The Power of Wobbly Types

A bachelor thesis by **Vincent Driessen**
Supervised by **Prof.dr.ir. M.J. Plasmeijer**
Radboud University Nijmegen, The Netherlands

Summer of 2006

Contents

List of Figures

Chapter 1

Introduction

FUNCTIONAL PROGRAMMING LANGUAGES are equipped with type mechanisms in all sorts—some more powerful than others. The reverse of the medal is that the more powerful a type system is, the more complex it becomes to understand and wield for its users. For exactly that reason, research is carried out in the field of finding the "optimum" in the area of tension between expressiveness and complexity.

Classical type systems can be extended with all kinds of constructions to put more and more detailed restrictions on the possible terms someone may denote in the target programming language. However, in practice, a simple looking extension of the type system can already lead to undecidability.

One example of an extension of a type system is the addition of so called *generalized algebraic data types* (GADT) to the language. The exact implementation of such an addition to the functional programming language **Haskell** is proposed and elaborated [9, 16, 22]. By now, the implementation is a fact. Although they may look simple at first sight, GADTs provide powerful expressiveness, but also have undecidability of the type inference algorithm as a consequence.

This is where *wobbly types* come in. Wobbly types are a construct that allow programmers in functional languages to annotate their code with type information on key places where the type inference algorithm would otherwise run into undecidability. Type inference will still be decidable for non-GADT types, as one might expect.

Although virtually any enhancement to existing type systems is worth studying, in this case, there might also be a practical use to it, because decidability is preserved.

Research question

The key question that I will try to answer in this paper is the following:

> *"Does the addition of wobbly types to a classic type system actually offer added value, or does the expression power remain the same?"*

Structure of this paper

This thesis will show that the addition of wobbly types is in fact a valuable one, by making a comparison between type systems with and without wobbly types, and further elaborate on that.

In Chapter 2, I will give a brief introduction to classical type systems and an explanation of terminology as it is used in this paper. Type inference will be explained and I will show its limitations. After that, in Chapter 3 we will pretty much directly dive into the GADT theory, explaining how they work and what the problems are that are caused with respect to type inference by their introduction to a type system. We will see in Chapter 4 that wobbly types are a practical solution to the type inference problems that come with the introduction of GADTs to a type system and we will. Finally, we will show the use of wobbly types and conclude with some words on practical usability.

Chapter 2

Classic Type Systems

THE KEY PURPOSE of a compiler is to accept or reject programs (more generally speaking, *terms*) and to translate this code to another language—typically a machine language understood by an operating system or a processor. The criteria for a compiler to accept or reject a term should depend on whether or not the term has conceptual or "real" meaning. Formally we say that term acceptance should be *sound* and *complete*.

Terms consist of sub terms following a defined grammar for the target language. Due to the coarseness that inherently follows from the syntactical structure of the grammar, 'incorrect' programs can get accepted while 'correct' programs are rejected at the same time. These limitations can be addressed, among other techniques, by adding types to terms. In this process, an effort is taken to extend the language with type information in order to narrow down the number of possible undesirable outputs a compiler produces when processing programs.

2.1 Type systems

The purpose of a type system is to be *statically* (i.e. at compile-time) denote as exact as possible what is legal and what is illegal. The key problem of this is how exact can you be without constructing an undecidable system.

Typing terms, whether explicitly or implicitly, can be best seen as putting constraints on the possible values of terms. In untyped languages, every term can be seen as a set of elements from Ω, the *universe of discourse*[1]. By giving a term a type, one can define a specific subset of Ω that specifies the exact values that a term can have. Because typing terms gives information on what elements of Ω may be valid inhabitants of the terms, typing is a restriction mechanism that allows the programmer to make implicit assumptions within a program. For example, when a programmer knows that a certain variable is of type **Int**, he or she knows that the only possible values of that variable are elements of $\mathbb{N}$.

In untyped languages, one has no knowledge at all about the possible values a certain term can have without looking at the program as a whole. Therefore, in theory, whenever the programmer uses the untyped variable as if it where $\in \mathbb{N}$, he or she should explicitly check for it first—and on *every occasion*. Not only is this tedious, programs can get tremendously verbose in that aspect. In practice, this causes programmers to often leave these verbose checks just plain out of their programs, simply because they "know" that the program fragments are correctly used within their own context. There's no need saying that this is evil by nature since program fragment semantics should ideally not depend on their context.

In order to sketch the context of this paper, it is important to have a look at how functional languages

[1]The universe of discourse Ω is the set of all objects presumed or hypothesised to exist for some specific purpose. With a little imagination, within the context of this paper, Ω can be seen as the set of all terms that can be denoted.

work. First of all, all programming languages are constructed to let a human being "talk" to a machine. Because a machine handles instructions on a much more basic level than how humans reason, language constructs are designed with an eye on human mental models and reasoning [5, 15].

The formal mathematical language of the λ-calculus forms the basic for every functional language designer. Although several variants of the λ-calculus exist, all of them have one thing in common: their grammar and language rules are simple and minimalistic. This makes the language a great intermediate language, easily translatable into a machine language. The disadvantage is that programs in λ-calculus are pretty baroque and therefor not feasible for human beings to grasp or write.

This makes practically every functional language in essence nothing more than a huge framework of syntactic sugar, translating its programs to some variant of the λ-calculus. All language constructs and complexity should therefor preferably be implemented in the higher-order language layers.

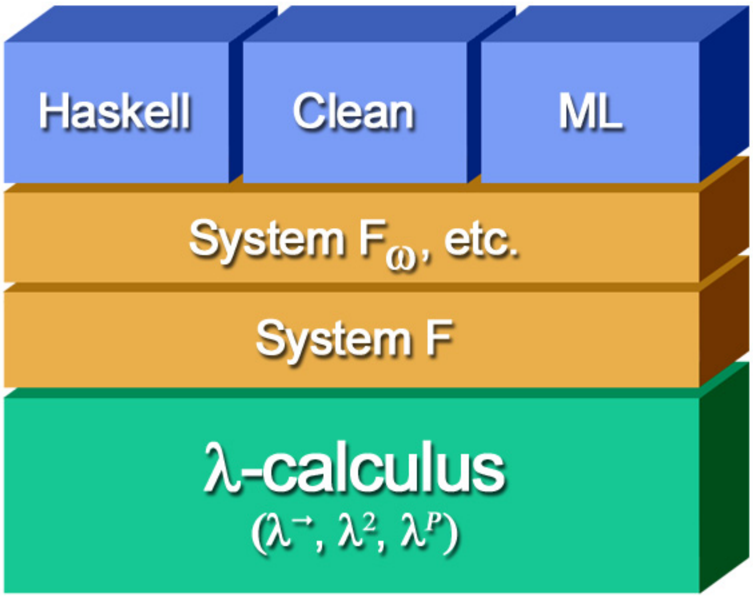

Figure 2.1: The pyramid of functional programming languages

2.2 Typing terms

Typing terms comes down to appointing subsets of Ω to them and thereby letting meta-information be carried with them on what their possible contents are. A term t of type σ will be denoted as $t : \sigma$ from now on. This tells you that regardless what term t is, it will only hold values $\in \sigma$, where, of course, $\sigma \subseteq \Omega$. Most programming languages specify the type relation with a double colon: $t :: \sigma$. We will use this notation in this paper when it concerns language constructs.

Definition 1. A type is a non-empty subset of Ω.

Well-known types are $\mathbb{N}$, the set of all natural numbers, often represented[2] as **Int** in implementations, or **String**, which holds all possible string values. Of course, types don't necessarily have to be infinite: the **Bool** type is an excellent example of a finite set—it only holds the values `True` and `False`. The smallest type is **Unit**, which holds only one value: :: **Unit** $= \{$`Unit`$\}$.

Property 1 (Unicity)**.** These sets, as they are used in functional programming languages like Haskell or Clean, will not overlap. Each term is of exactly one type. This makes it impossible for the term 1

[2]Of course, $\mathbb{N}$ is an infinite collection, whereas **Int** is finite due to machine limitations.

to be of both the types **Int** and **Real**, for example. The corresponding term for denoting the number 1 of type **Real** is `1.0`. It is worth noting that this concession is made for practical use, in order to avoid numerous decidability problems and to simplify the implementation of the unification algorithm. By doing so, Ω actually gets partitioned and for every term there is a mapping the other way around, which tells us of which type the term is.

Programs follow the same pattern—they consist of terms representing both functions and (typed) data. Functional languages have the property that they easily support the extension of functions to the program, but at the same time data types are hard to extend. This is because all data members are kept together in one place, where it is defined. Object oriented languages have the reverse problem—they are good in extending data (think of inheritance), but functions are hard to add [11].

In the remainder of this paper, the focus will lie on data types only. We will take a look at how data types are constructed and used in functions.

2.3 Data Types

2.3.1 Algebraic Data Types

The most basic sort of types in functional languages are the so called algebraic data types. Algebraic data types are types of which each value is data from other data types, that are "wrapped" in one of the constructors of the data type, by way of the parameters of the data constructors. The foremost way to look at data constructors of algebraic data types is that they are functions that can construct data (actually constants), but they cannot be rewritten as such, because there is no rewrite rule for it. They cannot be simplified further. Instead, they are merely operated upon by unwrapping the terms, using pattern matching.

Typical algebraic data types are enumerations, like:

```
:: DayOfWeek = Monday | Tuesday | Wednesday | Thursday
             | Friday | Saturday | Sunday
```

Note that where | is written, you can as well read $\cup$, which might be more consistent with the idea of types as mathematical sets.

On the left hand of the definition, the *type constructor* is given which, at this time, is just the name of the type. It is denoted in a bold font, such as **DayOfWeek**. On the right side, all possible *data constructors* are listed for the type, separated by |'s.

In this example, the seven data constructors (Monday, . . . , Sunday) have no parameters and are actually basic constants that are used for pattern matching purposes, for example:

```
IsWeekend :: DayOfWeek  →  Boolean
IsWeekend    Saturday   =  true
IsWeekend    Sunday     =  true
IsWeekend    _          =  false
```

Note that data constructors may be used in only one type, in order to satisfy unicity (property 1). This makes it possible to infer a type just by looking at the name of the constructor.

2.3.2 Recursive types

Other well-known types are recursive types, which are defined in terms of themselves, meaning that their data constructors have parameters that are of the type that is currently being defined. The most common examples of recursive types are **List**s and **Tree**s.

:: **List** = Nil
  | Cons Ω **List**

Binary **Tree**s are defined as follows. (n-ary **Tree**s can be trivially derived from this definition.)

:: **Tree** = Leaf
  | Node Ω **Tree Tree**

But not only direct recursion is allowed. Recursion may be indirect, too, like in the example below where mutual recursion is the case. Here, an **Object** may be either constructed of an Egg (which is just a constant) or a Surprise, in which case the surprise will contain a **Box**. The **Box**, for example, may be Empty or contain another **Object**.

:: **Object** = Egg
  | Surprise **Box**

:: **Box** = Empty
  | Filled **Object**

2.3.3 Polymorphic types

In order to narrow down the possible **Tree**s or **List**s that can be denoted, the types can be parameterized as follows:

:: **Tree** α β = Leaf α
  | Node β (**Tree** α β) (**Tree** α β)

This restricts **Tree** terms so that concrete pieces of data that are put in the tree when constructing it are of the same type—all leaf elements are of type α and all node elements are of type β.

The rationale behind this seemingly simple, yet powerful restriction is that there is an implicit equality constraint within the type definition, since the α on the left side of the $=$-sign is the same as the α on the right side of it. Furthermore, within the second line of the definition, there is an implicit equality between the three β's. Due to transitiveness, all types in a tree are trivially of an equal type.

Note that, whatever concrete type α or β is, the definition works. This is called *polymorphism*. For example, one could parametrise **Tree** with the concrete type **Int**, to generate a tree of integers, but also, using the same definition of **Tree**, to generate a tree containing any kind of other data. This is the uniform substitution rule: α can be of *any* type, but it has to be substituted by the same type in all occurrences. Actually, this is a shorthand for:

:: $\forall\alpha.\ \forall\beta.$ **Tree** α β = Leaf α
  | Node β (**Tree** α β) (**Tree** α β)

The order of parameters is, of course, not important. While it might not seem to make any sense to swap the arguments in the data constructors, the programmer is of course free to do so. In which case parameter types of the **Tree**s are swapped at the left-hand side of each Node.

:: **Tree** α β = ... | Node β (**Tree** $\underline{\beta\ \alpha}$) (**Tree** α β)

NOTE:
One could still "simulate" an untyped **Tree** type, by allowing for existential quantification of types:

$$\begin{aligned} ::\ \textbf{Tree}\ &= \text{Leaf}\ \exists\alpha.\alpha \\ &\mid \text{Node}\ \exists\alpha.\alpha\ \textbf{Tree}\ \textbf{Tree} \end{aligned}$$

In which $\exists\alpha.\alpha$ could be looked at as a synonym for Ω.

The main reason why such a definition is not useful is because it does not do any justice to the reason why types are introduced in the first place. However, there *are* useful constructs possible with the existential quantor, for example it could store tuples of a certain type α and a function on that domain in a Node:

$$\begin{aligned} ::\ \textbf{Tree}\ &= \text{Leaf}\ \exists\alpha.\alpha \\ &\mid \text{Node}\ \exists\alpha.\,(\alpha, \alpha \to \alpha)\ \ \textbf{Tree}\ \textbf{Tree} \end{aligned}$$

This is in fact a valid and meaningful restriction, because of the implicit equality of the type α and the α's in $\alpha \to \alpha$, within the scope of the quantor.

2.4 Type inference

Type inference is the process of deriving the types of a term of which the types are not explicitly annotated by the programmer. For example, consider the following program fragment, in which a binary tree data type **Tree** is defined and used to sum up all the values of the leafs of the tree: Clearly,

$$\begin{aligned} ::\ \textbf{Tree}\ \alpha\ \beta &= \text{Leaf}\ \alpha \\ &\mid \text{Node}\ \beta\ (\textbf{Tree}\ \alpha\ \beta)\ (\textbf{Tree}\ \alpha\ \beta) \\ \text{sumleafs}\ (\text{Leaf}\ x) &= x \\ \text{sumleafs}\ (\text{Node}\ \_\ t_1\ t_2) &= \text{sumleafs}\ t_1 + \text{sumleafs}\ t_2 \end{aligned}$$

Figure 2.2: The algebraic data type definition and the sumleafs function

in the example program fragment from Figure 2.2, we want to derive that:

$$\text{sumleafs}\ ::\ (\textbf{Tree}\ \textbf{Int}\ \alpha) \to \textbf{Int}$$

This is pretty obvious to see, because the result of the second sumleafs function body line is an **Int**. The function operates on inputs of type **Tree Int** α. In this case, we can derive the most general type for **Tree**, using polymorphic variable α, because the second type parameter (α) is never used in the calculation of sum, so any type is allowed here.

2.4.1 An example

To get a basic understanding of how the algorithm works, lets have a look at the example above. If you are not interested, you can skip to the next section on the formalisation details directly. First, we consider the line:

$$\text{sumleafs}\ (\text{Leaf}\ x) = x$$

Apparently, sumleafs is a function, due to the fact that is has a parameter. Actually, it is function from "something to something". If we formalise the "somethings", we read:

$$\text{sumleafs} :: \alpha \to \beta \tag{2.1}$$

The α and β are unknown by know, so we need to generate equations that specify what α and β are. From the first body line, we can derive the following equations:

$$\alpha = \textbf{Tree}\ \gamma\ \delta \tag{2.2}$$
$$\beta = \gamma \tag{2.3}$$

From 2.1, we see that α is the parameter(s) to the function. From the first line of the body definition, we see that there is only one parameter, of type "some **Tree**". Hence, we find 2.2.

Furthermore, we see that there is an implicit relation between the two x's, so the type of the output of the sumleafs function (β) is equal to the type of the parameter of the Leaf data constructor, for a tree of type **Tree** γ δ, i.e. it equals γ. This is shown in 2.3.

Next, we arrive at the second line of the body definition. Again, we conclude that the type sumleafs is a function type. We should currently not be smart already and naively use new type variables, so we find:

$$\text{sumleafs} :: \varepsilon \rightarrow \zeta \tag{2.4}$$

Let's recall the second line of the function body:

$$\text{sumleafs (Node \_}\ t_1\ t_2) = \text{sumleafs}\ t_1 + \text{sumleafs}\ t_2$$

Also, again, we find that the first parameter of the function is a **Tree**, because we know that a node is a tree, because of the unicity property (see page 8). However, we know nothing about its parameters, so we suppose we can have any kind of tree. This is expressed in equation 2.6. Furthermore, we see that the output type of the function is equal to the output type of the $+$-operator, so it is an **Int**. We suppose $+ ::$ **Int** $\rightarrow$ **Int** $\rightarrow$ **Int**, and ignore overloading of operators for now. Even more so, since an application of the $+$-operator requires two **Int** parameters, we have for both parameters:

$$\underbrace{\underbrace{\overbrace{\text{sumleafs}}^{::?}\ \overbrace{t_1}^{::\varepsilon_1}}_{::\textbf{Int}} + \underbrace{\overbrace{\text{sumleafs}}^{::?}\ \overbrace{t_2}^{::\varepsilon_2}}_{::\textbf{Int}}}_{::\textbf{Int}} \tag{2.5}$$

This yields equations 2.6 and 2.7:

$$\varepsilon_i = \textbf{Tree}\ \eta_i\ \theta_i \tag{2.6}$$
$$\zeta_i = \textbf{Int} \tag{2.7}$$

As you can see, all of this is very tedious and error-prone when done by hand. Certainly, naïveté comes in quite useful, but the result is a pretty big number of equations to solve, most of which are pretty trivial:

$$\begin{aligned}
\alpha \rightarrow \beta &= \varepsilon \rightarrow \zeta \\
\alpha &= \textbf{Tree}\ \gamma\ \delta \\
\beta &= \gamma \\
\varepsilon &= \varepsilon_1 = \varepsilon_2 \\
\eta_1 &= \eta_2 \\
\theta_1 &= \theta_2 \\
\varepsilon_1 &= \textbf{Tree}\ \eta_1\ \theta_1 \\
\varepsilon_2 &= \textbf{Tree}\ \eta_2\ \theta_2 \\
\zeta &= \zeta_1 = \zeta_2 \\
\zeta_1 &= \textbf{Int} \\
\zeta_2 &= \textbf{Int}
\end{aligned}$$

After some calculation, we find:

$$\textbf{Tree Int } \delta \to \textbf{Int}$$

Which is exactly the type of the sumleafs function. Since δ is the only variable that cannot be solved further, we know that this parameter does not matter and we can derive the most general (polymorphic) type for sumleafs.

2.5 The Hindley-Milner algorithm

Back in 1958, Haskell B. Curry and Robert Feys devised the origin of the type inference algorithm for the simply typed lambda calculus, $\lambda^{\to}$. In $\lambda^{\to}$, initially the types of terms are basically unknown and using an algorithm it is tried to derive a type for the terms. Curry's algorithm always finds the most general type for any given term. This was proved by Roger Hindley in 1969 and Robin Milner in 1978 (both independently).

The basic structure of the algorithm consists of two steps. First, based on the terms at hand, a set of equations that need to hold is derived. This set of equations can be used to *check* the types—this is what a type checker does. Secondly, the set of equations is tried to be solved. "Solving" in this context means finding a *substitution*—a function S mapping variables to variables. In contrast to type *checking*, this is type *inference*.

2.5.1 Definitions

In order to formalise the type inference algorithm, we will start off by making an abstraction of the target language, by using the polymorphic λ-calculus, λ^P. We define the following language grammar:

$$\begin{aligned} \mu &= \textbf{Int} \mid \textbf{Bool} \mid \text{etc.} \\ \tau &= \mu \mid \alpha \mid \tau \to \tau \\ \sigma &= \tau \mid \forall\alpha.\sigma \mid (\forall\alpha.\sigma)\sigma' \end{aligned}$$

By this definition, τ is an unquantified type, meaning it can only be an atomic type μ, a type variable α or a compound function. For example, τ can be **Int**, or **Int** $\to$ **Boolean**, or even (**Int** $\to$ **Boolean**) $\to$ **Int** $\to$ **Boolean**.

Furthermore, we allow quantification over type variables (e.g. polymorphism) on the outermost level, which is expressed in the definition of σ. In the remainder of this thesis, we will use σ where we mean a "possibly quantified (polymorphic) type", constructed by type abstraction or application, and τ if we know for sure that that variable is a monomorphic type.

The free type variables in these types are defined by the FTV function:

$$\begin{aligned} \mathsf{FTV}(\mu) &= \emptyset \\ \mathsf{FTV}(\alpha) &= \{\alpha\} \\ \mathsf{FTV}(\tau \to \tau') &= \mathsf{FTV}(\tau) \cup \mathsf{FTV}(\tau') \\ \mathsf{FTV}(\forall\alpha.\sigma) &= \mathsf{FTV}(\sigma) \setminus \{\alpha\} \\ \mathsf{FTV}((\forall\alpha.\sigma)\sigma') &= \mathsf{FTV}(\sigma[\sigma'/\alpha]) \end{aligned}$$

Contexts

Contexts are used during compilation to keep track of the currently resolved program fragments and their types. In essence, the context is nothing more than a mapping of terms to types or type variables. Typing contexts are defined in the usual way, by the grammar below:

$$\Gamma \quad = \quad \emptyset \mid \Gamma,\ x : \sigma \mid \Gamma,\ T : \star$$

Next, the free type variables (FTV) of a context are defined as the union of the free type variables in all bindings of that context:

$$\begin{array}{rcl}\mathsf{FTV}\,(\emptyset) & = & \emptyset \\ \mathsf{FTV}\,(\Gamma,\ x:\sigma) & = & \mathsf{FTV}\,(\Gamma)\cup\mathsf{FTV}\,(\sigma) \\ \mathsf{FTV}\,(\Gamma,\ T:\star) & = & \mathsf{FTV}\,(\Gamma)\end{array}$$

Substitutions

A substitution S is a function that replaces type variables by types. For example, when we write $S = [\beta/\alpha]$ and $S\tau$, we mean "β replaces α in term τ". Substitutions may be concatenated, by using the $\circ$ operator:

$$(S_1 \circ S_2 \circ \cdots \circ S_n)\tau$$

Substitution over a context is defined straightforward, applying the substitution to each variable in the context:

$$\begin{array}{rcl}S(\emptyset) & = & \emptyset \\ S(\Gamma,\ x:\sigma) & = & S(\Gamma),\ x:S(\sigma) \\ S(\Gamma,\ T:\star) & = & S(\Gamma)\end{array}$$

Furthermore, we define an "is more general" operator, $\sqsubseteq$. We could then say, for example, that σ is more general than σ' by $\sigma \sqsubseteq \sigma'$. A typical example would be:

$$\forall\alpha.\alpha\to\alpha \sqsubseteq \forall\beta.(\beta\to\beta)\to\beta\to\beta \sqsubseteq (\gamma\to\gamma)\to\gamma\to\gamma$$

The formal definition of this operator can be found in [24]. The important intuition to have right now is that type σ is more general than σ' iff there exists a uniform substitution S for type variables in σ, such that $S\sigma = \sigma'$, or there has been a type application.

2.5.2 Typing rules

The formal typing rules that specify how programs are correctly constructed are shown in Figure 2.3. In these rules, the barred vector notation of for example $\overline{\alpha}$ means $\alpha_1, \ldots, \alpha_n$, were n is of no relevance.

There are several language constructs. Above in the figure are all the rules for the construction of types. The next group of rules tells how expressions are constructed. Finally, there are rules for declarations of new types and for the construction of a whole program that wraps all the rules together.

Inferring the types

In order to infer the types of our example program (Figure 2.2), we must first translate the code into more basic λ-calculus, so that we can read the program fragment in the same language as where the typing rules are defined in.

The translated example program is illustrated in Figure 2.4. In order to achieve the translation of the recursion in the function, we have to use the fixed point combinator **Y**, but we will not dive into the implementation details of that right now. The basic idea of the fixed point calculations is that we encode one step of the recursion (i.e. the function sumleafs_step), while the **Y**-combinator takes care of the recursive invocation of the function[3].

[3]The **Y**-combinator is typically defined as λ f.(λ x.f (x x)) (λ x.f (x x)).

$$\boxed{\Gamma \vdash_t \sigma : \star}$$

The $\vdash_t$ rules define how types are constructed. Note that the symbol $\star$ means "kind", which is the collection of all types. A type is either a type by way of language construct or a user-defined type, δ.

$$\frac{}{\Gamma \vdash_t \sigma : \star}\ \text{(Atom)} \qquad \frac{\delta : \star \in \Gamma}{\Gamma \vdash_t \delta : \star}\ \text{(Cust)}$$

$$\boxed{\Gamma \vdash_e e : \sigma}$$

The $\vdash_e$ rules define how expressions are evaluated.

$$\frac{e : \sigma \in \Gamma}{\Gamma \vdash_e e : \sigma}\ \text{(Var)} \qquad \frac{\tau = \Gamma(x) \quad \overline{\Gamma \vdash_e p : \tau} \quad \overline{\Gamma \vdash_e e : \sigma}}{\Gamma \vdash_e \textbf{case}\ x\ \textbf{of}\ \overline{p \to e} : \sigma}\ \text{(Case)}$$

$$\frac{\Gamma \vdash_t \tau : \star \quad \Gamma, x : \tau \vdash_e e : \tau'}{\Gamma \vdash_e \lambda x : \tau . e : \tau \to \tau'}\ \text{(Abs)} \qquad \frac{\Gamma \vdash_e e : \tau' \to \tau \quad \Gamma \vdash_e e' : \tau'}{\Gamma \vdash_e ee' : \tau}\ \text{(App)}$$

$$\boxed{\Gamma \vdash_d decl : \Gamma'}$$

The $\vdash_d$ rules define how data declarations are constructed. The result of this rule is a new environment Γ', which contains the declarations. Here, T indicates a type and C a data constructor.

$$\frac{\overline{\Gamma, T\ \overline{\alpha} : \star \vdash_t \sigma : \star}}{\Gamma \vdash_d\ ::\ T\ \overline{\alpha} = \overline{C\ \overline{\sigma}} : \left\{T\ \overline{\alpha} : \star,\ \overline{C\ \overline{\sigma} : T\ \overline{\alpha}}\right\}}\ \text{(Data)}$$

$$\boxed{\Gamma_0 \vdash_p P : \sigma}$$

The $\vdash_p$ rules define how programs may be constructed. A program is a series of type and function declarations, followed by an expression. The program rule is the only rule that does not require a context.

$$\frac{\overline{\Gamma_0 \vdash_d decl : \Gamma_d} \qquad \Gamma = \overline{\Gamma_d} \qquad \Gamma \text{ is consistent} \qquad \Gamma \vdash_e e : \sigma}{\Gamma_0 \vdash_p \overline{decl};\ e : \sigma}\ \text{(Program)}$$

Figure 2.3: Type inference rules

Note, by the way, that the functions that we defined (i.e. sumleafs, sumleafs_step and **Y**) are nothing more than shorthand *names* that can be substituted by their function bodies.

Actually inferring the example program using these rules is way too verbose to print here. The essence would be that, using the Program rule, first all declarations get loaded into the environment, so:

$$\Gamma = \{\textbf{Tree}\ \alpha\ \beta : \star,\ \text{Node}\ \alpha : \textbf{Tree}\ \alpha\ \beta,\ \text{Leaf}\ \beta : \textbf{Tree}\ \alpha\ \beta\}$$

Then, using rules Abs and App, are used to infer the types of sumleafs. This is not trivial, because of the fixed point combinator. To infer the type of sumleafs_step, first there are two applications of the

$$
\begin{array}{ll}
\multicolumn{2}{l}{:: \textbf{Tree}\ \alpha\ \beta = \textsf{Leaf}\ \alpha} \\
\multicolumn{2}{l}{\qquad\qquad\quad \mid \textsf{Node}\ \beta\ (\textbf{Tree}\ \alpha\ \beta)\ (\textbf{Tree}\ \alpha\ \beta)} \\
\textsf{sumleafs\_step} & = \lambda f.\lambda e.\textbf{case}\ e\ \textbf{of} \\
 & \qquad (\textsf{Leaf}\ x) \rightarrow x \\
 & \qquad (\textsf{Node}\ \_\ t_1\ t_2) \rightarrow \textsf{plus}\ (f\ t_1)\ (f\ t_2) \\
\textsf{sumleafs} & = \lambda e.\mathbf{Y}\ \textsf{sumleafs\_step}\ e
\end{array}
$$

Figure 2.4: The example program translated to the intermediate language

Abs rule, after which we can use the rule Case to infer the type of the **case**-construct. Then, we have found the type of the function body, and finally from the program as a whole.

Chapter 3

Generalized abstract data types (GADTs)

As we have seen in the first chapter, data type constructors can be regarded as regular functions, with the only difference being that they are merely atomic terms that can be utilised in using pattern matching and therefore cannot be rewritten as such. In this chapter, we are going to take a look at an extension of the type system that allows for more specific term construction.

Suppose we have this polymorphic, recursive data type **Term**, as defined below. Its purpose is to design a grammar for a small language that allows for expressing certain simple terms. The reason why the polymorphic variable α is used everywhere in this definition, is that we need it in the base case for the expression induction, Const. This means that any expression can be parsed into a tree whose leafs are all constants of a certain type α.

(The **Term** data type will be the running example for the rest of this paper.)

$$\begin{aligned} :: \textbf{Term}\ \alpha = {} & \text{Const}\ \alpha \\ \mid {} & \text{Inc}\ (\textbf{Term Int}) \\ \mid {} & \text{IsZero}\ (\textbf{Term Int}) \\ \mid {} & \text{If}\ (\textbf{Term Bool})\ (\textbf{Term}\ \alpha)\ (\textbf{Term}\ \alpha) \end{aligned}$$

This definition introduces a new structured data type (**Term** α) and a set of constructor functions.

An example term that is allowed to be expressed using this grammar is:

If (IsZero (Const 13)) (Const 1) (Const 2) :: **Term Int**

However, the type signature of **Term** is not strictly enough, so we may as well write terms which have no semantic meaning, but are perfectly correct according to the grammar:

Inc (IsZero (Const 13)) :: **Term Int**

Clearly, this is a problem with expressiveness of types in the programming language. The reason why this expression problem arises, becomes clear when we explicitly write out the data constructors:

$$\begin{aligned} &\text{Const} :: \forall\alpha\ .\ \alpha \rightarrow \underline{\textbf{Term}\ \alpha} \\ &\text{Inc} :: \forall\alpha\ .\ (\textbf{Term Int}) \rightarrow \underline{\textbf{Term}\ \alpha} \\ &\text{IsZero} :: \forall\alpha\ .\ (\textbf{Term Int}) \rightarrow \underline{\textbf{Term}\ \alpha} \\ &\text{If} :: \forall\alpha\ .\ (\textbf{Term Bool})\ (\textbf{Term}\ \alpha)\ (\textbf{Term}\ \alpha) \rightarrow \underline{\textbf{Term}\ \alpha} \end{aligned}$$

The data constructors return phantom types [6]. The range of every constructor is a phantom instance of the type being defined, **Term** α. The restriction here is that we cannot take control over the output types. Actually, what is missing is that it is not possible to say that IsZero always returns a (**Term Bool**). Generalized algebraic data types remove exactly this restriction.

3.1 What is a GADT?

What we actually would like to express in the example above, is that the data constructors will accept parameters of any type, but that they can possibly return a term of an explicit type. Note that this is different from polymorphism! In Haskell, for example, GADTs can be utilised to construct the following non-trivial data type definition: GADTs provide syntactic and semantic support to express *return type*

$$\begin{aligned} :: \textbf{Term}\ \alpha\ &= \text{Const}\ :: \forall\alpha\ .\ \alpha \rightarrow \textbf{Term}\ \alpha \\ &\mid \text{Inc}\ :: \forall\alpha\ .\ (\textbf{Term Int}) \rightarrow \textbf{Term}\ \underline{\textbf{Int}} \\ &\mid \text{IsZero}\ :: \forall\alpha\ .\ (\textbf{Term Int}) \rightarrow \textbf{Term}\ \underline{\textbf{Bool}} \\ &\mid \text{If}\ :: \forall\alpha\ .\ (\textbf{Term Bool})\ (\textbf{Term}\ \alpha)\ (\textbf{Term}\ \alpha) \rightarrow \textbf{Term}\ \alpha \end{aligned}$$

Figure 3.1: Return type refinement

refinement. In order to see how this works, let's turn to the example using no GADTs, and build an abstract syntax tree (AST) for it (see Figure 3.2). The @-sign in the ASTs indicate an application of the right-hand node to the left-hand one. To show the contrast, we show how the same expression gets rejected in the case we use GADTs with return type refinement (Figure 3.3).

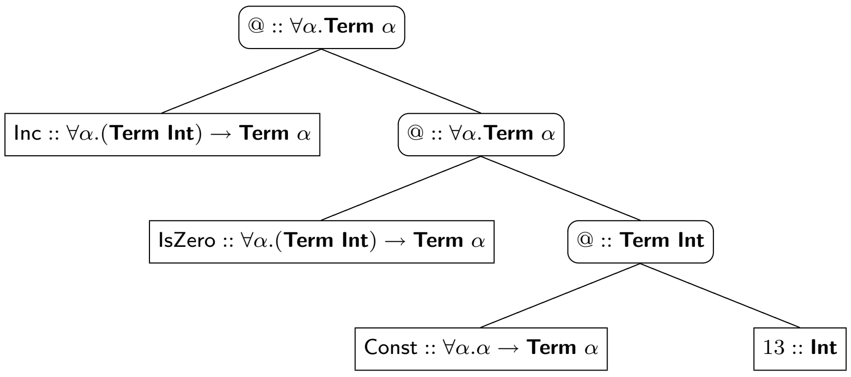

Figure 3.2: Abstract syntax tree *without* GADTs

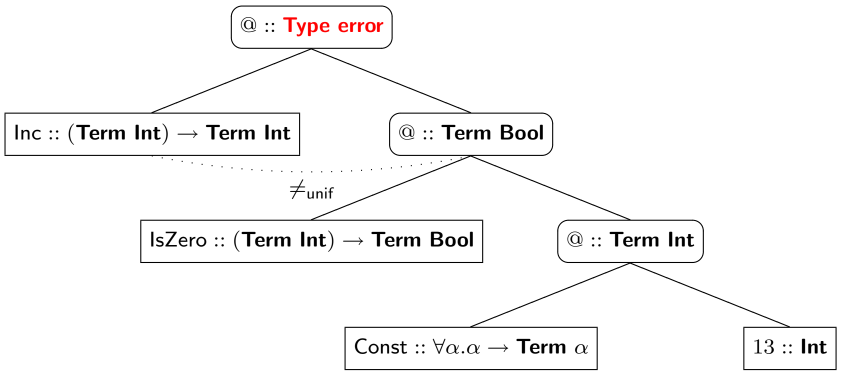

Figure 3.3: Abstract syntax tree *with* GADTs

Thus, we can actively restrict the terms that are possible to denote in our target language. What we are doing here is actually very odd—we have some semantics in our heads about how the evaluator should work and are able to express that at compile time. $\boxed{\text{This is an important new concept!}}$

Compared to normal abstract data types, GADTs allow for a more precise grammar. Hence the name *generalized* abstract data types. As stated in Section 2.1, the purpose of a type system is to be as specific as possible statically about what is legal and illegal to write. GADTs allow for describing data structures more precisely at compile-time, so there is less checking involved at run-time and the code will be more elegant.

The fact that we have statically added some information is also visible in the function definition of the evaluation function (see Figure 3.4). It is clean, and actually becomes very straightforward.

$$\begin{aligned}
&\text{eval (Const } x) &&= x\\
&\text{eval (Inc } i) &&= \text{eval } i + 1\\
&\text{eval (IsZero } n) &&= \text{eval } n == 0\\
&\text{eval (If } b\ x\ y) &&= \text{if (eval } b) \text{ (eval } x) \text{ (eval } y)
\end{aligned}$$

Figure 3.4: The implementation of the eval function

In the example in Figure 3.4, the type of eval should be $\forall\alpha.\textbf{Term}\ \alpha \rightarrow \alpha$, but this is not possible to infer automatically.

3.2 Type inference with GADTs

To illustrate why type inference is undecidable in this scenario, let's build the AST for each line of the function body. Figures 3.5–3.7 show the ASTs for the first three lines. The AST for the If-term has been omitted, because it is verbose. The first three lines suffice as they already illustrate the problem. The maroon colored boxes show the inferred types for the eval function.

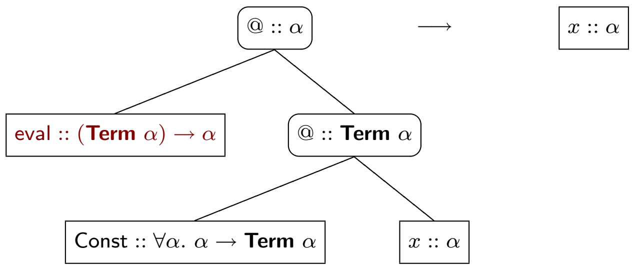

Figure 3.5: AST for: eval (Const x) = x

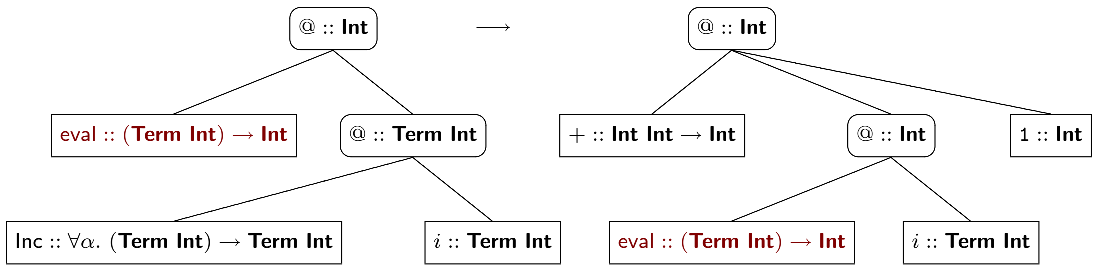

Figure 3.6: AST for: eval (Inc i) = eval i + 1

Looking at Figure 3.5, the type is inferred directly from the known types of the application (Const x), by way of unicity. Also, the right-hand side of this line is of type α, which is obvious by definition.

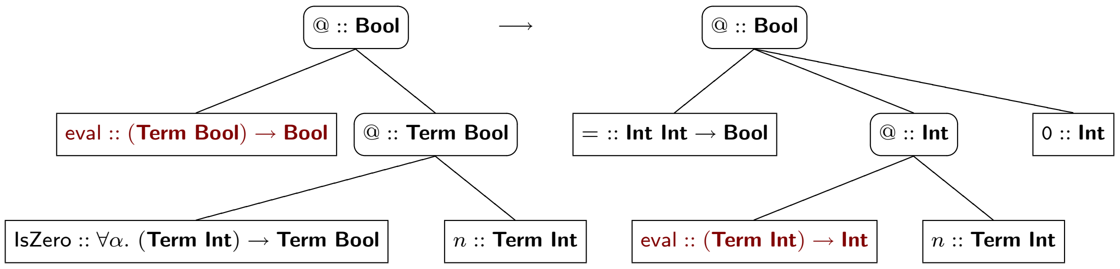

Figure 3.7: AST for: eval (IsZero n) = eval n == 0

However, in Figure 3.6, we deal with a more difficult case of inference. The type for the eval function in the left tree is inferred partly by unicity, so we know

$$\text{eval} :: (\textbf{Term Int}) \rightarrow \gamma$$

But we do not know what γ is, yet. In order to know this, we must first evaluate the right-hand side of this line. From the definition of the +-operator, we know that its two arguments and its result are of type **Int**, and because we know from the left tree that i :: **Term Int**, we may derive that

$$\text{eval} :: (\textbf{Term Int}) \rightarrow \textbf{Int}$$

As you can see, type inference is not straightforward anymore and we have to incorporate a smart (complex?) way of combining information from different contexts. Assuming that the types of the eval functions in the left and right trees have the same type, we are able to derive the type in the left tree.

However, looking at the third body line of eval, as shown in Figure 3.7, we see that this assumption is wrong! In the left context, eval is specifically instantiated with **Bool** types, while in the right context, it is instantiated with **Int**s.

Since every line of the function body renders a different type for eval, the type system gets confused and does not know how it should infer a general type for eval. The rules of the game have to change for this new type system. We will detail this in the next section.

3.3 Interline type unification

First of all, what remains the same is that the types of the left and right hand sides of the equality sign should be unifiable (left $=_{\text{unif}}$ right).

$$\begin{array}{lcl}
\boxed{\text{eval (Const x)}} & =_{\text{unif}} & \boxed{\text{x}} \\
\boxed{\text{eval (Inc i)}} & =_{\text{unif}} & \boxed{\text{eval i + 1}} \\
\boxed{\text{eval (IsZero n)}} & =_{\text{unif}} & \boxed{\text{eval n == 0}} \\
\boxed{\text{eval (If b x y)}} & =_{\text{unif}} & \boxed{\text{if (eval b) (eval x)\ \ (eval y)}}
\end{array}$$

Figure 3.8: Line-based unification rules remain the same

However, although there should be inferred a single, unambiguous type for the function that each line is unifiable with, the new rules of the game do not demand each line of the function body to be mutually unifiable with each other. The type of the function therefore should be of a more general type (remember the $\sqsubseteq$-operator) than each of his body lines individually.

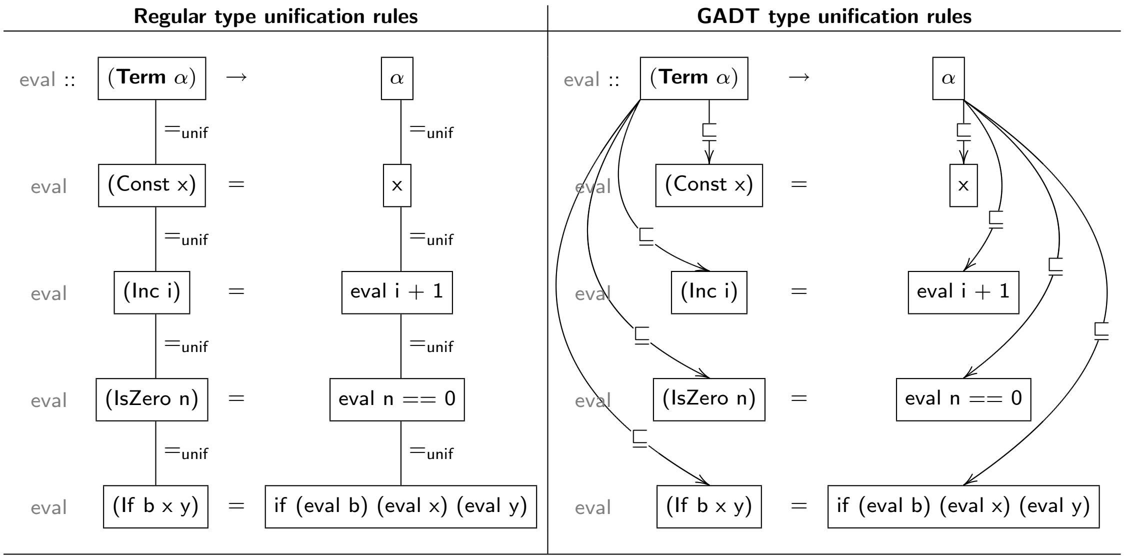
Regular type unification rules
GADT type unification rules
eval :: (Term α) → α
eval (Const x) = x
eval (Inc i) = eval i + 1
eval (IsZero n) = eval n == 0
eval (If b x y) = if (eval b) (eval x) (eval y)
=unif
⊑

Figure 3.9: Old and new interline type unification rules

Figure 3.9 shows how the rules change for interline dependencies. Each of the arrows without labels in this diagram expresses a regular unifiability relationship. The other, directed arrow means "is more general", to the definition of the ⊑-operator from Section 2.5.1.

The new rules for interline type unification require a new type inference algorithm. In classic type systems, the function body for eval would obviously be rejected immediately with a message that the type system "cannot unify **Int** with **Bool**" (respectively the right-hand sides of lines 2 and 3 of the function body). However, with GADTs, these terms are not obligated to unify mutually. Instead, there should both adhere to a single type which is more general—and exactly *that* type should be the type of the eval function's result.

The most crucial question in this new theory is which conclusions we may draw from the fact that line x is of type σ and line y is of type σ'. It is almost philosophical. Generally, we may draw three conclusions:

- **Reject.** Assume there is an irreconcilable type conflict and that it is not possible to infer a single type.
- **Be exact and minimalistic.** The exact type is actually the union of both individual types, so the type to infer is their union:
$$\forall_{\rho \in \{\sigma,\sigma'\}}.\rho$$
- **Over-approximate.** Assume the type may be anything and infer a polymorphic type:
$$\forall \alpha.\alpha$$

The first option is of course exactly the classic type system way of dealing with these inconsistencies. It follows immediately from the direct interline unification requirement.

If we want to support GADTs, from a theoretical perspective, the second option would be the best choice to make. We already know that both types apparently are valid, but we know nothing more. From this limited set of information, we can merely distill that the type to infer should be the union of its individual body parts' types. However, drawing this conclusion leads to numerous problems for language designers, so it is not very practical. Languages that implement these constructs require carrying proofs along with types, assuring safe content [2, 26].

Finally, there is over-approximation, which seems wrong by definition. Of course, type systems are essentially introduced to prevent these kind of conclusions! Allowing this reasoning, we may accept every program again, possibly drawing the conclusion that the program would be of type $\forall\alpha.\alpha$. Hard thinking required!

Chapter 4

Wobbly types

Unlike the previous chapter implied, over-approximation actually is not such a bad idea as it seems. It should however only be used in specific contexts, not in general reasoning. Programmers may have specific knowledge about their programs that the type system cannot know of. Therefore, when a programmer is certain that, within that context, it is safe to generalise the type, he or she may chose to do so. What is important with GADTs, is that the body lines can be *checked* to be of the correct type.

This over-approximation is achieved by annotating term types manually, using *wobbly variables*. A wobbly variable is a type variable, but its concrete type is not clear in general. In specific contexts, the type variable "refines" to specific types.

4.1 Manual annotation

What wobbly types all come down to is manual annotation. Wobbly types are not inferred, but are explicitly annotated by human programmers, that actually tell the compiler that the types may be generalized in a certain context. The running example of the evaluator from the previous chapter with its type annotated is shown in Figure 4.1.

$$\begin{array}{lll}
\text{eval} :: (\textbf{Term}\ \alpha) & \rightarrow \alpha \\
\text{eval} & (\text{Const}\ x) & = x \\
\text{eval} & (\text{Inc}\ i) & = \text{eval}\ i + 1 \\
\text{eval} & (\text{IsZero}\ n) & = \text{eval}\ n == 0 \\
\text{eval} & (\text{If}\ b\ x\ y) & = \text{if}\ (\text{eval}\ b)\ (\text{eval}\ x)\ (\text{eval}\ y)
\end{array}$$

Figure 4.1: An example annotation using wobbly types

Within the context of the second line of the body, the type system may know that the wobbly type α can only be **Int**, so we may accept the usage of the $+$-operator. Realise that this is very strange! Normally, when using polymorphic functions, we are not allowed to actually *use* the polymorphic data, but we can merely *pass them through*. Compare this to a typical polymorphic function like map (see Figure 4.2), and note how the actual data within the lists is not used concretely, but only passed around and rearranged. Nothing can be assumed about the types of the terms within the body of map, nor about the available operators. The only reason we may apply x to f, is because f itself is polymorphic again.

Only abstract (polymorphic) function applications are legal—so nothing concrete like the $+$-operator.

$$\begin{array}{lll} \mathsf{map} :: (\alpha \to \beta)\ (\mathbf{List}\ \alpha) & \to (\mathbf{List}\ \beta) \\ \mathsf{map}\quad f\ \mathsf{Nil} & = \mathsf{Nil} \\ \mathsf{map}\quad f\ (\mathsf{Cons}\ x\ xs) & = \mathsf{Cons}\ (f\ x)\ (\mathsf{map}\ f\ xs) \end{array}$$

Figure 4.2: Polymorphic variables can be merely passed around

When we would allow for concrete function applications, then we would immediately infer a non-polymorphic type (e.g. map :: $(\alpha \to \beta)$ (**List Int**) $\to$ (**List Int**)).

4.2 Type inference with wobbly types

Type inference with wobbly types can be made decidable again, by letting the programmer specify the blueprint that describes how each line of the function body must look like. The right side of Figure 4.1 shows exactly that. What is expressed by these annotations is that there is a relation between the α's in the annotation. Informally, you express that every term of some α will evaluate to α itself, but in each context the concrete instantiation of α may differ.

Knowing this, let's look back at the inference process that was shown in Figures 3.5–3.7. Recall that the troubles lay in both the second and the third line. The problem with the second line (Figure 3.6) was the inference eval's type in the left tree. Recall that the type system could basically go as far as inferring

$$\mathsf{eval} :: (\mathbf{Term\ Int}) \to \gamma$$

but γ would remain unknown without information from the right tree. When we can make use of the knowledge from annotated wobbly types, we are able to directly infer

$$\mathsf{eval} :: (\mathbf{Term\ Int}) \to \mathbf{Int}$$

because of the direct relation with the blueprint's wobbly variables.

The other issue, from Figure 3.7, was that there were two different instantiations of the type of eval. With wobbly types, this is not a problem, because both types individually satisfy the blueprint.

4.3 Type inference rules

Recall the type inference rules from Figure 2.3. The new rules for type inference with wobbly types are displayed in Figure 4.3. In this new rule set, two notable details have changed.

First, we add GADT support to the type constructor (rule Data). We do this by adding the type refinement operator -> to the syntax of the data constructors. The refinement should be of the form $T\ \overline{\tau}$. Since τ can hold type variables as well as concrete types, we can specify the refinement in detail. Next, the data constructors have their refined types stored in the environment.

Next, we need to introduce a new rule for the specific manual annotations of wobbly types (rule Ann). This rule does nothing more than declare a certain term x to be of type σ and stores this in the environment.

Finally, we add a line that reflects the change we made in the set of unification rules. We require the right-hand side of each line of the function body (i.e. the **case**-construct) to be of a more specific type than the type of the individual cases (i.e. the requirement $\sigma_e \sqsubseteq \sigma$). The type of the **case**-construct as a whole is now that general type σ.

$$\boxed{\Gamma \vdash_t \sigma : \star}$$

The $\vdash_t$ rules define how types are constructed. Note that the symbol $\star$ means "kind", which is the collection of all types. A type is either a type by way of language construct or a user-defined type, δ.

$$\frac{}{\Gamma \vdash_t \sigma : \star}\ \text{(Atom)} \qquad \frac{\delta : \star \in \Gamma}{\Gamma \vdash_t \delta : \star}\ \text{(Cust)}$$

$$\boxed{\Gamma \vdash_e e : \sigma}$$

The $\vdash_e$ rules define how expressions are evaluated.

$$\frac{e : \sigma \in \Gamma}{\Gamma \vdash_e e : \sigma}\ \text{(Var)} \qquad \frac{\tau = \Gamma(x) \quad \overline{\Gamma \vdash_e p : \tau} \quad \overline{\Gamma \vdash_e e : \sigma_e} \quad \overline{\sigma_e \sqsubseteq \sigma}}{\Gamma \vdash_e \textbf{case}\ x\ \textbf{of}\ \overline{p \rightarrow e} : \sigma}\ \text{(Case)}$$

$$\frac{\Gamma \vdash_t \tau : \star \quad \Gamma, x : \tau \vdash_e e : \tau'}{\Gamma \vdash_e \lambda x : \tau . e : \tau \rightarrow \tau'}\ \text{(Abs)} \qquad \frac{\Gamma \vdash_e e : \tau' \rightarrow \tau \quad \Gamma \vdash_e e' : \tau'}{\Gamma \vdash_e ee' : \tau}\ \text{(App)}$$

$$\boxed{\Gamma \vdash_d decl : \Gamma'}$$

The $\vdash_d$ rules define how data declarations are constructed. The result of this rule is a new environment Γ', which contains the declarations. Here, T indicates a type and C a data constructor.

$$\frac{\overline{\Gamma, T\ \overline{\alpha} : \star \vdash_t \sigma : \star}}{\Gamma \vdash_d\ ::\ T\ \overline{\alpha}\ \texttt{=}\ \overline{C\ \overline{\sigma}\ \texttt{->}\ T\ \overline{\tau}} : \left\{T\ \overline{\alpha} : \star,\ \overline{C\ \overline{\sigma} : T\ \overline{\tau}}\right\}}\ \text{(Data)}$$

$$\frac{\Gamma \vdash_t \sigma : \star}{\Gamma \vdash_d x\ ::\ \sigma : \{\Gamma, x : \sigma\}}\ \text{(Ann)}$$

$$\boxed{\Gamma_0 \vdash_p P : \sigma}$$

The $\vdash_p$ rules define how programs may be constructed. A program is a series of type and function declarations, followed by an expression. The program rule is the only rule that does not require a context.

$$\frac{\overline{\Gamma_0 \vdash_d decl : \Gamma_d} \qquad \Gamma = \overline{\Gamma_d} \qquad \Gamma \text{ is consistent} \qquad \Gamma \vdash_e e : \sigma}{\Gamma_0 \vdash_p \overline{decl};\ e : \sigma}\ \text{(Program)}$$

Figure 4.3: Type inference rules for wobbly types

Critical notes

There are some notes that need to be made regarding these inference rules for wobbly types. Note that, at this point, the Case rule contains in invalid construct, namely the sudden introduction of the variable σ. Of course, the type system cannot calculate this σ, so here type inference halts.

What is actually intended here, is that σ is a construct that should be passed downwards by the type system, as it should be declared higher in the AST of the program—by the programmer's manual annotation of the wobbly type, of course.

Due to time limitations, I was not able to extend the type inference framework with support for passing type information downward. Martin Sulzmann and Simon Peyton Jones have published on this topic in [22]. Although their approach uses coercions, the same basic ideas can be applied to see the actual implementation of such a construct.

Chapter 5

Conclusion

Without a doubt, wobbly types are useful. In Chapter 4, we already saw that inference with GADT-supporting type systems can be made decidable again with wobbly types. However, it must be noted that this is stated kind of unfair. It might not be honest to still keep calling it *type inference*. After all, although the type system can infer specific types in specific contexts, without the help of a human being, the system is unable to infer the general types by itself. Instead, the real inference process has shifted to the programmer's mind, filling up gaps that are left open by the type system. Although programming languages, type systems and all other luxury that is found in programming tools nowadays has been developed to prevent just that, wobbly types are still worth while, because they offer a nice balance of both expressiveness with predictability and simplicity of type inference.

Two other established examples of systems with programmer-specified type annotations that have these properties are higher-rank types and polymorphic recursion. It can be expected that expressive languages will shift increasingly towards type systems that exploit and propagate programmer annotations [9].

5.1 Related work

A lot of related work can be found in the works by Simon Peyton Jones [9, 16], Thorsten Altenkirch [3], Ralph Hinze [6, 11], Martin Sulzmann [21, 22] and Hongwei Xi [27]. We will now quickly cover some the most interesting literature that is around, regarding wobbly types.

The founder of wobbly types, Simon Peyton Jones, is very involved in the currently available literature. He acknowledges the existence of GADTs and shows that there are many innovations around regarding type systems that all fall under the same category as GADTs do. Several techniques (dependant types, first-class phantom types, guarded recursive data types) try to address more or less the same kind of challenge.

Significant research has been carried out by Hongwei Xi on the implementation of GADTs in System F. In [27], he introduced an explicitly typed variant of the λ-calculus ($\lambda_{2,G\mu}$), an implicitly typed source language that supports GADTs and established a direct translation between them.

In [22], in search of an alternative to Xi's approach, Simon Peyton Jones, together with Martin Sulzmann, have even shown how one extension of the intermediate language System F$_\omega$ with type-equality coercions is suitable and powerful enough to directly map not only wobbly types, but also associated types, functional dependencies and closed type functions to this system, without the need of a native implementation of each of the mechanisms individually. Unfortunately, the public release of this paper came too late to involve some of these interesting discoveries in this thesis.

Sulzmann has even carried out more research to optimise type inference support in real-life programming languages. In [21], he argues that in essence the wobbly types approach demands that every polymorphic recursive function and every use of a GADT must be annotated. His contributions are the introduction of an efficient type inference method for GADTs, that minimises the amount of user-specified annotations that are required to generally infer types. Furthermore, if possible, the method provides feedback to the programmer which parts of information are missing and need to be annotated manually.

There is a lot going on in the field of functional programming and, as often, the most interesting and cutting-edge innovations on type systems are introduced in this particular field. Having noticed the frequency of publications on wobbly types in the last year, it is pretty thinkable that we are on the verge of a breakthrough discovery in type systems as we may know them today.

5.2 Acknowledgements

First and foremost, I would like to thank Rinus Plasmeijer. Without his help, I would not have been able to come to the result you are reading right now. Thanks go out, not only for being an excellent guide that I absolutely needed to stay on track during this research, but mainly for waking my interest in functional languages and type theory in the first place.

Bibliography

[1] *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.4*. URL www.haskell.org/ghc/docs/6.4/html/users_guide/index.html.

[2] Thorsten Altenkirch and Thierry Coquand. A Functional Programming View of Type Theory. 1995.

[3] Thorsten Altenkirch, Conor McBride, and James McKinna. Why Dependant Types Matter. *ACM*, 2005.

[4] Henk Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon*, volume 2. 1992. URL citeseer.ist.psu.edu/barendregt92lambda.html.

[5] Alan F. Blackwell. Metacognitive Theories of Visual Programming: What do we think we are doing? *IEEE Symposium on Visual Languages*, page 240, 1996.

[6] James Cheney and Ralf Hinze. First-Class Phantom Types. 2003.

[7] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison-Wesley, 2000.

[8] Simon Peyton Jones. *The Implementation Of Functional Programming Languages*. Prentice Hall, 1987.

[9] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. *PLDI'06*, 2005.

[10] Yang Jun, Greg Michaelson, and Phil Trinder. Explaining Polymorphic Types. *Computer Journal, Feb 2001*, 2002. URL citeseer.ist.psu.edu/jun02explaining.html.

[11] Andres Löh and Ralf Hinze. Open data types and open functions. 2006.

[12] Alberto Martelli. An Efficient Unification Algorithm. In *ACM Transactions on Programming Languages and Systems*, volume 4, pages 258–282. No. 2, April 1982.

[13] Dr.ir. Hans Meijer. *Inleiding Vertalerbouw*. 2003.

[14] Hanne Riis Nielson and Flemming Nielson. Semantics With Applications—A Formal Introduction. 1999.

[15] Marian Petre and Alan F. Blackwell. Mental Imagery in Program Design and Visual Programming. *International Journal of Human-Computer Studies*, pages 7–30, No. 51, 1999.

[16] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, Computer and Information Science Department, Levine Hall, 3330 Walnut Street, Philadelphia, Pennsylvania, 19104-6389, July 2004.

[17] Rinus Plasmeijer and Marko van Eekelen. *The Concurrent Clean Language Report*, 2001.

[18] François Pottier and Yann Régis-Gianas. Stratified Type Inference For Generalized Algebraic Data Types. *POPL'06*, 2006.

[19] Tim Sheard. Languages of the Future. *ACM Conference on Object-Oriented Programming, Systems*, 2004.

[20] Thomas A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison Wesley Longman, Inc., second edition, 1997.

[21] Martin Sulzmann and Peter J. Stuckey. Type Inference for Guarded Recursive Data Types. 2005.

[22] Martin Sulzmann, Manual Chakravarty, and Simon Peyton Jones. System F with Type Equality Coercions. *ICFP'06*, 2006.

[23] Peter Thiemann. WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms. *PADL'02*, 2002.

[24] Jeff Vaughan. A proof of correctness for the Hindley-Milner type inference algorithm. 2005,2006.

[25] Joep Verkoelen and Vincent Driessen. Formal semantics of loosely typed languages. 2004.

[26] Philip Wadler. Proofs are Programs: 19th Century Logic and 21st Century Computing. 2000.

[27] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded Recursive Datatype Constructors. *POPL'03*, 2003.