describes a method that allows a programmer to reuse heap space once it has been used. The idea behind this paper is that if there is a **case**-expression on a unique variable, then the heap cell for that variable can be reused in the **case**-arms. This is safe because we know that the variable has just been used and since it is unique it will not be used any further.

We had a couple of difficulties incorporating this analysis into our own, so it deserves its own chapter. Since the paper uses uniqueness analysis, it is assumed in this chapter that the analysis is configured for uniqueness analysis (Section 3.6).

6.2 Static semantics

The paper suggests a new application construct when the argument is a constructor. However, our analysis expects that the argument of an application is always a variable. It turns out that this is quite simple to solve.

The language in the paper is monomorphic, only supports lists and can only recycle a heap cell when its replacement value has the exact same constructor as the original value (because memory sizes of different constructors might differ). Let's try to find a way to remove all these limitations.

The extension for the static semantics (Figure 6.1) introduces a new type, new terms and new inference rules.

The new type $\#n$ describes a heap cell with n bytes of memory. The function $[\cdot]$ is used to find out how many bytes a specific value requires in memory.

There are two new terms. The new **let** defines the variable x while recycling the heap cell h and only allows values to be defined. Since the value now has a variable to reference it, this solves the issue with the new application construct defined in the paper. The new **case** matches variable x and allows its heap cell h to be recycled in the case arms. The heap cells h are actually normal variables in the type system. However, their types are the new heap cell descriptor type.

The Let'-rule first types the body. Then it is checked that the type of the definition matches its use in the body. Lastly, we have to make sure that the heap cell h has size $[K]$ and that it is demanded once.

New type & terms

$$
\begin{array}{rcll}
\tau & ::= & \ldots & \mid \ \# n \\
\mathbf{e} & ::= & \ldots & \mid \ \textbf{let } h@x =^{\nu,\delta} K^{\nu}\ \overline{x_i} \textbf{ in } \mathbf{e} \ \mid \ \textbf{case } h@x \textbf{ of } \overline{K_i\ \overline{x_{ij}} \to \mathbf{e_i}}
\end{array}
$$

New inference rules $\boxed{\Gamma \vdash \mathbf{e} : \eta_{\tau}}$

$$
\dfrac{\begin{array}{c}\Gamma_1, x : (\forall \emptyset\,.\ \emptyset \Rightarrow \tau)^{\nu,\delta} \vdash \mathbf{e} : \eta \rightsquigarrow C_1 \\ \Gamma_2 \vdash K^{\nu}\ \overline{x_i} : \tau^{\nu} \rightsquigarrow C_2 \qquad \Gamma_1 \oplus \Gamma_2 \oplus (h : (\forall \emptyset\,.\ \emptyset \Rightarrow \#[K])^{\nu,\mathbb{1}}) = \Gamma \rightsquigarrow C_3\end{array}}{\Gamma \vdash \textbf{let } h@x =^{\nu,\delta} K^{\nu}\ \overline{x_i} \textbf{ in } \mathbf{e} : \eta \rightsquigarrow C_1 \cup C_2 \cup C_3}\ \textsc{Let'}
$$

$$
\dfrac{\begin{array}{c}\textbf{data } T\ \overline{u_l}\ \overline{\alpha_k} = \overline{K_i\ \overline{\rho_{ij}}} \\ {\tau_{ij}}^{\nu_{ij},\delta_{ij}} = \rho_{ij}\ [\overline{\varphi_l}/\overline{u_l}, \overline{\tau_k}/\overline{\alpha_k}] \qquad \Gamma_0 \vdash x : (T\ \overline{\varphi_l}\ \overline{\tau_k})^{\mathbb{1}} \rightsquigarrow C_1 \\ \Gamma_i, h : (\forall \emptyset\,.\ \emptyset \Rightarrow \# n_i)^{\nu,\mathbb{1}}, \overline{x_{ij} : (\forall \emptyset\,.\ \emptyset \Rightarrow \tau_{ij})^{\nu_{ij},\delta_{ij}}} \vdash \mathbf{e_i} : \eta \rightsquigarrow C_2 \\ \Gamma_0 \oplus (\bigsqcup_i \Gamma_i) = \Gamma \rightsquigarrow C_3 \qquad n_i \leqslant [K_i]\end{array}}{\Gamma \vdash \textbf{case } h@x \textbf{ of } \overline{K_i\ \overline{x_{ij}}} \to \mathbf{e_i} : \eta \rightsquigarrow C_1 \cup C_2 \cup C_3}\ \textsc{Case'}
$$

Figure 6.1: Extension for static semantics

The Case'-rule is very similar to the Case-rule. The main differences are that (i) what is matched upon (x) must be a variable (instead of any expression) and must be used exactly once (instead of at least once); and (ii) each **case**-arm has the heap cell h in the environment. It is important to note that the size of the heap cell is different in each **case**-arm (since each constructor can have a different size) and that its demand must be $\mathbb{1}$ (otherwise the heap cell might be recycled twice). The size of the heap cell must be less or equal to its respective constructor ($n_i \leqslant [K_i]$). Since size does not place any restrictions on the type we could change the constructor when recycling. Also, since the size does not have to be exact, it is possible to recycle a *Cons* as a *Nil* for example.

6.3 Operational semantics

There is no point in extending the static semantics with new language constructs if we don't add them to operational semantics too. The extended operational semantics can be found in Figure 6.2.

The terms of the operational semantics are extended in the same way the terms of the static semantics were.

The normal Let-rule creates fresh heap bindings. However, the Let'-rule doesn't. Since we know that h is a valid heap binding that isn't used anymore, we do not need to create a fresh heap binding for x. We can simply use h. To do that we update h with M_2 and replace every occurrence of x with h in M_3. However, since we are overwriting an existing heap binding we have to verify that the existing binding has exactly zero use and demand. We should not be overwriting heap bindings that could still be accessed.

In the Case'-rule we know that x has a heap binding. So, we can replace all h with x in $\overline{M_i}$ and then evaluate it further as if it is a normal **case**-expression.

New terms

$$M \quad ::= \quad \ldots \quad | \; \mathbf{let}\ h@x =^{\nu,\delta} M_1\ \mathbf{in}\ M_2 \quad | \; \mathbf{case}\ h@x\ \mathbf{of}\ \overline{K_i\ \overline{x_{ij}} \to M_i}$$

New inference rules $\boxed{\langle H; M; S\rangle \rightarrowtail \langle H; M; S\rangle}$

$$\frac{}{\langle H, h =^{\mathbb{0},\mathbb{0}} M_1; \mathbf{let}\ h@x =^{\nu,\delta} M_2\ \mathbf{in}\ M_3; S\rangle \rightarrowtail \langle H, h =^{\nu,\delta} M_2; M_3\ [^{h}/_{x}]; S\rangle}\ \textsc{Let'}$$

$$\frac{}{\langle H; \mathbf{case}\ h@x\ \mathbf{of}\ \overline{K_i\ \overline{x_{ij}} \to M_i}; S\rangle \rightarrowtail \langle H; x; \mathbf{case}\ [\cdot]\ \mathbf{of}\ \overline{K_i\ \overline{x_{ij}} \to M_i\ [^{x}/_{h}]}, S\rangle}\ \textsc{Case'}$$

Figure 6.2: Extension for operational semantics

```
id1 h@[ ] = h@[ ]
id1 h@(x : xs) = h@(x : (id1 xs))
id2 = λa → case a of
  [ ] → (λb → b) a@[ ]
  (x : xs) → let tmp = id2 xs
    in (λb → b) a@(x : tmp)
id3 = λa → case h@a of
  [ ] → let h@b = [ ] in b
  (x : xs) → let tmp = id3 xs
    in let h@b = x : tmp in b
```

Figure 6.3: Identity function for lists

6.4 Improvement

Let's take a look at some examples. The figures 6.3 and 6.4 contain the functions *id* and *reverse* respectively. Both work on lists. In each of these figures the first version looks somewhat readable and shows the intent of the function. The second version uses the notation from the original paper and the last version uses the notation from this chapter.

The first thing we see is that both notations look very similar. The second thing to note is that we have to introduce a *tmp* variable to make sure the invariants regarding function and constructor application are not broken (Section 3.2, under the definition of terms). This means that an extra heap cell is allocated for every cons. This is the case in both *id* and *reverse*, while we'd expect them to do no extra allocations. Without heap recycling it would be worse (2 extra heap cell allocations for every cons). However, it is still worse than our expectation. So, we should try to improve it.

It is important that we maintain the invariants regarding function application and constructor application. If we don't, all the rules in the analysis might become overly complex. Are there any ways we can keep the new variable, but avoid the extra allocation? Lets make another observation: both *id* and *reverse*

```
reverse1 l =
  let rev [] acc = acc
      rev h@(x : xs) acc = rev xs h@(x : acc)
  in rev l []
reverse2 =
  let rev = λl → λacc → case l of
     [] → acc
     (x : xs) → let tmp = rev xs
        in tmp l@(x : acc)
  inλl → let f = rev l in f Nil
reverse3 =
  let rev = λl → λacc → case h@l of
     [] → acc
     (x : xs) → let tmp = rev xs
        in let h@r = x : acc in tmp r
  inλl → let f = rev l in f Nil
```

Figure 6.4: Reverse function for lists

actually return a value directly from a heap binding.

So, one way of solving the issue would be to introduce a new **let** construct that requires the value of the definition to be associated directly with a heap binding. This would also need an analysis to verify that this is the case. Then the new **let** would not have to allocate a heap cell.

A more simple approach would be to detect these cases at runtime. This would avoid a complex static analysis. Currently a runtime configuration consists of $\langle H; M; S \rangle$. Changing this to $\langle H; M; \mathit{Maybe}\ x; S \rangle$ would allow us to track at runtime whether or not M is associated directly with a heap cell; and if so, which one. The LET-rule in the operational semantics would have to be altered to only allocate a new heap cell when there is no heap cell associated with M.

Further research is necessary to determine the best way to solve this issue.

Chapter 7

Example

7.1 Overview

In this chapter we will take a look at a simple example. With this example we attempt to show how the static semantics, the dynamic semantics and heap recycling work in practice.

First, the example is introduced in Section 7.2. Then, we will look at how we should type (and thus annotate) the program (Section 7.3). Lastly we will run the program (Section 7.4).

7.2 Code

Lets look at a simple example:

$$\begin{aligned} &\mathit{swap}\ h@(a, b) = h@(b, a) \\ &\mathit{main} = \mathit{swap}\ (0, 1) \end{aligned}$$

The idea behind the *swap* function is that it swaps the elements of the pair without doing any additional heap allocations.

Unfortunately this code is not usable directly in our system, so it has to be transformed to fit our type system. The result of the transformation can be found in Figure 7.1.

```
let main =
  let swap = λx → case h@x of
    MkPair a b → let h@r = MkPair b a in r
  in let zero = Z
  in let one = S zero
  in let v = MkPair zero one
  in swap v
in main
```

Figure 7.1: Transformed code

```
$\textbf{data}\ \mathit{Pair}\ [\beta_1, \beta_2, \beta_3, \beta_4]\ [\alpha_1, \alpha_2]$
    $= \mathit{MkPair}\ {\alpha_1}^{\beta_1, \beta_2}\ {\alpha_2}^{\beta_3, \beta_4}$
$\textbf{data}\ \mathit{Nat}\ [\beta_1, \beta_2]\ [\,]$
    $= Z$
    $\mid S\ \mathit{Nat}^{\beta_1, \beta_2}$
```

Figure 7.2: Annotated datatype definitions

```
$\textbf{let}\ \mathit{main} :: (\mathit{Pair}\ [\mathbb{0}, \mathbb{0}, \mathbb{0}, \mathbb{0}]\ [\mathit{Nat}\ [\mathbb{0}, \mathbb{0}]\ [\,], \mathit{Nat}\ [\mathbb{0}, \mathbb{0}]\ [\,]])\ ^{\mathbb{1}}$
  $\mathit{main} =^{\mathbb{1}, \mathbb{1}}$
  $\textbf{let}\ \mathit{swap} :: (\forall \{\alpha_1, \alpha_2, \beta_1, \beta_2, \beta_3, \beta_4, \beta_5\}\ .\ [\,] \Rightarrow$
    $(\mathit{Pair}\ [\beta_1, \beta_2, \beta_3, \beta_4]\ [\alpha_1, \alpha_2])^{\mathbb{1}, \mathbb{1}} \rightarrow$
    $(\mathit{Pair}\ [\beta_3, \beta_4, \beta_1, \beta_2]\ [\alpha_2, \alpha_1])^{\beta_5})\ ^{\mathbb{1}}$
    $\mathit{swap} =^{\mathbb{1}, \mathbb{1}} \lambda^{\mathbb{1}}\ x \rightarrow \textbf{case}\ h@x\ \textbf{of}$
      $\mathit{MkPair}\ a\ b \rightarrow \textbf{let}\ h@r =^{\mathbb{1}, \mathbb{1}}\ \mathit{MkPair}^{\mathbb{1}}\ b\ a\ \textbf{in}\ r$
  $\textbf{in let}\ \mathit{zero} :: (\forall \{\beta_1, \beta_2\}\ .\ [\,] \Rightarrow \mathit{Nat}\ [\beta_1, \beta_2]\ [\,])\ ^{\mathbb{0}}$
    $\mathit{zero} =^{\mathbb{0}, \omega}\ Z^{\mathbb{0}}$
  $\textbf{in let}\ \mathit{one} :: (\forall \{\beta_1, \beta_2\}\ .\ [\,] \Rightarrow \mathit{Nat}\ [\beta_1, \beta_2]\ [\,])\ ^{\mathbb{0}}$
    $\mathit{one} =^{\mathbb{0}, \mathbb{1}}\ S^{\mathbb{0}}\ \mathit{zero}$
  $\textbf{in let}\ v :: (\forall \{\beta_1, \beta_2, \beta_3, \beta_4, \beta_5, \beta_6, \beta_7, \beta_8\}\ .\ [\,] \Rightarrow$
    $\mathit{Pair}\ [\beta_1, \beta_2, \beta_3, \beta_4]\ [\mathit{Nat}\ [\beta_5, \beta_6]\ [\,], \mathit{Nat}\ [\beta_7, \beta_8]\ [\,]])\ ^{\mathbb{1}}$
    $v =^{\mathbb{1}, \mathbb{1}}\ \mathit{MkPair}^{\mathbb{1}}\ \mathit{zero}\ \mathit{one}$
  $\textbf{in}\ \mathit{swap}\ v$
$\textbf{in}\ \mathit{main}$
```

Figure 7.3: Annotated code

The transformed code contains some datatypes that are not yet defined. Figure 7.2 contains the annotated datatype definitions.

7.3 Type inference

The next step is to let the type inference algorithm annotate the source code (Figure 7.3). Nothing uses the result of the *main* expression, so let's set the use/demand variables within its result to $\mathbb{0}$ (for simplicity's sake). Also, the outer ν has been set to $\mathbb{1}$ to drive evaluation, as required in Section 4.7. In the type of *swap* we can see that the argument is required to be unique, which is logical if we consider that the associated heap binding will be reused. Since the function does not use the values of the pair, its types and annotations are variables.

Please note that *main*, *swap* and v are all used exactly once. So, we could perform all the optimizations associated with sharing analysis (requires "at most once") and strictness analysis (requires "at least once"). The expressions *zero* and *one* are not used at all, so the optimizations for absence analysis could be used. The optimizations are described in Section 2.4.

```
let $main =^{\mathbb{1},\mathbb{1}}$
  let $swap =^{\mathbb{1},\mathbb{1}} (\lambda x \to$ case $h@x$ of
    $MkPair\ a\ b \to$ let $h@r =^{\mathbb{1},\mathbb{1}} (MkPair\ b\ a)^{\mathbb{1}}$ in $r)\ ^{\mathbb{1}}$
  in let $zero =^{\mathbb{0},\omega} Z^{\mathbb{0}}$
  in let $one =^{\mathbb{0},\mathbb{1}} (S\ zero)^{\mathbb{0}}$
  in let $v =^{\mathbb{1},\mathbb{1}} (MkPair\ zero\ one)^{\mathbb{1}}$
  in $([\cdot]\ v)\ [swap]$
in $main$
```

Figure 7.4: Evaluable code

7.4 Evaluation

Evaluation uses yet another representation of the code, so lets start by transforming the result from type inference to the valid representation (Figure 7.4).

The valid representation can be put directly into a configuration. For completeness we included the entire evaluation sequence. There is not a whole lot to explain, since each configuration forces us to use one specific evaluation rule to go to the next configuration.

Start with:

$$\begin{array}{l}
\langle \emptyset; \\
\textbf{let } swap =^{\mathbb{1},\mathbb{1}} \cdots \textbf{in} \\
\textbf{let } zero =^{\mathbb{0},\omega} \cdots \textbf{in} \\
\textbf{let } one =^{\mathbb{0},\mathbb{1}} \cdots \textbf{in} \\
\textbf{let } v =^{\mathbb{1},\mathbb{1}} \cdots \textbf{in} \\
([\cdot]\ v)\ [swap]; \\
\epsilon \rangle
\end{array}$$

Apply Let multiple times:

$$\begin{array}{l}
\langle swap =^{\mathbb{1},\mathbb{1}} \cdots , \\
zero =^{\mathbb{0},\omega} \cdots , \\
one =^{\mathbb{0},\mathbb{1}} \cdots , \\
v =^{\mathbb{1},\mathbb{1}} \cdots ; \\
([\cdot]\ v)\ [swap]; \\
\epsilon \rangle
\end{array}$$

Apply Unwind:

$$\begin{array}{l}
\langle swap =^{\mathbb{1},\mathbb{1}} \cdots , \\
zero =^{\mathbb{0},\omega} \cdots , \\
one =^{\mathbb{0},\mathbb{1}} \cdots , \\
v =^{\mathbb{1},\mathbb{1}} \cdots ; \\
swap; \\
[\cdot]\ v, \epsilon \rangle
\end{array}$$

Apply Lookup:

$$\begin{array}{l}
\langle \mathit{zero} =^{\mathbb{0},\omega} \cdots , \\
\mathit{one} =^{\mathbb{0},\mathbb{1}} \cdots , \\
v =^{\mathbb{1},\mathbb{1}} \cdots ; \\
(\lambda x \to \cdots)\ \mathbb{1}; \\
\#^{\mathbb{0},\mathbb{0}}\ \mathit{swap}, [\cdot]\ v, \epsilon\rangle
\end{array}$$

Apply Update:

$$\begin{array}{l}
\langle \mathit{swap} =^{\mathbb{0},\mathbb{0}} \cdots , \\
\mathit{zero} =^{\mathbb{0},\omega} \cdots , \\
\mathit{one} =^{\mathbb{0},\mathbb{1}} \cdots , \\
v =^{\mathbb{1},\mathbb{1}} \cdots ; \\
(\lambda x \to \cdots)\ \mathbb{1}; \\
[\cdot]\ v, \epsilon\rangle
\end{array}$$

Apply Reduce, App:

$$\begin{array}{l}
\langle \mathit{swap} =^{\mathbb{0},\mathbb{0}} \cdots , \\
\mathit{zero} =^{\mathbb{0},\omega} \cdots , \\
\mathit{one} =^{\mathbb{0},\mathbb{1}} \cdots , \\
v =^{\mathbb{1},\mathbb{1}} \cdots ; \\
\textbf{case}\ h@v\ \textbf{of} \\
\quad \mathit{MkPair}\ a\ b \to \cdots ; \\
\epsilon\rangle
\end{array}$$

Apply Unwind:

$$\begin{array}{l}
\langle \mathit{swap} =^{\mathbb{0},\mathbb{0}} \cdots , \\
\mathit{zero} =^{\mathbb{0},\omega} \cdots , \\
\mathit{one} =^{\mathbb{0},\mathbb{1}} \cdots , \\
v =^{\mathbb{1},\mathbb{1}} \cdots ; \\
v; \\
\textbf{case}\ [\cdot]\ \textbf{of}\ \mathit{MkPair}\ a\ b \to [{}^{v}/_{h}] \cdots , \\
\epsilon\rangle
\end{array}$$

Apply Lookup:

$$\begin{array}{l}
\langle \mathit{swap} =^{\mathbb{0},\mathbb{0}} \cdots , \\
\mathit{zero} =^{\mathbb{0},\omega} \cdots , \\
\mathit{one} =^{\mathbb{0},\mathbb{1}} \cdots , \\
(\mathit{MkPair}\ \mathit{zero}\ \mathit{one})^{\mathbb{1}}; \\
\#^{\mathbb{0},\mathbb{0}}\ v, \\
\textbf{case}\ [\cdot]\ \textbf{of}\ \mathit{MkPair}\ a\ b \to [{}^{v}/_{h}] \cdots , \\
\epsilon\rangle
\end{array}$$

Apply Update:

$$\begin{array}{l}
\langle \mathit{swap} =^{\mathbb{0},\mathbb{0}} \cdots , \\
\mathit{zero} =^{\mathbb{0},\omega} \cdots , \\
\mathit{one} =^{\mathbb{0},\mathbb{1}} \cdots , \\
v =^{\mathbb{0},\mathbb{0}} \cdots ; \\
(\mathit{MkPair}\ \mathit{zero}\ \mathit{one})^{\mathbb{1}}; \\
\textbf{case}\ [\cdot]\ \textbf{of}\ \mathit{MkPair}\ a\ b \to [{}^{v}/_{h}] \cdots , \\
\epsilon\rangle
\end{array}$$

Apply Reduce:

$$\begin{array}{l}\langle \mathit{swap} =^{\mathbb{0},\mathbb{0}} \cdots ,\\ \mathit{zero} =^{\mathbb{0},\omega} \cdots ,\\ \mathit{one} =^{\mathbb{0},\mathbb{1}} \cdots ,\\ v =^{\mathbb{1},\mathbb{1}} \mathit{MkPair}\ \mathit{one}\ \mathit{zero};\\ v;\\ \epsilon\rangle\end{array}$$

Apply Lookup:

$$\begin{array}{l}\langle \mathit{swap} =^{\mathbb{0},\mathbb{0}} \cdots ,\\ \mathit{zero} =^{\mathbb{0},\omega} \cdots ,\\ \mathit{one} =^{\mathbb{0},\mathbb{1}} \cdots ;\\ \mathit{MkPair}\ \mathit{one}\ \mathit{zero};\\ \#^{\mathbb{0},\mathbb{0}}\ v, \epsilon\rangle\end{array}$$

Apply Update:

$$\begin{array}{l}\langle \mathit{swap} =^{\mathbb{0},\mathbb{0}} \cdots ,\\ \mathit{zero} =^{\mathbb{0},\omega} \cdots ,\\ \mathit{one} =^{\mathbb{0},\mathbb{1}} \cdots ,\\ v =^{\mathbb{0},\mathbb{0}} \mathit{MkPair}\ \mathit{one}\ \mathit{zero},\\ \mathit{MkPair}\ \mathit{one}\ \mathit{zero};\\ \epsilon\rangle\end{array}$$

Since we evaluated to a value and the stack is empty, this is where the program terminates. Note that the termination condition does not hold. This is because we cheated a little bit during type inference. The generalized type of *main* contained demand variables (which is not allowed according to Section 5.3). These were all set to $\mathbb{0}$ for simplicity. The demand variables express the demand on *zero* and *one*, and these are exactly the values that make the termination condition fail.

So, we should not have evaluated this program in this way to begin with. However, *swap* and v are indeed correctly annotated with $\mathbb{0}, \mathbb{0}$.

Chapter 8

Implementation

An implementation of the analysis described in this paper has been made. The language used for the implementation was Haskell and the library to parse and represent the language was `haskell-src-exts`.

The implementation can be found in the folder `prototype/` and only does constraint generation (Section 3) and constraint solving (Section 4). Evaluation (Section 5) was omitted. However, a partial proof of the theorems in Section 5.4 can be found in the folder `proofs/`.

Figure 8.1 contains an overview of the important folders and files.

`proofs/Data.v`	Definitions from sections 3.2 and 5.2
`proofs/Dynamic.v`	Proof tree representation for Figure 5.8
`proofs/Proofs.v`	Partial proof of theorems in Section 5.4
`proofs/Static.v`	Proof tree representation for Figure 3.18
`prototype/src/Data`	Environments, types, annotations, variables, etc.
`prototype/src/Utils`	Utilities (substitution, monads)
`prototype/src/Datatypes.hs`	Annotated definitions of common datatypes
`prototype/src/Main.hs`	Main
`prototype/src/Inference.hs`	Constraint generation
`prototype/src/Simplify.hs`	Constraint solving

Figure 8.1: Overview of important folders and files

Chapter 9

Related Work

Until the early 1990s a lot of work on the different analyses has been done using abstract interpretation (Section 2.3.2): Hudak [18], Goldberg [11], Benton [3], Nöcker [28], Marlow [23]. However, interest in abstract interpretation has faded because it is difficult to analyze higher order functions, which occur very often in functional languages. Very recently abstract interpretation has been used again for the GHC compiler (Sergey [31], Sergey et al. [32]). Here, due to aggressive inlining, there are fewer higher order functions. When precise analysis of higher-order functions is not required, implementing abstract interpretation is said to be faster.

In the meantime the most popular approach was type and effect systems (Section 2.3.1), which has been used in this master thesis. A lot of work has been done on individual analyses, while the papers that do look at more than one analysis are relatively recent. In the following paragraphs several important papers will be discussed.

Launchbury et al. [22] present one of the first papers on sharing analysis. The targeted language is based on the one used by Jones et al. [20], which introduces an abstract machine for implementing functional languages and is actually used in GHC. Annotations in this paper are of the form: (i) **Zero**, never used; (ii) **One**, used at most once; and (iii) **Many**, used any number of times. So it is always safe to decide on **One** over **Zero** or on **Many** over **One** and – by transitivity – on **Many** over **Zero** when annotating an expression. The analysis is not polymorphic or polyvariant, however, it does have subeffecting.

The language used by Turner et al. [33] (also on sharing analysis), is similar to the one used by Launchbury et al. [22]. However, annotations are changed to: (i) 1, used at most once; and (ii) ω, used any number of times. This analysis also allows lists to be used and carefully considers recursion. Even though it is mentioned, polymorphism and polyvariance are not used. It does include a dynamic semantics – which were missing from [22] – and proves the analysis sound.

Mogensen [26] adds the 0 annotation back to the work of Turner et al. [33], which they had removed from Launchbury et al. [22]. However, lists are removed from the analysis. Instead, tuples are added, including a special language construct to access both parts of a tuple while maintaining a single use of the value. This paper does not include polymorphism or polyvariance. It does contain subtyping and subeffecting. A soundness proof is omitted, however, they argue it is similar to that of Turner et al. [33]. Instead they focus on solving the generated constraints by transforming them into Horn-clauses. The least solution can be found in time linear in the size of the constraint set.

Wansbrough and Jones [36] extend the work of Turner et al. [33] with polymorphism and algebraic data types. It includes subtyping and subeffecting. Polyvariance is not used, because the authors argue it is to expensive and does not provide enough additional precision compared to subeffecting. It also shows which program transformations can be used if usage information is available. The language includes recursive lets, constructors and case statements. Types can now contain data types and it is possible to use type schemes. The annotations are still the same as in [33]. For algebraic data types the annotation is placed on the type itself, but not on the arguments. It does discuss a couple of options for annotating data types. However, we think Gedell et al. [9] explain this more clearly, so we will not discuss it now. The paper does include a soundness proof.

Wansbrough and Jones [37] present new work, based on their previous paper: Wansbrough and Jones [36]. The sharing analysis in the earlier paper turned out to be worthless in practice. It turns out that the inferred usage information is very bad when functions are curried. They argue that polyvariance is needed, but that constrained polyvariance would cost too much. The new algorithm they present has polymorphism, unconstrained polyvariance, subtyping and subeffecting. In the paper they have omitted data types, however, these are supported in the implementation they have built. Measurements show an average 3% decrease in run-time. The algorithm still works worse than expected, the authors argue this is due to their treatment of data types – for which they use only a single annotation. As future work they suggest an extension to the 7-point Bierman lattice with the elements: $= 0$, $= 1$, >1, $\leqslant 1$, $\not\equiv 1$, $\geqslant 1$ and $\perp$. This could lead to a system that provides usage, strictness and absence information.

The paper of Wansbrough and Jones [37] is later accompanied by a PhD thesis by Wansbrough [35]. It contains the same sharing analysis, however, with more attention to details like data types, implementation and proofs. Especially Appendix C is interesting, where the author provides a concept for an extended lattice. Which – when restricted – is similar to the Bierman lattice. This extended lattice could provide sharing, strictness and absence information. A language, static semantics and dynamic semantics are all provided. The static semantics do not support polymorphism or polyvariance. However, recursive lets, data types, subtyping and subeffecting are supported. Another interesting point is that demand and actual use are separated in this type system. Since this is only a (very detailed) concept, no implementation or proofs are provided.

Gedell et al. [9] study the effects of polymorphism/polyvariance, subtyping/subeffecting, whole program optimization, data type annotations and existing optimizations on sharing analysis. They have an implementation of a sharing analysis for GHC where they can enable/disable any of these features. The options they consider for the different features are:

- Polymorphism/polyvariance: polymorphism with (i) monovariance; (ii) polyvariance, with type schemes where the usage variables may not be constrained; (iii) monovariant recursion/constrained polyvariance; or (iv) polyvariant recursion.

- Subtyping/subeffecting: subeffecting, either on or off.
- Algebraic data types: given a data type definition **data** $T\ \overline{\alpha} = K_1\ \overline{\tau_1} \mid \cdots \mid K_n\ \overline{\tau_n}$, annotate each type at the right-hand with a fresh usage variable and store it in $\overline{u}$. Then alter the definition of T to: **data** $T\ \overline{u}\ \overline{\alpha} = K_1\ \overline{\tau_1'} \mid \cdots \mid K_n\ \overline{\tau_n'}$ and replace recursive occurrences of T with $T\ \overline{u}$. Mutually recursive data types need to be annotated simultaneously. This process is very simple, however, it can lead to a huge number of usage variables.

  The different approaches to annotating algebraic data types are thus defined by how they limit the number of variables. The approaches are: (i) no limitations; (ii) limit the number of fresh variables per data type, reuse the variables by cycling through them (used limits where 100, 10 and 1); or (iii) assign ω to each type (so no variables at all).
- Whole program optimization: either on or off.
- Existing optimizations: either on or off.

They did not test all combinations of all the options, instead they chose some sensible configurations. Results showed that with each extra enabled feature, and the higher the precision, the performance increase lowered. Which is expected, since many features overlap. Most important results where: (i) data type annotations barely matter if existing optimizations are used; (ii) whole program optimization has a large impact on runtime and memory consumpton; and (iii) results – even for the most precise analysis – are pretty poor.

De Vries et al. [7] implement uniqueness analysis assuming there is a sharing analysis available. It uses the sharing information everywhere, but places some extra constraints to deal with the problems specified in Section 2.4.2: (i) In the abstraction rule: functions have an extra annotation, whenever a function has access to a unique parameter this is set to unique. (ii) In the application rule: the usual annotation on the function may not exceed the extra annotation of the function. This way it is possible to keep subtyping and subeffecting enabled: whenever it is applied to a function on which it shouldn't, it is impossible to use it thanks to the extra constraint in the application rule.

The first paper with an analysis that can be instantiated (with a single parameter) to two different analyses is written by Hage et al. [14]. The language is polymorphic, polyvariant and includes subeffecting. The parameter specifies how the subeffecting rule works, which is one of the only differences between the analyses (see Section 2.4.5). However, the other differences are intentionally ignored. This can lead to problems, as De Vries et al. [8] later point out. Subtyping is also intentionally left out, since it is not necessary when types are generic enough. They also felt it would make the analysis overly complex. The paper also includes dynamic semantics, which are used to provide a theorem for soundness.

De Vries et al. [8] write a new paper that reimplements their previous work – De Vries et al. [7] – in a more elegant way. Uniqueness attributes are now special type constructors, constraints are removed and encoded as boolean expressions and subtyping has been made obsolete. This enabled the authors to delete the extra annotation on functions that was introduced in their previous work. Instead of marking something unique when subtyping could lift it to non-unique, the annotation is left a variable. However, when something is necessarily unique – like functions with access to unique parameters – it is marked unique. The authors have a formal proof of Soundness and an implementation for the language "Morrow".

Hage and Holdermans [13] present a uniqueness analysis, based on the earlier work by Hage et al. [14]. However, usage analysis has been removed and the problems with uniqueness analysis w.r.t. referential transparency have been fixed using the method by De Vries et al. [7] (an extra annotation on functions). The analysis is polymorphic, polyvariant and includes subeffecting. A new language construct is added that allows the programmer to perform destructive updates on heap cells whenever it is used uniquely. Special care is taken to ensure that heap cells are of sufficient size, since different constructors have different sizes. The paper also includes a soundness theorem and a property that states that the updates do not change the meaning of the program or negatively impact the space behaviour.

Holdermans and Hage [17] introduce a strictness analysis. The language is extended with booleans, integers and most importantly a special "strict application" construct that allows a programmer to force an argument to be evaluated. Languages like Haskell and Clean contain similar constructs, it is interesting to see how this influences a strictness analysis. The analysis is monomorphic, monovariant and contains subeffecting. Normally strictness analyses only keep track of whether a value is demanded or not. However, to improve precision this analysis also keeps track of whether or not a value will be applied to an argument. The authors call this "applicativeness". No dynamic semantics is included, however the paper does include a proof sketch for the correctness of the analysis. I have implemented a polymorphic and polyvariant version of this paper during the APA course (with Gabe Dijkstra).

Chapter 10

Conclusion and Further Research

The main contribution of this thesis is extending the type system from [35] with polymorphism/polyvariance and adding the option for uniqueness analysis. There are two important things to note about uniqueness analysis:

- You can't simply "swap" the subtyping/subeffecting operator used in sharing analysis to get one for uniqueness analysis. This leads to the problem described in 2.4.2. We have solved this by using equality, effectively disabling subtyping/subeffecting for uniqueness analysis. This leads to a loss of precision in some cases [39].
- Analysis dependent subtyping/subeffecting should only be used for the argument of an application. The rest of the places in the type system that use subtyping/subeffecting use the same rule as sharing analysis uses. Using analysis dependent subtyping/subeffecting in these other places would actually make the type system more restrictive than necessary and thus less usable.

Although an implementation was made to show that it is possible to solve all the relatively complex constraints, it isn't very fast. It is possible to optimize this particular implementation. However, in the end it might still be quite slow because of all the complexity. Even though the results are very precise, the trade-off between running the analysis (and thus increased compile time) and the run-time improvement the results might give should be considered when implementing this in an actual compiler.

We have not written a lot about deriving a annotated datatype definition from an unannotated one. Since there are relatively few papers that talk about datatypes at all, this might still be something interesting to study.

Section 4 provided an algorithm to solve the constraints. There is no guarantee that it is correct (we suspect it is though). To prove this correct it would be necessary to implement it in Coq (or another theorem prover) and verify that it works as intended.

Section 6 showed an elegant way to integrate heap recycling from [13] into the analysis. However, it also showed that only allowing variables as arguments to function and constructor application requires more **let**s, and thus more heap allocations. This invariant is quite convenient when defining a counting analysis and we have seen it used in a lot of papers. Further research is required to investigate if: (a) we should do away with this invariant entirely; or (b) we can keep the invariant, but solve the issue using additional analysis; or (c) we can keep the invariant, but solve the issue using some run-time tricks.

Bibliography

[1] E. Barendsen and S. Smetsers. Conventional and uniqueness typing in graph rewrite systems. In *Foundations of Software Technology and Theoretical Computer Science*, pages 41–51. Springer, 1993.

[2] E. Barendsen, S. Smetsers, et al. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

[3] P.N. Benton. *Strictness analysis of lazy functional programs.* PhD thesis, University of Cambridge, 1993.

[4] M. Coppo, F. Damiani, and P. Giannini. Strictness, totality, and non-standard-type inference. *Theoretical Computer Science*, 272(1):69–112, 2002.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[6] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[7] E. De Vries, R. Plasmeijer, and D. Abrahamson. Uniqueness typing redefined. *Implementation and Application of Functional Languages*, pages 181–198, 2007.

[8] E. De Vries, R. Plasmeijer, and D. Abrahamson. Uniqueness typing simplified. *Implementation and Application of Functional Languages*, pages 201–218, 2008.

[9] T. Gedell, J. Gustavsson, and J. Svenningsson. Polymorphism, subtyping, whole program analysis and accurate data types in usage analysis. *Programming Languages and Systems*, pages 200–216, 2006.

[10] K. Glynn, P. Stuckey, and M. Sulzmann. Effective strictness analysis with horn constraints. *Static Analysis*, pages 73–92, 2001.

[11] B. Goldberg. Detecting sharing of partial applications in functional programs. In *Functional Programming Languages and Computer Architecture*, pages 408–425. Springer, 1987.

[12] J. Gustavsson. *A type based sharing analysis for update avoidance and optimisation*, volume 34. ACM, 1998.

[13] J. Hage and S. Holdermans. Heap recycling for lazy languages. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 189–197. ACM, 2008.

[14] J. Hage, S. Holdermans, and A. Middelkoop. A generic usage analysis with subeffect qualifiers. In *ACM SIGPLAN Notices*, volume 42, pages 235–246. ACM, 2007.

[15] D. Harrington. Uniqueness logic. *Theoretical Computer Science*, 354(1): 24–41, 2006.

[16] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[17] S. Holdermans and J. Hage. Making stricterness more relevant. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 121–130. ACM, 2010.

[18] P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 351–363. ACM, 1986.

[19] N.D. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. 1994.

[20] S.L.P. Jones et al. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of functional programming*, 2(2):127–202, 1992.

[21] S.P. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[22] J. Launchbury, A. Gill, J. Hughes, S. Marlow, S.P. Jones, and P. Wadler. Avoiding unnecessary updates. *Functional Programming, Glasgow*, pages 144–153, 1992.

[23] S. Marlow. Update avoidance analysis by abstract interpretation. In *Glasgow Workshop on Functional Programming, Ayr, Springer Verlag Workshops in Computing Series*, 1993.

[24] A. Middelkoop. *Improved uniqueness typing for Haskell*. PhD thesis, Masters Thesis, INF/SCR-06-08, Universiteit Utrecht, 2006.

[25] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[26] T. Mogensen. Types for 0, 1 or many uses. *Implementation of Functional Languages*, pages 112–122, 1998.

[27] F. Nielson and H. Nielson. Type and effect systems. *Correct System Design*, pages 114–136, 1999.

[28] E. Nöcker. Strictness analysis using abstract reduction. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 255–265. ACM, 1993.

[29] R. Plasmeijer and M. van Eekelen. Concurrent clean language report. *High Level Software Tools BV and University of Nijmegen, version*, 1, 1998.

[30] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for gadts. *ACM Sigplan Notices*, 44(9):341–352, 2009.

[31] I. Sergey. Theory and practice of demand analysis in haskell, 2012.

[32] I. Sergey, S. P. Jones, and D. Vytiniotis. Higher-order cardinality analysis. 2013.

[33] D.N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 1–11. ACM, 1995.

[34] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.

[35] K. Wansbrough. *Simple polymorphic usage analysis*. PhD thesis, University of Cambridge, 2002.

[36] K. Wansbrough and S.P. Jones. Once upon a polymorphic type. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 15–28. ACM, 1999.

[37] K. Wansbrough and S.P. Jones. Simple usage polymorphism. In *Proceedings of the Third ACM SIGPLAN Workshop on Types in Compilation*, 2000.

[38] Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

[39] Guangyu Zhang. Binding-time analysis: Subtyping versus subeffecting, 2008.