Top Quality Type Error Messages

Top Kwaliteit Typeringsfoutmeldingen

(met een samenvatting in het Nederlands)

Proefschrift ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de Rector Magnificus, Prof. dr. W. H. Gispen, ingevolge het besluit van het College voor Promoties in het openbaar te verdedigen op dinsdag 20 september 2005 des ochtends te 10.30 uur

door

Bastiaan Johannes Heeren

geboren op 28 mei 1978, te Alphen aan den Rijn

promotor: Prof. dr. S. Doaitse Swierstra
co-promotor: Dr. Jurriaan Hage

Departement Informatica, Universiteit Utrecht

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Printed by Ridderprint offsetdrukkerij BV, Ridderkerk.
Cover painting *“Why did we reach Top?”* by Lydia van den Ham.

ISBN 90-393-4005-6

Contents

1 Introduction

Overview. *This chapter provides a gentle introduction to type systems, functional languages, and type error messages. We give examples of reported type error messages to illustrate the problems one typically has to face when programming in a statically typed language, such as Haskell. To improve the quality of type error messages, we introduce the* TOP *type inference framework. An outline of the topics covered by this framework is given.*

Flaws in computer software are a fact of life, both in small and large-scale applications. From a user's perspective, these errors may take the form of crashing programs that quit unexpectedly, security holes in computer systems, or internet pages that are inaccessible due to scripting errors. In some cases, a malfunctioning application is only a nuisance that frustrates the user, but in other cases such an error may have severe consequences. Creating reliable programs is one of the most challenging problems for the software industry.

Some programming mistakes, such as division by zero, are generally detected *dynamically*, that is, while a program is being executed. Dynamic checks, often inserted by a compiler, offer the programmer a high degree of freedom: programs are always accepted by the compiler, but at the expense of leaving potential errors unexposed. Some errors are found by testing, but testing alone, even if done thoroughly, cannot guarantee the reliability of software.

Compilers for modern programming languages accommodate many program analyses for finding mistakes at compile-time, or *statically*. These analyses detect a significant portion of errors automatically: this makes program analyses a valuable and indispensable tool for developing high quality software. Static analysis gives us two major advantages: certain classes of errors are completely eliminated, and the immediate compiler feedback to the programmer increases efficiency in program development. Moreover, the programmer (rather than the user) is confronted with his own mistakes.

Type checking is perhaps the most popular and best studied form of static analysis. This analysis guarantees that functions (or methods) are never applied to incompatible arguments. This property is best summarized by Milner's well-known phrase *"Well-typed programs do not go wrong"* [42]. Type checking is traditionally an important part of functional programming languages.

Functional programming is an alternative paradigm for the widely-used imperative style of programming. In a functional language, a program is described by a collection of mathematical functions, and not by a sequence of instructions. Functional languages are based on the lambda calculus, which allows for a high degree of abstraction. Examples of functional programming languages are LISP, Scheme, and ML. More recently, the language Haskell was designed (named after the logician Haskell Curry). One common feature of ML and Haskell is that they are implicitly typed: no type annotations in the program are required, although both languages allow specifying types for parts of a program. Instead, these languages rely on an effective type inference algorithm, which automatically deduces all types. This inference algorithm has the pleasing property that it finds principal (or most general) types. In fact, the inference algorithm not only recovers type information, it also detects type inconsistencies in programs, which are generally caused by mistakes made by the programmer.

In most compilers, the algorithm for type inference is not designed to provide good feedback, but rather focuses on performance. The type error messages that are reported by these algorithms can be difficult to understand. For experienced programmers, this is not a problem. They are mainly interested in whether the program is well-typed or not, and may have the tendency to take only a brief look at error messages. Beginning programmers, on the other hand, may get the impression that they have to fight the type system before they can execute their programs. Their main concern is *where* and *how* the program should be corrected, which is not always clear from the error message. This makes static typing a double-edged sword: it is highly appreciated by experts, but it can easily become a source of frustration for inexperienced programmers. As a result, functional languages have a steep learning curve, which is a serious barrier to start appreciating functional programming. This thesis is entirely devoted to type inference for programming languages such as Haskell, and to the process of notifying a programmer about type errors in particular.

But why is it so difficult for compilers to report constructive type error messages? Haskell, for example, hosts a number of features that pose extra challenges for type recovery algorithms, and thus also for error reporting. First of all, it supports higher-order functions (a function can be passed as an argument to a function, and can be the result of a function), and functions are allowed to be curried: instead of supplying all arguments at once, we may supply one argument at a time. Furthermore, the types inferred for functions may be parametrically polymorphic: type variables represent arbitrary types. And, finally, absence of type information in a program may lead to a mismatch between the types inferred by the compiler and the types assumed by the programmer.

Our goal is to improve type error messages for higher-order, polymorphic programming languages, especially targeted to the beginning programmer. To a great extent, this boils down to management of information: having more information available implies being able to produce more detailed error reports. Ideally, we should follow a human-like inference strategy, and depart from the mechanical order in which standard inference algorithms proceed. This entails, for example, a global analysis of a program, which depends on heuristics that capture expert knowledge. We may even anticipate common errors and provide additional information for special classes of mistakes.

We have reached the above results by pursuing a constraint-based approach to type inference. The typing problem is first mapped to a set of type constraints, which is then passed to a specialized constraint solver.

Other researchers have addressed the problem of type error messages (Chapter 3 provides an overview of related work), but our approach distinguishes itself by the following characteristics.

- *Practical experience.* All type inference techniques have been implemented in the Helium compiler [26], which covers almost the entire Haskell 98 standard (see also Section 2.5 on Haskell implementations). This compiler has been used for the introductory functional programming courses at Utrecht University, by students without any prior knowledge of functional programming. The feedback received from the students helped to further enhance our inference techniques. The implementation affirms that our approach scales to a full-blown programming language.
- *Customizable.* There is no best type inference algorithm that suits everyone. Instead, we present a parameterized framework, which can be instantiated and customized according to one's personal preferences.
- *Scriptable.* The type inference process can be personalized even further by scripting the type inference process. A small special purpose language is presented to influence the inference process and change the error message facility, without losing soundness of the underlying type system. For instance, we can tune the error messages to the level of expertise of a specific group of programmers.
- *Domain specific.* Functional languages are very suitable for embedding domain specific languages (DSL), such as combinator libraries. Unfortunately, type error messages are typically reported in terms of the host language, and not in terms of the actual domain. The scripting facility that comes with our framework makes it possible to phrase error messages in terms of the specific domain, creating a close integration of the DSL and the host language.

1.1 Motivating examples

We present three examples to illustrate what kind of error messages are typically reported for incorrect programs, and we discuss why these reports are hard to comprehend. Although our examples are written in Haskell, the type error messages are characteristic for all programming languages with a type system that is based on the Hindley-Milner type rules. These rules will be discussed in the next section.

Example 1.1. Let *plus* be the addition function with the type $Int \to Int \to Int$. We define a function f, which given a list of integers returns a pair consisting of the length and the sum of the list.

$$\begin{array}{l} f :: [Int] \to (Int, Int) \\ f = \lambda xs \to (length\ xs, foldr\ plus\ xs) \end{array}$$

The type signature provided for f is in accordance with our intention, but the definition contains a type error. The standard function *foldr* has the type

$$(a \to b \to b) \to b \to [a] \to b,$$

and thus expects three arguments. The application *foldr plus xs* implies that the type variables a and b in the type of *foldr* must both be *Int*, and, as a result, *xs* must also have type *Int*. Consequently, the type error message reported by Hugs [35], focuses on the other application in the definition of f, that is, *length xs*.

```
ERROR "A.hs":2 - Type error in application
*** Expression     : length xs
*** Term           : xs
*** Type           : Int
*** Does not match : [a]
```

This error message is not satisfactory, because the type signature that was supplied by the programmer has not been taken into consideration. Moreover, the type signature supports the idea that the application *foldr plus xs* should have been reported instead.

The error message reveals that at some point during type inference, *xs* is assumed to be of type *Int*. In fact, this assertions arises from the application *foldr plus xs*. Note that the bias in this type error message is caused by the fact that the second component of a pair is considered before the first component, which is rather counterintuitive.

Example 1.2. Function g computes the sine of the floating-point number 0.2, and multiplies the outcome with some parameter r. However, we make a syntax error in writing the floating-point number.

$$g\ r = r * \mathit{sin}\ .2$$

Nevertheless, g's definition is accepted by the Glasgow Haskell Compiler [18]. If we query the type that has been inferred for g, we find

$$g :: \mathit{forall}\ b\ a\ .\ (\mathit{Num}\ (a \to b), \mathit{Floating}\ b) \Rightarrow (a \to b) \to a \to b.$$

The spacing in the expression $r * \mathit{sin}\ .2$ is misleading: it is interpreted by the compiler as $r * (\mathit{sin}\ .\ 2)$, where the dot (.) denotes function composition. Because multiplication, *sin*, and the constant 2 are all overloaded in Haskell 98, the type inference engine miraculously finds the type above for g. It is likely that the error in g's definition goes unnoticed by a novice programmer. That is, until g is used elsewhere in the program.

In contrast with GHC, the Hugs interpreter does report a type error message for the definition of g.

```
ERROR "B.hs":1 - Illegal Haskell 98 class constraint in inferred type
*** Expression : g
*** Type       : (Num (a → b), Floating b) ⇒ (a → b) → a → b
```

This error report offers almost no help for the inexperienced programmer, who probably does not fully understand class constraints. In Chapter 11, we describe directives to improve the quality of type error messages for overloaded functions.

Example 1.3. Not only beginning Haskell programmers suffer from inaccurate type error messages. The more complex functions and their types become, the more difficult it is to extract useful information from a type error report. For instance, consider the following error message by Hugs. (We are not interested in the program which resulted in this error message.)

```
ERROR "C.hs":6 - Type error in application
*** Expression     : semLam <$ pKey "\\" <*> pFoldr1 (semCons, semNil) pVarid
 <*> pKey "->"
*** Term           : semLam <$ pKey "\\" <*> pFoldr1 (semCons, semNil) pVarid
*** Type           : [Token] → [((Type → Int → [([Char], (Type, Int, Int))] → Int
 → Int → [(Int, (Bool, Int))] → (Doc, Type, a, b, [c] → [Level], [S] → [S])) → Type →
d → [([Char], (Type, Int, Int))] → Int → Int → e → (Doc, Type, a, b, f → f, [S] → [S])
, [Token])]
*** Does not match : [Token] → [([Char] → Type → d → [([Char], (Type, Int, Int))]
 → Int → Int → e → (Doc, Type, a, b, f → f, [S] → [S]), [Token])]
```

Type error messages of this size are not exceptional, and they may arise from time to time in a typical laboratory assignment. The two contradicting types are rather complicated, and it is not immediately obvious why the types cannot be unified. And even if we understand the differences between the types, it remains a challenging task to apply the desired correction to the program at hand.

1.2 Overview of the framework

Figure 1.1 presents an overview of the constraint-based type inference framework Top, and gives correlations between the topics introduced in this thesis. For each topic, the figure indicates in which chapter or section it is discussed: these references are displayed in gray. Top is not only a theoretical framework, but also a library for building program analyses that offer high quality feedback to users.

Constraints play a central role in the framework, and in particular the set of type constraints that we use to describe the Haskell 98 type system. A type analysis conducted within the framework can be divided into three phases. Firstly, type constraints are collected for a program or an expression. For this, we use a set of syntax-directed constraint collecting type rules. These rules are formulated in a bottom-up fashion. In the second phase, we choose an ordering for the collected type constraints. If the constraints are inconsistent, then the relative order determines at which point we discover the error (in particular when the constraints are handled sequentially). Instead of collecting the type constraints in a set, we generate constraint trees that closely follow the shape of the abstract syntax tree of the program that is analyzed, and these trees are flattened using tree walks. As it turns out, the well-known type inference algorithms $\mathcal{W}$ and $\mathcal{M}$ (explained in Section 2.4) can be emulated by choosing a certain tree walk.

Thirdly, we discuss how to solve an ordered list of constraints. We take a flexible approach that allows new kinds of constraints to be included easily, and in which different solving strategies can live side by side. Special attention is paid to

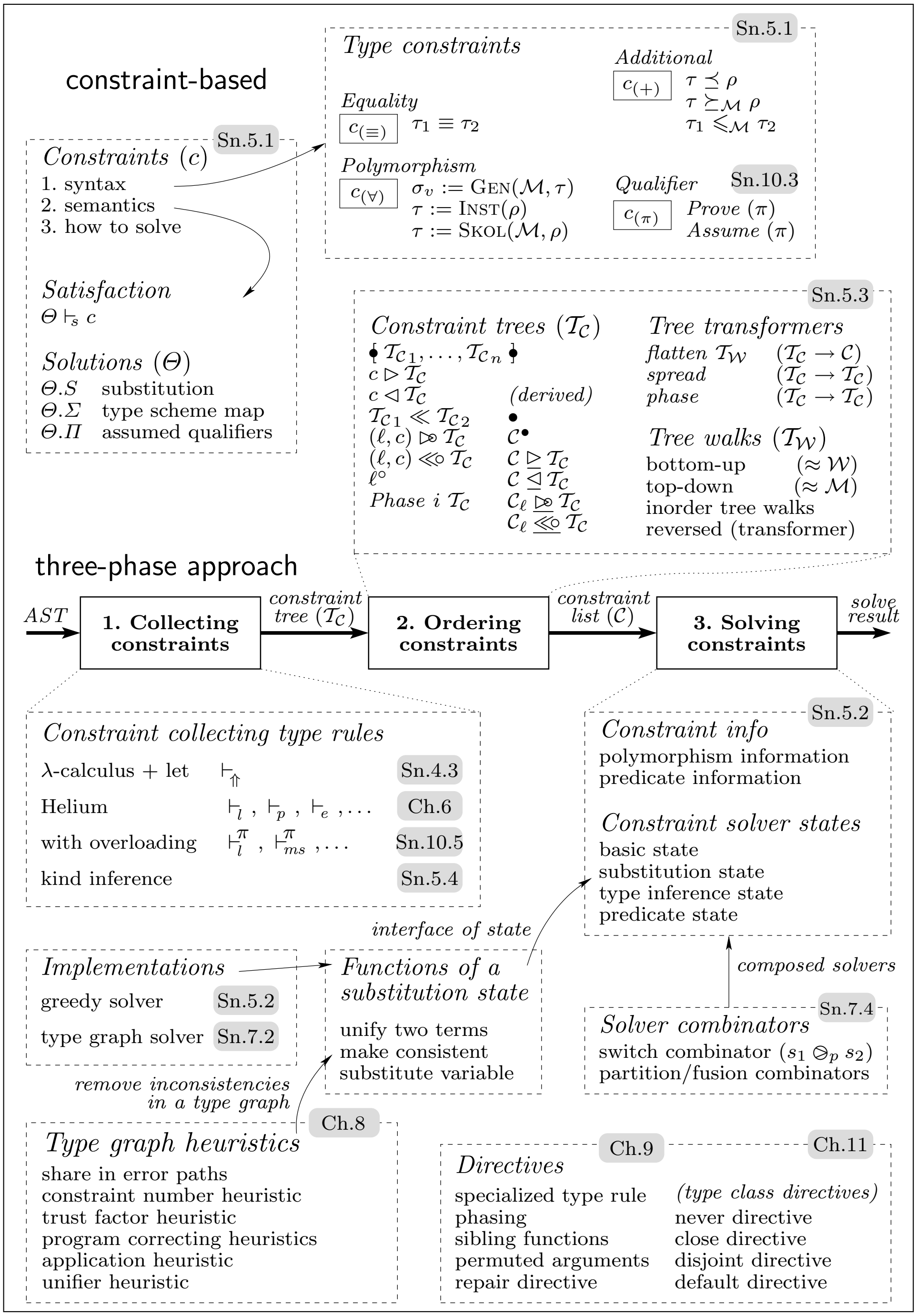

constraint-based
Sn.5.1
Constraints (c)
1. syntax
2. semantics
3. how to solve
Satisfaction
$\Theta \vdash_s c$
Solutions (Θ)
$\Theta.S$ substitution
$\Theta.\Sigma$ type scheme map
$\Theta.\Pi$ assumed qualifiers
Type constraints
Sn.5.1
Equality
$\boxed{c_{(\equiv)}}$ $\tau_1 \equiv \tau_2$
Polymorphism
$\boxed{c_{(\forall)}}$ $\sigma_v := \text{GEN}(\mathcal{M}, \tau)$
$\tau := \text{INST}(\rho)$
$\tau := \text{SKOL}(\mathcal{M}, \rho)$
Additional
$\boxed{c_{(+)}}$ $\tau \preceq \rho$
$\tau \succeq_{\mathcal{M}} \rho$
$\tau_1 \leqslant_{\mathcal{M}} \tau_2$
Qualifier
Sn.10.3
$\boxed{c_{(\pi)}}$ Prove (π)
Assume (π)
Sn.5.3
Constraint trees $(\mathcal{T_C})$
$\bullet\!\!\lbrack \mathcal{T_C}_1, \ldots, \mathcal{T_C}_n \rbrack\!\!\bullet$
$c \rhd \mathcal{T_C}$
$c \lhd \mathcal{T_C}$
$\mathcal{T_C}_1 \ll \mathcal{T_C}_2$
$(\ell, c) \rhd\!\!\circ\, \mathcal{T_C}$
$(\ell, c) \ll\!\circ\, \mathcal{T_C}$
ℓ°
Phase i $\mathcal{T_C}$
(derived)
$\bullet$
$\mathcal{C}^\bullet$
$\mathcal{C} \trianglerighteq \mathcal{T_C}$
$\mathcal{C} \trianglelefteq \mathcal{T_C}$
$\mathcal{C}_\ell \,\underline{\rhd\!\!\circ}\, \mathcal{T_C}$
$\mathcal{C}_\ell \,\underline{\ll\!\circ}\, \mathcal{T_C}$
Tree transformers
flatten $\mathcal{T_W}$ $(\mathcal{T_C} \to \mathcal{C})$
spread $(\mathcal{T_C} \to \mathcal{T_C})$
phase $(\mathcal{T_C} \to \mathcal{T_C})$
Tree walks $(\mathcal{T_W})$
bottom-up $(\approx \mathcal{W})$
top-down $(\approx \mathcal{M})$
inorder tree walks
reversed (transformer)
three-phase approach
AST
1. Collecting constraints
constraint tree $(\mathcal{T_C})$
2. Ordering constraints
constraint list $(\mathcal{C})$
3. Solving constraints
solve result
Constraint collecting type rules
λ-calculus + let $\vdash_{\Uparrow}$ Sn.4.3
Helium $\vdash_l$, $\vdash_p$, $\vdash_e$, ... Ch.6
with overloading $\vdash^\pi_l$, $\vdash^\pi_{ms}$, ... Sn.10.5
kind inference Sn.5.4
Sn.5.2
Constraint info
polymorphism information
predicate information
Constraint solver states
basic state
substitution state
type inference state
predicate state
interface of state
composed solvers
Implementations
greedy solver Sn.5.2
type graph solver Sn.7.2
Functions of a substitution state
unify two terms
make consistent
substitute variable
Sn.7.4
Solver combinators
switch combinator $(s_1 \oslash_p s_2)$
partition/fusion combinators
remove inconsistencies in a type graph
Ch.8
Type graph heuristics
share in error paths
constraint number heuristic
trust factor heuristic
program correcting heuristics
application heuristic
unifier heuristic
Ch.9
Ch.11
Directives
specialized type rule
phasing
sibling functions
permuted arguments
repair directive
(type class directives)
never directive
close directive
disjoint directive
default directive

Figure 1.1. Overview of the constraint-based type inference framework TOP

heuristics for dealing with type inconsistencies. These heuristics help to pinpoint the most likely source of a type error, and enhance the type error messages. Moreover, they can be used to give suggestions how to correct the problem (for a number of common mistakes).

In the Top framework, developers of a library can specify their own type inference directives. These directives are particularly useful in the context of domain specific languages. The directives change all three phases of the type inference process. For some directives we can guarantee that the underlying type system remains intact (i.e., it affects only the error reporting facility).

1.3 Structure of this thesis

Chapters 2 and 3 provide background material required for the remainder of the thesis. We introduce some notation, and give a short introduction to the Hindley-Milner type system, which is used in many modern programming languages. We explain the set of formal type rules, as well as two type inference algorithms that implement these rules. In Chapter 3, we give an extensive overview of the literature on type error messages, and we categorize all proposals.

A formal treatment of constraint-based type inference is given in the fourth chapter, and we explain how we deal with parametric polymorphism. Bottom-up type rules are used to collect constraints for a small expression language, and an algorithm to solve the constraints is discussed. We conclude this chapter with a correctness proof for our constraint-based approach with respect to the type rules formulated by Hindley and Milner.

We present an introduction to the Top type inference framework in Chapter 5, which is a continuation of the approach proposed in the previous chapter. Top is designed to be customizable and extendable, which are two essential properties to encourage reusability of the framework. In the Top framework, each constraint carries information, which can be used to create an error message if the constraint is blamed for a type inconsistency, and which describes properties of the constraint at hand. Three independent phases are distinguished: collecting constraints, ordering constraints, and solving constraints. Each of these phases is explained in detail. We conclude with a brief discussion on how to use the framework to perform kind inference, and we sum up the advantages of the approach taken.

Chapter 6 contains the complete set of constraint collecting type rules that is used by the Helium compiler [26]. Hence, these rules cover almost the entire Haskell 98 language. The most notable omission is that we do not consider overloading at this point. The rules are not only given for completeness reasons: they possess various strategies to enable high quality type error messages in later chapters. In fact, we use the rules to create a constraint tree for ordering the constraints in the second phase of the Top framework. The chapter includes a detailed presentation of binding group analysis.

Type graphs are introduced in Chapter 7. A type graph is a data structure for representing a substitution with the remarkable property that it can be in an inconsistent state. This property is the key to an alternative constraint solving algorithm, which can proceed in case of an inconsistency. With type graphs, we can efficiently analyze an entire set of type constraints, which significantly improves the quality of the type error messages. Finally, we introduce two combinators to compose constraint solvers.

Chapter 8 presents a number of predefined heuristics, which help to pinpoint the most likely source of an error. These heuristics work on the type graphs introduced in the previous chapter. In Chapter 9, we discuss type inference directives, which are user-defined heuristics. Directives enable a user to adapt the error reporting facility of a type system externally (that is, without changing the compiler). Directives are very suitable for experimenting with different type inference strategies, and are particularly useful to support domain specific libraries. Soundness of the presented directives is automatically checked, which is an important feature to guarantee that the underlying type system remains intact.

In Chapter 10, we extend the type inference framework to handle overloading of identifiers. We take an approach for dealing with qualifiers in general, which is closely related to Jones' theory on qualified types [31]. New constraints for dealing with qualifiers are added to the framework, and new constraint collecting type rules are given for the overloaded language constructs (to replace the rules from Chapter 6). We give a short introduction to improving substitutions, which are needed to capture other qualifiers.

Overloading of identifiers is a very pervasive language feature, and it strongly influences what is reported, and where. To remedy this problem, Chapter 11 proposes four type class directives to improve type error messages that are related to overloading. In addition, the directives from Chapter 9 are generalized to cope with overloading.

Finally, Chapter 12 sums up the main contributions of this thesis. We conclude with a list of topics that require further research.

2 Preliminaries

Overview. *In this chapter we fix the notation, and introduce basic concepts that are needed for the rest of this thesis. Furthermore, we provide an introduction to the Hindley-Milner type rules, and the type inference algorithms $\mathcal{W}$ and $\mathcal{M}$. We conclude with a note on existing Haskell implementations.*

Most of the notation in this thesis is customary, and readers familiar with type systems can safely skip this chapter. New symbols and special syntax are gradually introduced and explained within each chapter.

One issue that deserves special attention is the distinction we make between sets and lists. In a set, duplicates are ignored, and the order of the elements is irrelevant. On the contrary, lists specify an ordering of the elements, and may contain duplicates. To distinguish these collections, we write $\{x_1, \ldots, x_n\}$ for sets, and $[x_1, \ldots, x_n]$ for lists. Furthermore, $\emptyset$ denotes the empty set, we write $A \cup B$ for the union of A and B, and we use $A - B$ for set difference. The empty list is written $[\,]$, and we use $+\!\!+$ to concatenate two lists. Occasionally, we write $\overline{x}$ instead of $\{x_1, \ldots, x_n\}$.

A finite map is a set of key-value pairs, written as *key* : *value*. We write $A(x)$ to get the value associated with the key x in the finite map A. We use $dom(A)$ and $ran(A)$ to obtain the keys and the values (respectively) from A, and $A \backslash x$ to remove all pairs with key x. Also, we use $A \backslash \{x_1, \ldots, x_n\}$ as an abbreviation for $((A \backslash x_1) \backslash \ldots) \backslash x_n$.

The rest of this chapter is organized as follows. In Section 2.1, we define a small expression language, and we discuss types and type schemes. Various operations on types, such as substitution and unification, are defined in Section 2.2. In the sections 2.3 and 2.4 we give a minimal introduction to the Hindley-Milner type rules, and the type inference algorithms $\mathcal{W}$ and $\mathcal{M}$. For more background reading on this subject, the interested reader is referred to Pierce's excellent textbook on type systems [52]. Finally, some notes on existing Haskell implementations are given in Section 2.5.

2.1 Expressions, types, and type schemes

We take the untyped lambda calculus as the starting point. The lambda calculus is a minimal programming language which captures the core of languages such as ML, Haskell, and Scheme. It is widely used to design, specify, and implement language features. In particular, it plays a central role in the study of type systems. We enrich the lambda calculus with local definitions (or let expressions): although a local definition can be translated into the lambda calculus, it introduces a form of *polymorphism*[1] to the language. This means that one expression can be used in a diversity of contexts. The expression language we use is:

Expression:

$$\begin{array}{lll} e ::= & x & \textit{(identifier)} \\ \mid & e_1\ e_2 & \textit{(application)} \\ \mid & \lambda x \rightarrow e & \textit{(abstraction)} \\ \mid & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 & \textit{(local definition)} \end{array}$$

Applications are left associative, and parentheses are used to disambiguate. Besides identifiers and (function) applications, we have lambda abstractions to construct a function. Given an identifier (say x), this function yields an expression (in which x may appear). Similarly, a local definition $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$ introduces the identifier x, which can be used in e_2. Local definitions are not recursive: free occurrences of x in e_1 are not bound. (We deal with recursive definitions in later chapters.) Scoping of identifiers is defined by the function *fev*, which determines the free (or unbound) expression variables.

$$\begin{array}{ll} \mathit{fev}(x) & = \{x\} \\ \mathit{fev}(e_1\ e_2) & = \mathit{fev}(e_1) \cup \mathit{fev}(e_2) \\ \mathit{fev}(\lambda x \rightarrow e) & = \mathit{fev}(e) - \{x\} \\ \mathit{fev}(\textbf{let}\ x = e_1\ \textbf{in}\ e_2) & = \mathit{fev}(e_1) \cup (\mathit{fev}(e_2) - \{x\}) \end{array}$$

An expression is *closed* if it contains no free expression variables.

In addition to the expression language, we have types.

Type:

$$\begin{array}{lll} \tau ::= & a & \textit{(type variable)} \\ \mid & T & \textit{(type constant)} \\ \mid & \tau_1\ \tau_2 & \textit{(type application)} \end{array}$$

Type variables are place-holders, and represent arbitrary types. In type inference algorithms, type variables are often used as an implementation technique to denote an unknown type, but which can be refined later on. We assume to have infinitely many type variables at our disposal, for which we usually write v_0, v_1, v_2, and so on. We write β for a fresh type variable, that is, a type variable that has not been used yet. Furthermore, we have type constants, such as *Int* and *Bool*, and type constructors to build composite types. For instance, $(\rightarrow)$ denotes function space, and creates a function type out of two types. We use an infix to denote function types: $(\mathit{Int} \rightarrow \mathit{Bool})$ denotes the type applications $(((\rightarrow)\ \mathit{Int})\ \mathit{Bool})$. The function arrow is right associative. We also have special syntax for lists and pairs, e.g., $[\mathit{Int}]$ and $(\mathit{Int}, \mathit{Bool})$.

Actually, the type language is too liberal. Meaningless types, such as $(\mathit{Int}\ \mathit{Bool})$ and $((\rightarrow)\ \mathit{Int})$ should be ruled out in advance. We make the explicit assumption

[1] translation: *more than one shape*

that we only deal with well-formed types. A common approach is to use a kind system to guarantee the well-formedness of types. In Section 5.4, we take a closer look at kind inference.

Finally, we also consider polymorphic types or type schemes (as opposed to the types above, which are monomorphic). In a type scheme, some type variables are universally quantified.

Type scheme:

$$\begin{array}{lll} \sigma & ::= \forall a.\sigma & \textit{(polymorphic type scheme)} \\ & \mid\ \ \tau & \textit{(monomorphic type)} \end{array}$$

Because we introduce an extra syntactic class for type schemes rather than adding universal quantification to the type language, we are able to predict at which locations quantifiers can show up. In the presentation of a type scheme, we use $a, b, c, \ldots$ for the quantified type variables. $\forall abc.\tau$ is a short-hand notation for $\forall a.\forall b.\forall c.\tau$, and, similarly, we write $\forall \overline{a}.\tau$ to quantify the type variables in $\overline{a}$.

2.2 Operations on types

The function *ftv* returns all free type variables. For types, this function is defined as follows.

$$\mathit{ftv}(a) = \{a\} \qquad \mathit{ftv}(T) = \emptyset \qquad \mathit{ftv}(\tau_1\ \tau_2) = \mathit{ftv}(\tau_1) \cup \mathit{ftv}(\tau_2)$$

This function also collects the free type variables of larger syntactic constructs that contain types. The free type variables of a type scheme, however, do not include the quantified variables: $\mathit{ftv}(\forall \overline{a}.\tau) = \mathit{ftv}(\tau) - \overline{a}$.

A substitution, denoted by S, is a mapping from type variables to types. Only free type variables are replaced by a substitution. The empty substitution is the identity function, and is written *id*. We write Sx to denote the application of S to x, where x is some syntactic construct containing free type variables (for instance a type). Often, we use finite maps to represent a substitution. For instance, the substitution $[v_1 := \tau_1, \ldots, v_n := \tau_n]$ (also written as $[\overline{v} := \overline{\tau}]$) maps v_i to τ_i $(1 \leqslant i \leqslant n)$ and all other type variables to themselves. For notational convenience, $[v_1, \ldots, v_n := \tau]$ abbreviates $[v_1 := \tau, \ldots, v_n := \tau]$.

The domain of a substitution contains the type variables that are mapped to a different type: $\mathit{dom}(S) = \{a \mid Sa \neq a\}$. We define the range (or co-domain) of a substitution as $\mathit{ran}(S) = \{Sa \mid a \in \mathit{dom}(S)\}$. A substitution S is idempotent if and only if $\mathit{dom}(S) \cap \mathit{ran}(S) = \emptyset$, which implies that Sx equals $S(Sx)$. We only consider substitutions with this property. The substitutions S_1 and S_2 are equal only if $S_1 a = S_2 a$ for all $a \in \mathit{dom}(S_1) \cup \mathit{dom}(S_2)$. Furthermore, we define a partial order on substitutions: $S_1 \sqsubseteq S_2$ if and only if $\exists R. R \circ S_1 = S_2$. Clearly, the minimal element of this poset is the empty substitution. We define $\top$ to be the error substitution, and the maximal element of the poset. This substitution maps all types to the error type *ErrorType*. We define the function $\mathit{tv}(S)$, which returns all type variables that are part of the substitution S.

$$\begin{array}{lll}
\mathit{mgu} :: (\mathit{Type}, \mathit{Type}) \to \mathit{Substitution} & & \\
\mathit{mgu}(a, a) & = \mathit{id} & \\
\mathit{mgu}(T, T) & = \mathit{id} & \\
\mathit{mgu}(a, \tau) & = [\, a := \tau \,] & a \notin \mathit{ftv}(\tau) \\
\mathit{mgu}(\tau, a) & = [\, a := \tau \,] & a \notin \mathit{ftv}(\tau) \\
\mathit{mgu}(\tau_1\ \tau_2, \tau_3\ \tau_4) & = \mathbf{let}\ S = \mathit{mgu}(\tau_1, \tau_3) & \\
 & \quad \mathbf{in}\ \mathit{mgu}(S\tau_2, S\tau_4)\ \circ\ S & \\
\mathit{mgu}(\_, \_) & = \top & \mathit{otherwise}
\end{array}$$

Figure 2.1. Robinson's unification algorithm

Definition 2.1 (*tv*). *The set of type variables in substitution S is*

$$tv(S) \quad =_{def} \quad dom(S) \cup ftv(ran(S)).$$

Substitution S is a unifier for τ_1 and τ_2 if $S\tau_1 = S\tau_2$, and a most general unifier (or principal unifier) if $S \sqsubseteq S'$ holds whenever S' is a unifier of τ_1 and τ_2. A unification algorithm calculates a substitution which makes two types equal. Let *mgu* be Robinson's unification algorithm [55] to compute a most general unifier for two types. Algorithm *mgu* is presented in Figure 2.1: the error substitution $\top$ is returned only if all other cases fail.

Proposition 2.1 (Robinson unification). *Algorithm $mgu(\tau_1, \tau_2)$ returns a most general unifier for τ_1 and τ_2, such that it only involves type variables in τ_1 and τ_2. If unification fails, then the error substitution $\top$ is returned.*

Lemma 2.1 (*mgu*). *Let S be $mgu(\tau_1, \tau_2)$. Then $tv(S) \subseteq ftv(\tau_1) \cup ftv(\tau_2)$.*

Proof. Follows directly from Proposition 2.1. □

We can generalize a type to a type scheme while excluding the free type variables of $\mathcal{M}$, which are to remain monomorphic (or unquantified):

$$generalize(\mathcal{M}, \tau) \quad =_{def} \quad \forall \overline{a}.\tau \qquad where\ \overline{a} = ftv(\tau) - ftv(\mathcal{M}).$$

We can also generalize a type with respect to a type environment Γ (which is a finite map) by taking the free type variables of the types in the co-domain of Γ. A type scheme $\forall \overline{a}.\tau$ can be instantiated into $S\tau$ if $dom(S) \subseteq \overline{a}$ (only quantified type variables are substituted). A typical way to use instantiation is to replace $\overline{a}$ by fresh type variables. Skolemization is a special form of instantiation in which the quantified type variables are replaced by fresh skolem constants: these are type constants that do not appear elsewhere.

We introduce an instance-of relation between type schemes: $\sigma_1 < \sigma_2$ indicates that σ_1 is an instance of σ_2 (or, that σ_2 is the more general type scheme). If we view type schemes as sets of types that can be obtained via instantiation, then the set for σ_1 is a subset of the set for σ_2, and $<$ can be understood as subset-of. Note that this notation is consistent with the conventional notation, except that often $\sigma_2 > \sigma_1$ is

$\boxed{\sigma < \sigma}$ *Instance-of relation*

$$\frac{}{\tau < \tau}\ (\textsc{Sub-Mono})$$

$$\frac{\sigma_1 < [a := \tau]\sigma_2}{\sigma_1 < \forall a.\sigma_2}\ (\textsc{Sub-Inst}) \qquad \frac{\sigma_1 < \sigma_2 \qquad a \notin \mathit{ftv}(\sigma_2)}{\forall a.\sigma_1 < \sigma_2}\ (\textsc{Sub-Skol})$$

Figure 2.2. Instance-of relation on type schemes

used (e.g., in [11, 37]). This is in contrast with a number of papers on higher-rank polymorphism that really use the inverse relation (for instance, [45, 51]). In most cases, σ_1 is just a type.

The instance-of relation is specified by three deduction rules (see Figure 2.2). Deduction rules can be interpreted as follows: if the premises hold (above the horizontal line), then we can deduce the judgement in the conclusion (below the horizontal line). First of all, the only instance of a (monomorphic) type is the type itself (Sub-Mono). A quantified type variable appearing on the right-hand side can be instantiated by every type (Sub-Inst). For instance, $\mathit{Int} \to \mathit{Int} < \forall a.a \to a$ (by choosing Int for a), but also $(v_0 \to v_0) \to (v_0 \to v_0) < \forall a.a \to a$ (by choosing $v_0 \to v_0$ for a). A quantifier on the left-hand side can be removed (Sub-Skol), but only if the quantified type variable is not free in the type scheme on the right-hand side. This is, in fact, similar to skolemizing the type variable. The instance-of relation is both reflexive and transitive. The following examples illustrate this relation.

Example 2.1.

$$\begin{aligned} v_1 \to v_1 &\not< v_2 \to v_2 \\ v_1 \to v_1 &< \forall a.a \to a \\ \mathit{Int} \to \mathit{Int} &< \forall a.a \to a \\ \forall a.(a \to a) \to a \to a &< \forall a.a \to a \\ \forall a.a \to a &\not< v_1 \to v_1 \end{aligned}$$

We define two type schemes to be equivalent whenever they are mutual instances.

$$\sigma_1 = \sigma_2 \qquad =_{def} \qquad \sigma_1 < \sigma_2 \ \wedge \ \sigma_2 < \sigma_1$$

As a result, type schemes are equivalent up to alpha conversion of quantified type variables. Also the order in which type variables are quantified becomes irrelevant.

Lemma 2.2 (Instance-of). $\tau_1 < \forall \overline{a}.\tau_2 \quad \Longleftrightarrow \quad \exists S : \mathit{dom}(S) \subseteq \overline{a} \ \wedge \ \tau_1 = S\tau_2$

Proof. Follows from the instance-of relation, defined in Figure 2.2. In fact, the substitution predicts precisely for each quantified type variable (in $\overline{a}$) to which type it should be mapped in the deduction rule (Sub-Inst). □

$\boxed{\Gamma \vdash_{\text{HM}} e : \tau}$ *Typing judgement*

$$\frac{\tau < \Gamma(x)}{\Gamma \vdash_{\text{HM}} x : \tau} \quad (\text{HM-V}\textsc{ar})$$

$$\frac{\Gamma \vdash_{\text{HM}} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\text{HM}} e_2 : \tau_1}{\Gamma \vdash_{\text{HM}} e_1 \; e_2 : \tau_2} \quad (\text{HM-A}\textsc{pp})$$

$$\frac{\Gamma \backslash x \cup \{x : \tau_1\} \vdash_{\text{HM}} e : \tau_2}{\Gamma \vdash_{\text{HM}} \lambda x \to e : (\tau_1 \to \tau_2)} \quad (\text{HM-A}\textsc{bs})$$

$$\frac{\Gamma \vdash_{\text{HM}} e_1 : \tau_1 \qquad \Gamma \backslash x \cup \{x : \mathit{generalize}(\Gamma, \tau_1)\} \vdash_{\text{HM}} e_2 : \tau_2}{\Gamma \vdash_{\text{HM}} \mathbf{let}\; x = e_1 \;\mathbf{in}\; e_2 : \tau_2} \quad (\text{HM-L}\textsc{et})$$

Figure 2.3. Syntax-directed Hindley-Milner type rules

2.3 Hindley-Milner type rules

In this section, we explore the famous type rules formulated by Roger Hindley and Robin Milner. The logician Hindley was the first to study a type discipline to conduct proofs [27]. Milner, computer scientist, and designer of the functional language ML, independently rediscovered a theory of type polymorphism for functional programming languages [42]. Nowadays, these rules are known as the Hindley-Milner type rules, and form the basis of type systems for programming languages, including Haskell.

The Hindley-Milner rules are constructed from typing judgements of the form $\Gamma \vdash_{\text{HM}} e : \tau$. Such a judgement can be read as "under the type assumptions in Γ, we can assign the type τ to expression e". The type assumptions Γ (or type environment) contains at most one type scheme for each identifier.

Figure 2.3 displays four type rules: one rule for each alternative of the expression language. The type rule for an identifier (HM-Var) states that the type environment should be consulted: any instance of the identifier's type scheme can be assigned to the identifier. For function applications (HM-App), we have to derive a type both for the function e_1 and the argument e_2. Only when we derive a function type for e_1 (say $\tau_1 \to \tau_2$), and the type assigned to e_2 matches τ_1, then τ_2 can be assigned to the function application. A lambda abstraction (HM-Abs) adds a type assumption about the abstracted identifier x to Γ, which is used to derive a type for the body of the abstraction. If Γ already contains an assumption about x, then this assumption is replaced. The lambda abstraction is assigned a function type, from the type associated with x in the type environment, to the type found for the body. In the type rule for a local definition (HM-Let), a type τ_1 is derived for the definition, which is then generalized with respect to the assumptions in Γ.

The resulting type scheme is associated with x, and used to type the body of the let expression. We assign the type of the body to the complete let expression.

The rules make up a system for constructing proofs that a type can be assigned to an expression. It is important to realize that in order to construct a proof, one needs at certain points an oracle to tell which type to choose. Furthermore, the rules cannot determine whether a type derivation exists for a given expression. For that, we need an algorithm that implements the rules. In the following example, we present a derivation with the Hindley-Milner type rules.

Example 2.2. The following derivation can be constructed with the rules in Figure 2.3. In this derivation, we use σ to abbreviate the type scheme $\forall v_0.v_0 \to v_0$.

$$\dfrac{\dfrac{\dfrac{v_0 < v_0}{\{x:v_0\} \vdash_{\mathrm{HM}} x : v_0}}{\emptyset \vdash_{\mathrm{HM}} \lambda x \to x : v_0 \to v_0} \qquad \dfrac{\dfrac{(v_1{\to}v_1){\to}v_1{\to}v_1 < \sigma}{\{i:\sigma\} \vdash_{\mathrm{HM}} i : (v_1{\to}v_1){\to}v_1{\to}v_1} \quad \dfrac{v_1{\to}v_1 < \sigma}{\{i:\sigma\} \vdash_{\mathrm{HM}} i : v_1{\to}v_1}}{\{i:\sigma\} \vdash_{\mathrm{HM}} i\ i : v_1 \to v_1}}{\emptyset \vdash_{\mathrm{HM}} \mathbf{let}\ i = \lambda x \to x\ \mathbf{in}\ i\ i : v_1 \to v_1}$$

The next two lemmas phrase under which conditions assumptions can be added to and removed from a type environment in a complete derivation.

Lemma 2.3 (Extend Γ). $\Gamma \vdash_{\mathrm{HM}} e : \tau \wedge x \notin \mathit{dom}(\Gamma) \implies \Gamma \cup \{x:\sigma\} \vdash_{\mathrm{HM}} e : \tau$

Proof. New assumptions may be added to Γ, because the deduction tree deriving $\Gamma \vdash_{\mathrm{HM}} e : \tau$ remains valid. □

Lemma 2.4 (Reduce Γ). $\Gamma \vdash_{\mathrm{HM}} e : \tau \ \wedge\ x \notin \mathit{fev}(e) \implies \Gamma \backslash x \vdash_{\mathrm{HM}} e : \tau$

Proof. Only the assumptions in Γ concerning free identifiers in e contribute in constructing a deduction tree. Hence, the other assumptions can be discarded. □

2.4 Type inference algorithms

Damas and Milner [11] presented a type assignment algorithm $\mathcal{W}$, which infers a type (if possible) for an expression and a type environment. Moreover, they prove that $\mathcal{W}$ is a correct implementation of the Hindley-Milner type rules (it is sound and complete), and that principal type schemes are derived (i.e., it finds the most general type possible).

Algorithm $\mathcal{W}$ is given in Figure 2.4: given a type environment and an expression, it returns a substitution and a type. Fresh type variables are introduced at various locations, and these type variables act as place-holders. The substitution we maintain refines the type variables, and is updated along the way. For each function application, Robinson's unification algorithm [55] is used to refine the type of the function and the argument such that the types fit. In fact, $\mathcal{W}$ fails when the types cannot be unified. In such a case we say that the expression is ill-typed (under the used type environment).

```
$\mathcal{W} :: (\mathit{TypeEnvironment}, \mathit{Expression}) \to (\mathit{Substitution}, \mathit{Type})$

$\mathcal{W}(\Gamma, x)$                  = let $\forall \overline{a}.\tau = \Gamma(x)$
                                in $(\mathit{id}, [\overline{a} := \overline{\beta}]\tau)$                 fresh $\overline{\beta}$

$\mathcal{W}(\Gamma, \lambda x \to e)$           = let $(S, \tau) = \mathcal{W}(\Gamma \backslash x \cup \{x : \beta\}, e)$       fresh $\beta$
                                in $(S, S\beta \to \tau)$

$\mathcal{W}(\Gamma, e_1\ e_2)$               = let $(S_1, \tau_1) = \mathcal{W}(\Gamma, e_1)$
                                    $(S_2, \tau_2) = \mathcal{W}(S_1\Gamma, e_2)$
                                    $S_3 = \mathit{mgu}(S_2\tau_1, \tau_2 \to \beta)$              fresh $\beta$
                                in $(S_3 S_2 S_1, S_3\beta)$

$\mathcal{W}(\Gamma,$ let $x = e_1$ in $e_2)$ = let $(S_1, \tau_1) = \mathcal{W}(\Gamma, e_1)$
                                    $\sigma = \mathit{generalize}(S_1\Gamma, \tau_1)$
                                    $(S_2, \tau_2) = \mathcal{W}(S_1\Gamma \backslash x \cup \{x : \sigma\}, e_2)$
                                in $(S_2 S_1, \tau_2)$
```

Figure 2.4. Standard type inference algorithm $\mathcal{W}$

Example 2.3. We show how algorithm $\mathcal{W}$ proceeds for the expression of Example 2.2. Bullets and indentation indicate the recursive calls to $\mathcal{W}$, and the call to the unification algorithm.

```
$\mathcal{W}(\emptyset, \textbf{let } i = \lambda x \to x \textbf{ in } i\ i)$
    • $\mathcal{W}(\emptyset, \lambda x \to x)$
        • $\mathcal{W}(\{x : v_0\}, x) = (\mathit{id}, v_0)$
    $= (\mathit{id}, v_0 \to v_0)$
    • $\mathcal{W}(\{i : \forall v_0.v_0 \to v_0\}, i\ i)$
        • $\mathcal{W}(\{i : \forall v_0.v_0 \to v_0\}, i) = (\mathit{id}, v_1 \to v_1)$
        • $\mathcal{W}(\{i : \forall v_0.v_0 \to v_0\}, i) = (\mathit{id}, v_2 \to v_2)$
        • $\mathit{mgu}(v_1 \to v_1, (v_2 \to v_2) \to v_3) = [v_1, v_3 := v_2 \to v_2]$
    $= ([v_1, v_3 := v_2 \to v_2], v_2 \to v_2)$
$= ([v_1, v_3 := v_2 \to v_2], v_2 \to v_2)$
```

The unification procedure corresponding to the only application in the expression refines the type variables v_1 and v_3.

$\mathcal{W}$ is not the only procedure which implements the Hindley-Milner type rules. A similar algorithm pushes down an expected type in the abstract syntax tree, and is therefore highly context-sensitive. This folklore algorithm, known as $\mathcal{M}$, has been formalized and proven correct with respect to the type rules by Lee and Yi [36].

Figure 2.5 displays algorithm $\mathcal{M}$. This algorithm only returns a substitution containing deductions made about type variables. Compared to $\mathcal{W}$, algorithm $\mathcal{M}$ introduces more type variables. Unification takes place for each identifier and each lambda abstraction: therefore, the potential locations where this algorithm can fail differ from $\mathcal{W}$'s locations. Lee and Yi [36] have proven that for ill-typed programs, $\mathcal{M}$ stops "earlier" than $\mathcal{W}$, that is, after inspecting fewer nodes of the abstract syntax tree.

$$\mathcal{M} :: (\mathit{TypeEnvironment}, \mathit{Expression}, \mathit{Type}) \rightarrow \mathit{Substitution}$$

$$\begin{array}{lll}
\mathcal{M}(\Gamma, x, \tau_1) & = \textbf{let } \forall\overline{a}.\tau_2 = \Gamma(x) & \\
 & \quad \textbf{in } \mathit{mgu}(\tau_1, [\overline{a} := \overline{\beta}]\tau_2) & \mathit{fresh}\ \overline{\beta} \\[1ex]
\mathcal{M}(\Gamma, \lambda x \rightarrow e, \tau) & = \textbf{let } S_1 = \mathit{mgu}(\tau, \beta_1 \rightarrow \beta_2) & \mathit{fresh}\ \beta_1, \beta_2 \\
 & \qquad\quad S_2 = \mathcal{M}(S_1\Gamma\backslash x \cup \{x : S_1\beta_1\}, e, S_1\beta_2) & \\
 & \quad \textbf{in } S_2 S_1 & \\[1ex]
\mathcal{M}(\Gamma, e_1\ e_2, \tau) & = \textbf{let } S_1 = \mathcal{M}(\Gamma, e_1, \beta \rightarrow \tau) & \mathit{fresh}\ \beta \\
 & \qquad\quad S_2 = \mathcal{M}(S_1\Gamma, e_2, S_1\beta) & \\
 & \quad \textbf{in } S_2 S_1 & \\[1ex]
\mathcal{M}(\Gamma, \textbf{ let } x = e_1 \textbf{ in } e_2, \tau) & = \textbf{let } S_1 = \mathcal{M}(\Gamma, e_1, \beta) & \mathit{fresh}\ \beta \\
 & \qquad\quad \sigma\ \ = \mathit{generalize}(S_1\Gamma, S_1\beta) & \\
 & \qquad\quad S_2 = \mathcal{M}(S_1\Gamma\backslash x \cup \{x : \sigma\}, e_2, S_1\tau) & \\
 & \quad \textbf{in } S_2 S_1 &
\end{array}$$

Figure 2.5. Folklore type inference algorithm $\mathcal{M}$

Most existing compilers use a hybrid algorithm for type inference, which is (in essence) a combination of algorithm $\mathcal{W}$ and algorithm $\mathcal{M}$. Algorithm $\mathcal{G}$ [37] is a generalization of these two algorithms. This generalized algorithm is particularly interesting since each instance behaves differently for incorrect expressions. Note that the constraint-based type inference algorithm we present in Chapter 4 generalizes $\mathcal{G}$, and, as a result, also $\mathcal{W}$ and $\mathcal{M}$.

2.5 Haskell implementations

Although the type inference framework presented in this thesis applies to type systems based on Hindley-Milner, we assume that the reader is comfortable with the programming language Haskell, and in particular with the Haskell 98 standard [49]. We make use of Haskell's syntax in examples and code fragments. For further reading, we suggest one of the many introductory textbooks on Haskell [29, 63, 6].

Two widely-used Haskell implementations are the Hugs interpreter [35] and the Glasgow Haskell Compiler (GHC) [18]. The small-scale interpreter Hugs is very suitable in the development of programs, and is frequently used in educational settings. GHC, on the other hand, is an industrial strength compiler, able to generate efficient code. This compiler comes with a large collection of libraries, and it supports several extensions to the language.

More recently, the Helium compiler [26] has been developed at the Utrecht University, which supports almost all of the Haskell 98 standard. This compiler has been designed especially for learning (and teaching) the language, and has been used in a number of introductory courses on functional programming. The type inference algorithm used in the Helium compiler has grown out of the research on the constraint-based type inference framework we describe in the following chapters.

In contrast with the Haskell 98 specification, we have deliberately chosen not to apply implicit overloading for numeric constraints. Hence, the expression $1 + 2.0$ is not accepted by the Helium compiler.

3

Literature on type errors

Overview. *We present a summary of the approaches that have been suggested in the literature for improving the quality of type error messages. The techniques are classified in a number of categories, and we discuss the advantages and disadvantages of each paradigm.*

A considerable amount of attention has been paid to improve the type error messages reported by Hindley-Milner type systems [27, 42]. Among the first proposals was a paper presented by Wand [64] at the Symposium on Principles of Programming Languages (POPL) in 1986. He proposed to trace reasons for type deductions during type inference, from which an explanation can be generated. At the same conference, Johnson and Walz [30] suggested that a compiler should report the most likely source of error. A number of heuristics select which location is reported.

The philosophy of these two influential papers are quite contradictory: the former traces *everything* that contributes to an error, whereas the latter attempts to pinpoint the *most likely* mistake. More recently, two doctoral theses were fully dedicated to the subject of type error messages. Yang [66] continued in the direction of Wand, and studied human-like explanations of polymorphic types in detail. At the same time, McAdam [41] proposed techniques to automatically correct ill-typed programs. Despite all the effort and the number of proposals, the current situation is still far from satisfactory: most implementations of modern compilers for functional languages are still suffering from the same old problems concerning type errors. The shortcomings of these compilers become most apparent when people are introduced to functional programming and have to deal with the reported type error messages. Recent extensions to the type system, such as multi-parameter type classes and rank-n polymorphism, have an additional negative effect on the quality of the type error reports.

Numerous approaches have been proposed over the last couple of years. The approaches can be classified in the following categories.

- *Order of unification.* Change the order in which types are unified, and thereby change the location where a type inconsistency is first detected. Alternative traversals over the abstract syntax tree have been defined, and also unification strategies to remove the left-to-right bias.
- *Explanation systems.* Describe why certain types have been inferred, for instance, in a text-based setting. A typical approach is to modify the type in-

ference algorithm such that it keeps track of information that is required to construct the explanation.

- *Reparation systems.* Report the most likely source of a type inconsistency. Heuristics are applied to discriminate between the candidate locations. More advanced systems do not only suggest where the program should be corrected, but also which program transformation would resolve the type error.
- *Program slicing.* Enumerate all the locations that contribute to a type conflict. Instead of the standard textual output, these systems often rely on other visualization techniques such as highlighting of program parts, for instance in an editor.
- *Interactive systems.* Let the user debug a program and find his mistake in an interactive session. There are various forms of interaction, such as inspection of inferred types, or questioning the programmer to find a function which is assigned an unintended type.

The rest of this chapter is organized as follows. First, the five classes of approaches are presented, and for each class the advantages and disadvantages are discussed. Some remaining proposals that do not fit in either one of the categories are discussed in Section 3.6. Finally, Section 3.7 provides an overall summary.

3.1 Order of unification

The most straightforward approach to change the error reporting behavior of a type inferencer is to modify the order in which types are unified. Typically, a type error is reported at the location where unification fails. Thus, by changing the unification order one can delay or speed up the detection of inconsistencies. Most type inference algorithms, including $\mathcal{W}$ and $\mathcal{M}$, traverse the abstract syntax tree of an expression in a predetermined order. As a consequence, subexpressions are treated asymmetrically, and often the left subexpression is considered before the subexpression on the right. This leads to the infamous left-to-right bias, which means that the algorithm has the tendency to report type errors towards the end of a program. Hugs in particular suffers from this bias, because it traverses some language constructs (for instance tuples) from right to left.

Lee and Yi (1998, 2000)
Lee and Yi [36] give a formal definition of the folklore, top-down type inference algorithm $\mathcal{M}$, which is an alternative for the standard $\mathcal{W}$ algorithm. Because an expected type is passed to the recursive calls, this algorithm is highly context aware.

In another paper, Lee and Yi present a generalized algorithm $\mathcal{G}$ [37], which has both $\mathcal{W}$ and $\mathcal{M}$ as instances. Other instances are the algorithms used in the SML-NJ and OCaml compilers. A correctness proof is given for this generalized algorithm, which implies soundness and completeness for all instances. Furthermore, they claim that some instances stop earlier in the presence of a type error than others. In particular, the top-down algorithm $\mathcal{M}$ detects a type inconsistency earlier than the standard algorithm $\mathcal{W}$. Earliness is defined with respect to the number of recursive calls made by the algorithm, which does not necessarily reflect the number of unifications that have been performed, nor the amount of work that has been done. Lee and Yi state that the two extreme algorithms ($\mathcal{W}$ and $\mathcal{M}$) are not suitable for producing understandable type error messages, but instead one should use a hybrid algorithm (a combination of the two) in practice.

Algorithm $\mathcal{G}$ could be defined even more generally. One possible direction, also mentioned by the authors, is to vary the constraining of the types in the type environment Γ as proposed by McAdam [38]: his proposal will be discussed next. A second direction is to parameterize the order in which the subexpressions are visited (for instance right-to-left instead of the standard left-to-right visits), or to depart completely from the abstract syntax tree. The correctness of these orderings can be validated easily in a constraint-based setting (see Chapter 5). However, regardless of the ordering one prefers, one fundamental problem remains: because such an algorithm proceeds in some specific order, it is not difficult to come up with a counter-example such that the type inconsistency is reported at an undesirable location.

McAdam (1998)

McAdam [38] describes how to modify existing inference algorithms to remove the left-to-right bias. To remove this bias, subexpressions have to be treated symmetrically. The subexpressions can be visited independently, but then the resulting substitutions have to be unified afterwards. Algorithm U_S is presented, which returns a most general unifier for a pair of substitutions. A modification to the standard algorithm $\mathcal{W}$ is suggested which removes the left-to-right bias in the reported type error messages. The implementation of algorithm $\mathcal{W}$ can be left unmodified, except for the case of an application. The resulting algorithm is symmetric, hence the name $\mathcal{W}^{SYM}$. The correctness of this modification has been proven [41]. Note that other asymmetries can be removed similarly for other language constructs. Likewise, $\mathcal{M}^{SYM}$ is a modified version of the folklore algorithm $\mathcal{M}$. Figure 3.1 shows the locations at which the inference algorithms detect an inconsistency for the given expression.

Discussion

An unfortunate effect of algorithms with a fixed unification order is that it is easy to construct examples for which the reported error site is far from the actual error. Another peculiarity is that inference algorithms that are in use by current compilers are hybrid: some types are pushed downwards, while in other places type information flows upwards. This can make the outcome of the type checking process even more mysterious to the average user, as it is no longer clear which parts of the program have been inspected by the compiler, and, hence, which parts may contribute to the inconsistency. Symmetrical treatment of subexpressions seems to be a promising alternative since it is less arbitrary in reporting an error site, although the reported sites are higher in the abstract syntax tree, and are thus less precise.

$$\lambda f \rightarrow \underbrace{(\ f\ 3\ ,\ \overbrace{f\ \underbrace{\mathit{True}}_{\mathcal{M}}}^{\mathcal{W}}\ )}_{\mathcal{W}^{SYM}\ \mathit{and}\ \mathcal{M}^{SYM}}$$

Figure 3.1. Error sites of various inference algorithms

3.2 Explanation systems

A different approach to gain insight in a type inconsistency is to explain *why* types have been inferred. A common approach is to record reasons for type deductions during unification. When unification fails, there is a trace available from which an explanation can be constructed. Most inference algorithms can be extended in a straightforward way to keep track of these reasons. We will see later that nicely presenting such a trace is a greater challenge.

Wand (1986)
A type inference algorithm detects a type error when it cannot proceed any further. Reporting this location of detection can be misleading as the source of the actual problem might be somewhere else. Wand [64] proposes a modified unification algorithm to keep track of the reasons for type deductions. For the simple lambda calculus, each type deduction is caused by some application. Although the output of the algorithm to explain a type inconsistency is repetitive and too verbose to be really helpful, many scientists continued their research in this direction, which resulted in several improvements of Wand's original proposal.

Soosaipillai (1990)
Soosaipillai [58] describes a facility which provides a stepwise explanation for inferred types in Standard ML. This tool, which has been implemented, lets a user ask *why* a type has been inferred. During type inference, explanations for all the subexpressions are recorded in a global list, which can later be inspected by a menu driven traversal. One complication of this tool is that the information is structured in a bottom-up fashion, which makes that results have to be remembered to comprehend the reasoning later on. Furthermore, only correct programs can be inspected with this tool.

Beaven and Stansifer (1993)
Beaven and Stansifer [4] follow Wand's approach to maintain information about each type deduction. Their system offers an explanation *why* an expression is assigned a certain type, and *how* a type is deduced for a particular type variable. They also discuss the let construct, which significantly complicates the generation of concise explanations. The generated explanations are presented in plain English,

which makes them lengthy and verbose. A second (and unavoidable) problem is that local type variables are used to explain some dependencies.

Bernstein and Stark (1995)
Bernstein and Stark [5] present an algorithm which can infer principal types for open expressions, that is, expressions containing free identifiers. As opposed to the type environment Γ which is passed top-down to supply the type of an identifier, an assumption environment is constructed bottom-up which contains the monomorphic type of each identifier. In the end, the inferred (monomorphic) types of the unbound variables are reported. These reported types can give some insights in type information from inside the program. Note that by constructing the assumption environment bottom-up, the left-to-right bias has disappeared. Although a prototype has been implemented, the ideas have not been tested in practice.

Duggan and Bent (1996)
Duggan and Bent [14] use Wand's concept: they propose a modification of algorithm $\mathcal{W}$ to record why a type has been assigned to a type variable. However, they refine Wand's algorithm by considering the aliasing of type variables, that is, internal type variables which turn out to be equal. As a result, the explanations produced by their system are simpler than explanations that are constructed while traversing the abstract syntax with algorithm $\mathcal{W}$. More importantly, the explanations correspond better to the reasoning of a human expert. Despite the improvements, the reported type traces are still lengthy and hard to comprehend. A prototype of the proposed algorithm has been implemented.

Yang (2000)
The location where a type error is detected is often not the location where the program should be modified. To tackle this problem, Yang [65] discusses two algorithms that report two conflicting program sites (both contributing to the inconsistency) instead of just one. Firstly, an algorithm that unifies assumption environments ($\mathcal{U}_{\text{AE}}$) is discussed, which was suggested by Agat and Gustavsson. The main idea is to handle the two subexpressions of an application independently to remove the left-to-right bias present in most algorithms. The algorithm uses two type environments. One type environment is passed top-down (inherited) and contains all the predefined functions. The types of all the other identifiers are recorded in an assumption environment, which is constructed bottom-up (synthesized). The only reason to have the predefined types in a separate environment is to gain efficiency. Figure 3.2 shows an error message reported by the $\mathcal{U}_{\text{AE}}$ algorithm containing two conflicting sites.

A second algorithm that is proposed is an incremental error inference ($\mathcal{IEI}$) algorithm, which just combines two inference algorithms. At first, an expression is checked with the $\mathcal{U}_{\text{AE}}$ algorithm. If this fails (an inconsistency is detected), then the folklore top-down algorithm $\mathcal{M}$ is used to find another site that contributes to the error.

```
fn x => map x [x+2]
```

```
Type conflicts in subexpressions:
map (x)
+(x, 2)

the common program variables have type conflicts in different sites
from the first expression
x: 'c -> 'd

but from the second expression
x: int
```

Figure 3.2. Two conflicting sites reported by Yang's $\mathcal{U}_{\mathrm{AE}}$ algorithm

Yang, Michaelson, and Trinder (2002)

Algorithms to infer polymorphic types are well-known, but these algorithms are not a suitable basis for explanation generation. The standard algorithms, on the one hand, proceed in a fixed order, introducing intermediate type variables for types that are not yet known. On the other hand, human experts use different techniques if they are asked to explain the type of an expression. Yang, Michaelson, and Trinder [68] have conducted experiments with programmers to identify human type inference techniques. For instance, experts focus on the concrete types of literals and known operators. Partial types, or type skeletons, are constructed instead of complete types. Another inference technique that is applied by experts is a two-dimensional inspection of the code. For example, patterns on the same argument position should have the same type. Experts tend to avoid using type variables in their explanation.

Yang, Michaelson, and Trinder [68] designed a new inference algorithm $\mathcal{H}$ that stores human-like explanations for the types that are inferred, and which mimics the identified human inference techniques. Their algorithm is built on top of $\mathcal{U}_{\mathrm{AE}}$, and the generated explanations are both succinct and non-repetitive. Figure 3.3 shows a human-like explanation for the inferred type of `twice`'s well-typed definition.

Discussion

At first, it might seem to be attractive to produce explanations as a human expert would do. However, in general it is very difficult to generate such a helpful textual explanation for a type inconsistency. The following problems are inherent to this approach.

- Choosing the right level of detail for explanations is a delicate matter as it depends on the expertise of the user and the complexity of the problem at hand. Explanations that are too detailed can be overwhelming by the amount of information that they contain. The task to extract the helpful parts is left to the user. As a result, such explanations are likely to be ignored completely. On the other hand, systems that do not show everything risk to leave out important

```
val twice = fn x => x*2
```

```
"twice" and "fn x => x * 2" have the same type
 - LHS and RHS of "val twice = fn x => x * 2"
1: "fn x => x * 2" is a function type: int -> int
Argument 1: int
2: the Bound variable "x" and "2" have the same type in "x * 2"
   - arguments/result of "*" have the same Number types
3: "2": int
Result: int
4: Function body "x * 2": int
   - result/arguments of "*" have the same Number types
   - see 2 above
```

Figure 3.3. Yang's explanation for `twice`'s inferred type

parts of an explanation. Generated explanations also have the tendency to be repetitive.

- Typically, explanations follow the underlying type checking algorithm, and, in some cases, knowledge about the algorithm is necessary to comprehend a report. In particular, the type variables that are introduced for polymorphic functions play a key role in understanding the problem. Inevitably, such type variables do show up in type error explanations. Explanations that contain numerous type variables are, however, hard to follow.
- This approach does not scale well to larger and more complicated programs as the size of an explanation will also increase rapidly.

Despite all this, an explanation system can be a valuable tool if it is used in combination with other techniques.

3.3 Reparation systems

This section describes type inference algorithms that try to report the most appropriate location for a type inconsistency. One site is selected from a list of candidate locations, because it is believed to be the location of the mistake. Heuristics can be employed to guide this selection. These heuristics can be straightforward, such as the isolation of a minority, or more advanced, such as recognizing common mistakes.

Johnson and Walz (1986)

Johnson and Walz [30] recognize the importance of error detection and correction in systems that are based on unification, especially since inconsistencies are typically detected far from the site of the actual mistake. Their motivation is an ML-oriented editor, in which they highlight exactly what contributed to an error. Because they observed that in most cases a variable is used in a correct way, and only a small number of uses are conflicting, they draw the user's attention to the location that appears to be the most likely source of the error. This is achieved by having different levels of intensities available for highlighting. Type constraints are used to express unification, and a maximum flow technique is used to determine the most likely cause of an inconsistency.

Gandhe, Venkatesh, and Sanyal (1996)

Gandhe, Venkatesh, and Sanyal [17] present an algorithm to suggest corrections for ill-typed terms. A correction is the replacement of a subterm by a free identifier. They use the inference algorithm of Wand and the maximal consistent subset algorithm given by Cox [10], after which they compute a set of minimal corrections. However, their system cannot handle polymorphism, but just covers the simple lambda calculus.

McAdam (2001)

One of the shortcomings of reported type error messages is that they do not advise the programmer how to fix the problem. McAdam [40] proposes to use a theory of unification modulo linear isomorphisms to repair type errors automatically. If unification fails, then we search for a pair of morphisms to witness that the two types are in fact equal up to isomorphism. These morphisms are normal lambda terms: for instance, the morphism $\mu = \lambda f\ (x, y) \rightarrow f\ y\ x$ “uncurries” and flips the arguments of a function. Inserting the right morphism into the source results in a type correct program.

For example, take a look at the ill-typed expression $\mathit{map}\ ([1, 2, 3], \mathit{toString})$, and observe that applying the morphism μ to map yields a type correct program.[1] This correction can, in combination with a default type error message, be suggested to a user as a possible fix. Instead of suggesting a lambda term to a programmer, McAdam suggests to unfold these morphisms by performing some form of partial evaluation. For instance, $\mu\ \mathit{map}\ ([1, 2, 3], \mathit{toString})$ can be reduced to $\mathit{map}\ \mathit{toString}\ [1, 2, 3]$, which is exactly how a human expert would correct the type error. Figure 3.4 displays the possible fix that is suggested to the programmer.

There are still some obstacles to overcome before these techniques can be used in a real implementation. First of all, this approach relies heavily on the fact that we consider n-ary function applications, and not a sequence of binary applications, which is a more common representation. Secondly, their prototype implementation is built on top of algorithm $\mathcal{W}$. Morphisms can change the type of the whole application: swapping the arguments in $\mathit{const}\ 1\ \mathit{True}$ changes the type of the expression to Bool. Since $\mathcal{W}$ is not aware of the type expected by the context of the function application, a program fix that is suggested could transfer the type inconsistency to a location higher in the abstract syntax tree of the program. Finally, it is not clear whether partial evaluation always prevents the programmer from seeing the morphisms.

[1] The standard function map has the polymorphic type $\forall ab.(a \rightarrow b) \rightarrow [a] \rightarrow [b]$.

```
Try changing
   map ([1, 2, 3], Int.toString)
To
   map Int.toString [1, 2, 3]
```

Figure 3.4. McAdam's error message offering constructive advice

Discussion
Systems that are equipped with heuristics can help programmers significantly in pinpointing the location of the actual mistake. The heuristics can be designed with specific knowledge about the language and some common mistakes, and they can even be tailor-made for an intended group of users and their level of expertise. However, there is one obvious downside to this approach: if the heuristics are applied incorrectly, then the reported message can be quite misleading. The following arguments can be used against systems based on heuristics.

- The compiler cannot be aware of the programmer's intentions, and should therefore not automatically decide what is correct and what is not. A wrong judgement of the compiler, that does not match with the programmer's beliefs about his program, can lead to even more cryptic type error messages.
- The approach is based on the selection of a single error site, but often there are just two contradicting sites involved in a type error. Discriminating against one or the other seems to be wrong.

In the forthcoming chapters, we explore a type inference mechanism which includes heuristics to pinpoint the most likely source of a type inconsistency, and with support to suggest probable fixes for an ill-typed program.

3.4 Program slicing

A type inconsistency can also be reported as a set of contributing program points. Such a set of program points is called a program slice, and contains the location of the actual mistake. Hence, the parts that are not in the program slice do not require a further inspection by the user. To report a program slice effectively, one cannot use standard textual output. Instead, these systems use editors and other visualization tools to highlight and underline parts of a program.

Choppella and Haynes (1994, 2002)
Choppella and Haynes [9] present a logical framework to find program slices for a type error. They observed that most error reports present irrelevant information, or miss the crucial parts. The presented framework has been designed to determine in a precise way which parts contribute to a type inconsistency, and which parts do not. To achieve this, they apply techniques used for proof normalization. Furthermore, their framework makes use of a variation of the graphs proposed by Port [53]. These graphs are used to perform unification, which is called diagnostic unification

by the authors. Unfortunately, they do not truly address the problems introduced by let-polymorphism, but only sketch along which lines the framework should be generalized. The PhD thesis of Choppella [8] discusses the logical framework in more detail.

Dinesh and Tip (1997)
Dinesh and Tip [12] compute program slices for ill-typed programs that contain exactly the constructs that cause the inconsistency. They emphasize the importance of accurate positional information that is associated with a type error message, as this helps to determine the location where a change is required. Position information becomes even more important in the context of programming environments. Maintaining exact position information is hardly a scientific issue, but there is a good reason why most modern compilers do a poor job in keeping the locations: adding position information to the abstract syntax involves changes throughout the compiler, which make the implementation less readable. Dinesh and Tip solve this by generating a compiler (including a type inference algorithm) with precise positional information from a specification. Their proposed approach is language independent.

Haack and Wells (2003)
Haack and Wells [20] follow the approach of Tip and Dinesh, and compute a program slice (a set of program points) which contains everything that contributes to a type inconsistency. In addition, they present an algorithm to compute minimal type error slices by calculating minimal unsolvable constraint sets for a given set of equality type constraints. There is an online demo of a type error slicer for MiniML (a small sublanguage of Standard ML). Figure 3.5 shows a program slice for a type incorrect program. The upper half shows the complete source, the lower half is the program slice which contains precisely the parts of the program that contribute to the type inconsistency. The inconsistency is caused by a function type and an *Int* type that cannot be unified. The colors associated with these types are dark gray and black, respectively. These types arise from a function application and the numeric constant `1`, as indicated in both halves of the figure.

Discussion
Slicing systems depart from the traditional type error messages as they report all contributing program locations instead of the two conflicting types that are normally reported. The slices give a truthful impression to quickly discover the actual problem, especially since the presented information is complete, which is a truly beneficial property of such a system. However, it remains questionable whether this technique alone gives satisfactory results, or whether it should be considered a separate tool to be used in combination with other techniques. One disadvantage is that types do no longer relate to expressions. Thus, a user needs to be aware of the types present to compensate for this. Systems that use highlighting to present program slices introduce another subtlety. Highlighting conventions are necessary to precisely indicate what contributes to the error and what does not. Consider for

```
val average =
fn weight => fn list =>
let val iterator = fn (x,(sum,length)) => (sum+ weight   x,length+1)
    val (sum,length) = foldl iterator (0,0) list
in sum div length end

val find_best =
fn weight => fn lists =>
let val average = average weight
    val iterator = fn (list,(best,max)) =>
                     let val avg_list = average list
                     in if avg_list > max then
                          (list,avg_list)
                        else
                          (best,max)
                     end
    val (best,_) = foldl iterator (nil,0) lists
in best end

val find_best_simple = find_best 1
```

```
type constructor clash, endpoints: function vs. int.

(.. val average = fn weight => (.. weight   (..) ..)
.. val find_best = fn weight => (.. average weight ..)
.. find_best 1 ..)
```

Figure 3.5. A program slice produced by Haack and Wells

instance a function application. Since, in general, there is no special syntax for an application (it is *just a space* between expressions), it is important and far from trivial to decide what to highlight if the application itself contributes, but neither of its subexpressions does.

3.5 Interactive systems

Given that a compiler cannot be aware of a programmer's intentions and assumptions about his program, the only way to uncover these is through interaction. An interactive system can help a user to debug his program, and guide him to the location of the mistake. Most languages, for instance, provide ways to express the type of a part of the program via type declarations. Additional type information helps to find mismatches between the programmers assumptions about his program and the types inferred by the compiler

```
foldl f z []      = [z]
foldl f z (x:xs)  = foldl f (f z x) xs
flip f x y = f y x
reverse = foldl (flip (:)) []
palin xs = reverse xs == xs
```

```
Ex1.hs> :debug palin
reverse :: [a] -> [[a]]
Ex1.hs> is this type correct> n
flip :: (a -> b -> c) -> b -> a -> c
Ex1.hs> is this type correct> y
foldl :: (a -> b -> a) -> a -> [b] -> [a]
Ex1.hs> is this type correct> n
type error - contributing locations
foldl f z [] = [z]
foldl f z (x:xs) = foldl f (f z x) xs
```

Figure 3.6. A declarative debugging session with Chameleon

Rittri (1993)
Rittri [54] continues with Wand's approach, but states that not too much information should be given at once. He suggests an interactive system that helps a programmer in finding a mistake. Unfortunately, his ideas have not been implemented.

Huch, Chitil, and Simon (2000, 2001)
Huch, Chitil, and Simon describe Typeview [28]: an interactive tool to query the type of all expressions in a program. They suggest to report a set of possible types for a variable that cannot be typed, and claim that an interactive approach could give better support than complicated algorithms that are based on heuristics.

Chitil [7] argues that the most difficult part to understand why an ill-typed program is rejected by a compiler is to explain why a certain type has been assigned to an expression. He believes that an imprecise error location is unavoidable, and therefore suggests to let the programmer inspect and explore the assigned types in an interactive manner. Principal typings are used instead of the commonly used principal types. A typing consists of a type and a monomorphic type environment, and can be constructed completely bottom-up. As a result, the explanation of an assigned type does not rely on the context of the expression anymore, but is really compositional. Chitil implemented a prototype version for a small expression language, together with an interface for the systematic debugging of programs.

Stuckey, Sulzmann, and Wazny (2003, 2004)
Stuckey, Sulzmann, and Wazny [59] have produced the Chameleon system, which is an interactive programming environment to debug type errors. Their system supports Haskell with some extensions (such as functional dependencies), and can be used as a front-end to existing Haskell compilers. The typing problem is mapped

to a set of constraints. Attached to each type constraint is the program code that is responsible for the restriction. Constraint handling rules (or CHRs for short) are then used to manipulate the set of constraints at hand, that is, by either simplifying or propagating constraints. If the need arises, minimal unsatisfiable constraint sets are determined, which are then used to underline parts of the program that contribute to the type error. Constraints that are member of all minimal unsatisfiable constraint sets are doubly underlined: these are the most likely locations to find a mistake. Two other features for debugging are supported by Chameleon. An explanation why a certain type has been assigned to an expression can be requested, and the shape of a type can be explained. To construct such an explanation, a set of minimal implicants is computed. In addition, there is an interactive interface to let the programmer localize an error. Figure 3.6 shows an interactive debugging session to find the mistake.

In a more recent paper [60], the authors discuss techniques to enhance the Chameleon system. Additional information is extracted from the minimal unsatisfiable sets, which is used to generate more constructive error messages. For instance, in case of an ambiguous type scheme, a programmer could resolve the ambiguity by inserting some type annotations. To assist the programmer, Chameleon computes all locations in a program where such a type annotation could be inserted, and presents these locations as additional information in the reported error messages. Their system has the advantage that it offers users flexibility in selecting which heuristics to employ, and new heuristics can be included to enhance the type error reports. Devising better heuristics is an area of ongoing research.

Discussion
For most programs it is obvious which fix to the program is required, but it is the remaining minority which is time consuming to correct. In this light, interactive systems are a suitable tool to locate these hard-to-find mistakes in cases you really don't understand the source of the problem. Moreover, it is a standard debugging technique that can equally well be applied to other domains, and which does not make any prior assumptions. A different point of consideration is the kind of information that the user has to supply to the system. If the questions to be answered are too difficult – for instance, the type of a complex function is asked for – then a debugging session becomes overly complicated, and perhaps even more complex than the original problem. Another pitfall is that debugging sessions quickly become too elaborate and time consuming to be really effective.

3.6 Other approaches

Finally, we address some relevant papers that do not fit in any of the other categories.

Yang and Michaelson (2000)
Yang and Michaelson [67] describe a way to visualize types. At its simplest, each common base type is assigned a unique color, and special conventions are suggested

to deal with composite types, such as lists, Cartesian products, and function types. Although the idea of visualizing types is original and different from other proposals, there are some downsides involved. For example, problems arise when the types are polymorphic, that is, contain type variables. A more fundamental problem is that, in general, people find it harder to interpret a graphical visualization than the corresponding textual representation. Experiments (conducted by the authors) to compare the visualized types with text-based representations showed that the visualization did not contribute to a greater understanding, although they conjecture that combining the two approaches might achieve more than either can alone.

Yang, Michaelson, Trinder, and Wells (2000, 2001)
Yang and others [69, 66] suggest a manifesto to measure the quality of reported type error messages. According to this manifesto, a good error report should have the following properties.

1. *Correct.* An error message is emitted only for illegal programs, while correct programs are accepted without an error report.
2. *Accurate.* The report should be comprehensive, and the reported error sites should all be relevant and contribute to the problem. Moreover, to understand the problem, only these sites should be inspected.
3. *Intuitive.* It should be close to human reasoning, and not follow any mechanical inference techniques. In particular, internal type variables should be introduced with extreme care.
4. *Succinct.* An error report should maximize the amount of helpful information, and minimize irrelevant details. This requires a delicate balance between being too detailed and being too terse.
5. *Source-based.* Reported fragments should come from the actual source, and should not be introduced by the compiler. In particular, no desugared expressions should be reported.

A warning is issued for the emission of multiple errors: this advice is not included in the manifesto. Reporting several errors at once comes with the risk of generating a cascade of bogus error messages, although it does provide insight on the number of corrections that are needed.

McAdam (2000)
McAdam [39] defines a graph that captures type information of a program, and which can be used to generate type error messages. This graph follows the structure of types rather than the shape of the abstract syntax tree. His graphs can represent both well-typed and ill-typed expressions, and it can deal with unbound identifiers. More important, his work generalizes earlier approaches suggested by Bernstein and Stark, Wand, and Duggan and Bent, because their error reports can be generated from the information available in his graphs. However, it is not straightforward to read from a graph whether the program is well-typed or not. Furthermore, parts of the graph are duplicated to deal with let-constructs. As a consequence, the size of a graph can grow rapidly (exponentially in the size of the program), and, even

more unpleasant, duplication of inconsistent parts of the program will lead to new inconsistencies in the graph.

The graphs presented by McAdam have evident similarities to the type graph data structure that we present in Chapter 7. The algorithm proposed by McAdam constructs graphs directly from abstract syntax trees. We, on the other hand, follow a constraint-based approach: we generate constraints for an abstract syntax tree, and then construct a graph using the collected constraints. In fact, it is this extra level of indirection that paves the way for a more flexible and scalable type inference framework. The constraint-based approach also helps us to deal with let-polymorphism in a different way, without the need to duplicate parts of the graph.

Findler and others (2002)
The DrScheme programming environment [15] has been designed to let students gradually become familiar with the Scheme programming language. Language levels are introduced to tackle the typical problems one encounters when learning a language: these levels are syntactically restricted variants of Scheme. Because analyses become more straightforward for a smaller language, better error messages can be reported. An additional advantage is that beginners are not confronted with the more advanced features of a programming language.[2] A number of add-ons have been developed to further enhance the programming environment, such as an interactive static debugger. The project was initially targeted at students. Particularly interesting is the TeachScheme! project, which is a spin-off from the DrScheme project. This project focuses on introductory computer science curricula, and is not only in use at universities, but also at a large number of high schools.

Neubauer and Thiemann (2003, 2004)
Neubauer and Thiemann [43] describe a type system that is based on discriminative sum types. For example, the sum type $(\mathit{Int} \,;\, \mathit{Bool} \,;\, \beta)$ expresses that the type variable β is either *Int* or *Bool*. A flow analysis is used to propagate type information in a program. Parts of the program are marked as producer (or as consumer), which is used later to trace the origin of type constants. An advantage of a type system with discriminative sum types is that every term can be typed. In fact, an annotated program can be considered a principal description of all type errors that are present. Note that the proposed flow analysis solves only half the problem, since we still have to interpret a type derivation before we can construct a type error report. A nice property of this approach is that it decouples type derivation and error reporting, just as these are two separate issues in a constraint-based setting.

Recently, the authors have reported on the Haskell Type Browser [44], which is a tool to inspect discriminative sum types assigned to parts of a program. Given a Haskell program, an interactive XHTML page is generated, which displays the source program with some highlighting conventions. An extra window assists the user in navigating through the abstract syntax tree. It is unknown whether the

[2] For example, novice Haskell programmers have to deal with type error messages that report unresolved overloading from day one.

browser leads to a better understanding of type inconsistencies in programs, in particular in the case of a novice user. Clearly, the user has to be familiar with discriminative sum types, and needs some training to grasp the dependencies between the types assigned to parts of the program.

3.7 Summary

In this chapter, we have presented several alternatives to improve the quality of type error messages. The techniques are often complementary rather than exclusive, and their effectiveness is strongly subjected to personal preferences. Although an objective comparison is thus out of the question, it should be possible to scientifically compare these methods by investigating their effects on groups of programmers. Because of the diversity in the presented techniques, we pose a list of criteria to help select a suitable approach.

- *Target audience.* Type error messages have to match with the level of expertise of a typical user. For instance, tools designed for inexperienced users have a different set of requirements.
- *Output format.* Most error reporting tools impose restrictions on the format in which error messages are presented. Often, error reports are text-based (as opposed to graphical), which impedes other visualization techniques.
- *Interaction.* Some debugging techniques rely on some form of interaction with the programmer. Such a question and answer session can take the form of letting a user inspect the types that are assigned to parts of the program.
- *Heuristics.* It is debatable whether to consult a set of heuristics or not. The safest approach is to report all contributing program locations, which includes the location where a correction is required. Alternatively, we could use heuristics based on expert-knowledge to select and report the most likely cause of the problem. Taking this approach to the extreme leads to suggesting program fixes to a user (or even making the correction).
- *Primary or external.* Fortunately, the majority of type errors require little effort to correct, and for these cases, standard techniques work fine. Hence, we have to discriminate between error messages reported for the general case, and the techniques used by a tool dedicated to debug the hard-to-find cases.

4

Constraint-based type inference

Overview. *This chapter introduces a constraint-based approach to type inference. Various kinds of type constraints are presented for the Hindley-Milner type system, and special attention is given to dealing with polymorphism. Our constraint-based approach consists of two parts: bottom-up type rules for collecting type constraints, and an algorithm to solve the collected constraints. We prove the correctness of our approach.*

We introduced the Hindley-Milner type system in Chapter 2. This system consists of two parts: a set of deduction rules formalizes which types are valid for a given expression (in a certain context), and a type inference algorithm that assigns types to expressions according to these deduction rules. The type rules are the specification of the type system, whereas type inference algorithms are actual implementations. Various algorithms can implement the same set of deduction rules. For instance, both $\mathcal{W}$ and $\mathcal{M}$ implement the Hindley-Milner type rules. These algorithms are equivalent for well-typed programs, but may react differently to incorrect input.

Typically, type inference algorithms proceed by introducing fresh type variables for various parts of an expression. The type variables function as place-holders for types that are still to be computed. At the same time, a substitution is maintained which maps the type variables to a more refined type. A unification algorithm is used to record equality between two types by finding a most general unifier. This is necessary to meet the restrictions introduced by the type rules. Such a unifier is incorporated into the substitution. Inconsistencies can arise at these unification points for ill-typed expressions. An alternative approach is to delay the (possibly failing) unifications, but capture these in a type constraint instead. Hence, collecting the type constraints cannot fail, although the set of collected type constraints may be inconsistent. What are the advantages of first collecting type constraints and solving them afterwards?

- Separation of concerns: specification and implementation are two separate issues. A specification is expressed by enumerating which type constraints are collected without worrying how to solve these constraints. For a collected constraint set, multiple constraint solvers can be available to find a solution that meets the imposed constraints.
- The only difference between the algorithms $\mathcal{W}$ and $\mathcal{M}$ is the order in which types are unified. The unification order can be tuned by choosing an ordering

of the type constraints. As a result, this approach will not only generalize these two well known algorithms, but also numerous other variants.

- Separating constraint collection and constraint solving results in a high level of control in case the set is unsatisfiable. Reasoning over a large set of constraints helps to produce better and more informative type error messages.

The idea to formulate a type system in terms of type constraints is not new. Pierce [52] dedicates one section of his textbook on type systems to constraint-based typing, and presents type rules for a simple expression language without polymorphism. Aiken and Wimmers [2, 1] propose a constraint solving system that can handle inclusion constraints over type expressions. The Hindley-Milner type system, including let-polymorphism, can be formulated in terms of these constraints, although the treatment of polymorphism is not completely satisfactory (see Section 4.2). Another interesting direction is the HM(X) framework, presented by Odersky, Sulzmann and Wehr [46]. This framework is a constraint-based formulation of the Hindley-Milner system including let-polymorphism, which is parameterized over some constraint system X. All kinds of extensions to the type system can be captured via this abstraction. In principle, the HM(X) framework and the constraint system of Aiken and Wimmers resemble the system that is introduced in this chapter. However, since our main interest is producing clear error messages for ill-typed expressions, we will follow a different direction, in particular in the treatment of let-polymorphism.

The organization of this chapter is as follows. We start with a discussion on constraint-based type rules for the lambda calculus (Section 4.1). Then we extend the lambda calculus with let expressions in Section 4.2. This extension introduces polymorphism, and we compare three different approaches to deal with this. One of the presented approaches is adopted in Section 4.3, which presents a set of bottom-up type rules to collect type constraints. The collected constraints can be solved with an algorithm presented in Section 4.4. Section 4.5 concludes this chapter with a correctness proof of the proposed constraint-based type inference process.

4.1 Type constraints

We start by exploring constraint-based type rules for the lambda calculus without let-polymorphism. Equality constraints are introduced to capture unification of types.

Type constraint:

$$c ::= \tau_1 \equiv \tau_2 \qquad \textit{(equality)}$$

An equality constraint expresses that two types should eventually become the same. Types can be specialized by applying a substitution. Let S be a substitution. We say that S satisfies the equality constraint $(\tau_1 \equiv \tau_2)$, denoted by $S \vdash_s \tau_1 \equiv \tau_2$, if $S\tau_1$ and $S\tau_2$ are syntactically equivalent.

$\boxed{\Gamma \vdash_{\Downarrow} e : \tau}$ *Top-down typing*

$$\frac{\Gamma(x) \equiv \beta}{\Gamma \vdash_{\Downarrow} x : \beta}\ (\text{SD-Var}) \qquad \frac{\begin{array}{c}\tau_1 \equiv \tau_2 \rightarrow \beta \\ \Gamma \vdash_{\Downarrow} e_1 : \tau_1 \quad \Gamma \vdash_{\Downarrow} e_2 : \tau_2\end{array}}{\Gamma \vdash_{\Downarrow} e_1\ e_2 : \beta}\ (\text{SD-App})$$

$$\frac{\Gamma \backslash x \cup \{x : \beta\} \vdash_{\Downarrow} e : \tau}{\Gamma \vdash_{\Downarrow} \lambda x \rightarrow e : (\beta \rightarrow \tau)}\ (\text{SD-Abs})$$

Figure 4.1. Simple top-down constraint-based type rules

Definition 4.1 (Constraint satisfaction).

$$S \vdash_s (\tau_1 \equiv \tau_2) \ =_{def}\ S\tau_1 = S\tau_2 \qquad (\vdash_s \equiv)$$

Likewise, we define constraint satisfaction for constraint sets.

$$S \vdash_s \{c_1, \ldots, c_n\} \ =_{def}\ S \vdash_s c_1 \ \wedge\ \ldots\ \wedge\ S \vdash_s c_n \qquad (\vdash_s\ set)$$

From a different perspective, one can think of a substitution as a set of equality constraints in a solved form.

Let us take a look at the first set of constraint-based type rules, shown in Figure 4.1. A top-down typing, written as $\Gamma \vdash_{\Downarrow} e : \tau$, expresses that expression e can be assigned type τ under a type environment Γ. Since we ignore polymorphism for the moment, we assume that Γ contains only monotypes. The downward arrow in the turnstyle is a reminder for the direction of the type environment (not to be confused with the direction of the type). The type rule (SD-Var) for a variable x is straightforward: find the (monomorphic) type of x in the type environment Γ, and equate this with a newly introduced type variable β. This type variable is also returned as the type of the variable; of course, this is effectively the same as returning the type found in Γ for x. The type rule (SD-App) for function application is more interesting, as it is the equation ($\tau_1 \equiv \tau_2 \rightarrow \beta$) that relates the types of the two subexpressions and the result type. Normally, the type of the function and the type of the argument are related by writing types that share type meta-variables. In the type rule (SD-Abs) for lambda abstraction, the type environment that is used to type the body of the abstraction is extended with a variable x, which is assigned a fresh type variable β. In the conclusion, we have a function type from this β to the type assigned to e. An alternative would be to introduce a second fresh type variable, say β', generate the constraint ($\beta' \equiv \beta \rightarrow \tau$), and return the type variable β' instead. In this way we make no assumptions about the types in the premises (except for the constraints that should hold), and neither do we for the type appearing in the conclusion.

Example 4.1. Consider the function *twice* $= \lambda f \rightarrow \lambda x \rightarrow f\ (f\ x)$. Applying the type rules results in the following derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{v_0 \equiv v_2}{\Gamma \vdash_{\Downarrow} f : v_2}\,\text{(SD-Var)} \qquad v_2 \equiv v_5 \rightarrow v_6 \qquad \dfrac{\dfrac{v_0 \equiv v_3}{\Gamma \vdash_{\Downarrow} f : v_3}\,\text{(SD-Var)} \quad v_3 \equiv v_4 \rightarrow v_5 \quad \dfrac{v_1 \equiv v_4}{\Gamma \vdash_{\Downarrow} x : v_4}\,\text{(SD-Var)}}{\Gamma \vdash_{\Downarrow} f\ x : v_5}\,\text{(SD-App)}}{\Gamma = \{f : v_0, x : v_1\} \vdash_{\Downarrow} f\ (f\ x) : v_6}\,\text{(SD-App)}}{\{f : v_0\} \vdash_{\Downarrow} \lambda x \rightarrow f\ (f\ x) : (v_1 \rightarrow v_6)}\,\text{(SD-Abs)}}{\emptyset \vdash_{\Downarrow} \lambda f \rightarrow \lambda x \rightarrow f\ (f\ x) : (v_0 \rightarrow v_1 \rightarrow v_6)}\,\text{(SD-Abs)}$$

This derivation tree contains five equality constraints. We can give a substitution S that satisfies each of the five constraints (where α is a new type variable).

$$S = [\ v_0, v_2, v_3 := \alpha \rightarrow \alpha\ \ ,\ \ v_1, v_4, v_5, v_6 := \alpha\ ]$$

Note that this is a minimal substitution: $S \sqsubseteq S'$ holds for any substitution S' that satisfies the five type constraints. The inferred type for *twice* is $S(v_0 \rightarrow v_1 \rightarrow v_6) = (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$.

There is a slightly different, but equivalent, set of type rules. The previous set of type rules makes use of a type environment Γ which associates a type with each identifier that is in scope. For instance, in the type rule (SD-Var), we write $\Gamma(x)$ to get the type assigned to the identifier x. Moreover, x must be present in the type environment, or else we have found an unbound variable. This type environment is computed and passed on in a top-down fashion. The type rules can also be formulated in terms of an assumption set, denoted by $\mathcal{A}$. This assumption set contains *all* the types that have been assigned to a (yet unbound) identifier, and this assumption set is constructed in a bottom-up way. The assumption set collected for a closed lambda term is empty. A substantial difference is that such an assumption set can contain multiple assertions about a variable, which makes no sense in the traditional top-down type environment.

Consider the type rules in Figure 4.2. The type rule for a variable is almost trivial: a fresh type variable is returned, and the fact that this type variable is assigned to the variable at hand is recorded in the assumption set. The type rule for function application is similar to the top-down type rule, except that we combine the two assumption sets constructed for the subexpressions by taking the union. In the case of a lambda abstraction, we remove all the occurrences that are bound by this abstraction from the assumption set, and we equate each type associated with this identifier to β, which is a fresh type variable.

$\boxed{\mathcal{A} \vdash_{\Uparrow} e : \tau}$ *Bottom-up typing*

$$\dfrac{}{\{x{:}\beta\} \vdash_{\Uparrow} x : \beta}\ (\text{SU-VAR}) \qquad \dfrac{\begin{array}{c}\tau_1 \equiv \tau_2 \to \beta \\ \mathcal{A}_1 \vdash_{\Uparrow} e_1 : \tau_1 \qquad \mathcal{A}_2 \vdash_{\Uparrow} e_2 : \tau_2\end{array}}{\mathcal{A}_1 \cup \mathcal{A}_2 \vdash_{\Uparrow} e_1\ e_2 : \beta}\ (\text{SU-APP})$$

$$\dfrac{\mathcal{A} \vdash_{\Uparrow} e : \tau \qquad \{\beta \equiv \tau' \mid x{:}\tau' \in \mathcal{A}\}}{\mathcal{A} \backslash x \vdash_{\Uparrow} \lambda x \to e : (\beta \to \tau)}\ (\text{SU-ABS})$$

Figure 4.2. Simple bottom-up constraint-based type rules

Example 4.1 (continued). Consider the derivation for *twice* using the bottom-up type rules.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{f{:}v_0\} \vdash_{\Uparrow} f : v_0}\ (\text{SU-VAR}) \quad v_0 \equiv v_3 \to v_4 \quad \dfrac{\dfrac{}{\{f{:}v_1\} \vdash_{\Uparrow} f : v_1}\ (\text{SU-VAR}) \quad v_1 \equiv v_2 \to v_3 \quad \dfrac{}{\{x{:}v_2\} \vdash_{\Uparrow} x : v_2}\ (\text{SU-VAR})}{\{f{:}v_1, x{:}v_2\} \vdash_{\Uparrow} f\ x : v_3}\ (\text{SU-APP})}{\{f{:}v_0, f{:}v_1, x{:}v_2\} \vdash_{\Uparrow} f\ (f\ x) : v_4 \qquad v_2 \equiv v_5}\ (\text{SU-APP})}{\{f{:}v_0, f{:}v_1\} \vdash_{\Uparrow} \lambda x \to f\ (f\ x) : (v_5 \to v_4) \quad v_0 \equiv v_6 \quad v_1 \equiv v_6}\ (\text{SU-ABS})}{\emptyset \vdash_{\Uparrow} \lambda f \to \lambda x \to f\ (f\ x) : (v_6 \to v_5 \to v_4)}\ (\text{SU-ABS})$$

The types and equality constraints found in this derivation are the same as the derivation with the top-down type rules, except for the renaming of type variables.

4.2 Dealing with polymorphism

An equality constraint captures unification of types. The constraint language must be extended with other kinds of constraints to deal with polymorphism.

Example 4.2. Consider the expression $idid = \textbf{let}\ id = \lambda x \to x\ \textbf{in}\ id\ id$, which applies the identity function to itself. This expression is well-typed because the function *id* is assigned the polymorphic type $\forall\alpha.\alpha \to \alpha$. This type scheme is found by generalizing the type inferred for $\lambda x \to x$, and is instantiated for both uses of *id* in the body of the let expression.

Polymorphism in the type language poses two challenges in a constraint-based setting. Firstly, we need to generalize types to type schemes, and instantiate these type schemes. Somehow, we must be able to express these two operations in our constraint language. Secondly, let expressions pose restrictions on the order in which

$\boxed{\mathcal{C}, \Gamma \vdash_{\Downarrow} e : \tau}$ *Top-down typing (alternative 1)*

$$\frac{\Gamma(x) = (\forall \overline{a}.\mathcal{C} \Rightarrow \tau) \qquad \beta' \equiv [\overline{a} := \overline{\beta}]\tau}{[\overline{a} := \overline{\beta}]\mathcal{C}, \Gamma \vdash_{\Downarrow} x : \beta'} \quad \text{(SD-VAR)}$$

$$\frac{\begin{array}{c}\sigma = \mathit{generalize}(\Gamma, \mathcal{C}_1 \Rightarrow \tau_1) \\ \mathcal{C}_1, \Gamma \vdash_{\Downarrow} e_1 : \tau_1 \qquad \mathcal{C}_2, \Gamma \backslash x \cup \{x : \sigma\} \vdash_{\Downarrow} e_2 : \tau_2\end{array}}{\mathcal{C}_1 \cup \mathcal{C}_2, \Gamma \vdash_{\Downarrow} \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2} \quad \text{(SD-LET)}$$

Figure 4.3. Type rules for dealing with polymorphism (alternative 1)

type inference may proceed. More specifically, the polymorphic type of a local definition must be inferred before we can proceed with typing the body.

As it is not obvious how one can express polymorphism using type constraints, we discuss and compare three alternative formulations. Each approach comes with a different view on type schemes.

Alternative 1: qualification of equality constraints

In a constraint-based setting, we can assign the type $v_0 \rightarrow v_1$ to the expression $\lambda x \rightarrow x$ under the condition that the constraint $v_0 \equiv v_1$ is satisfied. The type variables v_0 and v_1 are only internal place-holders, and do not appear outside the typing judgements for the given expression. Hence, we can generalize over these type variables, and associate the qualified type scheme $\forall ab.(a \equiv b) \Rightarrow a \rightarrow b$ with *id*. Of course, this scheme can be simplified to the more familiar scheme $\forall a.a \rightarrow a$, which can be seen as a normal form of the former type scheme. However, we will refrain from scheme normalization until the constraint solving phase: solving the constraints at this point, or doing some kind of normalization, would break the separation of constraint generation and constraint resolution. The type schemes that are needed for this approach have the following shape.

Type scheme (for the first alternative):

$$\begin{array}{lll} \sigma ::= & \forall a.\sigma & \textit{(polymorphic type scheme)} \\ \quad\;\; \mid & \mathcal{C} \Rightarrow \tau & \textit{(qualified type)} \end{array}$$

The constraint set $\mathcal{C}$ that is part of the qualified type consists of zero or more equality constraints.

Because we intend to qualify over the constraints that are collected for a local definition, we include the set of collected type constraints in the top-down typing judgement. The type rules for function application and lambda abstraction can be left unchanged. Figure 4.3 presents two (top-down) type rules that reflect the intention of this approach.

We have to change the type rule for a variable: the type scheme that is associated with the identifier at hand in the type environment is instantiated, which yields an instantiated constraint set and an instantiated type. In the type rule for a let expression, a type scheme is constructed from the type assigned to the local definition and the collected constraints. The type environment that is used to type the body of the let expression includes this type scheme.

The type rule (SD-Let) has one issue which might not be apparent at first sight. The constraints collected for the definition, in the type rule denoted by $\mathcal{C}_1$, should not only be included in the generalized type that is inserted in the type environment, but should also be part of the final constraint set. Otherwise, a type error in the definition would go unnoticed in case the definition is not used in the body of the let. To be more precise, we require that the constraint set $\mathcal{C}_1$ is satisfiable. Existential quantification can be used here to express satisfiability – an approach which is also followed in the HM(X) framework [46]. The constraint set in the conclusion of the (SD-Let) type rule would then be

$$(\exists\overline{a}.\mathcal{C}_1) \cup \mathcal{C}_2 \qquad \textit{where } \overline{a} = \mathit{ftv}(\mathcal{C}_1) - \mathit{ftv}(\Gamma).$$

Alternatively, the presented type rules can be formulated bottom-up, although this requires some additional computation to know over which type variables we are allowed to generalize, and which should remain monomorphic.

We continue with *idid* from Example 4.2. After generalizing the constraints collected for $\lambda x \to x$, we have the type scheme $\forall ab.(a \equiv b) \Rightarrow a \to b$ for *id*. The body (*id id*) can then be typed by instantiating this type scheme twice, and then applying the application type rule. At the end, we find the type v_6 under the type constraints

$$\{v_0 \equiv v_1, v_2 \equiv v_3, v_4 \equiv v_5, v_2 \to v_3 \equiv (v_4 \to v_5) \to v_6\}.$$

Not surprisingly, this type is equal to $\alpha \to \alpha$ for some type variable α.

This approach for dealing with let-polymorphism is in essence the solution of Aiken and Wimmers in their constraint system [2], and can be summarized as follows.

- In the constraint collecting phase, we generalize over the type and constraints of a local definition. This generalized type is instantiated for each usage. Conceptually, this corresponds to the *inlining* of local definitions before doing type inference.
- An advantage of this approach is the simplicity of constraint solving, as we only have to deal with constraints that express type equality.
- Inlining of constraint sets implies duplication of work when solving the constraints, and this does not seem to scale very well. Even worse, if the duplicated constraint set is inconsistent, then this inconsistency is also duplicated. This will have a negative impact on the quality of the reported type error messages. The two approaches that are discussed next avoid constraint duplication by sharing intermediate typing results for local definitions.

$\boxed{\mathcal{C}, \Gamma \vdash_{\Downarrow} e : \tau}$ *Top-down typing (alternative 2)*

$$\frac{(x : \sigma) \in \Gamma}{\{\beta := \textsc{Inst}(\sigma)\}, \Gamma \vdash_{\Downarrow} x : \beta} \quad (\textsc{SD-Var})$$

$$\frac{\mathcal{C}_1, \Gamma \vdash_{\Downarrow} e_1 : \tau_1 \qquad \mathcal{C}_2, \Gamma \backslash x \cup \{x : \sigma\} \vdash_{\Downarrow} e_2 : \tau_2}{\mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\sigma := \textsc{Gen}(\Gamma, \tau_1)\}, \Gamma \vdash_{\Downarrow} \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2} \quad (\textsc{SD-Let})$$

Figure 4.4. Type rules for dealing with polymorphism (alternative 2)

Alternative 2: introducing type scheme variables

Type schemes represent polymorphic types, and the problem we face in a constraint-based setting is that the polymorphic types are not known beforehand – they are the result of solving a part of the collected type constraints. To circumvent this problem, we can use type scheme variables as place-holders for polymorphic types that are not yet known, but that become available at some time during constraint solving. This is nothing but an implementation technique similar to the introduction of fresh type variables in type inference algorithms. We first extend the type constraint language with *generalization* and *instantiation* constraints.

Type constraint (for the second alternative):

$$\begin{array}{llr} c ::= & \tau_1 \equiv \tau_2 & \textit{(equality)} \\ \mid & \sigma := \textsc{Gen}(\Gamma, \tau) & \textit{(generalization)} \\ \mid & \tau := \textsc{Inst}(\sigma) & \textit{(instantiation)} \end{array}$$

A generalization constraint, written as $\sigma := \textsc{Gen}(\Gamma, \tau)$, produces a type scheme by generalizing a type τ with respect to some type environment Γ. The σ in this constraint is a type scheme variable, which is assigned the type scheme obtained from the generalization. The other constraint that is introduced is an instantiation constraint, $\tau := \textsc{Inst}(\sigma)$. This constraint should be read as "the type τ is an instance of the type scheme σ". Again, σ can be a type scheme variable. A special case is when the type scheme to be instantiated is a monotype. In this case, $\tau_1 := \textsc{Inst}(\tau_2)$ boils down to $\tau_1 \equiv \tau_2$. Altogether, we can now record and instantiate type schemes.

It is important to realize that monotypes (the τ's) and type schemes (the σ's) come from two different domains, and that type variables will never be mapped to a type scheme. Furthermore, type scheme variables should really act as place-holders. For instance, type schemes found by generalization do not contain type scheme variables.

In Figure 4.4, we present a new type rule for variables and one for let expressions. A type scheme variable σ is introduced in the (SD-Let) type rule which is constrained to the generalized type of the definition. This type scheme variable is used in the type environment to type the body of the let. For each identifier we find its corresponding type scheme in the type environment and constrain the returned type (the type variable β) to be an instance of this type scheme. The type rules for application and lambda abstraction are unchanged.

We continue with the expression *idid* from Example 4.2. The type assigned to this expression is v_4, and the following constraints are collected using our newly formulated type rules.

$$\{v_1 := \textsc{Inst}(v_0), \sigma_0 := \textsc{Gen}(\emptyset, v_0 \to v_1),\\ v_2 := \textsc{Inst}(\sigma_0), v_3 := \textsc{Inst}(\sigma_0), v_2 \equiv v_3 \to v_4\}$$

The first constraint, $v_1 := \textsc{Inst}(v_0)$, is equivalent to $v_0 \equiv v_1$: these two type variables have to map to the same type. This instantiation constraint results from the monotype introduced by the lambda abstraction. As a result, σ_0 is mapped to the type scheme $\forall a.a \to a$. The type variables v_2 and v_3 are both instances of this type scheme, say $v_5 \to v_5$ and $v_6 \to v_6$ respectively. The final constraint restricts the type of the complete expression (v_4) and v_5 to be equivalent to $v_6 \to v_6$.

In conclusion, the advantages and disadvantages of this approach are:

- The polymorphic type of a locally defined variable is computed only once, regardless of the number of times it is used.
- The exact point when generalization and instantiation constraints can be *solved* is restricted: type variables over which we generalize should not be restricted at a later point in time by another constraint, and a type scheme variable cannot be instantiated until it has been assigned a value.
- Implementing this alternative requires type scheme variables and three kinds of constraints. A special purpose substitution should be maintained that maps type scheme variables to inferred type schemes. In particular, this type scheme substitution should be kept strictly separate from the (normal) type substitution.

Alternative 3: collecting implicit instance constraints

A third alternative for dealing with polymorphism is to use implicit instance constraints, which combine the generalization and instantiation constraints from the previous approach, but circumvent the introduction of type schemes and type scheme variables in the constraint language. The constraints of this approach are of the following form:

Type constraint (for the third alternative):

$$\begin{array}{lll} c ::= & \tau_1 \equiv \tau_2 & \textit{(equality)} \\ \quad\mid & \tau_1 \leqslant_{\mathcal{M}} \tau_2 & \textit{(implicit instance)} \end{array}$$

An implicit instance constraint should be interpreted as follows: the type τ_1 should be an instance of the type scheme obtained by generalizing the type τ_2 without quantifying the type variables that are free in $\mathcal{M}$. The idea is that such a constraint can be solved only if no more deductions can be found in the remaining constraints about the type variables over which we generalize. The bottom-up type rule for a let expression is formulated as follows.

$$\frac{\begin{array}{c}\mathcal{C}_{new} = \{\tau' \leqslant_{\mathcal{M}} \tau_1 \mid x : \tau' \in \mathcal{A}_2\} \\ \mathcal{C}_1, \mathcal{A}_1 \vdash_{\Uparrow} e_1 : \tau_1 \qquad \mathcal{C}_2, \mathcal{A}_2 \vdash_{\Uparrow} e_2 : \tau_2\end{array}}{\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_{new}, \mathcal{A}_1 \cup \mathcal{A}_2 \backslash x \vdash_{\Uparrow} \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2} \quad (\text{SU-L}\textsc{et})$$

For each occurrence of x in the body of the let expression we create one implicit instance constraint. These constraints are annotated with a set of monomorphic types $\mathcal{M}$, which is supplied by the context. The details of this set with monomorphic types are discussed in the next section.

Consider the running Example 4.2 once more. Applying the type rules results in the type v_4 and the following constraint set.

$$\{v_1 \equiv v_0, v_2 \leqslant_{\emptyset} v_1 \rightarrow v_0, v_3 \leqslant_{\emptyset} v_1 \rightarrow v_0, v_2 \equiv v_3 \rightarrow v_4\}$$

Informally speaking, the two implicit instance constraints can only be solved after the constraint $v_1 \equiv v_0$ has been taken into account. Consequently, the intermediate type scheme is $\forall a.a \rightarrow a$, the type of the identity function. After two instantiations of this type scheme are unified with v_2 and v_3 respectively, we obtain the type $\alpha \rightarrow \alpha$ for v_4 via the last equality constraint, where α is some fresh type variable.

A summary of this approach:

- The type of an identifier defined by a let is inferred only once, independently of the number of occurrences in the body. No type constraints are duplicated, but for each implicit instance constraint we generalize and instantiate. Ideally, we generalize a type only once.
- No type schemes appear in the type environment or in the type constraints. One extra constraint, an implicit instance constraint, is introduced for dealing with polymorphism.
- An implicit instance constraint is solved in two steps: first a type is generalized to a type scheme, and then this type scheme is instantiated. The approach discussed previously (alternative 2) is more explicit as it distinguishes these two steps. An implicit instance constraint imposes a restriction on the order in which the type constraints can be solved.

4.3 Bottom-up type rules

The previous section discusses several alternative approaches for dealing with let-polymorphism in a constraint-based setting. In the remaining sections of this chapter, we adopt the implicit instance constraints in dealing with polymorphism: this keeps the proofs relatively simple, and it is close enough to the alternative with type scheme variables that we use in later chapters. In addition to the implicit instance constraint, we introduce the explicit instance constraint to express that a type is to be an instance of a type scheme that we know before solving (i.e., during constraint collection).

Definition 4.2 (Type constraints).

Type constraint:

$$\begin{array}{lll} c ::= & \tau_1 \equiv \tau_2 & \textit{(equality)} \\ \mid & \tau \preceq \sigma & \textit{(explicit instance)} \\ \mid & \tau_1 \leqslant_{\mathcal{M}} \tau_2 & \textit{(implicit instance)} \end{array}$$

We define substitution on a type constraint in the obvious way, as well as determining the free type variables of a type constraint. The free type variables of an implicit instance constraint include the type variables of the monomorphic set, and this set may change by applying a substitution. We now define when a substitution satisfies a type constraint (refinement of Definition 4.1).

Definition 4.3 (Constraint satisfaction).

$$\begin{array}{llll} S \vdash_s (\tau_1 \equiv \tau_2) & =_{def} & S\tau_1 = S\tau_2 & (\vdash_s \equiv) \\ S \vdash_s (\tau \preceq \sigma) & =_{def} & S\tau < S\sigma & (\vdash_s \preceq) \\ S \vdash_s (\tau_1 \leqslant_{\mathcal{M}} \tau_2) & =_{def} & S\tau_1 < \mathit{generalize}(S\mathcal{M}, S\tau_2) & (\vdash_s \leqslant) \end{array}$$

With this definition of constraint satisfaction, a number of properties can be established.

Lemma 4.1. *Assume that* $S_1 \vdash_s (\tau_1 \equiv \tau_2)$*. Then for all substitutions* S_2 *it holds that* $S_2 \circ S_1 \vdash_s (\tau_1 \equiv \tau_2)$*.*

Proof. If $S_1 \vdash_s (\tau_1 \equiv \tau_2)$, then $S_1\tau_1 = S_1\tau_2$. As a result, $S_2(S_1\tau_1)$ is equal to $S_2(S_1\tau_2)$. Then, by definition, we have $S_2 \circ S_1 \vdash_s (\tau_1 \equiv \tau_2)$. □

Lemma 4.2. $S \vdash_s \tau_1 \equiv \tau_2 \quad \Leftrightarrow \quad S \vdash_s \tau_1 \preceq \tau_2$

Proof. Because $S\tau_1 < S\tau_2 \;\Leftrightarrow\; S\tau_1 = S\tau_2$ by instance-of relation (Figure 2.2). Of course, this only holds because τ_2 is monomorphic (and not a type scheme). □

Lemma 4.3. $S \vdash_s \tau_1 \leqslant_{\mathcal{M}} \tau_2 \quad \Leftrightarrow \quad S \vdash_s \tau_1 \preceq \mathit{generalize}(S\mathcal{M}, S\tau_2)$

Proof. According to Definition 4.3 (constraint satisfaction), we need to prove that $S\tau_1 < \mathit{generalize}(S\mathcal{M}, S\tau_2) \;\Leftrightarrow\; S\tau_1 < S(\mathit{generalize}(S\mathcal{M}, S\tau_2))$. This holds because $S(\mathit{generalize}(S\mathcal{M}, S\tau_2))$ equals $\mathit{generalize}(S\mathcal{M}, S\tau_2)$ since S is an idempotent substitution. □

The type rules that we will present next have a bottom-up nature: type information about an expression is (almost) independent of its context. The type rules are also syntax-directed, which makes them relatively straightforward to implement. In fact, the rules describe an algorithm for collecting type constraints. Because the relation between types within an expression is expressed as a set of type constraints, every expression has a derivation. This is in contrast with, for instance, the Hindley-Milner type rules, that fail to find a derivation for an ill-typed expression. However, a derivation for an ill-typed expression will, of course, result in an inconsistent constraint set. Furthermore, there is one unique derivation for each expression – modulo the choice of fresh type variables. An other property of

$\boxed{\mathcal{M}, \mathcal{A}, \mathcal{C} \vdash_{\Uparrow} e : \tau}$ *Bottom-up typing*

$$\frac{}{\mathcal{M}, \{x{:}\beta\}, \emptyset \vdash_{\Uparrow} x : \beta} \quad \text{(BU-Var)}$$

$$\frac{\begin{array}{c}\mathcal{C}_{new} = \{\tau_1 \equiv \tau_2 \rightarrow \beta\} \\ \mathcal{M}, \mathcal{A}_1, \mathcal{C}_1 \vdash_{\Uparrow} e_1 : \tau_1 \qquad \mathcal{M}, \mathcal{A}_2, \mathcal{C}_2 \vdash_{\Uparrow} e_2 : \tau_2\end{array}}{\mathcal{M}, \mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_{new} \vdash_{\Uparrow} e_1\ e_2 : \beta} \quad \text{(BU-App)}$$

$$\frac{\begin{array}{c}\mathcal{C}_{new} = \{\beta_2 \equiv \beta_1 \rightarrow \tau\} \cup \{\beta_1 \equiv \tau' \mid x{:}\tau' \in \mathcal{A}\} \\ \mathcal{M} \cup \{\beta_1\}, \mathcal{A}, \mathcal{C} \vdash_{\Uparrow} e : \tau\end{array}}{\mathcal{M}, \mathcal{A} \backslash x, \mathcal{C} \cup \mathcal{C}_{new} \vdash_{\Uparrow} \lambda x \rightarrow e : \beta_2} \quad \text{(BU-Abs)}$$

$$\frac{\begin{array}{c}\mathcal{C}_{new} = \{\beta \equiv \tau_2\} \cup \{\tau' \leqslant_{\mathcal{M}} \tau_1 \mid x{:}\tau' \in \mathcal{A}_2\} \\ \mathcal{M}, \mathcal{A}_1, \mathcal{C}_1 \vdash_{\Uparrow} e_1 : \tau_1 \qquad \mathcal{M}, \mathcal{A}_2, \mathcal{C}_2 \vdash_{\Uparrow} e_2 : \tau_2\end{array}}{\mathcal{M}, \mathcal{A}_1 \cup \mathcal{A}_2 \backslash x, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_{new} \vdash_{\Uparrow} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \beta} \quad \text{(BU-Let)}$$

Figure 4.5. Bottom-up type rules

the bottom-up type rules is that while constraints are collected, not a single constraint needs to be solved in order to proceed. Collecting constraints and solving constraints are two separate concerns.

The type rules are formulated in terms of judgements of the form

$$\mathcal{M}, \mathcal{A}, \mathcal{C} \vdash_{\Uparrow} e : \tau.$$

Such a judgement should be read as: "given a set of types $\mathcal{M}$ that are to remain monomorphic, we can assign type τ to expression e if the type constraints in $\mathcal{C}$ are satisfied, and if $\mathcal{A}$ enumerates all the types that have been assigned to the identifiers that are free in e". The set of monomorphic types ($\mathcal{M}$) is provided by the context: it is passed top-down. This is only to simplify constraint collection: alternatively, we could change the monomorphic sets of the implicit instance constraints when they are collected upwards. The assumption set ($\mathcal{A}$) contains an assumption for each *occurrence* of an unbound identifier. Hence, $\mathcal{A}$ can have multiple assertions for the same identifier.

The four bottom-up type rules are listed in Figure 4.5. The type rule (BU-Var) for a variable x is straightforward: a fresh type variable β is returned as the type for x, and this assertion is recorded in the assumption set. No type constraints are introduced by this rule.

Each function application introduces a fresh type variable β, which represents the result of the application. Due to the bottom-up formulation of the type rules, we may assume to be able to find a judgement for the function and the argument of an application without making any restrictions on these judgements. We constrain the type of the function to be a function type which, when applied to something of type τ_2, returns a value of type β. This new equality constraint ($\tau_1 \equiv \tau_2 \to \beta$) is combined with the constraints that were already collected for e_1 and e_2. The assumption sets $\mathcal{A}_1$ and $\mathcal{A}_2$ for the two subexpressions are combined. Note that this follows the intuition that an assumption set records the types assigned to unbound identifiers.

The type rule for a lambda abstraction introduces two fresh type variables. The first (β_1) represents the type of the abstracted value x. The second (β_2) corresponds to the type of the whole lambda abstraction. Each assertion about the type of x collected for the body of the abstraction produces one new equality constraint. Furthermore, the type of the abstraction is equal to a function type from the type of x to the type of the body, hence $\beta_2 \equiv \beta_1 \to \tau$. The assumption set in the conclusion is the set of assumptions collected for the body without the assertions about x. Observe that β_1 is included in the set of monomorphic values in the judgement for e.

We assume to have two judgements for the subexpressions of a let expression. One fresh type variable (β) is introduced to represent the result type of the let expression. The definition cannot be recursive – an x that is unbound in e_1 remains free. On the other hand, types associated with x from e_2 have to be an instance of the type that is inferred for the definition of x. An implicit instance constraint is generated for each assertion about x in $\mathcal{A}_2$. The set of monomorphic types $\mathcal{M}$ is stored with each of these constraints. In the conclusion, the assertions about x are removed from the assumption set $\mathcal{A}_2$. The constraint $\beta \equiv \tau_2$ states that the type of the let expression equals the type of the body.

Example 4.3. Applying the bottom-up type rules to the expression of Example 4.2 results in the following derivation tree.

$$\dfrac{\dfrac{\dfrac{}{\{v_1\},\{x:v_0\},\emptyset \vdash_{\Uparrow} x : v_0}}{\emptyset,\emptyset,\mathcal{C}_0 \vdash_{\Uparrow} \lambda x \to x : v_2} \qquad \dfrac{\dfrac{}{\emptyset,\{i:v_3\},\emptyset \vdash_{\Uparrow} i : v_3} \quad \dfrac{}{\emptyset,\{i:v_4\},\emptyset \vdash_{\Uparrow} i : v_4}}{\emptyset,\{i:v_3,i:v_4\},\mathcal{C}_1 \vdash_{\Uparrow} i\ i : v_5}}{\emptyset,\emptyset,\mathcal{C}_2 \vdash_{\Uparrow} \mathbf{let}\ i = \lambda x \to x\ \mathbf{in}\ i\ i : v_6}$$

$$\begin{aligned}
\mathcal{C}_0 &= \{v_2 \equiv v_1 \to v_0, v_1 \equiv v_0\}\\
\mathcal{C}_1 &= \{v_3 \equiv v_4 \to v_5\}\\
\mathcal{C}_2 &= \mathcal{C}_0 \cup \mathcal{C}_1 \cup \{v_6 \equiv v_5, v_3 \leqslant_{\emptyset} v_2, v_4 \leqslant_{\emptyset} v_2\}
\end{aligned}$$

Six constraints are collected for this particular expression. The empty assumption set at top-level indicates that the expression is closed: it has no unbound identifiers.

An invariant of the bottom-up type rules is that the type of a judgement in the conclusion is always a fresh type variable, and that no assumptions are made about the types of subexpressions. Types are thus only related by the type constraints, and not by the context of an expression. We associate a type variable with each subexpression, which helps us at a later stage to produce precise error messages. A closer look at the type rules may give rise to the question whether the rules can be defined more succinctly, using fewer type constraints, and introducing fewer type variables. Indeed: the rules can be formulated more compactly. However, the presented rules are fine-grained, and give us more control in reporting the cause of a type inconsistency.

A second observation is that some of the generated equality constraints correspond to unifications that are performed by algorithm $\mathcal{W}$ (the type associated with a subexpression is restricted, as is the case for the constraint created by the (BU-App) rule), whereas some other constraints restrict the type in the conclusion as algorithm $\mathcal{M}$ would do. Some equality constraints, like $\beta \equiv \tau_2$ in the (BU-Let) type rule, do not correspond to a unification in $\mathcal{W}$ or $\mathcal{M}$. This is an indication that our constraint-based formulation is indeed a very general approach to the Hindley-Milner type system, which has the algorithms $\mathcal{W}$ and $\mathcal{M}$ as an instance. We formalize and explore this claim in the next chapter.

4.4 Solving type constraints

So far, a set of type rules has been presented to collect type constraints from an expression. This section explains how such a set of type constraints can be solved. Or to put it differently: how to construct a solution for a given set of type constraints such that all the constraints are satisfied. Before we present an algorithm, we define when a type variable is *active*.

Definition 4.4 (Active). *The set of active type variables for a given type constraint is*

$$\begin{array}{lll} \mathit{active}(\tau_1 \equiv \tau_2) & =_{def} & \mathit{ftv}(\tau_1) \cup \mathit{ftv}(\tau_2) \\ \mathit{active}(\tau \preceq \sigma) & =_{def} & \mathit{ftv}(\tau) \ \cup \mathit{ftv}(\sigma) \\ \mathit{active}(\tau_1 \leqslant_{\mathcal{M}} \tau_2) & =_{def} & \mathit{ftv}(\tau_1) \cup (\mathit{ftv}(\mathcal{M}) \cap \mathit{ftv}(\tau_2)). \end{array}$$

This notion is lifted to constraint sets by taking the union. As a result, $\mathcal{C}_1 \subseteq \mathcal{C}_2$ implies $\mathit{active}(\mathcal{C}_1) \subseteq \mathit{active}(\mathcal{C}_2)$.

The intuition behind this definition is that only active type variables can be mapped to other types as a result of solving the type constraint (set). Inactive type variables of a constraint remain unchanged when solving the constraints. Note that not all the type variables in the right-hand side type of an implicit instance constraint are active, but only those that are also in $\mathcal{M}$, which is the essential difference between active and free type variables. Hence, the set of active type variables is always a subset of the free type variables.

We present a non-deterministic algorithm Solve (see Figure 4.6) to find a substitution that satisfies all the type constraints. The algorithm deals with one constraint at a time, and it is non-deterministic in the sense that it is not explicit in the order in which it deals with the constraints. This gives us the freedom to experiment with different orderings of the constraints. One aspect of the algorithm is that it introduces fresh type variables. We write $\text{Solve}_{\chi}(\mathcal{C})$ for solving constraint set $\mathcal{C}$ with a set of fresh type variables χ, assuming that χ and $\mathit{ftv}(\mathcal{C})$ are disjoint (and remain disjoint). The set χ is omitted where possible.

```
$\textsc{Solve} :: \mathit{ConstraintSet} \rightarrow \mathit{Substitution}$

$\textsc{Solve}_\chi(\emptyset) = \mathit{id}$

$\textsc{Solve}_\chi(\{\tau_1 \equiv \tau_2\} \cup \mathcal{C}) = \textbf{let } S = \mathit{mgu}(\tau_1, \tau_2)$
                                  $\textbf{in } \textsc{Solve}_\chi(S\mathcal{C}) \circ S$

$\textsc{Solve}_\chi(\{\tau_1 \preceq \forall\overline{a}.\tau_2\} \cup \mathcal{C}) = \textbf{let } \chi = \chi_1 \cup \chi_2 \quad (\chi_1 \cap \chi_2 = \emptyset)$
                                      $S = [\overline{a} := \chi_1]$
                                  $\textbf{in } \textsc{Solve}_{\chi_2}(\{\tau_1 \equiv S\tau_2\} \cup \mathcal{C})$

$\textsc{Solve}_\chi(\{\tau_1 \leqslant_\mathcal{M} \tau_2\} \cup \mathcal{C}) = \textbf{let } \sigma = \mathit{generalize}(\mathcal{M}, \tau_2)$
                                  $\textbf{in } \textsc{Solve}_\chi(\{\tau_1 \preceq \sigma\} \cup \mathcal{C})$
      $\textit{only if } \mathit{ftv}(\tau_2) \cap \mathit{active}(\{\tau_1 \leqslant_\mathcal{M} \tau_2\} \cup \mathcal{C}) \subseteq \mathit{ftv}(\mathcal{M})$
```

Figure 4.6. A non-deterministic algorithm to solve type constraints

Solving the empty constraint set results in the empty substitution. The other three cases correspond to the three constraint forms. An equality constraint is solved by computing a most general unifier of the two types. The unifier is applied to the remaining constraints. The error substitution (written as $\top$) is used in case the two types cannot be unified. By definition, the error substitution satisfies every constraint set. An explicit instance constraint is translated into an equality constraint by instantiating the type scheme with fresh type variables. Note that the set of fresh type variables is first split into two disjoint sets. Solving an implicit instance constraint ($\tau_1 \leqslant_\mathcal{M} \tau_2$) is less straightforward. We are interested in the type scheme that we obtain by generalizing τ_2 with respect to $\mathcal{M}$. However, the type variables that are quantified should not be changed afterwards (as a result of solving some other constraint). Because substitutions do not affect quantified type variables, we impose a condition on the moment that an implicit instance constraint can be taken into account: the type variables that are about to be quantified ($\mathit{ftv}(\tau_2) - \mathit{ftv}(\mathcal{M})$) should not be active in the current constraint set. Once this condition holds, an implicit instance constraint can be translated into an explicit instance constraint.

Example 4.4. Consider the constraints that were collected in Example 4.3. We illustrate algorithm Solve by presenting one possible solution for this constraint set. For this example, we solve the constraints in the given order.

$$\textsc{Solve}(\{v_2 \equiv v_1 \rightarrow v_0, v_1 \equiv v_0, v_3 \equiv v_4 \rightarrow v_5, v_3 \leqslant_\emptyset v_2, v_4 \leqslant_\emptyset v_2, v_6 \equiv v_5\})$$

$$= \textsc{Solve}(\{v_1 \equiv v_0, v_3 \equiv v_4 \rightarrow v_5, v_3 \leqslant_\emptyset v_1 \rightarrow v_0, v_4 \leqslant_\emptyset v_1 \rightarrow v_0, v_6 \equiv v_5\})$$
$$\circ [v_2 := v_1 \rightarrow v_0]$$

$$= \textsc{Solve}(\{v_3 \equiv v_4 \rightarrow v_5, v_3 \leqslant_\emptyset v_0 \rightarrow v_0, v_4 \leqslant_\emptyset v_0 \rightarrow v_0, v_6 \equiv v_5\})$$
$$\circ [v_1 := v_0, v_2 := v_0 \rightarrow v_0]$$

$$= \textsc{Solve}(\{v_4 \rightarrow v_5 \leqslant_\emptyset v_0 \rightarrow v_0, v_4 \leqslant_\emptyset v_0 \rightarrow v_0, v_6 \equiv v_5\})$$
$$\circ [v_1 := v_0, v_2 := v_0 \rightarrow v_0, v_3 := v_4 \rightarrow v_5]$$

$$
\begin{aligned}
&= \textsc{Solve}(\{v_4 \to v_5 \preceq \forall a.a \to a, v_4 \leqslant_\emptyset v_0 \to v_0, v_6 \equiv v_5\}) \\
&\qquad \circ [v_1 := v_0, v_2 := v_0 \to v_0, v_3 := v_4 \to v_5] \\
&= \textsc{Solve}(\{v_4 \to v_5 \equiv v_7 \to v_7, v_4 \leqslant_\emptyset v_0 \to v_0, v_6 \equiv v_5\}) \\
&\qquad \circ [v_1 := v_0, v_2 := v_0 \to v_0, v_3 := v_4 \to v_5] \\
&= \textsc{Solve}(\{v_7 \leqslant_\emptyset v_0 \to v_0, v_6 \equiv v_7\}) \\
&\qquad \circ [v_1 := v_0, v_2 := v_0 \to v_0, v_3 := v_7 \to v_7, \{v_4, v_5\} := v_7] \\
&= \textsc{Solve}(\{v_7 \preceq \forall a.a \to a, v_6 \equiv v_7\}) \\
&\qquad \circ [v_1 := v_0, v_2 := v_0 \to v_0, v_3 := v_7 \to v_7, \{v_4, v_5\} := v_7] \\
&= \textsc{Solve}(\{v_7 \equiv v_8 \to v_8, v_6 \equiv v_7\}) \\
&\qquad \circ [v_1 := v_0, v_2 := v_0 \to v_0, v_3 := v_7 \to v_7, \{v_4, v_5\} := v_7] \\
&= \textsc{Solve}(\{v_6 \equiv v_8 \to v_8\}) \\
&\qquad \circ [v_1 := v_0, v_2 := v_0 \to v_0, v_3 := (v_8 \to v_8) \to v_8 \to v_8, \{v_4, v_5, v_7\} := v_8 \to v_8] \\
&= \textsc{Solve}(\emptyset) \\
&\qquad \circ [v_1 := v_0, v_2 := v_0 \to v_0, v_3 := (v_8 \to v_8) \to v_8 \to v_8, \{v_4, v_5, v_6, v_7\} := v_8 \to v_8] \\
&= [v_1 := v_0, v_2 := v_0 \to v_0, v_3 := (v_8 \to v_8) \to v_8 \to v_8, \{v_4, v_5, v_6, v_7\} := v_8 \to v_8]
\end{aligned}
$$

In this derivation, the substitution on the right is kept idempotent. Observe that the conditions that are imposed by the two implicit instance constraints are both met at the time the constraints are taken into account, and that the explicit instance constraints introduce fresh type variables (v_7 and v_8). The set of constraints was collected for the expression **let** $i = \lambda x \to x$ **in** $i\ i$, which was assigned the type v_6. Hence, we informally claim that we can assign $v_8 \to v_8$ to this expression.

The previous example also provides some intuition why solving the two implicit instance constraints is subjected to certain conditions. Dealing with these two constraints right away would result in translating the first instance constraint, $v_3 \leqslant_\emptyset v_2$, into $v_3 \preceq \forall a.a$. This type scheme is clearly too polymorphic for the identity function, especially since we cannot update the type scheme (and v_3) after solving the constraint $v_2 \equiv v_1 \to v_0$. One may wonder whether there will always be a point in time at which an implicit instance constraint can be solved. The answer is a plain no, as the following example shows.

Example 4.5. Let $\mathcal{C}$ be $\{v_0 \leqslant_\emptyset v_1, v_1 \leqslant_\emptyset v_0\}$. Then $\mathit{active}(\mathcal{C}) = \{v_0, v_1\}$ and, hence, none of the two constraints is candidate for being solved at this point. Since none of the rules can be applied, algorithm Solve cannot continue. Despite this, there are substitutions satisfying this constraint set, including the empty substitution.

Example 4.6. Consider the implicit instance constraint $c = v_1 \to \mathit{Bool} \leqslant_\emptyset \mathit{Int} \to v_1$. This constraint cannot be handled at this point by algorithm Solve, because $\mathit{ftv}(\mathit{Int} \to v_1) \cap \mathit{active}(c) \not\subseteq \emptyset$. If we would drop the restriction posed on implicit instance constraints, and turn c into the explicit instance constraint $v_1 \to \mathit{Bool} \preceq \forall a.\mathit{Int} \to a$, then we would get an incorrect substitution (i.e., one that does not satisfy c). Note that this example shows why the implicit instance constraint itself must be included in determining the active type variables of a constraint set. An implicit instance constraint (say $\tau_1 \leqslant_{\mathcal{M}} \tau_2$) that is created by the bottom-up type rules does not have this problem, since we know that $ftv(\tau_1)$ and $ftv(\tau_2)$ are disjoint.

We define when a set of type constraints can be ordered, such that the conditions imposed by implicit instance constraints are met. Constraint sets without this property, such as $\mathcal{C}$ in Example 4.5, cannot be solved by algorithm Solve.

Definition 4.5 (Orderable). *A constraint set $\mathcal{C}$ is orderable whenever the constraints can be ordered such that for each implicit instance constraint $c = \tau_1 \leqslant_{\mathcal{M}} \tau_2$ in $\mathcal{C}$ the following condition holds, where $\mathcal{C}'$ are those constraints from $\mathcal{C}$ that appear after c in the ordering.*

$$ftv(\tau_2) \cap active(\{c\} \cup \mathcal{C}') \subseteq ftv(\mathcal{M})$$

Constraint sets that are created with the $\vdash_{\Uparrow}$ type rules can always be ordered. This property of the bottom-up type rules is expressed in Lemma 4.4, which is illustrated by an example.

Lemma 4.4 (Orderable). *If $\mathcal{M}, \mathcal{A}, \mathcal{C} \vdash_{\Uparrow} e : \tau$, then $\mathcal{C}$ is orderable.*

Proof. To prove that each constraint set constructed with the bottom-up type rules is orderable, we indicate for each type rule how its constraints can be ordered. This gives us a valid constraint ordering for all derivations.

(BU-Var)	$\emptyset$	(BU-Abs)	$\mathcal{C}_{new} +\!\!+ \mathcal{C}$
(BU-App)	$\mathcal{C}_1 +\!\!+ \mathcal{C}_2 +\!\!+ \mathcal{C}_{new}$	(BU-Let)	$\mathcal{C}_1 +\!\!+ \mathcal{C}_{new} +\!\!+ \mathcal{C}_2$

All implicit instance constraints are generated by (BU-Let), and are part of its $\mathcal{C}_{new}$. Note that $\mathcal{C}_{new}$ also contains one equality constraint. The implicit instance constraints are placed after the constraints of the definition ($\mathcal{C}_1$), but before the constraints of the body of the let expression ($\mathcal{C}_2$). Furthermore, it is of great importance that the equality constraints for the variables bound by a lambda abstraction are taken into account before the constraints of the body of the abstraction. If not, the ordering can be invalid as we may discover that a type variable is monomorphic after having generalized that particular type variable. Equality constraints can "change" the set of monomorphic type variables carried by implicit instance constraints. The relative order of the constraint sets in (BU-App) is irrelevant. □

Example 4.7. Consider the following expression.

```
λx → let f = let g = x
             in λa → g
     in let h = f
        in h x
```

The bottom-up type rules create eight equality constraints, and three implicit instance constraints (one for f, one for g, and one for h). The constraint ordering that was suggested in the proof of Lemma 4.4 results in the following list of constraints.

For each constraint, we indicate at which node of the abstract syntax tree it was created (that is, which type rule is responsible for the constraint).

$[\ v_{12} \equiv v_{11} \rightarrow v_{10}\ ,\ v_{11} \equiv v_0\ ,\ v_{11} \equiv v_7$	(BU-Abs) *for x*
$,\ v_4 \equiv v_3\ ,\ v_1 \leqslant_{\{v_{11}\}} v_0$	(BU-Let) *for g*
$,\ v_3 \equiv v_2 \rightarrow v_1$	(BU-Abs) *for a*
$,\ v_{10} \equiv v_9\ ,\ v_5 \leqslant_{\{v_{11}\}} v_4$	(BU-Let) *for f*
$,\ v_9 \equiv v_8\ ,\ v_6 \leqslant_{\{v_{11}\}} v_5$	(BU-Let) *for h*
$,\ v_6 \equiv v_7 \rightarrow v_8$	(BU-App)
$]$	

This ordering of the constraints conforms with Definition 4.5. Let us take a closer look at the relative order of the three implicit instance constraints. The implicit instance constraint for g is the first which is solved, followed by the constraint for f, and finally for h. This order corresponds precisely to the order in which algorithm $\mathcal{W}$ (and also algorithm $\mathcal{M}$) would infer the types of the local definitions.

Given a constraint set that is orderable, algorithm Solve terminates and returns a substitution. Note that for each constraint set that cannot be ordered, there is a point at which Solve cannot proceed any more.

Theorem 4.5 (Progress of Solve). *Let $\mathcal{C}$ be an orderable constraint set. Then* Solve($\mathcal{C}$) *returns a substitution, and does not loop.*

Proof. We define a weight function to prove that algorithm Solve terminates for each orderable constraint set.

$$\begin{aligned} \mathit{weight}(\tau_1 \equiv \tau_2) &= 1 \\ \mathit{weight}(\tau \preceq \sigma) &= 2 \\ \mathit{weight}(\tau_1 \leqslant_{\mathcal{M}} \tau_2) &= 3 \end{aligned}$$

The weight of a constraint set $\mathcal{C}$ is defined as $\sum_{c \in \mathcal{C}} \mathit{weight}(c)$. The weight of the constraint set (strictly) decreases for each recursive call to Solve. □

An important property of algorithm Solve is that the returned substitution really *solves* the constraints. All the constraints that are solved must be satisfied by this substitution. First, three related lemmas are discussed. The following lemma formulates under which condition substitution distributes over generalization.

Lemma 4.6 (Generalization). *Given a type τ, a substitution S, and a set of types $\mathcal{M}$. Then*

$$\mathit{ftv}(\tau) \cap \mathit{tv}(S) \subseteq \mathit{ftv}(\mathcal{M})\ ^{\mathbf{(1)}} \quad \Rightarrow \quad S(\mathit{generalize}(\mathcal{M}, \tau)) = \mathit{generalize}(S\mathcal{M}, S\tau).$$

Proof. By the definition of *generalize*, we have $S(\mathit{generalize}(\mathcal{M}, \tau)) = S(\forall \overline{a}.\tau)$ with $\overline{a} = \mathit{ftv}(\tau) - \mathit{ftv}(\mathcal{M})$. Furthermore, $S(\forall \overline{a}.\tau)$ equals $\forall \overline{a}.S\tau$ for the following reason: for each type variable v in $\mathit{ftv}(\tau) \cap \mathit{dom}(S)$, we conclude that $v \in \mathit{tv}(S)$, $v \in \mathit{ftv}(\mathcal{M})$ (by **(1)**), and $v \notin \overline{a}$. Hence, applying S before or after quantification gives the same result. To complete the proof, we have to show that $\forall \overline{a}.S\tau = \mathit{generalize}(S\mathcal{M}, S\tau)$: we prove that both type schemes quantify the same set of type variables.

$$\mathit{ftv}(\tau) - \mathit{ftv}(\mathcal{M}) \quad = \quad \mathit{ftv}(S\tau) - \mathit{ftv}(S\mathcal{M})$$

We prove this equation in two steps.

Direction 1. Given is a type variable $v \in \mathit{ftv}(\tau) - \mathit{ftv}(\mathcal{M})$. Hence, $v \in \mathit{ftv}(\tau)$, and $v \notin \mathit{ftv}(\mathcal{M})$. This implies $v \notin \mathit{tv}(S)$ (by **(1)**), thus v does not appear in the substitution S. Because $v \in \mathit{ftv}(\tau)$ and $v \notin \mathit{dom}(S)$, we know that $v \in \mathit{ftv}(S\tau)$. Similarly, $v \notin \mathit{ftv}(S\mathcal{M})$ since $v \notin \mathit{ftv}(\mathcal{M})$ and $v \notin \mathit{ftv}(\mathit{ran}(S))$. Hence, we conclude that $v \in (\mathit{ftv}(S\tau) - \mathit{ftv}(S\mathcal{M}))$.

Direction 2. Let type variable v be in $\mathit{ftv}(S\tau)$, but not in $\mathit{ftv}(S\mathcal{M})$. If $v \notin \mathit{tv}(S)$, then clearly $v \in \mathit{ftv}(\tau) - \mathit{ftv}(\mathcal{M})$ holds. The case for $v \in \mathit{tv}(S)$ is more complex. If $v \in \mathit{dom}(S)$, then v cannot be in $\mathit{ftv}(S\tau)$ since S is idempotent, which gives a contradiction. Thus, $v \in \mathit{ftv}(\mathit{ran}(S))$ (and $v \notin \mathit{dom}(S)$). Because $v \notin \mathit{ftv}(S\mathcal{M})$, we get $v \notin \mathit{ftv}(\mathcal{M})$, and this implies that $v \notin \mathit{ftv}(\tau)$ by **(1)**. Because $v \in \mathit{ftv}(S\tau)$, there must be a $v' \in \mathit{dom}(S)$ such that $v' \in \mathit{ftv}(\tau)$ and $v \in \mathit{ftv}(Sv')$. However, we will argue that such a v' cannot exist. Because $v \notin \mathit{ftv}(S\mathcal{M})$, v' cannot be in $\mathit{ftv}(\mathcal{M})$. Recall that $v' \in \mathit{tv}(S)$, and thus $v' \notin \mathit{ftv}(\tau)$ (by **(1)**), which is in contradiction. Hence, the assertion $v \in \mathit{tv}(S)$ cannot hold. □

Our next lemma explains that algorithm SOLVE does not invent new type variables. Only type variables from the constraint set and type variables from χ can be part of the solution's domain or co-domain.

Lemma 4.7. *Given a constraint set $\mathcal{C}$, there is a set of fresh type variables χ such that $\mathit{ftv}(\mathcal{C}) \cap \chi = \emptyset$. Let substitution S be* $\textsc{Solve}_\chi(\mathcal{C})$. *Then*

$$\mathit{tv}(S) \;\subseteq\; \mathit{ftv}(\mathcal{C}) \cup \chi.$$

Proof. This follows directly from the definition of SOLVE. The case for solving an equality constraint $\tau_1 \equiv \tau_2$ is the only location where the substitution is extended. The unifier that extends the substitution contains only type variables that are free in τ_1 or τ_2 (Lemma 2.1). The case that deals with an explicit instance constraint may introduce fresh type variables in the constraint set that can thus show up in the final substitution. □

The intuition of the following lemma is that type variables that are inactive (they occur in the set of free type variables of a constraint set $\mathcal{C}$, but are not in the set of active type variables) cannot be in the domain of the substitution that is returned by algorithm SOLVE for $\mathcal{C}$.

Lemma 4.8 (Active). *Given a constraint set $\mathcal{C}$, there is a set of fresh type variables χ such that $\mathit{ftv}(\mathcal{C}) \cap \chi = \emptyset$. Then*

$$\mathit{tv}(\textsc{Solve}_\chi(\mathcal{C})) \cap \mathit{ftv}(\mathcal{C}) \subseteq \mathit{active}(\mathcal{C}).$$

Proof. By induction on the four cases of algorithm SOLVE. For the three cases where $\mathcal{C}$ is non-empty, let x be a type variable such that $x \in \mathit{ftv}(\mathcal{C})$ and $x \notin \mathit{active}(\mathcal{C})$. We proceed by proving that this implies $x \notin \mathit{tv}(\textsc{Solve}_\chi(\mathcal{C}))$. In these cases, $\mathcal{C}_2$ denotes the constraint set that is solved recursively. Hence, the induction hypothesis is

$$\mathit{tv}(\textsc{Solve}_\chi(\mathcal{C}_2)) \cap \mathit{ftv}(\mathcal{C}_2) \subseteq \mathit{active}(\mathcal{C}_2).$$

The crucial steps in the proof are marked, and discussed afterwards.

- **Case $\mathcal{C} = \emptyset$:** This holds trivially, since $\mathit{ftv}(\mathcal{C}) = \emptyset$.
- **Case $\mathcal{C} = \{\tau_1 \equiv \tau_2\} \cup \mathcal{C}_1$:** Let $\mathcal{C}_2 = S\mathcal{C}_1$ and $S = \mathit{mgu}(\tau_1, \tau_2)$, so that $\textsc{Solve}_\chi(\mathcal{C}) = \textsc{Solve}_\chi(\mathcal{C}_2) \circ S$.

  $x \in \mathit{ftv}(\mathcal{C}) \ \wedge\ x \notin \mathit{active}(\mathcal{C})$
  $\quad\Rightarrow \{$ *Definition 4.4 (active)* $\}$
  $x \in \mathit{ftv}(\mathcal{C}) \ \wedge\ x \notin \mathit{ftv}(\tau_1) \ \wedge\ x \notin \mathit{ftv}(\tau_2) \ \wedge\ x \notin \mathit{active}(\mathcal{C}_1)$
  $\quad\Rightarrow \{$ *def. ftv* $\}$
  $x \in \mathit{ftv}(\mathcal{C}_1) \ \wedge\ x \notin \mathit{ftv}(\tau_1) \ \wedge\ x \notin \mathit{ftv}(\tau_2) \ \wedge\ x \notin \mathit{active}(\mathcal{C}_1)$
  $\quad\Rightarrow \{$ *Lemma 2.1 (mgu)* $\}$
  $x \in \mathit{ftv}(\mathcal{C}_1) \ \wedge\ x \notin \mathit{active}(\mathcal{C}_1) \ \wedge\ x \notin \mathit{tv}(S)$
  $\quad\Rightarrow \{$ *def. ftv* $\}$
  $x \in \mathit{ftv}(\mathcal{C}_2) \ \wedge\ x \notin \mathit{active}(\mathcal{C}_2) \ \wedge\ x \notin \mathit{tv}(S)$
  $\quad\Rightarrow \{$ *i.h.* $\}$
  $x \notin \mathit{tv}(\textsc{Solve}_\chi(\mathcal{C}_2)) \ \wedge\ x \notin \mathit{tv}(S)$
  $\quad\Rightarrow$
  $x \notin \mathit{tv}(\textsc{Solve}_\chi(\mathcal{C}))$

- **Case $\mathcal{C} = \{\tau_1 \preceq \forall\overline{a}.\tau_2\} \cup \mathcal{C}_1$:** Let $\mathcal{C}_2 = \{\tau_1 \equiv S\tau_2\} \cup \mathcal{C}_1$ and $S = [\overline{a} := \chi_1]$, so that $\textsc{Solve}_\chi(\mathcal{C}) = \textsc{Solve}_{\chi_2}(\mathcal{C}_2)$. Furthermore, we know that $\chi = \chi_1 \cup \chi_2$, $\chi_1 \cap \chi_2 = \emptyset$, and $\chi \cap \mathit{ftv}(\mathcal{C}) = \emptyset$.

  $x \in \mathit{ftv}(\mathcal{C}) \ \wedge\ x \notin \mathit{active}(\mathcal{C})$
  $\quad\Rightarrow \{$ *Definition 4.4 (active)* $\}$
  $x \in \mathit{ftv}(\mathcal{C}) \ \wedge\ x \notin \mathit{ftv}(\tau_1) \ \wedge\ x \notin \mathit{ftv}(\forall\overline{a}.\tau_2) \ \wedge\ x \notin \mathit{active}(\mathcal{C}_1)$
  $\quad\Rightarrow \{$ *def. ftv* $\}$
  $x \in \mathit{ftv}(\mathcal{C}_1) \ \wedge\ x \notin \mathit{ftv}(\tau_1) \ \wedge\ x \notin \mathit{ftv}(\forall\overline{a}.\tau_2) \ \wedge\ x \notin \mathit{active}(\mathcal{C}_1)$
  $\quad\Rightarrow \{$ *def. ftv* $\}$
  $x \in \mathit{ftv}(\mathcal{C}_2) \ \wedge\ x \notin \mathit{ftv}(\tau_1) \ \wedge\ x \notin \mathit{ftv}(\forall\overline{a}.\tau_2) \ \wedge\ x \notin \mathit{active}(\mathcal{C}_1)$
  $\quad\Rightarrow \{$ $x \notin \chi_1$ *(because* $x \in \mathit{ftv}(\mathcal{C})$ *implies* $x \notin \chi$*)* $\}$ **(2)**
  $x \in \mathit{ftv}(\mathcal{C}_2) \ \wedge\ x \notin \mathit{ftv}(\tau_1) \ \wedge\ x \notin \mathit{ftv}(S\tau_2) \ \wedge\ x \notin \mathit{active}(\mathcal{C}_1)$
  $\quad\Rightarrow \{$ *Definition 4.4 (active)* $\}$
  $x \in \mathit{ftv}(\mathcal{C}_2) \ \wedge\ x \notin \mathit{active}(\mathcal{C}_2)$
  $\quad\Rightarrow \{$ *i.h.* $\}$
  $x \notin \mathit{tv}(\textsc{Solve}_{\chi_2}(\mathcal{C}_2))$
  $\quad\Rightarrow$
  $x \notin \mathit{tv}(\textsc{Solve}_\chi(\mathcal{C}))$

  (2) The type scheme $\forall\overline{a}.\tau_2$ is instantiated with fresh type variables from χ_1.

- **Case $\mathcal{C} = \{\tau_1 \leqslant_{\mathcal{M}} \tau_2\} \cup \mathcal{C}_1$:** Let $\mathcal{C}_2 = \{\tau_1 \preceq \sigma\} \cup \mathcal{C}_1$ and $\sigma = \mathit{generalize}(\mathcal{M}, \tau_2)$, so that $\textsc{Solve}_\chi(\mathcal{C}) = \textsc{Solve}_\chi(\mathcal{C}_2)$.

$x \in \mathit{ftv}(\mathcal{C}) \;\wedge\; x \notin \mathit{active}(\mathcal{C})$
  $\Rightarrow \{$ *disjointness* χ *and* $\mathit{ftv}(\mathcal{C})$ $\}$
$x \in \mathit{ftv}(\mathcal{C}) \;\wedge\; x \notin \mathit{active}(\mathcal{C}) \;\wedge\; x \notin \chi$
  $\Rightarrow \{$ *Definition 4.4 (active)* $\}$
$x \notin \mathit{ftv}(\tau_1) \;\wedge\; x \notin (\mathit{ftv}(\mathcal{M}) \cap \mathit{ftv}(\tau_2)) \;\wedge\; x \notin \mathit{active}(\mathcal{C}_1) \;\wedge\; x \notin \chi$
  $\Rightarrow \{$ *def. generalize, def. ftv* $\}$ **(3)**
$x \notin \mathit{ftv}(\tau_1) \;\wedge\; x \notin \mathit{ftv}(\sigma) \;\wedge\; x \notin \mathit{active}(\mathcal{C}_1) \;\wedge\; x \notin \chi$
  $\Rightarrow \{$ *Definition 4.4 (active)* $\}$
$x \notin \mathit{active}(\mathcal{C}_2) \;\wedge\; x \notin \chi$
  $\Rightarrow \{$ *i.h.* $\}$
$(x \in \mathit{ftv}(\mathcal{C}_2) \;\Rightarrow\; x \notin \mathit{tv}(\textsc{Solve}_\chi(\mathcal{C}_2))) \;\wedge\; x \notin \chi$
  $\Rightarrow \{$ *Lemma 4.7* $\}$ **(4)**
$(x \in \mathit{ftv}(\mathcal{C}_2) \;\Rightarrow\; x \notin \mathit{tv}(\textsc{Solve}_\chi(\mathcal{C}_2))) \;\wedge\; (x \notin \mathit{ftv}(\mathcal{C}_2) \;\Rightarrow\; x \notin \mathit{tv}(\textsc{Solve}_\chi(\mathcal{C}_2)))$
  $\Rightarrow$
$x \notin \mathit{tv}(\textsc{Solve}_\chi(\mathcal{C}_2))$
  $\Rightarrow$
$x \notin \mathit{tv}(\textsc{Solve}_\chi(\mathcal{C}))$

(3) $\mathit{generalize}(\mathcal{M}, \tau_2)$ equals $\forall\overline{a}.\tau_2$ with $\overline{a} = \mathit{ftv}(\tau_2) - \mathit{ftv}(\mathcal{M})$. Hence, $\mathit{ftv}(\forall\overline{a}.\tau_2) = \mathit{ftv}(\tau_2) - \overline{a} = \mathit{ftv}(\tau_2) - (\mathit{ftv}(\tau_2) - \mathit{ftv}(\mathcal{M})) = \mathit{ftv}(\mathcal{M}) \cap \mathit{ftv}(\tau_2)$.
(4) Suppose that $x \notin \mathit{ftv}(\mathcal{C}_2)$. Then Lemma 4.7 states that x cannot be in the substitution returned by algorithm Solve for $\mathcal{C}_2$ when we use χ to supply fresh type variables. □

We continue with a proof that algorithm Solve is sound with respect to constraint satisfaction.

Theorem 4.9 (Soundness of Solve). *Let $\mathcal{C}$ be a set of type constraints. Then it holds that* $\textsc{Solve}(\mathcal{C}) \vdash_s \mathcal{C}$.

Proof. By induction on the four cases of algorithm Solve.

- **Case $\mathcal{C} = \emptyset$:** Holds trivially, since $\mathit{id} \vdash_s \mathcal{C}$.
- **Case $\mathcal{C} = \{\tau_1 \equiv \tau_2\} \cup \mathcal{C}_1$:** Let $S = \mathit{mgu}(\tau_1, \tau_2)$.

Proposition 2.1 (mgu)
  $\Rightarrow$
$S\tau_1 = S\tau_2$
  $\Rightarrow \{$ *def.* $(\vdash_s \equiv)$ $\}$
$S \vdash_s \tau_1 \equiv \tau_2$
  $\Rightarrow \{$ *Lemma 4.1* $\}$
$\textsc{Solve}(S\mathcal{C}_1) \circ S \vdash_s \tau_1 \equiv \tau_2$
  $\Rightarrow \{$ *i.h.* $\}$
$\textsc{Solve}(S\mathcal{C}_1) \circ S \vdash_s \tau_1 \equiv \tau_2 \;\wedge\; \textsc{Solve}(S\mathcal{C}_1) \vdash_s S\mathcal{C}_1$
  $\Rightarrow \{$ *idempotency of* S $\}$
$\textsc{Solve}(S\mathcal{C}_1) \circ S \vdash_s \tau_1 \equiv \tau_2 \;\wedge\; \textsc{Solve}(S\mathcal{C}_1) \circ S \vdash_s S\mathcal{C}_1$

$\Rightarrow$ { *def.* $(\vdash_s \;\textit{set})$ }
$\textsc{Solve}(S\mathcal{C}_1) \circ S \vdash_s \mathcal{C}$

- **Case** $\mathcal{C} = \{\tau_1 \preceq \forall\overline{a}.\tau_2\} \cup \mathcal{C}_1$**:** Let $S = \textsc{Solve}_{\chi_2}(\{\tau_1 \equiv [\overline{a} := \chi_1]\tau_2\} \cup \mathcal{C}_1)$.

i.h.
$\Rightarrow$
$S \vdash_s \{\tau_1 \equiv [\overline{a} := \chi_1]\tau_2\} \cup \mathcal{C}_1$
$\Rightarrow$ { *def.* $(\vdash_s \;\textit{set})$ }
$S \vdash_s \tau_1 \equiv [\overline{a} := \chi_1]\tau_2 \;\wedge\; S \vdash_s \mathcal{C}_1$
$\Rightarrow$ { *def.* $(\vdash_s \equiv)$ }
$S\tau_1 = S([\overline{a} := \chi_1]\tau_2) \;\wedge\; S \vdash_s \mathcal{C}_1$
$\Rightarrow$ { *Lemma 2.2 (instance-of), witnessed by* $S\chi_1$ } **(5)**
$S\tau_1 < S(\forall\overline{a}.\tau_2) \;\wedge\; S \vdash_s \mathcal{C}_1$
$\Rightarrow$ { *def.* $(\vdash_s \preceq)$ }
$S \vdash_s \tau_1 \preceq \forall\overline{a}.\tau_2 \;\wedge\; S \vdash_s \mathcal{C}_1$
$\Rightarrow$ { *def.* $(\vdash_s \;\textit{set})$ }
$S \vdash_s \mathcal{C}$

(5) If the types $S\tau_1$ and $S([\overline{a} := \chi_1]\tau_2)$ are equal, then the former is also an instance of the latter (by (Sub-Mono)). The type $S([\overline{a} := \chi_1]\tau_2)$ can be rewritten to $[\overline{a} := S'\chi_1](S'\tau_2)$, where S' is S without $\overline{a}$ in its domain. Then by (Sub-Inst), we get that $S\tau_1 < \forall\overline{a}.S'\tau_2$, and, as a result, $S\tau_1 < S(\forall\overline{a}.\tau_2)$.

- **Case** $\mathcal{C} = \{\tau_1 \leqslant_{\mathcal{M}} \tau_2\} \cup \mathcal{C}_1$**:** Let $\sigma = \textit{generalize}(\mathcal{M}, \tau_2)$ and $S = \textsc{Solve}(\{\tau_1 \preceq \sigma\} \cup \mathcal{C}_1)$.

$\textit{ftv}(\tau_2) \cap \textit{active}(\mathcal{C}) \subseteq \textit{ftv}(\mathcal{M})$
$\Rightarrow$ { *Lemma 4.8* } **(6)**
$\textit{ftv}(\tau_2) \cap \textit{tv}(S) \subseteq \textit{ftv}(\mathcal{M})$
$\Rightarrow$ { *i.h.* }
$\textit{ftv}(\tau_2) \cap \textit{tv}(S) \subseteq \textit{ftv}(\mathcal{M}) \;\wedge\; S \vdash_s \{\tau_1 \preceq \sigma\} \cup \mathcal{C}_1$
$\Rightarrow$ { *def.* $(\vdash_s \;\textit{set})$ }
$\textit{ftv}(\tau_2) \cap \textit{tv}(S) \subseteq \textit{ftv}(\mathcal{M}) \;\wedge\; S \vdash_s \tau_1 \preceq \sigma \;\wedge\; S \vdash_s \mathcal{C}_1$
$\Rightarrow$ { *def.* $(\vdash_s \preceq)$ }
$\textit{ftv}(\tau_2) \cap \textit{tv}(S) \subseteq \textit{ftv}(\mathcal{M}) \;\wedge\; S\tau_1 < S\sigma \;\wedge\; S \vdash_s \mathcal{C}_1$
$\Rightarrow$ { *Lemma 4.6* } **(7)**
$S\tau_1 < \textit{generalize}(S\mathcal{M}, S\tau_2) \;\wedge\; S \vdash_s \mathcal{C}_1$
$\Rightarrow$ { *def.* $(\vdash_s \leqslant)$ }
$S \vdash_s \tau_1 \leqslant_{\mathcal{M}} \tau_2 \;\wedge\; S \vdash_s \mathcal{C}_1$
$\Rightarrow$ { *def.* $(\vdash_s \;\textit{set})$ }
$S \vdash_s \mathcal{C}$

(6) Lemma 4.8 expresses that $\textit{tv}(S) \cap \textit{ftv}(\mathcal{C}) \subseteq \textit{active}(\mathcal{C})$, which implies that $\textit{tv}(S) \cap \textit{ftv}(\tau_2) \subseteq \textit{active}(\mathcal{C})$ since $\textit{ftv}(\tau_2) \subseteq \textit{ftv}(\mathcal{C})$. Hence, we conclude that $\textit{ftv}(\tau_2) \cap \textit{active}(\mathcal{C}) \supseteq \textit{ftv}(\tau_2) \cap \textit{tv}(S)$.
(7) Condition **(1)** in Lemma 4.6 holds for τ_2, $\mathcal{M}$, and S. As a result, we may replace $S\sigma$ by $\textit{generalize}(S\mathcal{M}, S\tau_2)$ in the right-hand side of the instance-of judgement. $\square$

We expect Solve not only to be sound, but also complete. If there exists a substitution that satisfies an orderable set of type constraints, then a substitution is returned by the algorithm, which is different from the error substitution. This completeness result does not hold for all constraint sets, but only for the sets that can be ordered. Theorem 4.10 elaborates on this property.

Theorem 4.10 (Completeness of Solve). *Let $\mathcal{C}$ be an orderable constraint set. If there exists a substitution S such that $S \vdash_s \mathcal{C}$, then* Solve$(\mathcal{C}) \neq \top$. *Moreover,* Solve$(\mathcal{C})$ *is a most general substitution satisfying $\mathcal{C}$.*

Proof. By induction on the four cases of algorithm Solve.

- **Case $\mathcal{C} = \emptyset$:** Holds trivially.
- **Case $\mathcal{C} = \{\tau_1 \equiv \tau_2\} \cup \mathcal{C}_1$:** $S = mgu(\tau_1, \tau_2)$ is a most general substitution that satisfies the constraint $\tau_1 \equiv \tau_2$. This substitution is applied to the constraints in $\mathcal{C}_1$, which are subsequently solved. Composing the two substitutions yields a (most general) substitution that satisfies all in $\mathcal{C}$.
- **Case $\mathcal{C} = \{\tau_1 \preceq \forall\overline{a}.\tau_2\} \cup \mathcal{C}_1$:** An explicit instance constraint is solved by instantiating the type scheme, i.e., replacing $\overline{a}$ with fresh type variables from χ_1. From Lemma 2.2 we get that

$$S \vdash_s \tau_1 \preceq \forall\overline{a}.\tau_2 \quad \Longleftrightarrow \quad \exists R : dom(R) \subseteq \overline{a} \ \wedge \ S \vdash_s \tau_1 \equiv R\tau_2 .$$

  This implies that we do return a most general substitution that satisfies $\mathcal{C}$, except that the domain of this substitution may contain newly introduced type variables from χ_1.
- **Case $\mathcal{C} = \{\tau_1 \leqslant_{\mathcal{M}} \tau_2\} \cup \mathcal{C}_1$:** Let S be Solve$(\mathcal{C})$. From Lemma 4.6, we may conclude that

$$S \vdash_s \tau_1 \leqslant_{\mathcal{M}} \tau_2 \quad \Longleftrightarrow \quad S \vdash_s \tau_1 \preceq generalize(\mathcal{M}, \tau_2)$$

  whenever $ftv(\tau_2) \cap tv(S) \subseteq \mathcal{M}$. This condition holds because the side condition for solving an implicit instance constraint ensures that type variables in $ftv(\tau_2) - ftv(\mathcal{M})$ are not active. Hence, these type variables cannot appear in the domain of S, nor in its co-domain (Lemma 4.8). Therefore, solving an implicit instance constraint by translating it into an explicit instance constraint returns a most general unifier. □

4.5 Correctness

Section 4.3 introduced a set of type rules for collecting type constraints for an expression. The collected constraints are solved with algorithm Solve from Section 4.4. These are the two main ingredients for a constraint-based type inference algorithm, for which we show that it is correct with respect to the Hindley-Milner type rules.

We start by lifting equality and explicit instance constraints to work for lists of pairs.

$$\begin{array}{lll} \mathcal{A}_1 \equiv \mathcal{A}_2 & =_{def} & \{\tau_1 \equiv \tau_2 \mid x : \tau_1 \in \mathcal{A}_1, x : \tau_2 \in \mathcal{A}_2\} \qquad (\mathit{lift} \equiv) \\ \mathcal{A} \preceq \Gamma & =_{def} & \{\tau \preceq \sigma \mid x : \tau \in \mathcal{A}, x : \sigma \in \Gamma\} \qquad (\mathit{lift} \preceq) \end{array}$$

From these definitions it follows immediately that the following properties hold.

$$\begin{array}{lll} (\mathcal{A} \backslash x) \preceq \Gamma & = & \mathcal{A} \preceq (\Gamma \backslash x) \qquad (\textit{lift-remove}) \\ (\mathcal{A}_1 \cup \mathcal{A}_2) \preceq \Gamma & = & (\mathcal{A}_1 \preceq \Gamma) \cup (\mathcal{A}_2 \preceq \Gamma) \qquad (\textit{split-left}) \end{array}$$

The first equation says that instead of removing assumptions about an identifier in the left operand, one can also remove the assumptions in the right operand (and vice versa). Property (*split-left*) shows that set union distributes over the lifted constraints. These propositions apply equally well to the lifted equality constraint on sets (that is, after replacing $\preceq$ by $\equiv$).

Next, we present Algorithm INFER, which infers a type for an expression in the context of a type environment Γ.

Definition 4.6 (Algorithm INFER).

$$\text{INFER} :: (\mathit{TypeEnvironment}, \mathit{Expression}) \rightarrow \mathit{Type}$$

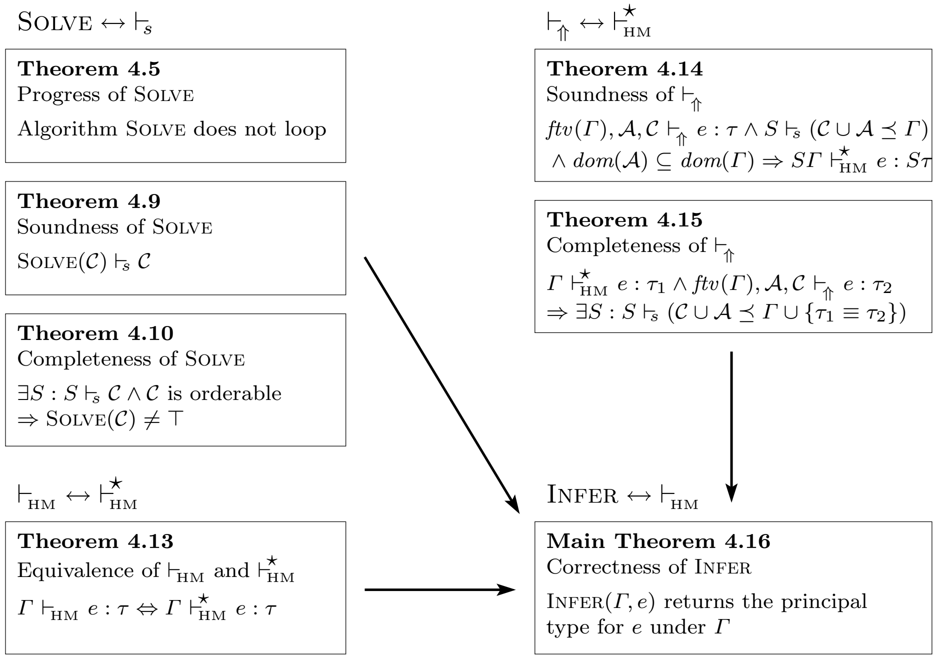
$$\text{INFER}(\Gamma, e) \;=_{def}\; \begin{cases} \text{SOLVE}(\mathcal{C} \cup \mathcal{A} \preceq \Gamma)\tau & \textit{if } \mathit{dom}(\mathcal{A}) \subseteq \mathit{dom}(\Gamma) \\ \mathit{ErrorType} & \textit{otherwise} \end{cases}$$

$$\textit{with } \mathcal{M}, \mathcal{A}, \mathcal{C} \vdash_{\Uparrow} e : \tau \textit{ and } \mathcal{M} = \mathit{ftv}(\Gamma)$$

The bottom-up type rules are used to collect type constraints for expression e. We choose $\mathcal{M}$ to be the set of free type variables of Γ. This gives us an assumption set $\mathcal{A}$, a constraint set $\mathcal{C}$, and a type τ. The set $\mathcal{A}$ contains assertions about the unbound identifiers of e. Hence, $\mathit{dom}(\mathcal{A}) \subseteq \mathit{dom}(\Gamma)$ must hold, or else, INFER returns *ErrorType* to indicate an error. We extend the constraint set $\mathcal{C}$ with $\mathcal{A} \preceq \Gamma$, which relates types assigned to the unbound identifiers of e to their corresponding type scheme in Γ. Solving all the constraints yields a substitution satisfying the constraints (or *ErrorType* in case the constraint set is inconsistent). Finally, we apply this substitution to τ.

In the remaining part of this chapter, we establish soundness and completeness for algorithm INFER, and prove that it is indeed a correct implementation of the Hindley-Milner type system. If desired, this correctness proof may be skipped altogether.

Figure 4.7 gives an outline of the correctness proof for algorithm INFER. Soundness, completeness, and progress of SOLVE have been proven in Section 4.4. However, we still have to prove that the set of constraints we are about to solve is orderable. Then we introduce a modified version of the Hindley-Milner type system ($\vdash^{\star}_{\text{HM}}$), and show that this is equivalent to the original system. Next, we prove that our constraint-based, bottom-up type rules are correct with respect to the modified Hindley-Milner type system. In fact, we construct a mapping between deduction trees of the two sets of type rules, in both directions. We conclude with a correctness proof for INFER, which combines earlier results.



$\textsc{Solve} \leftrightarrow \vdash_s$

Theorem 4.5
Progress of $\textsc{Solve}$
Algorithm $\textsc{Solve}$ does not loop

Theorem 4.9
Soundness of $\textsc{Solve}$
$\textsc{Solve}(\mathcal{C}) \vdash_s \mathcal{C}$

Theorem 4.10
Completeness of $\textsc{Solve}$
$\exists S : S \vdash_s \mathcal{C} \wedge \mathcal{C}$ is orderable
$\Rightarrow \textsc{Solve}(\mathcal{C}) \neq \top$

$\vdash_{\Uparrow} \leftrightarrow \vdash^{\star}_{\text{HM}}$

Theorem 4.14
Soundness of $\vdash_{\Uparrow}$
$\mathit{ftv}(\Gamma), \mathcal{A}, \mathcal{C} \vdash_{\Uparrow} e : \tau \wedge S \vdash_s (\mathcal{C} \cup \mathcal{A} \preceq \Gamma)$
$\wedge\ \mathit{dom}(\mathcal{A}) \subseteq \mathit{dom}(\Gamma) \Rightarrow S\Gamma \vdash^{\star}_{\text{HM}} e : S\tau$

Theorem 4.15
Completeness of $\vdash_{\Uparrow}$
$\Gamma \vdash^{\star}_{\text{HM}} e : \tau_1 \wedge \mathit{ftv}(\Gamma), \mathcal{A}, \mathcal{C} \vdash_{\Uparrow} e : \tau_2$
$\Rightarrow \exists S : S \vdash_s (\mathcal{C} \cup \mathcal{A} \preceq \Gamma \cup \{\tau_1 \equiv \tau_2\})$

$\vdash_{\text{HM}} \leftrightarrow \vdash^{\star}_{\text{HM}}$

Theorem 4.13
Equivalence of $\vdash_{\text{HM}}$ and $\vdash^{\star}_{\text{HM}}$
$\Gamma \vdash_{\text{HM}} e : \tau \Leftrightarrow \Gamma \vdash^{\star}_{\text{HM}} e : \tau$

$\textsc{Infer} \leftrightarrow \vdash_{\text{HM}}$

Main Theorem 4.16
Correctness of $\textsc{Infer}$
$\textsc{Infer}(\Gamma, e)$ returns the principal type for e under Γ

Figure 4.7. Outline of the correctness proof for algorithm $\textsc{Infer}$

One discrepancy between the Hindley-Milner type rules and the bottom type rules is how they record which type variables are monomorphic, and, in particular, how they cope with the shadowing of expression identifiers. The Hindley-Milner type rules for lambda abstraction and let expressions may remove assumptions about identifiers that are no longer in scope from Γ, thereby possibly shrinking the set of monomorphic type variables. On the other hand, the set $\mathcal{M}$ containing the monomorphic types in the $\vdash_{\Uparrow}$ type rules can only grow. This makes the two sets of type rules practically incomparable, since the monomorphic type variables play an important role in generalizing types to obtain polymorphic types for identifiers defined in let expressions. To remedy this problem, we define an operator to update a type environment without reducing the set of free type variables.

Definition 4.7 (Update operator). *Let $\underline{trash}$ be a special identifier that does not appear in expressions. The operator $\oslash$ updates a type environment, and is defined as follows.*

$$(x : \sigma_1) \oslash \Gamma \quad =_{def} \quad \begin{cases} \Gamma \backslash \{x, \underline{trash}\} \cup \{x : \sigma_1, \underline{trash} : \sigma'\} & \text{if } \{(x : \sigma_2), (\underline{trash} : \sigma_3)\} \subseteq \Gamma \\ \Gamma \backslash x \cup \{x : \sigma_1, \underline{trash} : \sigma_2\} & \text{if } (x : \sigma_2) \in \Gamma \\ \Gamma \cup \{x : \sigma_1\} & \text{otherwise} \end{cases}$$

where σ' is some type scheme such that $\mathit{ftv}(\sigma') = \mathit{ftv}(\sigma_2) \cup \mathit{ftv}(\sigma_3)$.

$\boxed{\Gamma \vdash^{\star}_{\mathrm{HM}} e : \tau}$ *Typing judgement*

$$\frac{(x : \tau_1) \oslash \Gamma \vdash^{\star}_{\mathrm{HM}} e : \tau_2}{\Gamma \vdash^{\star}_{\mathrm{HM}} \lambda x \to e : (\tau_1 \to \tau_2)} \quad (\text{HM-}\textsc{Abs})^{\star}$$

$$\frac{\Gamma \vdash^{\star}_{\mathrm{HM}} e_1 : \tau_1 \qquad (x : \mathit{generalize}(\Gamma, \tau_1)) \oslash \Gamma \vdash^{\star}_{\mathrm{HM}} e_2 : \tau_2}{\Gamma \vdash^{\star}_{\mathrm{HM}} \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2} \quad (\text{HM-}\textsc{Let})^{\star}$$

Figure 4.8. Modified Hindley-Milner type rules

Lemma 4.11 (Monotonicity). $\mathit{ftv}(\Gamma) \subseteq \mathit{ftv}((x : \sigma_1) \oslash \Gamma)$

Proof. Holds trivially for all three cases in the definition of $\oslash$. □

Lemma 4.12. $\Gamma \backslash x \cup \{x : \sigma\} \vdash_{\mathrm{HM}} e : \tau \Leftrightarrow (x : \sigma) \oslash \Gamma \vdash_{\mathrm{HM}} e : \tau$

Proof. By definition, $\Gamma \backslash x \cup \{x : \sigma\}$ equals $((x : \sigma) \oslash \Gamma) \backslash \underline{\mathit{trash}}$. Because $\underline{\mathit{trash}} \notin \mathit{fev}(e)$, we can use Lemma 2.4 and Lemma 2.3 to complete the proof. □

The following two properties of $\oslash$ follow directly from Definition 4.7.

$$\begin{array}{lcll} (\mathcal{A} \preceq \{x : \sigma\}) \cup (\mathcal{A} \preceq \Gamma \backslash x) & = & \mathcal{A} \preceq ((x : \sigma) \oslash \Gamma) & (\mathit{split} \oslash) \\ \mathit{dom}((x : \sigma) \oslash \Gamma) & \supseteq & \{x\} \cup \mathit{dom}(\Gamma) & (\mathit{domain} \oslash) \end{array}$$

We devise new type rules for lambda abstractions and let expressions (see Figure 4.8). This gives us a new type system ($\vdash^{\star}_{\mathrm{HM}}$), which is equivalent to the original set of type rules.

Definition 4.8 ($\vdash^{\star}_{\mathrm{HM}}$). *Let $\vdash^{\star}_{\mathrm{HM}}$ be a variant of the Hindley-Milner type system, in which* (HM-Abs) *and* (HM-Let) *are replaced by* (HM-Abs)* *and* (HM-Let)*. *Figure 4.8 contains the two modified type rules. To avoid confusion, we also write* (HM-Var)* *and* (HM-App)* *for the (unmodified) type rules when used in combination with the new rules.*

Theorem 4.13 (Equivalence of $\vdash_{\mathrm{HM}}$ and $\vdash^{\star}_{\mathrm{HM}}$). *The type systems $\vdash_{\mathrm{HM}}$ and $\vdash^{\star}_{\mathrm{HM}}$ are equivalent. Hence,*

$$\Gamma \vdash_{\mathrm{HM}} e : \tau \iff \Gamma \vdash^{\star}_{\mathrm{HM}} e : \tau.$$

Proof. Follows from Lemma 4.12. □

We proceed with a soundness and completeness proof for the $\vdash_{\Uparrow}$ type rules, with respect to the modified Hindley-Milner type system. For these two proofs, we use Definition 4.3 for constraint satisfaction. The intuition behind the soundness proof is that we can construct a derivation with the $\vdash^{\star}_{\mathrm{HM}}$ rules for each possible derivation with the bottom-up type rules once we are given a substitution that satisfies the type constraints.

Theorem 4.14 (Soundness of $\vdash_{\Uparrow}$). *Given expression e and type environment $\varGamma$, from which $\mathcal{M}, \mathcal{A}, \mathcal{C} \vdash_{\Uparrow} e : \tau$ with $\mathcal{M} = \mathit{ftv}(\varGamma)$. Then*

$$\forall S : S \vdash_s \mathcal{C} \;\wedge\; S \vdash_s \mathcal{A} \preceq \varGamma \;\wedge\; \mathit{dom}(\mathcal{A}) \subseteq \mathit{dom}(\varGamma) \quad\Longrightarrow\quad S\varGamma \vdash^{\star}_{\mathrm{HM}} e : S\tau.$$

Proof. By induction on the expression. To complete the proof, we have to strengthen our induction hypothesis: let invariant $\mathcal{I}$ be $\mathit{ftv}(S\mathcal{M}) = \mathit{ftv}(S\varGamma)$, which holds initially because $\mathcal{M} = \mathit{ftv}(\varGamma)$. In addition to the implication formulated in Theorem 4.14, we have to prove $\mathcal{I}$ whenever we use the induction hypothesis.

- **Case variable:** $\mathcal{M}, \{x : \beta\}, \emptyset \vdash_{\Uparrow} x : \beta$.

  $S \vdash_s \emptyset \;\wedge\; S \vdash_s \{x:\beta\} \preceq \varGamma \;\wedge\; \{x\} \subseteq \mathit{dom}(\varGamma)$
  $\quad\Rightarrow \{\ \textit{def.}\ (\textit{lift} \preceq)\ \}$
  $S\beta < S\varGamma(x)$
  $\quad\Rightarrow \{\ (\text{HM-Var})^{\star}\ \}$
  $S\varGamma \vdash^{\star}_{\mathrm{HM}} x : S\beta$

  ▶ *There is no induction hypothesis for which we have to prove $\mathcal{I}$.*

- **Case application:** $\mathcal{M}, \mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_{new} \vdash_{\Uparrow} e_1\ e_2 : \beta$ with $\mathcal{C}_{new} = \{\tau_1 \equiv \tau_2 \to \beta\}$.

  $S \vdash_s \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_{new} \;\wedge\; S \vdash_s (\mathcal{A}_1 \cup \mathcal{A}_2 \preceq \varGamma) \;\wedge\; \mathit{dom}(\mathcal{A}_1 \cup \mathcal{A}_2) \subseteq \mathit{dom}(\varGamma)$
  $\quad\Rightarrow \{\ (\vdash_s\ \textit{set}),\ \textit{prop.}\ (\textit{split-left})\ \}$
  $S \vdash_s \mathcal{C}_1 \;\wedge\; S \vdash_s \mathcal{C}_2 \;\wedge\; S \vdash_s \mathcal{C}_{new} \;\wedge\; S \vdash_s \mathcal{A}_1 \preceq \varGamma \;\wedge\; S \vdash_s \mathcal{A}_2 \preceq \varGamma$
  $\qquad\qquad \wedge\; \mathit{dom}(\mathcal{A}_1) \subseteq \mathit{dom}(\varGamma) \;\wedge\; \mathit{dom}(\mathcal{A}_2) \subseteq \mathit{dom}(\varGamma)$
  $\quad\Rightarrow \{\ \textit{i.h. for } e_1 \textit{ and } e_2\ \}$
  $S\varGamma \vdash^{\star}_{\mathrm{HM}} e_1 : S\tau_1 \;\wedge\; S\varGamma \vdash^{\star}_{\mathrm{HM}} e_2 : S\tau_2 \;\wedge\; S \vdash_s \mathcal{C}_{new}$
  $\quad\Rightarrow \{\ \textit{def.}\ (\vdash_s \equiv)\ \}$
  $S\varGamma \vdash^{\star}_{\mathrm{HM}} e_1 : S\tau_1 \;\wedge\; S\varGamma \vdash^{\star}_{\mathrm{HM}} e_2 : S\tau_2 \;\wedge\; S\tau_1 = S(\tau_2 \to \beta)$
  $\quad\Rightarrow$
  $S\varGamma \vdash^{\star}_{\mathrm{HM}} e_1 : S\tau_2 \to S\beta \;\wedge\; S\varGamma \vdash^{\star}_{\mathrm{HM}} e_2 : S\tau_2$
  $\quad\Rightarrow \{\ (\text{HM-App})^{\star}\ \}$
  $S\varGamma \vdash^{\star}_{\mathrm{HM}} e_1\ e_2 : S\beta$

  ▶ *Invariant $\mathcal{I}$ holds trivially because $\varGamma$ nor $\mathcal{M}$ changes in the premises of* (HM-App)* *and* (BU-App).

- **Case abstraction:** $\mathcal{M}, \mathcal{A}\backslash x, \mathcal{C} \cup \mathcal{C}_{new} \vdash_{\Uparrow} \lambda x \to e : \beta_2$ with $\mathcal{C}_{new} = \{\beta_2 \equiv \beta_1 \to \tau\} \cup \{\beta_1 \equiv \tau' \mid x : \tau' \in \mathcal{A}\}$.

  $S \vdash_s \mathcal{C} \cup \mathcal{C}_{new} \;\wedge\; S \vdash_s (\mathcal{A}\backslash x \preceq \varGamma) \;\wedge\; \mathit{dom}(\mathcal{A}\backslash x) \subseteq \mathit{dom}(\varGamma)$
  $\quad\Rightarrow \{\ \textit{prop.}\ (\textit{lift-remove})\ \}$
  $S \vdash_s \mathcal{C} \cup \mathcal{C}_{new} \;\wedge\; S \vdash_s (\mathcal{A} \preceq \varGamma\backslash x) \;\wedge\; \mathit{dom}(\mathcal{A}\backslash x) \subseteq \mathit{dom}(\varGamma)$
  $\quad\Rightarrow \{\ (\textit{lift} \equiv),\ \textit{constraints from } \mathcal{C}_{new}\ \}$
  $S \vdash_s \mathcal{C} \cup \mathcal{C}_{new} \;\wedge\; S \vdash_s (\mathcal{A} \equiv \{x:\beta_1\}) \;\wedge\; S \vdash_s (\mathcal{A} \preceq \varGamma\backslash x) \;\wedge\; \mathit{dom}(\mathcal{A}\backslash x) \subseteq \mathit{dom}(\varGamma)$
  $\quad\Rightarrow \{\ \textit{Lemma 4.2 lifted to sets}\ \}$
  $S \vdash_s \mathcal{C} \cup \mathcal{C}_{new} \;\wedge\; S \vdash_s (\mathcal{A} \preceq \{x:\beta_1\}) \;\wedge\; S \vdash_s (\mathcal{A} \preceq \varGamma\backslash x) \;\wedge\; \mathit{dom}(\mathcal{A}\backslash x) \subseteq \mathit{dom}(\varGamma)$

$\quad\Rightarrow \{ \textit{prop. (split} \oslash) \}$
$S \vdash_s \mathcal{C} \cup \mathcal{C}_{new} \ \wedge\ S \vdash_s (\mathcal{A} \preceq (x:\beta_1) \oslash \Gamma) \ \wedge\ dom(\mathcal{A}\backslash x) \subseteq dom(\Gamma)$
$\quad\Rightarrow \{ (\vdash_s\ \textit{set}), \textit{prop. (domain} \oslash) \}$
$S \vdash_s \mathcal{C} \ \wedge\ S \vdash_s \mathcal{C}_{new} \ \wedge\ S \vdash_s (\mathcal{A} \preceq (x:\beta_1) \oslash \Gamma) \ \wedge\ dom(\mathcal{A}) \subseteq dom((x:\beta_1) \oslash \Gamma)$
$\quad\Rightarrow \{ \textit{i.h.} \}$
$S((x:\beta_1) \oslash \Gamma) \vdash^{\star}_{\text{HM}} e : S\tau \ \wedge\ S \vdash_s \mathcal{C}_{new}$
$\quad\Rightarrow \{ (\text{HM-ABS})^{\star} \}$
$S\Gamma \vdash^{\star}_{\text{HM}} \lambda x \rightarrow e : S\beta_1 \rightarrow S\tau \ \wedge\ S \vdash_s \mathcal{C}_{new}$
$\quad\Rightarrow \{ \textit{def} (\vdash_s \equiv) \}$
$S\Gamma \vdash^{\star}_{\text{HM}} \lambda x \rightarrow e : S\beta_1 \rightarrow S\tau \ \wedge\ S\beta_2 = S\beta_1 \rightarrow S\tau$
$\quad\Rightarrow$
$S\Gamma \vdash^{\star}_{\text{HM}} \lambda x \rightarrow e : S\beta_2$

▶ *We still have to prove that invariant $\mathcal{I}$ holds at the point where we use the induction hypothesis. This is straightforward: because of Definition 4.7, we conclude that $ftv(S\mathcal{M}) = ftv(S\Gamma)$ implies $ftv(S(\mathcal{M} \cup \{\beta_1\})) = ftv(S((x:\beta_1) \oslash \Gamma))$.*

- **Case let expression:** $\mathcal{M}, \mathcal{A}_1 \cup \mathcal{A}_2\backslash x, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_{new} \vdash_{\Uparrow} \textbf{let } x = e_1 \textbf{ in } e_2 : \beta$ with $\mathcal{C}_{new} = \{\beta \equiv \tau_2\} \cup \{\tau' \leqslant_{\mathcal{M}} \tau_1 \mid x:\tau' \in \mathcal{A}_2\}$. Let $\sigma = generalize(S\mathcal{M}, S\tau_1)$. Note that σ equals $generalize(S\Gamma, S\tau_1)$ because of our invariant $\mathcal{I}$.

$S \vdash_s \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_{new} \ \wedge\ S \vdash_s ((\mathcal{A}_1 \cup \mathcal{A}_2\backslash x) \preceq \Gamma) \ \wedge\ dom(\mathcal{A}_1 \cup \mathcal{A}_2\backslash x) \subseteq dom(\Gamma)$
$\quad\Rightarrow \{ (\vdash_s\ \textit{set}), \textit{prop. (split-left)} \}$
$S \vdash_s \mathcal{C}_1 \ \wedge\ S \vdash_s \mathcal{C}_2 \ \wedge\ S \vdash_s \mathcal{C}_{new} \ \wedge\ S \vdash_s \mathcal{A}_1 \preceq \Gamma \ \wedge\ S \vdash_s (\mathcal{A}_2\backslash x \preceq \Gamma)$
$\qquad \wedge\ dom(\mathcal{A}_1) \subseteq dom(\Gamma) \ \wedge\ dom(\mathcal{A}_2\backslash x) \subseteq dom(\Gamma)$
$\quad\Rightarrow \{ \textit{i.h. for } e_1 \}$
$S\Gamma \vdash^{\star}_{\text{HM}} e_1 : S\tau_1 \ \wedge\ S \vdash_s \mathcal{C}_2 \ \wedge\ S \vdash_s \mathcal{C}_{new} \ \wedge\ S \vdash_s (\mathcal{A}_2\backslash x \preceq \Gamma)$
$\qquad \wedge\ dom(\mathcal{A}_2\backslash x) \subseteq dom(\Gamma)$
$\quad\Rightarrow \{ \textit{prop. (lift-remove)} \}$
$S\Gamma \vdash^{\star}_{\text{HM}} e_1 : S\tau_1 \ \wedge\ S \vdash_s \mathcal{C}_2 \ \wedge\ S \vdash_s \mathcal{C}_{new} \ \wedge\ S \vdash_s (\mathcal{A}_2 \preceq \Gamma\backslash x)$
$\qquad \wedge\ dom(\mathcal{A}_2\backslash x) \subseteq dom(\Gamma)$
$\quad\Rightarrow \{ \textit{Lemma 4.3, (lift} \preceq) \}$ **(8)**
$S\Gamma \vdash^{\star}_{\text{HM}} e_1 : S\tau_1 \ \wedge\ S \vdash_s \mathcal{C}_2 \ \wedge\ S \vdash_s \mathcal{C}_{new} \ \wedge\ S \vdash_s (\mathcal{A}_2 \preceq \Gamma\backslash x) \ \wedge\ S \vdash_s \mathcal{A}_2 \preceq \{x:\sigma\}$
$\qquad \wedge\ dom(\mathcal{A}_2\backslash x) \subseteq dom(\Gamma)$
$\quad\Rightarrow \{ \textit{prop. (split} \oslash) \}$
$S\Gamma \vdash^{\star}_{\text{HM}} e_1 : S\tau_1 \ \wedge\ S \vdash_s \mathcal{C}_2 \ \wedge\ S \vdash_s \mathcal{C}_{new} \ \wedge\ S \vdash_s (\mathcal{A}_2 \preceq (x:\sigma) \oslash \Gamma)$
$\qquad \wedge\ dom(\mathcal{A}_2\backslash x) \subseteq dom(\Gamma)$
$\quad\Rightarrow \{ \textit{prop. (domain} \oslash) \}$
$S\Gamma \vdash^{\star}_{\text{HM}} e_1 : S\tau_1 \ \wedge\ S \vdash_s \mathcal{C}_2 \ \wedge\ S \vdash_s \mathcal{C}_{new} \ \wedge\ S \vdash_s (\mathcal{A}_2 \preceq (x:\sigma) \oslash \Gamma)$
$\qquad \wedge\ dom(\mathcal{A}_2) \subseteq dom((x:\sigma) \oslash \Gamma)$
$\quad\Rightarrow \{ \textit{i.h. for } e_2, \textit{invariant } \mathcal{I} \}$ **(9)**
$S\Gamma \vdash^{\star}_{\text{HM}} e_1 : S\tau_1 \ \wedge\ S((x:\sigma) \oslash \Gamma) \vdash^{\star}_{\text{HM}} e_2 : S\tau_2 \ \wedge\ S \vdash_s \mathcal{C}_{new}$
$\quad\Rightarrow \{ (\text{HM-LET})^{\star} \}$
$S\Gamma \vdash^{\star}_{\text{HM}} \textbf{let } x = e_1 \textbf{ in } e_2 : S\tau_2 \ \wedge\ S \vdash_s \mathcal{C}_{new}$

$\Rightarrow \{ \textit{def}\,(\vdash_s \equiv) \}$
$S\Gamma \vdash^{\star}_{\text{HM}} \textbf{let } x = e_1 \textbf{ in } e_2 : S\tau_2 \ \wedge\ S\beta = S\tau_2$
$\Rightarrow$
$S\Gamma \vdash^{\star}_{\text{HM}} \textbf{let } x = e_1 \textbf{ in } e_2 : S\beta$

(8) The implicit instance constraints in $\mathcal{C}_{new}$ are satisfied by the substitution S. Lemma 4.3 states that an implicit instance constraint can be written as an explicit instance constraint. By combining this lemma with (*lift* $\preceq$), we conclude $S \vdash_s \mathcal{A}_2 \preceq \{x : \sigma\}$.
(9) Consider the type environment $(x : \sigma) \otimes \Gamma$, in which the type scheme σ is defined by *generalize*$(S\mathcal{M}, S\tau_1)$. The invariant $\mathcal{I}$ tells that *ftv*$(S\mathcal{M})$ equals *ftv*$(S\Gamma)$, and, as a result, we may use the induction hypothesis for e_2.

▶ *We have to establish $\mathcal{I}$ for both induction hypotheses. For e_1, the invariant clearly holds. For e_2, we need to show that ftv$(S\mathcal{M})$ = ftv$(S\Gamma)$ implies ftv$(S\mathcal{M})$ = ftv$(S((x:\sigma) \otimes \Gamma))$, where σ is generalize$(S\Gamma, S\tau_1)$. This holds since ftv$(\sigma) \subseteq$ ftv$(S\Gamma)$ by the definition of generalization.* □

We prove completeness for the bottom-up type rules. This involves showing that for each judgement that can be derived with the modified Hindley-Milner rules, there is a corresponding judgement with the $\vdash_{\Uparrow}$ rules. This is not a surprise, as we can construct a judgement for each expression – including ill-typed expressions. In addition, we prove that we can construct a substitution for this $\vdash_{\Uparrow}$ judgement that satisfies certain constraints.

Theorem 4.15 (Completeness of $\vdash_{\Uparrow}$). *Given expression e and type environment Γ, we construct $\mathcal{M}, \mathcal{A}, \mathcal{C} \vdash_{\Uparrow} e : \tau_1$ with $\mathcal{M}$ = ftv(Γ). Then*

$$\Gamma \vdash^{\star}_{\text{HM}} e : \tau_2 \implies \exists S : S \vdash_s \mathcal{C} \ \wedge\ S \vdash_s \mathcal{A} \preceq \Gamma \ \wedge\ S \vdash_s \tau_1 \equiv \tau_2 \ \wedge\ S \neq \top.$$

Proof. By induction on the expression. We use that for such a substitution *ftv*$(S\mathcal{M})$ = *ftv*$(S\Gamma)$ is invariant. The proof for this invariant is the same as for Theorem 4.14. The domains of the substitutions constructed in this proof are limited to the fresh type variables introduced by the bottom-up type rules. Hence, the domains of the substitutions constructed for two subexpressions do not overlap. We assume that the fresh type variables do not overlap with the type variables used in the Hindley-Milner type judgements.

- **Case variable:** $\mathcal{M}, \{x : \beta\}, \emptyset \vdash_{\Uparrow} x : \beta$.

$\Gamma \vdash^{\star}_{\text{HM}} x : \tau$
$\Rightarrow \{ \textit{syntax directness}, (\text{HM-VAR})^{\star} \}$
$\tau < \Gamma(x)$
$\Rightarrow \{ \textit{choose } S = [\beta := \tau] \}$
$S\beta < S\Gamma(x)$
$\Rightarrow \{ \textit{def.}\,(\vdash_s \ \textit{set}),\ \textit{def.}\,(\textit{lift} \preceq),\ \textit{def.}\,(\vdash_s \equiv) \}$
$S \vdash_s \emptyset \ \wedge\ S \vdash_s (\{x : \beta\} \preceq \Gamma) \ \wedge\ S \vdash_s \beta \equiv \tau$

- **Case application:** $\mathcal{M}, \mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_{new} \vdash_{\Uparrow} e_1\ e_2 : \beta$ with $\mathcal{C}_{new} = \{\tau_1' \equiv \tau_2' \to \beta\}$.

  $\Gamma \vdash^{\star}_{\text{HM}} e_1\ e_2 : \tau_2$
  $\quad \Rightarrow \{$ *syntax directness,* (HM-App)* $\}$
  $\Gamma \vdash^{\star}_{\text{HM}} e_1 : \tau_1 \to \tau_2 \ \wedge\ \Gamma \vdash^{\star}_{\text{HM}} e_2 : \tau_1$
  $\quad \Rightarrow \{$ *i.h., introduce S_1 for e_1 and S_2 for e_2 (s.t. $dom(S_1) \cap dom(S_2) = \emptyset$)* $\}$
  $S_1 \vdash_s \mathcal{C}_1 \ \wedge\ S_1 \vdash_s \mathcal{A}_1 \preceq \Gamma \ \wedge\ S_1 \vdash_s (\tau_1' \equiv \tau_1 \to \tau_2) \ \wedge\ S_2 \vdash_s \mathcal{C}_2$
  $\qquad \wedge\ S_2 \vdash_s \mathcal{A}_2 \preceq \Gamma \ \wedge\ S_2 \vdash_s \tau_2' \equiv \tau_1$
  $\quad \Rightarrow \{$ *choose* $S = [\beta := \tau_2] \circ S_2 \circ S_1$ $\}$ **(10)**
  $S \vdash_s \mathcal{C}_1 \ \wedge\ S \vdash_s \mathcal{A}_1 \preceq \Gamma \ \wedge\ S \vdash_s (\tau_1' \equiv \tau_1 \to \tau_2) \ \wedge\ S \vdash_s \mathcal{C}_2 \ \wedge\ S \vdash_s \mathcal{A}_2 \preceq \Gamma$
  $\qquad \wedge\ S \vdash_s \tau_2' \equiv \tau_1$
  $\quad \Rightarrow \{$ *prop. (split-left)* $\}$
  $S \vdash_s \mathcal{C}_1 \ \wedge\ S \vdash_s \mathcal{C}_2 \ \wedge\ S \vdash_s (\mathcal{A}_1 \cup \mathcal{A}_2 \preceq \Gamma) \ \wedge\ S \vdash_s (\tau_1' \equiv \tau_1 \to \tau_2) \ \wedge\ S \vdash_s \tau_2' \equiv \tau_1$
  $\quad \Rightarrow \{$ *def.* $(\vdash_s \equiv)$ $\}$
  $S \vdash_s \mathcal{C}_1 \ \wedge\ S \vdash_s \mathcal{C}_2 \ \wedge\ S \vdash_s (\mathcal{A}_1 \cup \mathcal{A}_2 \preceq \Gamma) \ \wedge\ S\tau_1' = S\tau_1 \to S\tau_2$
  $\qquad \wedge\ S\tau_2' = S\tau_1 \ \wedge\ S\beta = S\tau_2$
  $\quad \Rightarrow \{$ *def.* $(\vdash_s\ set)$, *def.* $(\vdash_s \equiv)$ $\}$
  $S \vdash_s (\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_{new}) \ \wedge\ S \vdash_s (\mathcal{A}_1 \cup \mathcal{A}_2 \preceq \Gamma) \ \wedge\ S \vdash_s \beta \equiv \tau_2$

  (10) The induction hypotheses for e_1 and e_2 introduce two substitutions, namely S_1 and S_2, respectively. Because of the bottom-up formulation of the constraint collecting type rules, these substitutions can be chosen such that their domains are disjoint. We compose S_1, S_2, and $[\beta := \tau_2]$. Because the type variable β is introduced for the application at hand, it does not appear in S_1, nor in S_2.

- **Case abstraction:** $\mathcal{M}, \mathcal{A}\backslash x, \mathcal{C} \cup \mathcal{C}_{new} \vdash_{\Uparrow} \lambda x \to e : \beta_2$ with $\mathcal{C}_{new} = \{\beta_2 \equiv \beta_1 \to \tau'\} \cup \{\beta_1 \equiv \tau \mid x{:}\tau \in \mathcal{A}\}$.

  $\Gamma \vdash^{\star}_{\text{HM}} \lambda x \to e : \tau_1 \to \tau_2$
  $\quad \Rightarrow \{$ *syntax directness,* (HM-Abs)* $\}$
  $(x{:}\tau_1) \otimes \Gamma \vdash^{\star}_{\text{HM}} e : \tau_2$
  $\quad \Rightarrow \{$ *i.h., introduce S_1 for e_1* $\}$
  $S_1 \vdash_s \mathcal{C} \ \wedge\ S_1 \vdash_s (\mathcal{A} \preceq (x{:}\tau_1) \otimes \Gamma) \ \wedge\ S_1 \vdash_s \tau' \equiv \tau_2$
  $\quad \Rightarrow \{$ *choose* $S = [\beta_1 := \tau_1, \beta_2 := \tau_1 \to \tau_2] \circ S_1$ $\}$
  $S \vdash_s \mathcal{C} \ \wedge\ S \vdash_s (\mathcal{A} \preceq (x{:}\tau_1) \otimes \Gamma) \ \wedge\ S \vdash_s \tau' \equiv \tau_2$
  $\quad \Rightarrow \{$ *prop. (split* $\otimes$*)* $\}$
  $S \vdash_s \mathcal{C} \ \wedge\ S \vdash_s \mathcal{A} \preceq \{x{:}\tau_1\} \ \wedge\ S \vdash_s (\mathcal{A} \preceq \Gamma\backslash x) \ \wedge\ S \vdash_s \tau' \equiv \tau_2$
  $\quad \Rightarrow \{$ *(lift* $\preceq$*)* $\}$
  $S \vdash_s \mathcal{C} \ \wedge\ S \vdash_s \{\tau \preceq \tau_1 \mid x{:}\tau \in \mathcal{A}\} \ \wedge\ S \vdash_s (\mathcal{A} \preceq \Gamma\backslash x) \ \wedge\ S \vdash_s \tau' \equiv \tau_2$
  $\quad \Rightarrow \{$ *Lemma 4.2* $\}$
  $S \vdash_s \mathcal{C} \ \wedge\ S \vdash_s \{\tau \equiv \tau_1 \mid x{:}\tau \in \mathcal{A}\} \ \wedge\ S \vdash_s (\mathcal{A} \preceq \Gamma\backslash x) \ \wedge\ S \vdash_s \tau' \equiv \tau_2$
  $\quad \Rightarrow \{$ *def.* $(\vdash_s \equiv)$, $S\beta_1 = S\tau_1$ $\}$
  $S \vdash_s \mathcal{C} \ \wedge\ S \vdash_s \{\beta_1 \equiv \tau \mid x{:}\tau \in \mathcal{A}\} \ \wedge\ S \vdash_s (\mathcal{A} \preceq \Gamma\backslash x) \ \wedge\ S \vdash_s \tau' \equiv \tau_2$
  $\quad \Rightarrow \{$ *prop. (lift-remove), def* $(\vdash_s \equiv)$ $\}$
  $S \vdash_s \mathcal{C} \ \wedge\ S \vdash_s \{\beta_1 \equiv \tau \mid x{:}\tau \in \mathcal{A}\} \ \wedge\ S \vdash_s (\mathcal{A}\backslash x \preceq \Gamma) \ \wedge\ S\tau' = S\tau_2$

$\quad\Rightarrow \{$ *def.* $(\vdash_s$ *set*$)$, *def.* $(\vdash_s \equiv) \}$
$S \vdash_s (\mathcal{C} \cup \mathcal{C}_{new}) \ \wedge\ S \vdash_s (\mathcal{A}\backslash x \preceq \Gamma) \ \wedge\ S \vdash_s (\beta_2 \equiv \tau_1 \to \tau_2)$

- **Case let expression:** $\mathcal{M}, \mathcal{A}_1 \cup \mathcal{A}_2\backslash x, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_{new} \vdash_{\Uparrow}$ **let** $x = e_1$ **in** $e_2 : \beta$ with $\mathcal{C}_{new} = \{\beta \equiv \tau_2'\} \cup \{\tau \leqslant_{\mathcal{M}} \tau_1' \mid x : \tau \in \mathcal{A}_2\}$. Let $\sigma = \mathit{generalize}(\Gamma, \tau_1)$.

$\Gamma \vdash^{\star}_{\text{HM}}$ **let** $x = e_1$ **in** $e_2 : \tau_2$
$\quad\Rightarrow \{$ *syntax directness,* $(\text{HM-Let})^{\star} \}$
$\Gamma \vdash^{\star}_{\text{HM}} e_1 : \tau_1 \ \wedge\ (x : \sigma) \oslash \Gamma \vdash^{\star}_{\text{HM}} e_2 : \tau_2$
$\quad\Rightarrow \{$ *i.h., introduce* S_1 *for* e_1 *and* S_2 *for* e_2 *(s.t.* $\mathit{dom}(S_1) \cap \mathit{dom}(S_2) = \emptyset$*)* $\}$
$S_1 \vdash_s \mathcal{C}_1 \ \wedge\ S_1 \vdash_s \mathcal{A}_1 \preceq \Gamma \ \wedge\ S_1 \vdash_s \tau_1' \equiv \tau_1 \ \wedge\ S_2 \vdash_s \mathcal{C}_2 \ \wedge\ S_2 \vdash_s (\mathcal{A}_2 \preceq (x : \sigma) \oslash \Gamma)$
$\qquad\qquad \wedge\ S_2 \vdash_s \tau_2' \equiv \tau_2$
$\quad\Rightarrow \{$ *choose* $S = [\beta := \tau_2] \circ S_2 \circ S_1 \}$
$S \vdash_s \mathcal{C}_1 \ \wedge\ S \vdash_s \mathcal{A}_1 \preceq \Gamma \ \wedge\ S \vdash_s \tau_1' \equiv \tau_1 \ \wedge\ S \vdash_s \mathcal{C}_2 \ \wedge\ S \vdash_s (\mathcal{A}_2 \preceq (x : \sigma) \oslash \Gamma)$
$\qquad\qquad \wedge\ S \vdash_s \tau_2' \equiv \tau_2$
$\quad\Rightarrow \{$ *prop.* $($*split* $\oslash) \}$
$S \vdash_s \mathcal{C}_1 \ \wedge\ S \vdash_s \mathcal{A}_1 \preceq \Gamma \ \wedge\ S \vdash_s \tau_1' \equiv \tau_1 \ \wedge\ S \vdash_s \mathcal{C}_2 \ \wedge\ S \vdash_s \mathcal{A}_2 \preceq \{x : \sigma\}$
$\qquad\qquad \wedge\ S \vdash_s (\mathcal{A}_2 \preceq \Gamma\backslash x) \ \wedge\ S \vdash_s \tau_2' \equiv \tau_2$
$\quad\Rightarrow \{$ *prop.* $($*lift-remove*$)$, *prop.* $($*split-left*$) \}$
$S \vdash_s \mathcal{C}_1 \ \wedge\ S \vdash_s (\mathcal{A}_1 \cup \mathcal{A}_2\backslash x \preceq \Gamma) \ \wedge\ S \vdash_s \tau_1' \equiv \tau_1 \ \wedge\ S \vdash_s \mathcal{C}_2 \ \wedge\ S \vdash_s \mathcal{A}_2 \preceq \{x : \sigma\}$
$\qquad\qquad \wedge\ S \vdash_s \tau_2' \equiv \tau_2$
$\quad\Rightarrow \{ ($*lift* $\preceq) \}$
$S \vdash_s \mathcal{C}_1 \ \wedge\ S \vdash_s (\mathcal{A}_1 \cup \mathcal{A}_2\backslash x \preceq \Gamma) \ \wedge\ S \vdash_s \tau_1' \equiv \tau_1 \ \wedge\ S \vdash_s \mathcal{C}_2$
$\qquad\qquad \wedge\ S \vdash_s \{\tau \preceq \sigma \mid x : \tau \in \mathcal{A}_2\} \ \wedge\ S \vdash_s \tau_2' \equiv \tau_2$
$\quad\Rightarrow \{$ *Lemma 4.3,* $(\vdash_s \equiv) \}$ **(11)**
$S \vdash_s \mathcal{C}_1 \ \wedge\ S \vdash_s (\mathcal{A}_1 \cup \mathcal{A}_2\backslash x \preceq \Gamma) \ \wedge\ S \vdash_s \tau_1' \equiv \tau_1 \ \wedge\ S \vdash_s \mathcal{C}_2$
$\qquad\qquad \wedge\ S \vdash_s \{\tau \leqslant_{\mathcal{M}} \tau_1' \mid x : \tau \in \mathcal{A}_2\} \ \wedge\ S \vdash_s \tau_2' \equiv \tau_2$
$\quad\Rightarrow \{$ *def.* $(\vdash_s$ *set*$)$, *def.* $(\vdash_s \equiv) \}$
$S \vdash_s (\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_{new}) \ \wedge\ S \vdash_s (\mathcal{A}_1 \cup \mathcal{A}_2\backslash x \preceq \Gamma) \ \wedge\ S \vdash_s \beta \equiv \tau_2$

(11) Similar to **(8)**: explicit instance constraints are converted to implicit instance constraints, and we use that $S\tau_1'$ equals $S\tau_1$. The invariant $(\mathit{ftv}(S\mathcal{M}) = \mathit{ftv}(S\Gamma))$ allows us to annotate the implicit instance constraints with $\mathcal{M}$. □

We conclude this chapter by claiming that algorithm Infer (see Definition 4.6) is a correct implementation of the Hindley-Milner type system.

Main Theorem 4.16 (Correctness of Infer). *Algorithm* Infer(Γ, e) *returns the principal type for* e *under* Γ.

Proof. Assume that $\mathit{fev}(e) \subseteq \mathit{dom}(\Gamma)$: otherwise, e has no principal type. Choose $\mathcal{M} = \mathit{ftv}(\Gamma)$, and derive $\mathcal{M}, \mathcal{A}, \mathcal{C} \vdash_{\Uparrow} e : \tau$. The constraints in $\mathcal{C}$ are orderable (Lemma 4.4), and therefore $\mathcal{C} \cup \Gamma \preceq \mathcal{A}$ is also orderable (the extra constraints can be placed in the front of the ordering). Hence, solving these constraints will result in a substitution S that satisfies the constraints (Theorem 4.9). If the set of constraints is satisfiable, then $S \neq \top$ (Theorem 4.10). Moreover, the process of solving the constraints will eventually stop (Theorem 4.5).

Soundness of $\vdash_{\Uparrow}$ gives us that $S\mathit{\Gamma} \vdash^{\star}_{\mathrm{HM}} e : S\tau$ (Theorem 4.14). Because the modified Hindley-Milner type system is equivalent to the original system (Theorem 4.13), $S\mathit{\Gamma} \vdash_{\mathrm{HM}} e : S\tau$ also holds. Furthermore, Theorem 4.15 guarantees that if a derivation exists for $\mathit{\Gamma}$ and e with the modified Hindley-Milner type rules (and thus also with the original rules), then the constraints that are collected by the bottom-up type rules are consistent.

As a result, algorithm Infer is a sound and complete implementation of the Hindley-Milner type rules, and returns the principal type of an expression under a type environment. $\square$

5

Top: a type inference framework

Overview. *This chapter describes the type inference framework* Top. *In addition to constraint generation and constraint resolution, we identify an intermediate step, which is the ordering of constraints. Each constraint carries information, which can be used for all kinds of purposes. The* Top *framework has been designed to support flexible and customizable type inference.*

This chapter presents the Top framework for constraint-based type inference. This framework focuses in particular on providing helpful feedback for inconsistent sets of type constraints. In a complete lattice, the top element (usually written as $\top$) is often used to represent an error situation – this explains the name of the framework. Although our approach applies to all kinds of program analyses for which good feedback is desirable, we limit ourselves to type inference. In this light, Top can also be interpreted as "typing our programs".

A constraint-based analysis can be divided into constraint generation and constraint resolution. Such an approach has a number of advantages: a framework can have several constraint solvers available, and we can choose freely which solver to use. Because of the clear separation between constraint collecting and constraint solving, it becomes much easier to reuse constraint solvers. Collecting constraints must be programmed for each compiler or tool, but solving the constraints is implemented once and for all.

The order in which the constraints are considered is of great importance, although the actual impact depends on which constraint solver is chosen. Generally, the constraints are solved one after another. The order should not influence the outcome of a consistent constraint set, but for an inconsistent set, it strongly determines at which point the inconsistency is detected. This bias can show up in the reported error messages. To prevent this, we distinguish a third phase in the inference process, which is to order the constraints.

In our framework, we incorporate this new phase in the following way. Instead of collecting a constraint set, we construct a tree with all the constraints. Typically, the shape of this constraint tree follows the shape of the abstract syntax tree on which we perform the analysis. In the second phase, we order the constraints by rearranging this tree, and then choosing a flattening of this tree, which gives us an ordered list of constraints. After the constraints have been ordered, we solve the constraints.

The Top framework is both customizable and extendable, which are two essential properties to encourage reusability. The framework enjoys the following characteristics.

1. *Extend the solution space.* We can add new (stateful) information that is available while we are solving the constraints.
2. *Abstract constraint information.* Each constraint carries information, but we make no assumption about the content of this information. Every user can decide which information is relevant to keep track of. Constraint information is also used to create error reports – thus, the formatting of the messages can be customized as well. Although the content is unspecified, there is some interaction between the constraint information and the process of constraint solving.
3. *New types of constraints can be added.* The framework can be extended to include new sorts of constraints. To solve a new type of constraint, we may have to extend the solution space (see item *1*).
4. *Multiple constraint solvers.* Our framework can support several constraint solvers, and new alternatives to solve the constraints can be included.
5. *Suitable for other analyses.* A part of the framework is not specific for type inference, and this part can be reused for other analyses that are formulated as a constraint problem.

The rest of this chapter is organized as follows. We first define a set of type constraints, which are currently supported by the framework (Section 5.1). Next, we discuss how to solve these constraints in Section 5.2. In Section 5.3, we present constraint trees, and define techniques to rearrange such a tree, as well as a function to flatten a constraint tree. This part of the framework does not depend in any way on the constraints that are used. We conclude this chapter with a discussion on how the framework can be used to perform kind inference (Section 5.4), and we summarize the advantages of our approach (Section 5.5).

5.1 Constraint generation

In this section, we revise the constraint language introduced in the previous chapter. Instead of choosing one fixed set of constraints, we explore a wide variety of constraints. And more important: this set can be extended to include other forms of constraints.

For each constraint in our framework, there are three properties we have to define. First of all, we have to define the syntax of a constraint (how to write it down). Secondly, we need to define the meaning of a constraint, that is, its semantics. This tells us whether a solution for a constraint set meets the restriction it imposes. This requires that we have some knowledge of the information available during constraint solving. Note that defining the semantics of a constraint is similar to the notion of constraint satisfaction in Definition 4.1. Finally, we have to describe how a constraint is to be solved. In theory, there can be multiple strategies to solve a constraint. The distinction between the semantics of a constraint and the way it is solved may seem to be insignificant, but there are some reasons for this distinction. Because the semantics relate to the (final) solution, it becomes straightforward to reason about the meaning of a set of constraints, without taking into account how such a set is solved. Secondly, we can verify the validity of the solution returned by the constraint solver, which gives us an internal sanity check for free.

In our framework, each constraint carries additional information. This constraint information can, for instance, contain the reason why the constraint was generated. Typically, the amount of information carried around should be enough to construct an error message if the constraint leads to an inconsistency. Constraint information will be denoted by $\langle i \rangle$, and we make no assumption about its content. This is a valuable abstraction within our framework: we can choose for ourselves which information we want to keep, and how the error messages will be presented. In the presentation, we omit the constraint information carried by a constraint whenever this is appropriate.

Once more, we start with equality constraints on types.

$$c_{(\equiv)} ::= (\tau_1 \equiv \tau_2)\ \langle i \rangle \qquad \textit{(equality constraint)}$$

The types of an equality constraint are monomorphic, and each equality constraint carries information about the constraint. A solution Θ solves an equality constraint if (and only if) applying the substitution in Θ to both types gives us two syntactically equivalent types.

$$\Theta \vdash_s \tau_1 \equiv \tau_2 \quad =_{def} \quad \Theta(\tau_1) = \Theta(\tau_2)$$

There is one noteworthy difference to our previous definition of constraint satisfaction for an equality constraint: we leave Θ unspecified. The only requirement for Θ is that it contains a substitution S, which we apply to the two types in the definition above. In fact, we use $\Theta(\tau)$ as a shorthand notation for $S_\Theta(\tau)$ (and with S_Θ we mean "taking the S component of Θ").

We introduce three constraints to capture polymorphism. For this, we follow the second alternative (proposed in Section 4.2) to cope with polymorphism, and use type scheme variables as place-holders for unknown type schemes. We use a three layer type language: besides mono types (τ), we have type schemes (σ), and ρ's, which are either type scheme variables (σ_v) or type schemes. These layers predict at which points we can expect a universal quantifier or a type scheme variable, and thus type our type language. With the following constraints we can express generalization, instantiation, and skolemization.

Polymorphism constraints:

$$\begin{array}{lll} c_{(\forall)} ::= & \sigma_v := \textsc{Gen}(\mathcal{M}, \tau)\ \langle i \rangle & \textit{(generalization)} \\ \quad\ \ \mid & \tau := \textsc{Inst}(\rho)\ \langle i \rangle & \textit{(instantiation)} \\ \quad\ \ \mid & \tau := \textsc{Skol}(\mathcal{M}, \rho)\ \langle i \rangle & \textit{(skolemization)} \end{array}$$

With a generalization constraint we can generalize a type with respect to a set of monomorphic type variables, and assign the resulting type scheme to a type scheme variable. Instantiation constraints express that a type should be an instance of a type scheme, or the type scheme corresponding to a type scheme variable. A skolemization constraint restricts a type to be "more general" than a type scheme. To determine this, we need to know which type variables in the type are polymorphic, and which are not. Hence, a set $\mathcal{M}$ is attached to each skolemization constraint.

We proceed with defining semantics for these constraints.

$$\begin{array}{llll} \Theta \vdash_s \sigma_v := \textsc{Gen}(\mathcal{M}, \tau) & =_{def} & \Theta(\sigma_v) = \mathit{generalize}(\Theta(\mathcal{M}), \Theta(\tau)) \\ \Theta \vdash_s \tau := \textsc{Inst}(\rho) & =_{def} & \Theta(\tau) < \Theta(\rho) \\ \Theta \vdash_s \tau := \textsc{Skol}(\mathcal{M}, \rho) & =_{def} & \Theta(\rho) < \mathit{generalize}(\Theta(\mathcal{M}), \Theta(\tau)) \end{array}$$

For these definitions, we require that Θ contains a *type scheme substitution* Σ (in addition to a "normal" substitution that works on types): this is a table which maps type scheme variables to type schemes. In these definitions, $\Theta(\rho)$ is a shorthand notation for $\Theta(\Sigma_\Theta(\rho))$: apply the substitution after having replaced all type scheme variables using Σ.

Skolemization constraints become important when expressions and definitions can be annotated with types. Such a type annotation should not be more general than the type we infer for its expression (or definition), although an annotation may be used to restrict the inferred type. We illustrate this with an example.

Example 5.1. Let e be some expression which is assigned the type variable v_0. Consider the annotated expression $(e :: \forall a.a \to a)$. To express that the (inferred) type of e is general enough, we use the type constraint $v_0 := \textsc{Skol}(\mathcal{M}, \forall a.a \to a)$, where $\mathcal{M}$ contains the monomorphic type variables from the context in which e appears. We continue this example with three hypothetical cases.

Case 1. Suppose we discover that v_0 should be $\mathit{Bool} \to \mathit{Bool}$. Observe that the type obtained from skolemizing $\forall a.a \to a$ cannot be unified with $\mathit{Bool} \to \mathit{Bool}$. Hence, e's annotation is too polymorphic, and $\forall a.a \to a \not< \mathit{Bool} \to \mathit{Bool}$. The skolemization constraint cannot be satisfied.

Case 2. Assume v_0 equals $v_1 \to v_1$, and none of the type variables is known to be monomorphic ($\mathcal{M} = \emptyset$). Then the skolemization constraint is satisfied, and $\forall a.a \to a < \mathit{generalize}(\emptyset, v_1 \to v_1)$. Thus, the type annotation is consistent with e.

Case 3. Again, v_0 is found to be $v_1 \to v_1$, but v_1 is to remain monomorphic. Then the annotation is in error, witnessed by $\forall a.a \to a \not< v_1 \to v_1$. In this particular case, we speak of an "escaping skolem constant", because it is the set with monomorphic type variables that prevents that the skolemization constraint is satisfied.

This definition of constraint satisfaction suggests that generalization and skolemization constraints are very much alike. In fact, the semantics of a generalization constraint can be safely loosened to $\Theta(\sigma_v) < \mathit{generalize}(\Theta(\mathcal{M}), \Theta(\tau))$, which suggests that these two constraints can be captured by a more general constraint. However, these constraints are solved in an opposite direction. For a generalization constraint, we assign a value to the type scheme variable, whereas for a skolemization constraint we restrict the type by skolemizing the (known) type scheme.

Given the semantics of each of the type constraints, we define an entailment relation on constraint sets, and an equivalence relation between sets of type constraints.

Definition 5.1 (Constraint entailment). *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be constraint sets. Then $\mathcal{C}_1$ entails $\mathcal{C}_2$, written $\mathcal{C}_1 \Vdash_e \mathcal{C}_2$, as defined below.*

$$\mathcal{C}_1 \Vdash_e \mathcal{C}_2 \quad =_{def} \quad \forall \Theta \; . \; \Theta \vdash_s \mathcal{C}_1 \Rightarrow \Theta \vdash_s \mathcal{C}_2$$

Definition 5.2 (Equivalence). *Equivalence of two constraint sets under the entailment relation is defined as follows.*

$$\mathcal{C}_1 =_e \mathcal{C}_2 \quad =_{def} \quad \mathcal{C}_1 \Vdash_e \mathcal{C}_2 \; \wedge \; \mathcal{C}_2 \Vdash_e \mathcal{C}_1$$

With equality and polymorphism constraints at our disposal, we can express the type system of a realistic functional language as a constraint problem.

Definition 5.3 (Translation). *Constraint set $\mathcal{C}_1$ can be translated into $\mathcal{C}_2$ (written as $\mathcal{C}_1 \rightsquigarrow \mathcal{C}_2$) if and only if $\mathcal{C}_2 \Vdash_e \mathcal{C}_1$. For singleton sets we write $c_1 \rightsquigarrow c_2$ (c_1 and c_2 are type constraints).*

Example 5.2. We could get rid of the equality constraints, and express these as instantiation constraints using the following translation. In our framework, however, we will not make use of this translation.

Translation. $(\tau_1 \equiv \tau_2) \; \langle i \rangle \quad \rightsquigarrow \quad \tau_1 := \textsc{Inst}(\tau_2) \; \langle i \rangle$

Proof. Let Θ be such that $\Theta \vdash_s \tau_1 := \textsc{Inst}(\tau_2)$. Then $\Theta(\tau_1) < \Theta(\tau_2)$ by the semantics of $\textsc{Inst}$, and $\Theta(\tau_1) = \Theta(\tau_2)$ by the definition of the instance-of relation. Thus, $\Theta \vdash_s \tau_1 \equiv \tau_2$ holds. □

For reasons of convenience, we introduce infix notation to write instantiation and skolemization constraints. The syntax for an infix instantiation constraint matches deliberately with the syntax of an explicit instance constraint (Chapter 4) since these are conceptually the same. Furthermore, we show how an implicit instance constraint can be formulated in terms of the polymorphism constraints.

Additional constraints:

$$\begin{array}{lll} c_{(+)} ::= & \tau \preceq \rho \; \langle i \rangle & \textit{(instantiation – infix)} \\ \mid & \tau \succeq_{\mathcal{M}} \rho \; \langle i \rangle & \textit{(skolemization – infix)} \\ \mid & (\tau_1 \leqslant_{\mathcal{M}} \tau_2) \; \langle i \rangle & \textit{(implicit instance)} \end{array}$$

The first two constraints are only syntactic sugar.

$$\begin{array}{lll} \tau \preceq \rho \; \langle i \rangle & =_{def} & \tau := \textsc{Inst}(\rho) \; \langle i \rangle \\ \tau \succeq_{\mathcal{M}} \rho \; \langle i \rangle & =_{def} & \tau := \textsc{Skol}(\mathcal{M}, \rho) \; \langle i \rangle \end{array}$$

Both $\preceq$ and $\succeq_{\mathcal{M}}$, but also equality constraints, are lifted to work on lists of pairs (in the line of (*lift* $\equiv$) and (*lift* $\preceq$), page 58). We define satisfaction for the implicit instance constraint, which matches the semantics defined in the previous chapter.

$$\Theta \vdash_s \tau_1 \leqslant_{\mathcal{M}} \tau_2 \quad =_{def} \quad \Theta(\tau_1) < \textit{generalize}(\Theta(\mathcal{M}), \Theta(\tau_2))$$

An implicit instance constraint can be translated into a combination of a generalization and an instantiation constraint. The order of these two constraints is irrelevant for the specification, although it may be important for an implementation.

Translation. $\{(\tau_1 \leqslant_{\mathcal{M}} \tau_2)\ \langle i \rangle\} \quad \rightsquigarrow \quad \{\sigma_v := \textsc{Gen}(\mathcal{M}, \tau_2), \tau_1 := \textsc{Inst}(\sigma_v)\}$
(where σ_v is a fresh type scheme variable)

Proof. Suppose that $\Theta \vdash_s \sigma_v := \textsc{Gen}(\mathcal{M}, \tau_2)$ and $\Theta \vdash_s \tau_1 := \textsc{Inst}(\sigma_v)$. Then, by definition, we have that $\Theta(\sigma_v) = \mathit{generalize}(\Theta(\mathcal{M}), \Theta(\tau_2))$ and $\Theta(\tau_1) < \Theta(\sigma_v)$. From this, we conclude that $\Theta(\tau_1) < \mathit{generalize}(\Theta(\mathcal{M}), \Theta(\tau_2))$. □

5.2 Constraint solving

Before we define how the constraints should be solved, we discuss which information we maintain when solving. We decompose the information into three states. The first state contains basic information, which can be used for other analyses as well. The second state consists of a substitution. Instead of choosing one representation for a substitution, we list the operations that can be performed on a substitution. The third state contains the remaining information that we have to keep to perform type inference. We investigate each of the three states in detail.

5.2.1 The basic state

We first introduce an abstract datatype to represent a constraint. This is a triple containing the strategy to solve the constraint, its semantics, and its syntax. We assume that solving this constraint takes place within some monad m.

```
data Constraint m =
   Constraint{ solve :: m (), semantics :: m Bool, syntax :: String }
```

The information we store in the basic state applies to constraint-based analyses in general. The basic state contains a stack of constraints that are not yet solved, and a list of constraints that are marked as being in error. In the beginning, the constraint stack contains all the constraints that we have to solve. We pop a constraint from the stack and solve it. This process continues until the constraint stack is empty. Constraints for which we discover that they lead to an inconsistency are added to the error list, which is initially empty.

We introduce a type class for monads that contain a basic state. This type class also mentions $\langle i \rangle$, which is the information we keep for each constraint. The functional dependency in the class declaration ensures that the type of the constraint information is uniquely determined by the monad.

```
class Monad m ⇒ HasBasic m ⟨i⟩ | m → ⟨i⟩ where

   -- constraints
   addConstraint  :: Constraint m    → m ()
   addConstraints :: [Constraint m] → m ()
   popConstraint  :: m (Maybe (Constraint m))
```

```
  -- errors
addError :: ⟨i⟩ → m ()
getErrors :: m [⟨i⟩]

  -- semantic check
addCheck :: String → m Bool → m ()
```

We use the functions that are supported by a basic state for solving the type constraints. The type class *HasBasic* has three functions to manipulate the constraint stack: we can add a single constraint or a list of constraints, and we can pop the first constraint from the stack. *Nothing* is returned for the empty constraint stack. Observe that we use the type variable *m* as parameter of the *Constraint* datatype.

Furthermore, we have a function to add one error, and a function to get all the errors discovered so far. Observe that we use constraint information for the list of errors. As mentioned before, the constraint information should be detailed enough to construct an error message. With the function *addCheck*, we can add sanity checks for the solved constraints. The idea is that these sanity checks are performed when all constraint have been solved. The first argument of *addCheck* is a message, which is reported when the second argument evaluates to *False* (in the monad *m*).

With the member function of the *HasBasic* type class, we can define how the constraints are solved.

```
startSolving :: HasBasic m ⟨i⟩ ⇒ m ()
startSolving =
  do mc ← popConstraint
     case mc of
       Nothing → return ()
       Just c   →
         do solve c
            addCheck (syntax c) (semantics c)
            startSolving
```

The function *startSolving* pops constraints from the stack, solves them, until the stack is empty. For each constraint that we solve, we add a sanity check to the state, which can be used to verify the final state.

The actual implementation of a basic state is straightforward given the description above, and we will not present it here. Clearly, we can store more information in this state, which would be convenient to have for other analyses as well. For instance, we can include information for debugging, or information for tracing the process of constraint solving.

5.2.2 The substitution state

Substitutions play a central role in our framework. We introduce a type class for monads containing a substitution. We plan ahead and take into account that future substitutions may be in an inconsistent state. This planning makes it possible to look for sophisticated substitutions with which we can improve the quality of the type error messages. The substitution in our state should at least support the following three functions.

```
class Monad m ⇒ HasSubst m ⟨i⟩ | m → ⟨i⟩ where
  unifyTerms     :: ⟨i⟩ → Type → Type → m ()
  makeConsistent :: m ()
  substVar       :: Int → m Type
```

The function *unifyTerms* receives two types, which it unifies. In the end, the two types should be equivalent under the substitution. However, unifying two types may bring the substitution into an inconsistent state. The function *makeConsistent* restores consistency for a substitution. With *substVar*, we can query the type the substitution assigns to a type variable (here represented by an *Int*). Given *substVar*, we can apply the substitution to all kind of values that contain type variables. We assume *applySubst* to be this function. We have to be in a consistent state before we can apply the substitution.

5.2.3 The type inference state

The remaining information we need during type inference is stored in the type inference state, for which we introduce a new type class.

```
class Monad m ⇒ HasTI m ⟨i⟩ | m → ⟨i⟩ where
  nextUnique      :: m Int
  getTypeSynonyms :: m TypeSynonyms
  storeScheme     :: ⟨i⟩ → Int → TypeScheme → m ()
  lookupScheme    :: Int → m TypeScheme
  addSkolems      :: ⟨i⟩ → Skolems → Mono → m ()
```

We keep a counter in this state, which we use to create fresh type variables, fresh type scheme variables, and so on. The function *nextUnique* returns and increments the value of this counter. We also store a list of type synonyms in this state.

Furthermore, we maintain a list of type scheme variables and their assigned type schemes. We can add a type scheme variable and its type scheme to this list (*storeScheme*), and we can find the type scheme that corresponds to a type scheme variable (*lookupScheme*). We assume that a type scheme variable is represented by a value of type *Int*. Remember that we introduced ρ types, which are either type scheme variables or type schemes. With the function *lookupScheme*, we define a function

```
findScheme :: HasTI m ⟨i⟩ ⇒ Rho → m TypeScheme
```

which expects a ρ type as its argument. If ρ is a type scheme variable, it returns the corresponding type scheme in the type scheme map. Otherwise, *findScheme* returns the type scheme at hand.

Finally, the type inference state contains a list of pairs to check that no skolem constant escapes. Such a pair consists of a list of skolem constants and a set of monomorphic types. New pairs can be added with the function *addSkolems*. After all the constraints have been solved, the substitution is applied to each of the sets of monomorphic types, and we check that none of the listed skolem constants appears in the substituted set. We produce an error if this check fails.

5.2.4 Constraint solving

We define how equality constraints and the three constraints to handle polymorphism can be solved. For an equality constraint, we only have to call the function *unifyTerms* with the constraint information and the two types.

```
$solve\ ((\tau_1 \equiv \tau_2)\ \ \langle i \rangle) = unifyTerms\ \langle i \rangle\ \tau_1\ \tau_2$
```

For a generalization constraint, we compute a type scheme. We apply the current substitution to the monomorphic type variables, and to the type. However, the substitution can be in an inconsistent state. Therefore, we *always* have to make the substitution consistent before we can apply it. The computed type scheme is then associated with the type scheme variable from the constraint, and this pair is stored in the type scheme map.

```
$solve\ ((\sigma_v := \textsc{Gen}(\mathcal{M}, \tau))\ \ \langle i \rangle) =$
  do $makeConsistent$
     $\mathcal{M}' \leftarrow applySubst\ \mathcal{M}$
     $\tau'\ \ \leftarrow applySubst\ \tau$
     let $\sigma = generalize(\mathcal{M}', \tau')$
     $storeScheme\ \langle i \rangle\ \sigma_v\ \sigma$
```

To solve an instantiation constraint $(\tau := \textsc{Inst}(\rho))\ \ \langle i \rangle$, we proceed as follows. We are given a ρ, which is either a type scheme or a type scheme variable. We use *findScheme* to get a type scheme for ρ. This type scheme is instantiated into a type with the function *instantiateM*, which uses the counter of the type inference state. We add an equality constraint between τ and the instantiated type scheme. However, we use the constraint information to store the original type scheme. We do this with the function *instScheme*, which will be discussed later on. Because we remember the original type scheme, we can create a more informative error message in case the created equality constraint turns out to be incorrect.

```
$solve\ ((\tau := \textsc{Inst}(\rho))\ \ \langle i \rangle) =$
  do $\sigma\ \ \leftarrow findScheme\ \rho$
     $\tau_1 \leftarrow instantiateM\ \sigma$
     let $\langle i \rangle' = instScheme\ \sigma\ \langle i \rangle$
     $addConstraint\ ((\tau \equiv \tau_1)\ \ \langle i \rangle')$
```

To deal with a skolemization constraint $(\tau := \textsc{Skol}(\mathcal{M}, \rho))\ \ \langle i \rangle$, we first find a type scheme for ρ (if this is a type scheme variable). This type scheme is skolemized, which yields a type τ_1 and a list of used skolem constants *skc*. We create the constraint $(\tau \equiv \tau_1)\ \ \langle i \rangle'$, where $\langle i \rangle'$ is the original constraint information updated with the skolemized type scheme. To remember the original type scheme, we use *skolScheme*. The skolem constants and $\mathcal{M}$ are stored together so that we can check afterwards that the skolem constants do not escape via a type variable in $\mathcal{M}$.

```
$solve\ ((\tau := \textsc{Skol}(\mathcal{M}, \rho))\ \ \langle i \rangle) =$
  **do** $\sigma \quad\quad\quad \leftarrow findScheme\ \rho$
     $(\tau_1, skc) \leftarrow skolemizeM\ \sigma$
     **let** $\langle i \rangle' = skolScheme\ \sigma\ \langle i \rangle$
     $addSkolems\ \langle i \rangle'\ skc\ \mathcal{M}$
     $addConstraint\ ((\tau \equiv \tau_1)\ \ \langle i \rangle')$
```

Remembering information that is available while we are solving constraints is one important aspect of the framework. Once more, we use the type class system to enrich the information carried by the constraints. The functions *instScheme* and *skolScheme*, used for solving instantiation and skolemization constraints respectively, are member-functions of this type class.

```
**class** $PolymorphismInfo\ \langle i \rangle$ **where**
   $instScheme :: TypeScheme \rightarrow \langle i \rangle \rightarrow \langle i \rangle$
   $skolScheme :: TypeScheme \rightarrow \langle i \rangle \rightarrow \langle i \rangle$
     -- default definitions
   $instScheme\ \_ = id$
   $skolScheme\ \_ = id$
```

Observe that we are still not making any assumption about the information that is carried by the constraints. Instead, we only require that the carried constraint information is an instance of the *PolymorphismInfo* type class. In the instance definition one can define how the type schemes should be stored. If no definition is given, then the default definitions are used, and the type schemes are simply ignored and forgotten.

In conclusion, we can solve the type constraints proposed in this chapter using some monad m, and under the assumption that each constraint carries information of the type $\langle i \rangle$, if the following type class predicates hold for m and $\langle i \rangle$.

$$(HasBasic\ m\ \langle i \rangle, HasSubst\ m\ \langle i \rangle, HasTI\ m\ \langle i \rangle, PolymorphismInfo\ \langle i \rangle)$$

If we only have to solve equality constraints, then having $HasBasic\ m\ \langle i \rangle$ and $HasSubst\ m\ \langle i \rangle$ is sufficient.

5.2.5 Faking skolemization

Skolemizing a type scheme is an implementation technique to ensure that no assumptions can be made about certain type variables. A drawback of this approach is that skolem constants can show up in types. If these types are used in unifications, then skolem constants can also appear in reported type error messages. This is highly undesirable, because for a programmer it will not be apparent where these constants came from, or why they were introduced. Fortunately, this problem can easily be circumvented by *faking* skolemization of type schemes.

We use ordinary type variables for skolemization, but we remember which fresh type variables were introduced. Afterwards (when all constraints are handled), we have to check that the substitution does not contain deductions about these type variables. Firstly, all skolem variables that are mapped by the substitution to a type other than a type variable are in violation, and for these variables, we create an appropriate error message. Secondly, we have to make sure that different skolem variables are mapped to different type variables. After these two checks, we continue with checking that no skolem escapes via a monomorphic type variable. This last step was already present in the type inference framework. The following examples illustrate these three checks.

Example 5.3. The type signature given for *length* is too general.

$$\begin{array}{l} \mathit{length} :: a \rightarrow \mathit{Int} \\ \mathit{length}\ [\,] \qquad\quad = 0 \\ \mathit{length}\ (x : \mathit{xs}) = 1 + \mathit{length}\ \mathit{xs} \end{array}$$

The skolem variable introduced for the type signature's type variable a is unified with a list type (because of the left-hand side patterns). We report that the type signature for *length* is too general since type variable a should be a list type.

Example 5.4. The type signature of f is inconsistent with f's definition.

$$\begin{array}{l} f :: a \rightarrow b \rightarrow [a] \\ f\ x\ y = [x, y] \end{array}$$

Because x and y are in the same list, they should have the same type. The substitution obtained after handling all the type constraints maps the skolem variables for a and b to the same type (variable). We report that f's type signature is too general because a and b should be the same.

Example 5.5. The type of f is too general.

$$\begin{array}{l} \mathit{test}\ x = \textbf{let}\ f :: a \rightarrow [a] \\ \qquad\qquad\quad\ \ f\ y = [x, y] \\ \qquad\qquad\ \ \textbf{in}\ f\ x \end{array}$$

The skolem variable introduced for the type variable a in f's type signature should have the same type as the type variable assigned to the identifier x. This type variable is monomorphic in f's definition. Hence, we report that this type variable escapes.

In our type inference framework, only small changes are required to fake skolemization. We modify the function *skolemizeM*, which is used to solve a skolemization constraint, such that it introduces fresh type variables instead of new skolem constants for the quantified type variables of the type scheme. A list of skolem variables is maintained in the type inference state, and when all type constraints have been handled, we perform the necessary checks, and may report errors.

5.2.6 A greedy solver

We now define a first attempt to implementing the substitution state. The most obvious instance of this state is one which contains an association list to represent a substitution. We call a solver that uses such a substitution state for dealing with the equality constraints a *greedy constraint solver.* This solver has the property that it performs the unifications issued by equality constraints immediately, and that it deals with inconsistencies right away. The substitution we keep can be, for instance, a finite map which assigns types to type variables, and which is kept idempotent. We give definitions for *unifyTerms*, *makeConsistent*, and *substVar* to complete the definition of a greedy constraint solver.

The implementation of *unifyTerms* returns the most general unifier of two types. The result of this unification is incorporated into the substitution. When two types cannot be unified, we deal with the inconsistency, and add the constraint information corresponding to the equality constraint to the list of errors. After discovering an error, we ignore the constraint and continue solving the remaining constraints, which may lead to the detection of more type errors. Because the substitution will always be in a consistent state, the action to be taken by *makeConsistent* is void: each *makeConsistent* leaves the state unchanged. The function *substVar* consults the substitution, and returns the type that is associated with a type variable.

Definition 5.4 (Greedy constraint solver). *A greedy constraint solver is a solver that implements the substitution state with a standard substitution.*

5.3 Constraint ordering

The order in which type constraints are solved may influence at which point in the process of constraint solving we detect an inconsistency. This also depends on the constraint solver that is used: a greedy constraint solver, for instance, is highly sensitive to the order in which it solves equality constraints, and it will report different constraints for different orders. In fact, choosing an ordering for the type constraints closely relates to the order in which types are unified.

Instead of limiting ourselves to one order in which the constraints are solved, we consider a family of constraint orderings from which a user can select one. As it turns out, we are able to emulate the two type inference algorithms $\mathcal{W}$ and $\mathcal{M}$, and many other variations. Hence, our approach generalizes these two standard algorithms.

The four bottom-up type inference rules, which are presented in Figure 4.5, collect the type constraints in a list. To obtain more control over the order of the constraints, we collect the constraints in a tree instead. This constraint tree has the same shape as the abstract syntax tree of the expression for which the constraints are generated. However, we are not completely free to choose a constraint ordering. Some constraints have to be solved before others, and we encode this too in the constraint tree.

We consider four alternatives for constructing a constraint tree.

Constraint tree:

$$
\begin{array}{lll}
\mathcal{T}_{\mathcal{C}} & ::= \; \langle\!\bullet\; \mathcal{T}_{\mathcal{C}1}, \ldots, \mathcal{T}_{\mathcal{C}n} \;\bullet\!\rangle & \textit{(node)} \\
& \mid \; c \rhd \mathcal{T}_{\mathcal{C}} & \textit{(attach constraint)} \\
& \mid \; c \lhd \mathcal{T}_{\mathcal{C}} & \textit{(attach constraint from parent)} \\
& \mid \; \mathcal{T}_{\mathcal{C}1} \ll \mathcal{T}_{\mathcal{C}2} & \textit{(strict node)}
\end{array}
$$

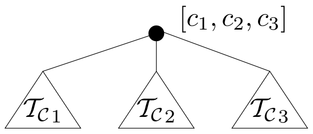
To minimize the use of parentheses, all operators for building constraint trees are right associative. With the first alternative we can combine a list of constraint trees into a single tree. The second and third alternatives let us add one constraint to a tree. The difference between these two alternatives lies in where the constraint was created: in the case of $c \rhd \mathcal{T}_{\mathcal{C}}$, the constraint c belongs to the first node encountered in the constraint tree $\mathcal{T}_{\mathcal{C}}$, and both c and this node were generated at the same location in the abstract syntax tree. The case of $c \lhd \mathcal{T}_{\mathcal{C}}$ is conceptually harder to comprehend. Such a constraint is generated by its parent node, which is not yet part of the constraint tree. However, it relates specifically to one of the constraint subtrees of this parent node, namely the one to which it is attached. The last case for constructing a constraint tree ($\mathcal{T}_{\mathcal{C}1} \ll \mathcal{T}_{\mathcal{C}2}$) combines two trees in a strict way: all the constraints in $\mathcal{T}_{\mathcal{C}1}$ should be considered before the constraints in $\mathcal{T}_{\mathcal{C}2}$. In the following example, we illustrate the two alternatives for attaching a constraint to a constraint tree.

Example 5.6. We construct a constraint tree for a conditional expression, say **if** e_1 **then** e_2 **else** e_3. Suppose we have created constraint tree $\mathcal{T}_{\mathcal{C}1}$ for e_1, in combination with the type τ_1. Similarly, we have $\mathcal{T}_{\mathcal{C}2}$ and τ_2 for e_2, and $\mathcal{T}_{\mathcal{C}3}$ and τ_3 for e_3. We introduce the fresh type variable β to represent the type of the branches of the conditional, and we generate three type constraints.

$$c_1 = (\tau_1 \equiv \mathit{Bool}) \qquad c_2 = (\tau_2 \equiv \beta) \qquad c_3 = (\tau_3 \equiv \beta)$$

We can first combine the constraint trees of the subexpressions, and then attach the three constraints. This results in the following constraint tree.

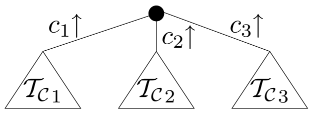
$$c_1 \rhd c_2 \rhd c_3 \rhd \langle\!\bullet\; \mathcal{T}_{\mathcal{C}1}, \mathcal{T}_{\mathcal{C}2}, \mathcal{T}_{\mathcal{C}3} \;\bullet\!\rangle$$



$[c_1, c_2, c_3]$
$\mathcal{T}_{\mathcal{C}1}$
$\mathcal{T}_{\mathcal{C}2}$
$\mathcal{T}_{\mathcal{C}3}$

However, each of the three type constraints relates to one subexpression in particular. Therefore, we first attach the constraints to their corresponding constraint tree, and then combine the subtrees. This gives us the following tree.

$$\langle\!\bullet\; c_1 \lhd \mathcal{T}_{\mathcal{C}1}, c_2 \lhd \mathcal{T}_{\mathcal{C}2}, c_3 \lhd \mathcal{T}_{\mathcal{C}3} \;\bullet\!\rangle$$



$c_1\uparrow$
$c_2\uparrow$
$c_3\uparrow$
$\mathcal{T}_{\mathcal{C}1}$
$\mathcal{T}_{\mathcal{C}2}$
$\mathcal{T}_{\mathcal{C}3}$

In the figure above, the constraints are written with an upward arrow to indicate that the constraint was created by their parent node.

Each node in the abstract syntax tree produces one node in the constraint tree. We make sure that all the type constraints that are generated in one node of the abstract syntax tree belong to the corresponding node in the constraint tree. Moreover, we will never generate constraint trees of the following form (where c corresponds to the current constraint tree node, and not by some node inside $\mathcal{T}_{\mathcal{C}1}$ or $\mathcal{T}_{\mathcal{C}2}$).

$$c \lhd \{\!\!\{\, \mathcal{T}_{\mathcal{C}1}, \mathcal{T}_{\mathcal{C}2} \,\}\!\!\} \qquad\qquad \{\!\!\{\, c \rhd \mathcal{T}_{\mathcal{C}1}, \mathcal{T}_{\mathcal{C}2} \,\}\!\!\}$$

In both of these cases, the constraint c no longer belongs to the node at which it was created.

Before we continue, we introduce some abbreviations.

$$\begin{array}{lll r}
\bullet & =_{def} & \{\!\!\{\ \}\!\!\} & \textit{(empty tree)} \\
\mathcal{C}^{\bullet} & =_{def} & \mathcal{C} \trianglerighteq \bullet & \textit{(constraint list)} \\
[c_1, \ldots, c_n] \trianglerighteq \mathcal{T}_{\mathcal{C}} & =_{def} & c_1 \rhd \ldots \rhd c_n \rhd \mathcal{T}_{\mathcal{C}} & \textit{(attach constraints)} \\
[c_1, \ldots, c_n] \trianglelefteq \mathcal{T}_{\mathcal{C}} & =_{def} & c_1 \lhd \ldots \lhd c_n \lhd \mathcal{T}_{\mathcal{C}} & \textit{(attach constraints from p.)}
\end{array}$$

In order of appearance, we have the empty constraint tree, the tree consisting of a single constraint set, and attaching a list of constraints (from the parent) to a tree.

In the remaining part of this section, we discuss various alternatives to flatten a constraint tree, which results in an ordered list of constraints. Furthermore, we present spreading and phasing of type constraints, which are techniques to transform a constraint tree.

5.3.1 Flattening a constraint tree

Our first concern is how to flatten a constraint tree to a list: for this, we define the function *flatten*. How a tree is flattened depends on the tree walk of our choice, which is a parameter of *flatten*. A tree walk specifies the order of the constraints for each node in the constraint tree. We use the following Haskell datatype to represent a tree walk.

$$\textbf{newtype}\ \textit{TreeWalk} = \mathcal{T}_{\mathcal{W}}\ (\forall a.[a] \to [([a],[a])] \to [a])$$

With this rank-two type we can represent tree walks that do not depend on the type of the elements in the tree. A number of example tree walks will be presented later on.

We define two mutually recursive helper-functions for *flatten*: *flattenTop* and *flattenRec*. In the definition of *flattenRec*, we maintain a list of constraints that are attached to the constraint tree ($c \rhd \mathcal{T}_{\mathcal{C}}$): this list, called *down*, is passed to all recursive calls. Furthermore, we use the list *up* to collect the constraints that are attached to the tree by their parent node ($c \lhd \mathcal{T}_{\mathcal{C}}$). This list is computed in a bottom-up fashion.

```
$flatten :: TreeWalk \rightarrow ConstraintTree \rightarrow [\,Constraint\,]$
$flatten\ (\mathcal{T}_{\mathcal{W}}\ f) = flattenTop$
  **where**
    $flattenTop :: ConstraintTree \rightarrow [\,Constraint\,]$
    $flattenTop\ tree =$
      **let** $pair = flattenRec\ [\,]\ tree$
      **in** $f\ [\,]\ [pair]$

    $flattenRec :: [\,Constraint\,] \rightarrow ConstraintTree$
                       $\rightarrow ([\,Constraint\,], [\,Constraint\,])$
    $flattenRec\ down\ tree =$
      **case** $tree$ **of**
        $\lbag\, t_1, \ldots, t_n \,\rbag$ $\rightarrow$ **let** $pairs = map\ (flattenRec\ [\,])\ [t_1, \ldots, t_n]$
                            **in** $(f\ \ down\ pairs, [\,])$
        $c \rhd t$          $\rightarrow flattenRec\ (down +\!\!+ [\,c\,])\ t$
        $c \lhd t$          $\rightarrow$ **let** $(cset, up) = flattenRec\ down\ t$
                            **in** $(cset, up +\!\!+ [\,c\,])$
        $t_1 \ll t_2$       $\rightarrow$ **let** $cs_1 = flattenTop\ t_1$
                                $cs_2 = flattenTop\ t_2$
                            **in** $(f\ \ down\ [(cs_1 +\!\!+ cs_2, [\,])], [\,])$
```

Observe that for the case of a node, we let the tree walk decide how the constraint lists at that point are to be combined – i.e., the list of downward constraints, and for each subtree in $\lbag\, t_1, \ldots, t_n \,\rbag$, its flattened constraint set and the upward constraints. If the constraints of two constraint trees should be ordered in a strict way (the case $t_1 \ll t_2$), then we flatten the two constraint trees, which gives us the constraint sets cs_1 and cs_2. These sets are combined in a fixed way, regardless of the tree walk, namely $cs_1 +\!\!+ cs_2$. However, we let the tree walk decide how this combined list and the downward constraints are ordered.

The first tree walk we define is truly bottom-up.

```
$bottomUp :: TreeWalk$
$bottomUp = \mathcal{T}_{\mathcal{W}}\ (\lambda down\ list \rightarrow f\ \ (unzip\ list) +\!\!+ down)$
  **where** $f\ \ (csets, ups) = concat\ csets +\!\!+ concat\ ups$
```

This tree walk puts the recursively flattened constraint subtrees in the front, while preserving the order of the trees. These are followed by the upward constraints for each subtree in turn. Finally, we append the downward constraints.

Example 5.7. Assume that we have the following constraint tree.

$$\mathcal{T}_{\mathcal{C}} = down \trianglerighteq \lbag\, up_1 \trianglelefteq \mathcal{C}_1^{\bullet}, \ldots, up_n \trianglelefteq \mathcal{C}_n^{\bullet} \,\rbag$$

Flattening this constraint tree with the bottom-up tree walk gives us the following constraint list.

$$flatten\ \ bottomUp\ \mathcal{T}_{\mathcal{C}} \quad = \quad \mathcal{C}_1 +\!\!+ \ldots +\!\!+ \mathcal{C}_n +\!\!+ up_1 +\!\!+ \ldots +\!\!+ up_n +\!\!+ down$$

The second tree walk we discuss follows a top-down approach.

$$\begin{array}{l} \mathit{topDown} :: \mathit{TreeWalk} \\ \mathit{topDown} = \mathcal{T}_{\mathcal{W}}\ (\lambda \mathit{down}\ \mathit{list} \rightarrow \mathit{down} +\!\!+ f\ (\mathit{unzip}\ \mathit{list})) \\ \quad \textbf{where}\ f\ (\mathit{csets}, \mathit{ups}) = \mathit{concat}\ \mathit{ups} +\!\!+ \mathit{concat}\ \mathit{csets} \end{array}$$

This tree walk puts the downward constraints in front, directly followed by all the upward constraints. Lastly, the recursively flattened subtrees are included in the list.

Example 5.7 (continued). We use the top-down tree walk to flatten $\mathcal{T}_{\mathcal{C}}$.

$$\mathit{flatten}\ \mathit{topDown}\ \mathcal{T}_{\mathcal{C}} \quad = \quad \mathit{down} +\!\!+ \mathit{up}_1 +\!\!+ \ldots +\!\!+ \mathit{up}_n +\!\!+ \mathcal{C}_1 +\!\!+ \ldots +\!\!+ \mathcal{C}_n$$

Obviously, there are more interesting tree walks. For instance, we could interleave the upward constraints and the flattened constraint trees at each node. Such an order differs from the bottom-up and top-down tree walks, which consider *all* the upward constraints, and *all* the recursively flattened constraint trees, at once. We have two choices to make: what is the position of the upward constraints with respect to the recursively flattened tree, and do we put the downward constraints in front or at the end of the list. These two options lead to the following helper-function.

$$\begin{array}{l} \mathit{variation} :: (\forall a.[\,a\,] \rightarrow [\,a\,] \rightarrow [\,a\,]) \rightarrow (\forall a.[\,a\,] \rightarrow [\,a\,] \rightarrow [\,a\,]) \rightarrow \mathit{TreeWalk} \\ \mathit{variation}\ f_1\ f_2 = \mathcal{T}_{\mathcal{W}}\ (\lambda \mathit{down}\ \mathit{list} \rightarrow f_1\ \mathit{down}\ (\mathit{concatMap}\ (\mathit{uncurry}\ f_2)\ \mathit{list})) \end{array}$$

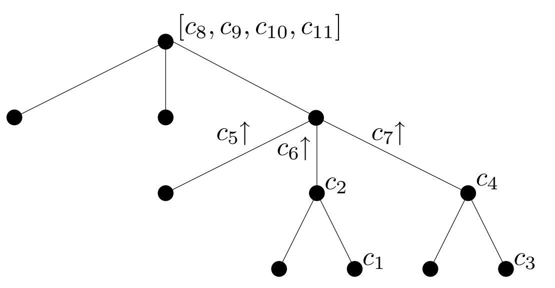
For both arguments of *variation*, we consider two alternatives: combine the lists in the order given ($+\!\!+$), or flip the order of the lists (*flip* ($+\!\!+$)). Hence, we get four more tree walks.

$$\begin{array}{ll} \mathit{inorderFirstBU} & = \mathit{variation}\ (+\!\!+)\ (+\!\!+) \\ \mathit{inorderFirstTD} & = \mathit{variation}\ (+\!\!+)\ (\mathit{flip}\ (+\!\!+)) \\ \mathit{inorderLastBU} & = \mathit{variation}\ (\mathit{flip}\ (+\!\!+))\ (+\!\!+) \\ \mathit{inorderLastTD} & = \mathit{variation}\ (\mathit{flip}\ (+\!\!+))\ (\mathit{flip}\ (+\!\!+)) \end{array}$$

Example 5.7 (continued). Once more, we consider the constraint tree $\mathcal{T}_{\mathcal{C}}$, and flatten this tree with the four tree walk defined above.

$$\begin{array}{lcl} \mathit{flatten}\ \mathit{inorderFirstBU}\ \mathcal{T}_{\mathcal{C}} & = & \mathit{down} +\!\!+ \mathcal{C}_1 +\!\!+ \mathit{up}_1 +\!\!+ \ldots +\!\!+ \mathcal{C}_n +\!\!+ \mathit{up}_n \\ \mathit{flatten}\ \mathit{inorderFirstTD}\ \mathcal{T}_{\mathcal{C}} & = & \mathit{down} +\!\!+ \mathit{up}_1 +\!\!+ \mathcal{C}_1 +\!\!+ \ldots +\!\!+ \mathit{up}_n +\!\!+ \mathcal{C}_n \\ \mathit{flatten}\ \mathit{inorderLastBU}\ \mathcal{T}_{\mathcal{C}} & = & \mathcal{C}_1 +\!\!+ \mathit{up}_1 +\!\!+ \ldots +\!\!+ \mathcal{C}_n +\!\!+ \mathit{up}_n +\!\!+ \mathit{down} \\ \mathit{flatten}\ \mathit{inorderLastTD}\ \mathcal{T}_{\mathcal{C}} & = & \mathit{up}_1 +\!\!+ \mathcal{C}_1 +\!\!+ \ldots +\!\!+ \mathit{up}_n +\!\!+ \mathcal{C}_n +\!\!+ \mathit{down} \end{array}$$

Our last example is a tree walk transformer: at each node in the constraint tree, the children are inspected in reversed order. Of course, this reversal is not applied to nodes with a strict ordering. With this transformer, we can inspect a program from right-to-left, instead of the standard left-to-right order.



$$\begin{aligned}
&c_1{}^* = v_4 \equiv \mathit{Int}\\
&c_2{}^* = v_3 \equiv v_4 \rightarrow v_5\\
&c_3{}^* = v_7 := \textsc{Inst}(\forall a.[a])\\
&c_4{}^* = v_6 \equiv v_7 \rightarrow v_8\\
&c_5\ \ = v_2 \equiv \mathit{Bool}\\
&c_6\ \ = v_5 \equiv v_9\\
&c_7\ \ = v_8 \equiv v_9\\
&c_8{}^* = v_0 \equiv v_3\\
&c_9{}^* = v_0 \equiv v_6\\
&c_{10} = v_1 \equiv v_2\\
&c_{11} = v_{10} \equiv v_0 \rightarrow v_1 \rightarrow v_9
\end{aligned}$$

Figure 5.1. A constraint tree

$$\begin{aligned}
&\mathit{reversed} :: \mathit{TreeWalk} \rightarrow \mathit{TreeWalk}\\
&\mathit{reversed}\ (\mathcal{T}_{\mathcal{W}}\ f) = \mathcal{T}_{\mathcal{W}}\ (\lambda \mathit{down}\ \mathit{list} \rightarrow f\ \mathit{down}\ (\mathit{reverse}\ \mathit{list}))
\end{aligned}$$

All presented tree walks work uniformly on constraint trees, i.e., they are independent of the information in the tree. But also irregular tree walks can be of interest, and may lead to even more constraint orderings to choose from: for instance, a tree walk that visits the subtree with the most type constraints first, or an ordering which is specialized for the constraints of a particular set of expressions. In Chapter 9, we will discuss how to create custom constraint orderings by defining specialized type rules.

We conclude our discussion on flattening a constraint tree with an example, which illustrates the impact the order of constraints has on which constraint is reported.

Example 5.8. Let us consider the following ill-typed expression. Various parts of the expression are annotated with their assigned type variable. Furthermore, v_9 is assigned to the if-then-else expression, and v_{10} to the complete expression.

$$\lambda\ \underset{v_0}{\underset{|}{f}}\ \ \underset{v_1}{\underset{|}{b}} \rightarrow\ \mathbf{if}\ \underset{v_2}{\underset{|}{b}}\ \mathbf{then}\ \overbrace{\underset{v_3}{\underset{|}{f}}\ \ \underset{v_4}{\underset{|}{1}}}^{v_5}\ \mathbf{else}\ \overbrace{\underset{v_6}{\underset{|}{f}}\ \ \underset{v_7}{\underset{|}{[\,]}}}^{v_8}$$

The constraint tree $\mathcal{T}_{\mathcal{C}}$ constructed for this expression is shown in Figure 5.1. The constraints in this tree are inconsistent: the constraints in the only minimal inconsistent subset are marked with a star. Hence, a greedy constraint solver will report the last of the marked constraints in the list as incorrect. We consider three flattening strategies. The underlined constraints are the locations where the inconsistency is detected.

$$\begin{aligned}
&\mathit{flatten}\ \mathit{bottomUp}\ \mathcal{T}_{\mathcal{C}} &&= [c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, \underline{c_9}, c_{10}, c_{11}]\\
&\mathit{flatten}\ \mathit{topDown}\ \mathcal{T}_{\mathcal{C}} &&= [c_8, c_9, c_{10}, c_{11}, c_5, c_6, c_7, c_2, c_1, c_4, \underline{c_3}]\\
&\mathit{flatten}\ (\mathit{reversed}\ \mathit{topDown})\ \mathcal{T}_{\mathcal{C}} &&= [c_8, c_9, c_{10}, c_{11}, c_7, c_6, c_5, c_4, c_3, c_2, \underline{c_1}]
\end{aligned}$$

Observe that for each of the tree walks, the inconsistency shows up while solving a different constraint. These constraints originated from the whole expression, the subexpression [ ], and the subexpression 1, respectively.

5.3.2 Spreading type constraints

We present a technique to move type constraints from one place in the constraint tree to a different location. This can be useful if constraints generated at a certain place in the abstract syntax tree also correspond to a second location. In particular, we will consider constraints that relate the definition site and the use site of an identifier. An advantage of this approach (to spread the type constraints in a constraint tree) is that we get more interesting ways to organize the type constraints without changing the constraint collecting process. More specifically, by spreading constraints we can emulate algorithms that use a top-down type environment, while using a bottom-up assumption set to collect the constraints.

The grammar for constraint trees is extended with three cases.

Constraint tree:

$$\begin{array}{lll} \mathcal{T}_\mathcal{C} ::= & (\ldots) & \textit{(alternatives on page 79)} \\ \quad\mid & (\ell, c) \rhd\!\!\circ\, \mathcal{T}_\mathcal{C} & \textit{(spread constraint)} \\ \quad\mid & (\ell, c) \ll\!\!\circ\, \mathcal{T}_\mathcal{C} & \textit{(spread constraint strict)} \\ \quad\mid & \ell^\circ & \textit{(receiver)} \end{array}$$

The first two extra cases serve to spread a constraint downwards, whereas the third marks a position to receive a constraint that is being spread. Labels ℓ are used only to find matching spread-receive pairs. The scope of spreading a constraint is limited to the right argument of $\rhd\!\!\circ$ (and $\ll\!\!\circ$).

The cases $\rhd\!\!\circ$ and $\ll\!\!\circ$ to spread a constraint are equivalent when it comes to spreading. However, if we choose not to spread constraints, then the two cases have different behavior for flattening. The first attaches the constraint to the tree, whereas the second demands that the constraint is considered before all the constraints in the tree. We extend the definition to flatten a tree, and cover the three new cases.

```
flattenRec down tree =
  case tree of
    ...                          → ...
    $(\ell, c) \rhd\!\!\circ\, t$  → flattenRec down $(c \rhd t)$
    $(\ell, c) \ll\!\!\circ\, t$   → flattenRec down $([c]^\bullet \ll t)$
    $\ell^\circ$                  → flattenRec down $\bullet$
```

We introduce some syntactic sugar to spread labeled constraint sets.

$$\begin{array}{ll} [(\ell_1, c_1), \ldots, (\ell_n, c_n)] \;\underline{\rhd\!\!\circ}\; \mathcal{T}_\mathcal{C} & \\ \qquad =_{def} \; (\ell_1, c_1) \rhd\!\!\circ \ldots \rhd\!\!\circ (\ell_n, c_n) \rhd\!\!\circ\, \mathcal{T}_\mathcal{C} & \textit{(spread list)} \\ [(\ell_1, c_1), \ldots, (\ell_n, c_n)] \;\underline{\ll\!\!\circ}\; \mathcal{T}_\mathcal{C} & \\ \qquad =_{def} \; (\ell_1, c_1) \ll\!\!\circ \ldots \ll\!\!\circ (\ell_n, c_n) \ll\!\!\circ\, \mathcal{T}_\mathcal{C} & \textit{(spread list strict)} \end{array}$$

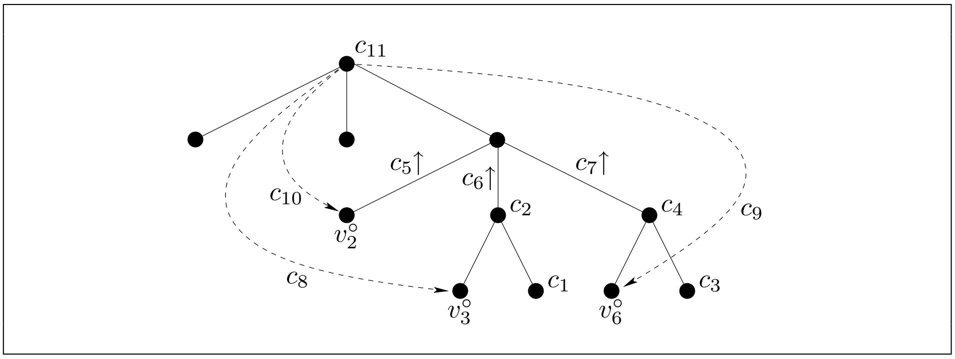


Figure 5.2. A constraint tree with type constraints that have been spread

We define a function *spread*, which spreads the type constraints in a tree. We pass a list of labeled type constraints that are spread as an inherited attribute.

```
$spread :: ConstraintTree \to ConstraintTree$
$spread = spreadRec\ [\,]$
  **where**
    $spreadRec :: [(Label, Constraint)] \to ConstraintTree \to ConstraintTree$
    $spreadRec\ list\ tree =$
      **case** $tree$ **of**
        $⦃\, t_1, \ldots, t_n\, ⦄ \to ⦃\, map\ (spreadRec\ list)\ [t_1, \ldots, t_n]\, ⦄$
        $c \rhd t$          $\to c \rhd spreadRec\ list\ t$
        $c \lhd t$          $\to c \lhd spreadRec\ list\ t$
        $t_1 \ll t_2$        $\to spreadRec\ list\ t_1 \ll spreadRec\ list\ t_2$
        $(\ell, c) \rhd\!\circ\, t$     $\to spreadRec\ ((\ell, c) : list)\ t$
        $(\ell, c) \ll\!\circ\, t$     $\to spreadRec\ ((\ell, c) : list)\ t$
        $\ell^\circ$             $\to [c \mid (\ell', c) \leftarrow list, \ell == \ell']^\bullet$
```

This definition of spreading comes with a warning: improper use may lead to disappearance or duplication of type constraints. For instance, we expect for every constraint that is spread to have exactly one receiver in its scope. The definition of *spread* can be extended straightforwardly to check this property for a given constraint tree.

Example 5.8 (continued). We take a second look at the constraint tree for the ill-typed expression introduced in Example 5.8. We spread the type constraints introduced for the monomorphic pattern variables f and b to their use sites. Hence, the constraints c_8, c_9, and c_{10} are moved to a different location in the constraint tree. At the three nodes of the variables (two for f, one for b), we put a receiver. The type variable that is assigned to a variable is also used as the label for the receiver. Hence, we get the receivers v_2°, v_3°, and v_6°. The constraint tree after spreading ($\mathcal{T_C}'$) is displayed in Figure 5.2.

$$\begin{array}{ll} \mathit{flatten\ bottomUp}\ \mathcal{T_C}' & = [c_{10}, c_8, c_1, c_2, c_9, c_3, \underline{c_4}, c_5, c_6, c_7, c_{11}] \\ \mathit{flatten\ topDown}\ \mathcal{T_C}' & = [c_{11}, c_5, c_6, c_7, c_{10}, c_2, c_8, c_1, c_4, c_9, \underline{c_3}] \\ \mathit{flatten}\ (\mathit{reversed\ bottomUp})\ \mathcal{T_C}' & = [c_3, c_9, c_4, c_1, c_8, \underline{c_2}, c_{10}, c_7, c_6, c_5, c_{11}] \end{array}$$

The *bottomUp* tree walk after spreading leads to reporting the constraint c_4 (created for the application f [ ]): without spreading type constraints, c_9 is reported.

One could say that spreading undoes the bottom-up construction of assumption sets for the free identifiers, and instead "applies" the more standard approach to pass down a type environment (usually denoted by Γ). Therefore, spreading type constraints gives a constraint tree that corresponds more closely to the type inference process of Hugs and GHC. Regarding the inference process for a conditional expression, both compilers constrain the type of the condition to be of type *Bool* before continuing with the then and else branches. GHC constrains the type of the condition even before its type is inferred: Hugs constrains this type afterwards. Therefore, the inference process of Hugs for a conditional expression corresponds to an inorder bottom-up tree walk. The behavior of GHC can be mimicked by an inorder top-down tree walk.

5.3.3 Phasing constraint trees

Phasing type constraints is another technique to rearrange a constraint tree, which is to be used before a tree is flattened into a list. The idea is very simple: we assign phase numbers to parts of the constraint tree. Constraints with a low phase number should be considered before constraints with a high phase number, although we respect the restrictions on the constraint order imposed by strict nodes.

We extend our definition of constraint trees with an extra case to assign a phase number to a constraint tree.

$$\begin{array}{lll} \multicolumn{3}{l}{\textit{Constraint tree:}} \\ \mathcal{T_C} ::= & (\ldots) & \textit{(alternatives on page 84)} \\ \quad\ \mid & \mathit{Phase}\ i\ \mathcal{T_C} & \textit{(phasing)} \end{array}$$

In the new case, i is the phase number. We assume 5 to be the default phase number. The definitions for flattening and spreading are extended to handle phased constraint trees. In both definitions, the assigned phase number is ignored.

```
flattenRec down tree =
  case tree of
    ...          → ...
    Phase i t → flattenRec down t

spreadRec list tree =
  case tree of
    ...          → ...
    Phase i t → Phase i (spreadRec list t)
```

We now define a function *phase*, which transforms a tree with phase numbers into a tree where the phases are made explicit by strict nodes. To perform this transformation, we use *phase maps*. A phase map is a list of pairs of a phase number and a constraint tree.

```
newtype PhaseMap = PM [(Int, ConstraintTree)]
```

We make sure that the phase numbers in a phase map are always strictly increasing. First, we define a number of helper-functions to construct, combine, and use phase maps.

```
pmEmpty :: PhaseMap
pmEmpty = PM []

pmSingleton :: Int → ConstraintTree → PhaseMap
pmSingleton i tree = PM [(i, tree)]
```

Our next function creates a constraint tree from a phase map by ordering the constraint trees of the phases in a strict way.

```
pmToTree :: PhaseMap → ConstraintTree
pmToTree (PM xs) = foldr (λ(_, t₁) t₂ → t₁ ≪ t₂) • xs
```

The following functions combine phase maps.

```
pmPlus :: PhaseMap → PhaseMap → PhaseMap
pmPlus (PM xs) (PM ys) = PM (f xs ys)
  where
    f [] ys = ys
    f xs [] = xs
    f xs@((i, tx) : restx) ys@((j, ty) : resty)
       | i == j = (i, ⦃ tx, ty ⦄) : f restx resty
       | i < j  = (i, tx) : f restx ys
       | i > j  = (j, ty) : f xs resty

pmConcat :: [PhaseMap] → PhaseMap
pmConcat = foldr pmPlus pmEmpty

pmAddDefault :: ConstraintTree → PhaseMap → PhaseMap
pmAddDefault tree = pmPlus (pmSingleton 5 tree)
```

With these helper-functions, we define how to phase a constraint tree. The local function *phaseRec* returns a constraint tree and a phase map. The tree that is returned by this function is the current constraint tree, for which we do not have a phase number.

```
$phase :: ConstraintTree \to ConstraintTree$
$phase = phaseTop$
  where
    $phaseTop :: ConstraintTree \to ConstraintTree$
    $phaseTop\ tree =$
      let $(t, pm) = phaseRec\ tree$
      in $pmToTree\ (pmAddDefault\ t\ pm)$

    $phaseRec :: ConstraintTree \to (ConstraintTree, PhaseMap)$
    $phaseRec\ tree =$
      case $tree$ of
        $\llbracket t_1, \ldots, t_n \rrbracket$ $\to$ let $pairs = map\ phaseRec\ [t_1, \ldots, t_n]$
                                      $([t'_1, \ldots, t'_n], pms) = unzip\ pairs$
                                  in $(\llbracket t'_1, \ldots, t'_n \rrbracket, pmConcat\ pms)$
        $c \rhd t$          $\to$ let $(t', pm) = phaseRec\ t$
                                  in $(c \rhd t', pm)$
        $c \lhd t$          $\to$ let $(t', pm) = phaseRec\ t$
                                  in $(c \lhd t', pm)$
        $t_1 \ll t_2$       $\to$ $(phaseTop\ t_1 \ll phaseTop\ t_2, pmEmpty)$
        $(\ell, c) \rhd\!\!\circ\ t$    $\to$ let $(t', pm) = phaseRec\ t$
                                  in $((\ell, c) \rhd\!\!\circ\ t', pm)$
        $(\ell, c) \ll\!\!\circ\ t$    $\to$ $((\ell, c) \ll\!\!\circ\ phaseTop\ t, pmEmpty)$
        $\ell^\circ$            $\to$ $(\ell^\circ, pmEmpty)$
        $Phase\ i\ t$       $\to$ let $(t', pm) = phaseRec\ t$
                                  in $(\bullet, pmPlus\ (pmSingleton\ i\ t')\ pm)$
```

Most definitions are relatively straightforward: we discuss those that are not. First, we explain the case for *Phase i ctree*. From the recursive call, we get the constraint tree t' which is assigned phase number i. This tree is added with the appropriate phase number to the phase map. We return the empty constraint tree as the first component of the pair since there are no more constraints for which we do not know their corresponding phase. Secondly, the two cases that impose a strict ordering on the constraints ($t_1 \ll t_2$ and $(\ell, c) \ll\!\!\circ\ t$) should be handled with care. In both cases, we use the function *phaseTop*, which uses *pmToTree* for converting the phase map to a constraint tree, and we return an empty phase map.

Example 5.8 (continued). Consider once more the expression

$$\lambda f\ b \to \textbf{if}\ b\ \textbf{then}\ f\ 1\ \textbf{else}\ f\ [\,],$$

and its constraint tree after spreading type constraints. This tree is shown in Figure 5.3 on the left. Suppose that we want to treat the subexpressions of conditionals in a special way. For example, we consider the constraints of the condition (including the constraint that this expression should have type *Bool*) before all the other type constraints, so we assign phase 3 to this part of the constraint tree. In a similar way, we postpone the constraints for the two branches, and use phase number 7 for these parts. The remaining type constraints are assigned to the default phase (which is 5). The right part of Figure 5.3 shows the constraint tree after phasing.

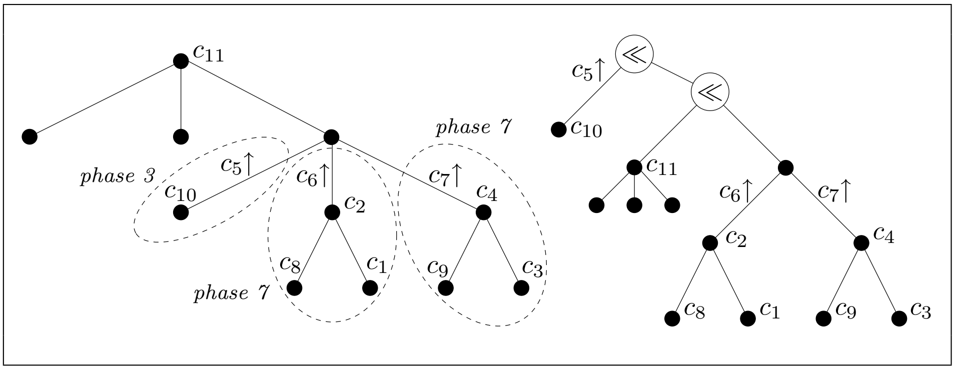


Figure 5.3. A constraint tree before and after phasing

The two strict nodes combine the three constraint trees of phase 3, 5, and 7 (from left to right). Note that a number of empty constraint trees have been omitted to simplify the presentation of the tree.

The phasing strategy of the previous example is not very realistic: it only illustrates the technique. In Chapter 9, we present convincing examples of how we can improve the quality of type error messages by assigning phase numbers to type constraints.

In conclusion: building a constraint tree for an expression that follows the shape of the abstract syntax tree provides the flexibility to come up with different orderings of the type constraints. Several tree walks have been proposed, including a bottom-up and a top-down approach. Furthermore, we have presented two techniques to rearrange the constraint tree, namely the spreading of type constraints, and assigning phase numbers to constraint trees.

5.4 Kind inference

In Chapter 2, we pointed out to the reader that all types are assumed to be well-formed. Normally, this is taken care of by kind inference. Each type (type constructor, type variable) is assigned a kind, and inference proceeds similarly to type inference. The kind constant $\star$ is assigned to all types that represent a value. For instance, *Int*, [*Bool*], and $v_0 \to v_0$ all have kind $\star$. The second kind constant $(\to)$ denotes function space, and should not be confused with the function space for types. A function kind is assigned to type constructors. For example, the list type constructor has kind $\star \to \star$, and $\to$ (function space for types!) has kind $\star \to \star \to \star$. For each type application $\tau_1\ \tau_2$ we have to check that the kind assigned to τ_1 is a function kind, and that τ_2's kind fits. Because kind inference is similar to type inference, we can easily adapt the constraint-based framework to perform kind analysis.

In fact, kind inference is much simpler than type inference, and there are two good reasons for this. Firstly, the number of alternatives of the type language is smaller than for the expression language. Secondly, no polymorphic kinds are inferred in Haskell 98: kind variables are used for unknown kinds, and all kind variables that remain are assumed to be of kind $\star$. This process is called defaulting, and it takes place instead of generalization. Note that, although all inferred kinds are monomorphic, we have to perform some sort of binding group analysis.

We do not present a set of rules to describe how the kind constraints are collected. Instead, we indicate which kind constraints are collected for a number of datatype declarations, and we discuss which kinds are inferred.

Example 5.9. Consider the following datatype for a binary tree.

data *Tree a* $=$ *Bin* (*Tree a*) (*Tree a*) $|$ *Leaf a*

We start with introducing fresh kind variables for the introduced datatype and the type arguments: this gives us *Tree* $:: \kappa_0$ and $a :: \kappa_1$. (We write $\kappa_0, \kappa_1, \kappa_2, \ldots$ to distinguish kind variables from type variables.) The first kind constraint we collect is $\kappa_0 \equiv \kappa_1 \rightarrow \star$, which expresses that the type *Tree a* is of kind $\star$, and that *Tree* is a type constructor expecting a type of kind κ_1. The fields of the datatype's alternatives are values, and thus of kind $\star$. We collect $\kappa_0 \equiv \kappa_1 \rightarrow \star$ twice for the two occurrences of *Tree a* in the alternative *Bin*. The last constraint we find is $\kappa_1 \equiv \star$ for the type variable a in *Leaf*'s alternative.

The collected constraints are consistent, and we find $\star$ for κ_1 (the type variable a), and $\star \rightarrow \star$ for κ_0 (the type constructor *Tree*).

Example 5.10. Slightly more difficult is the kind inferred for *GRose*.

data *GRose f a* $=$ *GRose a* (f a)

We assign κ_0 to *GRose*, κ_1 to f, κ_2 to a, and we create the constraint $\kappa_0 \equiv \kappa_1 \rightarrow \kappa_2 \rightarrow \star$. For the two fields of a *GRose*, we generate $\kappa_2 \equiv \star$ and $\kappa_1 \equiv \kappa_2 \rightarrow \star$. The substitution $[\kappa_0 := (\star \rightarrow \star) \rightarrow \star \rightarrow \star, \kappa_1 := \star \rightarrow \star, \kappa_2 := \star]$ satisfies all constraints. Hence, the first type argument of *GRose* must be a type constructor of kind $\star \rightarrow \star$, for example the list type constructor.

Example 5.11. Our last example considers the datatype *Apply*.

data *Apply f a* $=$ *Apply* (f a)

We collect the kind constraints $\kappa_0 \equiv \kappa_1 \rightarrow \kappa_2 \rightarrow \star$ and $\kappa_1 \equiv \kappa_2 \rightarrow \star$, where κ_0 is assigned to *Apply*. Observe that no deductions are found for the kind variable κ_2. Generalizing the kind of *Apply* would give us the polymorphic *kind scheme* $\forall \kappa.(\kappa \rightarrow \star) \rightarrow \kappa \rightarrow \star$. However, we default the inferred kind, and we get the monomorphic kind $(\star \rightarrow \star) \rightarrow \star \rightarrow \star$. Because the kind of *Apply* is defaulted, some types are considered incorrect, although they are well-formed when using the polymorphic kind scheme.

The kind constraints can also be collected in a bottom-up fashion, as for type inference. This approach leads to more kind variables being introduced, and more kind constraints being collected.

5.5 Advantages

We have presented a framework to tackle the problem of type inference. One distinct feature of this framework is that it is based on type constraints. We have divided the process of type inference into three consecutive steps: collecting constraints, ordering constraints, and solving constraints. Each constraint carries its own information, which can be used to create error messages (in case the constraint turns out to be unsatisfiable), or to provide guidance in the solving of the constraints. We did not make assumptions about the content of the information stored in a type constraint. Instead, this is left unspecified by the framework, and should be provided by the user of the framework. This gives us a considerable amount of flexibility in the amount of information that is present while the constraints are being solved, which is an important property if good feedback and clear error messages are important. Although the scope of this chapter is limited to type inference, it should be clear that the principles of our approach could be applied equally well to other program analyses. We conclude this chapter with a list of advantages of our approach.

1. *Separation of concerns.* The approach we follow is based on type constraints, and thus inherits all the advantages constraint-based analyses have in general. Typically, a constraint-based analysis consists of two parts. First, we collect constraints, which is the specification of the problem. The semantics of the constraints should be clear, i.e., we have to know when the constraints are satisfied (but not how we will achieve this). A completely separate issue is how the collected constraints should be solved. We deal with this question in the second step, which is thus the chosen implementation of the analysis at hand. Typically, implementations that do not follow the constraint-based approach do not make this distinction. This makes constraint-based solutions flexible and very suitable for experimentation. In our framework, we distinguish a third phase, in which an ordering of the constraints can be selected.
2. *Increased reusability.* All the efforts to perform type inference, including all the proposals to obtain good error messages, can be reused in a straightforward way. The only work that remains is collecting the type constraints from an abstract syntax tree, which is a relatively straightforward task. This setup is particularly convenient for developers of tools and compilers in which type inference is not a primary concern, although it is a desirable analysis to have. To build a type inferencer from scratch can be quite laborious and subtle to get right. Of course, this framework can also be used for languages other than Haskell. In particular, it supports unification-based program analyses.
3. *Extensibility.* This framework can be extended along several lines, as we discussed in the introduction of this chapter. The most powerful extension is that we can add new type constraints to the framework to deal with extensions to the (type) language. For example, in Chapter 10, we extend the framework to cope with type classes and overloading. In fact, we can also support other qualifiers (predicates).

4. *Alternative configurations.* Our framework includes several strategies to solve type constraints. Because the solvers possess different properties, such as a good run-time performance, a clean implementation, or high quality error reports, a user is free to choose one suited for his specific purpose. If the constraint solvers can be customized even further, one can really tune the type inference process. For greedy constraint solvers in particular, the order in which the constraints are solved is of great importance. For this, we introduced a special phase in which we order the constraints (by building, rearranging, and flattening a constraint tree). Here too, we have freedom of choice. All these options give a large number of possible configurations, which makes the framework a valuable environment for experimenting with type inference algorithms.
5. *Heuristics to improve error reporting.* Within this framework we can report clear and concise error messages: this was our initial motivation to conduct research in this area. In the next chapter, we discuss a solver which solves a constraint set in a global way. In this setting we can really benefit from the fact that we are free to choose which information is stored in the type constraints. It becomes easy to design and implement heuristics which decide what is reported for an inconsistent constraint set. Errors committed frequently can be captured by specialized heuristics, and it is easy to add new heuristics to the framework. Some of the heuristics can be specialized and customized in detail, resulting in so-called *type inference directives.* These directives give a user the possibility to reprogram the type inference process without changing the framework or the compiler. We will explore type inference directives in Chapter 9.

6

A full-blown type system

Overview. *A set of type rules is presented to collect constraints for nearly the entire Haskell 98 language (except for overloading). These constraint collecting rules are used in the Helium compiler. For a number of language constructs, we discuss how generating basic constraints helps to improve the quality of type error messages.*

In this chapter, we present a set of type rules that covers almost the language of the Haskell 98 standard. The most notable difference is that we do not consider overloading. The language features that are overloaded in the standard (for example, the numerical literals, enumerations, and do expressions) are specialized to work for one particular type only. In Chapter 10, we discuss how the type inference framework can be extended to cope with overloading, and how the presented type rules can be generalized accordingly.

Besides overloading, there are some minor differences, but none of great importance. For instance, we do not consider records. Moreover, the type rules we present cover exactly the language supported by the Helium compiler [26]. This compiler has been successfully employed in three introductory programming courses at Utrecht University. Thus, this chapter addresses a full-blown language.

Before we discuss the type rules, we have to ask ourselves whether this exercise is worth the effort. Most presentations of type systems are limited to the core of a language, such as the set of type rules given in Figure 4.5. Many language constructs can indeed be formulated in terms of a smaller core language. However, this approach (desugaring) is not to be preferred if good error messages are the primary concern. The set of presented type rules proves that the constraint-based approach is scalable.

In the type rules, we use the notation of Section 5.3 for building constraint trees. These rules are carefully designed to allow for high quality error messages. It is not our intention to minimize the number of generated constraints, nor the number of introduced type variables.

In particular, we devote attention to the binding group analysis, which is also formulated in a bottom-up fashion. A detailed description of this analysis can be found in Section 6.6.

6.1 Preliminaries

Before we present the type rules, we give an overview of the meta-variables and symbols that we use for identifiers, operators, non-terminals, and attributes that appear in the rules. Furthermore, we introduce some notational conventions.

Identifiers and operators

In Haskell, identifiers and operators have different name spaces. The set of identifiers can be split into two classes: variable identifiers, starting with a lower-case letter, and constructor identifiers, starting with an upper-case letter. To distinguish these two classes of identifiers in our type rules, we use the meta-variables x and C, respectively. Likewise, we have distinct name spaces for operators and constructor operators, for which we use the symbols $\otimes$ and $\bigotimes$, respectively. The following table summarizes which symbol is used for an identifier or operator.

C	constructor identifier	$\bigotimes$	constructor operator
x	variable identifier	$\otimes$	(function) operator

Note, however, that operators can be used in prefix notation by putting parentheses around the operator. For instance, we can write $(+)\ 3\ 4$ instead of $3 + 4$. Similarly, we can use identifiers in infix position by using back-quotes. For example, 3 ‘*plus*‘ 4 equals *plus* 3 4. The same rules hold for constructor identifiers and constructor operators. If we refer to an identifier, or we write a meta-variable x, then this includes parenthesized operators.

Non-terminals

The grammar for Haskell has many non-terminals, and in order to distinguish these in the type rules, each non-terminal is assigned a unique meta-variable. This meta-variable is used throughout the type system consistently. If necessary, we use variations of the meta-variable for different occurrences, such as a_1, a', or a'_1 for the meta-variable a. The following table lists the meta-variables for all the non-terminals we consider.

alt	alternative	*me*	maybe expression
d	declaration	*module*	module
e	expression	*ms*	(sequence of) statements
fb	function binding	*p*	pattern
ge	guarded expression	*qs*	(sequence of) qualifiers
l	literal	*rhs*	right-hand side
lhs	left-hand side		

Attributes and symbols

Most of the attributes used in the typing judgements have already been introduced in the previous chapters. For completeness, we present the following table with all the attributes and symbols occurring in the type rules.

Γ	(initial) type environment	$\mathcal{A}$	assumption set
Σ	type signatures	$\mathcal{B}$	binding group triple
β	(fresh) type variable	$\mathcal{C}$	constraint list
σ	(polymorphic) type annotation	$\mathcal{C}_\ell$	labeled constraint list
σ_v	(fresh) type scheme variable	$\mathcal{E}$	environment
τ	(monomorphic) type	$\mathcal{M}$	(monomorphic) types
$\tau^?$	maybe type	$\mathcal{T}_\mathcal{C}$	constraint tree
		c	type constraint

The type rules contain four different kinds of type environments. Each of the following environments associates identifiers with types.

$$\begin{array}{ll}
\Gamma = [x_1 : \sigma_1, \ldots, x_n : \sigma_n] & x_1, \ldots, x_n \textit{ are distinct, and } \mathit{ftv}(\Gamma) = \emptyset \\
\Sigma = [x_1 : \sigma_1, \ldots, x_n : \sigma_n] & x_1, \ldots, x_n \textit{ are distinct, and } \mathit{ftv}(\Sigma) = \emptyset \\
\mathcal{A} = [x_1 : \tau_1, \ldots, x_n : \tau_n] & \\
\mathcal{E} = [x_1 : \tau_1, \ldots, x_n : \tau_n] & x_1, \ldots, x_n \textit{ are distinct}
\end{array}$$

Γ is the initial type environment: it contains type schemes for all the imported functions, and for all data constructors. These are the types we have at top-level *before* we start inferring types. Σ is slightly different: it is used to collect declared type signatures. Both Γ and Σ contain only closed type schemes, i.e., no free type variables appear in these environments. $\mathcal{A}$ and $\mathcal{E}$ relate monomorphic types to identifiers. The former is used to record the types assigned to the unbound identifiers of an expression, whereas the latter is used for pattern variables that bind free identifiers. Note that $\mathcal{A}$ can have multiple assumptions about one identifier (but not $\mathcal{E}$). We write $\biguplus_{i=1}^{n} \mathcal{A}_i$ for $\mathcal{A}_1 +\!\!+ \ldots +\!\!+ \mathcal{A}_n$. The same notation is used to combine the other type environments.

We collect triples $\mathcal{B} = (\mathcal{E}, \mathcal{A}, \mathcal{T}_\mathcal{C})$ to perform a binding group analysis. Such a binding group triple contains all the information from a single binding group in which we are interested.

Notation

Some type rules contain judgements that are of the form $\vdash_0 C : \sigma$. This judgement should be interpreted as "under an initial type environment Γ, the type scheme σ is assigned to the constructor C". A more conventional notation for this judgement would thus be $\Gamma \vdash C : \sigma$. However, Γ does not appear in the judgement because its value does not vary: it is constant. One can consider Γ to be a global variable, or include it in every type rule.

The type rules we present can be translated directly into an attribute grammar [62]. Most attributes of the typing judgements are computed in a bottom-up fashion. Our convention is that the few inherited attributes are written enclosed in angle brackets. In particular, the set of monomorphic types $\mathcal{M}$ is passed on top-down, hence, we write $\langle \mathcal{M} \rangle$. As a result, the type rules presented in this chapter are not just a specification of constraint-based type inference, but also suggest an implementation.

Some typing judgements contain a list of types. To avoid confusion between a list of types and a *list type*, we introduce special notation for a type list attribute. We write $\{\!\![\ \tau_1, \ldots, \tau_n\ ]\!\!\}$ for a list containing n types.

The type rules contain maybe expressions (me) and maybe types ($\tau^?$). These are analogous to Haskell's datatype *Maybe*, which is either nothing, or just a value (an expression or a type, respectively). For "nothing", we write a dot ($\cdot$) to denote the empty attribute.

6.2 Literals and patterns

We start with the type rules for literals and patterns, shown in Figure 6.1. We consider four types of literals: integers (such as 28), floating-point values (2.75), characters (`'A'`), and string literals (`"hello!"`). The type judgement for a literal l is of the form $\vdash_{\!l}\ l : \tau$, where τ is the intended type of the literal. We do not overload the integer and float literals yet. In Haskell 98, these literals have the types *Num a* $\Rightarrow$ *a* and *Fractional a* $\Rightarrow$ *a*, respectively.

For a pattern p, we use a judgement $\mathcal{E}, \mathcal{T}_{\mathcal{C}} \vdash_{\!p}\ p : \tau$. In addition to the type τ that we assign to p, such a judgement contains a pattern variable environment $\mathcal{E}$, and a constraint tree $\mathcal{T}_{\mathcal{C}}$. We use the pattern variable environment to collect all the variables that are introduced by the pattern. Hence, this environment cannot have multiple assertions for the same pattern variable. Only the type rules for pattern variables (P-Var) and as-patterns (P-As) introduce new assertions: the other type rules only propagate environments of subpatterns. The constraint trees constructed by these two rules contain a receiver (in both cases β°), because constraints concerning pattern variables are spread downwards by future type rules.

Observe that each judgement in a conclusion mentions a fresh type variable. Furthermore, each type rule for a composed pattern combines the constraint trees of the subpatterns, and introduces a new node in the tree. In this way, the shape of the constraint tree matches exactly with the shape of the abstract syntax tree of the pattern.

In the type rules for pattern constructor application (P-Con) and infix pattern constructors (P-Infix), we create an instantiation constraint to instantiate the type of the constructor. We can assume to have the type schemes of all constructors before we start to infer types. These type schemes are closed.

Finally, we take a closer look at the type rule (P-List) for list patterns. Here, the types of the patterns in the list must all be the same. To achieve this, a fresh type variable is introduced (β_1), and each type must be equal to this new type variable. This gives us a list of constraints. Each constraint is added to the constraint tree that corresponds to the constrained type. A second fresh type variable (β_2) is introduced for the type of the pattern list, and this type variable is constrained by c.

$\boxed{\vdash_l l : \tau}$ *Literal*

$$\frac{}{\vdash_l \text{Integer} : \mathit{Int}}\ (\text{L-Int}) \qquad \frac{}{\vdash_l \text{Float} : \mathit{Float}}\ (\text{L-Float})$$

$$\frac{}{\vdash_l \text{Char} : \mathit{Char}}\ (\text{L-Char}) \qquad \frac{}{\vdash_l \text{String} : \mathit{String}}\ (\text{L-String})$$

$\boxed{\mathcal{E}, \mathcal{T}_{\mathcal{C}} \vdash_p p : \tau}$ *Pattern*

$$\frac{}{[x : \beta], \beta^{\circ} \vdash_p x : \beta}\ (\text{P-Var}) \qquad \frac{}{\emptyset, \bullet \vdash_p \_ : \beta}\ (\text{P-WC})$$

$$\frac{\vdash_l l : \tau}{\emptyset, [\beta \equiv \tau]^{\bullet} \vdash_p l : \beta}\ (\text{P-Lit}) \qquad \frac{\begin{array}{c}\mathcal{C} = [\tau \equiv \mathit{Int}, \beta \equiv \mathit{Int}] \\ \vdash_l l : \tau\end{array}}{\emptyset, \mathcal{C}^{\bullet} \vdash_p -l : \beta}\ (\text{P-Neg})$$

$$\frac{\mathcal{E}, \mathcal{T}_{\mathcal{C}} \vdash_p p : \tau}{[x : \beta] +\!\!+\ \mathcal{E}, (\beta \equiv \tau) \rhd \{\!\{ \beta^{\circ}, \mathcal{T}_{\mathcal{C}} \}\!\} \vdash_p x@p : \beta}\ (\text{P-As})$$

$$\frac{\begin{array}{c} c_1 = (\beta_1 \preceq \sigma) \qquad c_2 = (\beta_1 \equiv \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \beta_2) \\ \vdash_0 C : \sigma \qquad \mathcal{E}_i, \mathcal{T}_{\mathcal{C}i} \vdash_p p_i : \tau_i \quad 1 \leqslant i \leqslant n,\ n \geqslant 0 \end{array}}{\biguplus_{i=1}^{n} \mathcal{E}_i, c_2 \rhd \{\!\{ [c_1]^{\bullet}, \mathcal{T}_{\mathcal{C}1}, \ldots, \mathcal{T}_{\mathcal{C}n} \}\!\} \vdash_p C\ p_1\ \ldots\ p_n : \beta_2}\ (\text{P-Con})$$

$$\frac{\begin{array}{c} c_1 = (\beta_1 \preceq \sigma) \qquad c_2 = (\beta_1 \equiv \tau_1 \rightarrow \tau_2 \rightarrow \beta_2) \\ \mathcal{E}_1, \mathcal{T}_{\mathcal{C}1} \vdash_p p_1 : \tau_1 \qquad \vdash_0 \bigotimes : \sigma \qquad \mathcal{E}_2, \mathcal{T}_{\mathcal{C}2} \vdash_p p_2 : \tau_2 \end{array}}{\mathcal{E}_1 +\!\!+\ \mathcal{E}_2, c_2 \rhd \{\!\{ \mathcal{T}_{\mathcal{C}1}, [c_1]^{\bullet}, \mathcal{T}_{\mathcal{C}2} \}\!\} \vdash_p p_1 \bigotimes p_2 : \beta_2}\ (\text{P-Infix})$$

$$\frac{\begin{array}{c} c = (\beta_2 \equiv [\beta_1]) \\ c_i = (\beta_1 \equiv \tau_i) \qquad \mathcal{E}_i, \mathcal{T}_{\mathcal{C}i} \vdash_p p_i : \tau_i \qquad 1 \leqslant i \leqslant n,\ n \geqslant 0 \end{array}}{\biguplus_{i=1}^{n} \mathcal{E}_i, c \rhd \{\!\{ c_1 \lhd \mathcal{T}_{\mathcal{C}1}, \ldots, c_n \lhd \mathcal{T}_{\mathcal{C}n} \}\!\} \vdash_p [p_1, \ldots, p_n] : \beta_2}\ (\text{P-List})$$

$$\frac{\begin{array}{c} c = (\beta \equiv (\tau_1, \ldots, \tau_n)) \\ \mathcal{E}_i, \mathcal{T}_{\mathcal{C}i} \vdash_p p_i : \tau_i \qquad 1 \leqslant i \leqslant n,\ n \geqslant 0 \end{array}}{\biguplus_{i=1}^{n} \mathcal{E}_i, c \rhd \{\!\{ \mathcal{T}_{\mathcal{C}1}, \ldots, \mathcal{T}_{\mathcal{C}n} \}\!\} \vdash_p (p_1, \ldots, p_n) : \beta}\ (\text{P-Tuple})$$

Figure 6.1. Type rules for literals and patterns

6.3 Expressions: part one

We continue with the type rules for expressions, listed in Figure 6.2. For the time being, we skip let expressions, infix applications and sections, case expressions, enumerations, do expressions, and list comprehensions. These expressions all require new forms of judgement, and we will cover them later (most are presented in Section 6.7). The type rules presented here make use of judgements for literals (E-Lit) and patterns (E-Abs).

To type an expression e, we introduce judgements of the form $\langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T}_{\mathcal{C}} \vdash_e e : \tau$. The first component of such a judgement is the set of monomorphic types $\mathcal{M}$. This set is supplied by the context of the expression: it is an inherited attribute. The type rules leave $\mathcal{M}$ unchanged, except for the lambda abstraction type rule (E-Abs). In this type rule, the set of monomorphic types passed to the body of the lambda abstraction is extended with the type variables that are introduced by the patterns. This set of monomorphic type variables equals $ran(\mathcal{E})$, where $\mathcal{E}$ is the combined pattern variable environment of all the patterns of the lambda abstraction. Some of the identifiers in $\mathcal{A}$ are bound by the lambda patterns, and the type variables assigned to these identifiers are equated with the corresponding type variable in the pattern variable environment, expressed by $\mathcal{E} \equiv \mathcal{A}$. Thus, these type variables from $\mathcal{A}$ are also monomorphic. Therefore, we extend $\mathcal{M}$ with the free type variables in the constraint set $\mathcal{C}_\ell$. These extra type variables have to end up in $\mathcal{M}$ as well if we do not spread type constraints, since we do not know whether the constraints of $\mathcal{C}_\ell$ will be considered early enough. The constraints of $\mathcal{C}_\ell$ are spread in the constraint tree constructed for a lambda abstraction, and the assertions from $\mathcal{A}$ that are used to create $\mathcal{C}_\ell$ are removed from the assumption set.

The type rule for function application (E-Apply) differs from the type rules for application in most type systems, since it considers multiple arguments. An alternative is to use the following type rule for binary applications instead.

$$\frac{c = (\tau_1 \equiv \tau_2 \to \beta) \qquad \langle\mathcal{M}\rangle, \mathcal{A}_1, \mathcal{T}_{\mathcal{C}1} \vdash_e e_1 : \tau_1 \qquad \langle\mathcal{M}\rangle, \mathcal{A}_2, \mathcal{T}_{\mathcal{C}2} \vdash_e e_2 : \tau_2}{\langle\mathcal{M}\rangle, \mathcal{A}_1 +\!\!+ \mathcal{A}_2, c \rhd \bullet\!\![\, \mathcal{T}_{\mathcal{C}1}, \mathcal{T}_{\mathcal{C}2} \,]\!\!\bullet \vdash_e e_1\ e_2 : \beta} \text{ (E-BinApp)}$$

The constraints created by the two type rules for application are equivalent. However, dealing with multiple arguments at once (instead of viewing these as nested applications) has the advantage that the relation between the type of the function and the type of all its arguments is captured at one single tree node, and that it is not scattered through numerous constraints. We will profit from this fact when we define heuristics for improving type error messages involving applications, which work directly on the constraints created by the type rule (E-Apply).

The last type rule we discuss in this section is for expressions with a type annotation (E-Typed). These type annotations are closed type schemes. The type of the annotated expression must be at least as general as the annotation. We express this as a skolemization constraint (c_1), and we record the monomorphic types at this point in the constraint. The type assigned to an annotated expression

$\boxed{\langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T}_{\mathcal{C}} \vdash_e e : \tau}$ *Expression*

$$\frac{}{\langle\mathcal{M}\rangle, [x{:}\beta], \beta^{\circ} \vdash_e x : \beta}\ (\text{E-Var}) \qquad \frac{\vdash_0 C : \sigma}{\langle\mathcal{M}\rangle, \emptyset, [\beta \preceq \sigma]^{\bullet} \vdash_e C : \beta}\ (\text{E-Con})$$

$$\frac{\vdash_l l : \tau}{\langle\mathcal{M}\rangle, \emptyset, [\beta \equiv \tau]^{\bullet} \vdash_e l : \beta}\ (\text{E-Lit}) \qquad \frac{\begin{array}{c} c_1 = (\tau \equiv \mathit{Int}) \qquad c_2 = (\beta \equiv \mathit{Int}) \\ \langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T}_{\mathcal{C}} \vdash_e e : \tau \end{array}}{\langle\mathcal{M}\rangle, \mathcal{A}, c_2 \rhd \{\!\!\bullet\, c_1 \lhd \mathcal{T}_{\mathcal{C}} \,\bullet\!\!\} \vdash_e -e : \beta}\ (\text{E-Neg})$$

$$\frac{\begin{array}{c} c = (\tau \equiv \tau_1 \to \ldots \to \tau_n \to \beta) \\ \langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T}_{\mathcal{C}} \vdash_e e : \tau \\ \langle\mathcal{M}\rangle, \mathcal{A}_i, \mathcal{T}_{\mathcal{C}i} \vdash_e e_i : \tau_i \qquad 1 \leqslant i \leqslant n,\ n \geqslant 1 \end{array}}{\langle\mathcal{M}\rangle, \mathcal{A} + \!\!\!+ \biguplus_{i=1}^{n} \mathcal{A}_i, c \rhd \{\!\!\bullet\, \mathcal{T}_{\mathcal{C}}, \mathcal{T}_{\mathcal{C}1}, \ldots, \mathcal{T}_{\mathcal{C}n} \,\bullet\!\!\} \vdash_e e\ e_1\ \ldots\ e_n : \beta}\ (\text{E-Apply})$$

$$\frac{\begin{array}{c} \mathcal{T}_{\mathcal{C}new} = \{\!\!\bullet\, c_1 \lhd \mathcal{T}_{\mathcal{C}1}, c_2 \lhd \mathcal{T}_{\mathcal{C}2}, c_3 \lhd \mathcal{T}_{\mathcal{C}3} \,\bullet\!\!\} \\ c_1 = (\tau_1 \equiv \mathit{Bool}) \qquad c_2 = (\tau_2 \equiv \beta) \qquad c_3 = (\tau_3 \equiv \beta) \\ \langle\mathcal{M}\rangle, \mathcal{A}_1, \mathcal{T}_{\mathcal{C}1} \vdash_e e_1 : \tau_1 \\ \langle\mathcal{M}\rangle, \mathcal{A}_2, \mathcal{T}_{\mathcal{C}2} \vdash_e e_2 : \tau_2 \qquad \langle\mathcal{M}\rangle, \mathcal{A}_3, \mathcal{T}_{\mathcal{C}3} \vdash_e e_3 : \tau_3 \end{array}}{\langle\mathcal{M}\rangle, \mathcal{A}_1 + \!\!\!+ \mathcal{A}_2 + \!\!\!+ \mathcal{A}_3, \mathcal{T}_{\mathcal{C}new} \vdash_e \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \beta}\ (\text{E-Cond})$$

$$\frac{\begin{array}{c} \mathcal{C}_\ell = (\mathcal{E} \equiv \mathcal{A}) \qquad c = (\beta \equiv \tau_1 \to \ldots \to \tau_n \to \tau) \\ \mathcal{E} = (\biguplus_{i=1}^{n} \mathcal{E}_i) \qquad \mathcal{E}_i, \mathcal{T}_{\mathcal{C}i} \vdash_p p_i : \tau_i \qquad 1 \leqslant i \leqslant n,\ n \geqslant 1 \\ \langle\mathcal{M} + \!\!\!+ \mathit{ftv}(\mathcal{C}_\ell)\rangle, \mathcal{A}, \mathcal{T}_{\mathcal{C}} \vdash_e e : \tau \end{array}}{\langle\mathcal{M}\rangle, \mathcal{A} \backslash \mathit{dom}(\mathcal{E}), c \rhd \mathcal{C}_\ell \,\underline{\rhd\!\circ}\, \{\!\!\bullet\, \mathcal{T}_{\mathcal{C}1}, \ldots, \mathcal{T}_{\mathcal{C}n}, \mathcal{T}_{\mathcal{C}} \,\bullet\!\!\} \vdash_e \lambda p_1\ \ldots\ p_n \to e : \beta}\ (\text{E-Abs})$$

$$\frac{\begin{array}{c} c = (\beta_2 \equiv [\beta_1]) \\ c_i = (\beta_1 \equiv \tau_i) \qquad \langle\mathcal{M}\rangle, \mathcal{A}_i, \mathcal{T}_{\mathcal{C}i} \vdash_e e_i : \tau_i \qquad 1 \leqslant i \leqslant n,\ n \geqslant 0 \end{array}}{\langle\mathcal{M}\rangle, \biguplus_{i=1}^{n} \mathcal{A}_i, c \rhd \{\!\!\bullet\, c_1 \lhd \mathcal{T}_{\mathcal{C}1}, \ldots, c_n \lhd \mathcal{T}_{\mathcal{C}n} \,\bullet\!\!\} \vdash_e [e_1, \ldots, e_n] : \beta_2}\ (\text{E-List})$$

$$\frac{\begin{array}{c} c = (\beta \equiv (\tau_1, \ldots, \tau_n)) \\ \langle\mathcal{M}\rangle, \mathcal{A}_i, \mathcal{T}_{\mathcal{C}i} \vdash_e e_i : \tau_i \qquad 1 \leqslant i \leqslant n,\ n \geqslant 0 \end{array}}{\langle\mathcal{M}\rangle, \biguplus_{i=1}^{n} \mathcal{A}_i, c \rhd \{\!\!\bullet\, \mathcal{T}_{\mathcal{C}1}, \ldots, \mathcal{T}_{\mathcal{C}n} \,\bullet\!\!\} \vdash_e (e_1, \ldots, e_n) : \beta}\ (\text{E-Tuple})$$

$$\frac{c_1 = (\tau \succeq_{\mathcal{M}} \sigma) \qquad c_2 = (\beta \preceq \sigma) \qquad \langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T}_{\mathcal{C}} \vdash_e e : \tau}{\langle\mathcal{M}\rangle, \mathcal{A}, c_2 \rhd \{\!\!\bullet\, c_1 \lhd \mathcal{T}_{\mathcal{C}} \,\bullet\!\!\} \vdash_e e :: \sigma : \beta}\ (\text{E-Typed})$$

Figure 6.2. Type rules for expressions (partial)

is an instantiated version of the type scheme, which we formulate as an instantiation constraint (c_2). The following example illustrates the use of type annotations.

Example 6.1. Consider the definition of f, f's type signature, and the type annotation for the expression $\lambda x \to x$.

$$\begin{array}{l} f :: (\mathit{Int} \to \mathit{Int}) \to \mathit{Int} \to \mathit{Int} \\ f = (\lambda x \to x) :: (a \to a) \to a \to a \end{array}$$

We claim that this definition is well-typed by showing that the constraints created by (E-Typed) for the type annotation can be satisfied. The type annotation (σ) is $\forall a.(a \to a) \to a \to a$. Assume that the type variable v_0 is introduced for the pattern variable x. Then the annotated expression is given the type $v_0 \to v_0$, and v_0 does not appear in $\mathcal{M}$. Type constraint $c_1 = (\tau := \textsc{Skol}(\mathcal{M}, \sigma))$ holds since

$$\forall a.(a \to a) \to a \to a \quad < \quad \mathit{generalize}(\mathcal{M}, v_0 \to v_0).$$

To see that constraint c_2 holds too, we have to consider the fresh type variable β introduced by (E-Typed). This type variable is restricted to be $(\mathit{Int} \to \mathit{Int}) \to \mathit{Int} \to \mathit{Int}$ because of f's type signature. Hence, c_2 holds because

$$(\mathit{Int} \to \mathit{Int}) \to \mathit{Int} \to \mathit{Int} \quad < \quad \forall a.(a \to a) \to a \to a.$$

6.4 Left-hand sides and right-hand sides

The next step is to present typing judgements for left-hand sides and right-hand sides of function declarations. We discuss how to type declarations in the next section, for which we use the judgements of this section. The type rules are presented in Figure 6.3. We first discuss the left-hand sides.

Basically, the left-hand side of a declaration consists of two parts: the function or operator that is being defined, and a number of patterns (at least one), which are the arguments of the function or operator. Three alternatives exist to specify a left-hand side: they are presented in the following table.

left-hand side	*alternative*	*defines*	*arguments*
$f\ x\ y\ z$	function	f	$[x, y, z]$
$x \oplus y$	infix	$\oplus$	$[x, y]$
$(x \oplus y)\ z$	parenthesized + infix	$\oplus$	$[x, y, z]$

We use judgements of the form $x, \mathcal{E}, \mathcal{T}_{\mathcal{C}} \vdash_{\mathit{lhs}} \mathit{lhs} : \{\!|\ \tau_1, \ldots, \tau_n\ |\!\}$ for left-hand sides *lhs*, where x is the identifier which is being defined, and $\{\!|\ \tau_1, \ldots, \tau_n\ |\!\}$ is a list of types for the arguments. Environment $\mathcal{E}$ contains all pattern variables with their assigned type variables appearing in the arguments.

Let us look at the type rule (LHS-Par) for a parenthesized left-hand side (lhs) $p_1\ \ldots\ p_n$. The identifier defined by such a left-hand side is the identifier defined by *lhs*. The list of types consists of the types we find for *lhs*, followed by the types assigned to the n patterns.

$\boxed{x, \mathcal{E}, \mathcal{T}_{\mathcal{C}} \vdash_{lhs} lhs : \{\!|\tau, \ldots, \tau|\!\}}$ *Left-hand side*

$$\frac{\mathcal{E}_i, \mathcal{T}_{\mathcal{C}\,i} \vdash_p p_i : \tau_i \qquad 1 \leqslant i \leqslant n,\ n \geqslant 1}{x, \biguplus_{i=1}^{n} \mathcal{E}_i, \Lbag \mathcal{T}_{\mathcal{C}\,1}, \ldots, \mathcal{T}_{\mathcal{C}\,n} \Rbag \vdash_{lhs} x\ p_1\ \ldots\ p_n : \{\!|\tau_1, \ldots, \tau_n|\!\}} \ (\text{LHS-Fun})$$

$$\frac{\mathcal{E}_1, \mathcal{T}_{\mathcal{C}\,1} \vdash_p p_1 : \tau_1 \qquad \mathcal{E}_2, \mathcal{T}_{\mathcal{C}\,2} \vdash_p p_2 : \tau_2}{\otimes, \mathcal{E}_1 + \!\!+ \mathcal{E}_2, \Lbag \mathcal{T}_{\mathcal{C}\,1}, \mathcal{T}_{\mathcal{C}\,2} \Rbag \vdash_{lhs} p_1 \otimes p_2 : \{\!|\tau_1, \tau_2|\!\}} \ (\text{LHS-Infix})$$

$$\frac{\begin{array}{c} \mathcal{E}_{new} = (\mathcal{E} + \!\!+ \biguplus_{i=1}^{n} \mathcal{E}_i) \qquad \mathcal{T}_{\mathcal{C}\,new} = \Lbag \mathcal{T}_{\mathcal{C}}, \mathcal{T}_{\mathcal{C}\,1}, \ldots, \mathcal{T}_{\mathcal{C}\,n} \Rbag \\ x, \mathcal{E}, \mathcal{T}_{\mathcal{C}} \vdash_{lhs} lhs : \{\!|\tau'_1, \ldots, \tau'_m|\!\} \\ \mathcal{E}_i, \mathcal{T}_{\mathcal{C}\,i} \vdash_p p_i : \tau_i \qquad 1 \leqslant i \leqslant n,\ n \geqslant 1 \end{array}}{x, \mathcal{E}_{new}, \mathcal{T}_{\mathcal{C}\,new} \vdash_{lhs} (lhs)\ p_1\ \ldots\ p_n : \{\!|\tau'_1, \ldots, \tau'_m, \tau_1, \ldots, \tau_n|\!\}} \ (\text{LHS-Par})$$

$\boxed{\langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T}_{\mathcal{C}} \vdash_{rhs} rhs : \tau}$ *Right-hand side*

$$\frac{\begin{array}{c} (\mathcal{M}', \mathcal{A}', \mathcal{T}_{\mathcal{C}}{}') = bga(\mathcal{M}, [\mathcal{B}_{new}, \mathcal{B}_1, \ldots, \mathcal{B}_n], \biguplus_{i=1}^{n} \Sigma_i) \\ \langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T}_{\mathcal{C}} \vdash_e e : \tau \qquad \mathcal{B}_{new} = (\emptyset, \mathcal{A}, \mathcal{T}_{\mathcal{C}}) \\ \langle\mathcal{M}'\rangle, \mathcal{B}_i, \Sigma_i \vdash_d d_i \qquad 1 \leqslant i \leqslant n,\ n \geqslant 0 \end{array}}{\langle\mathcal{M}\rangle, \mathcal{A}', \mathcal{T}_{\mathcal{C}}{}' \vdash_{rhs} e\ \mathbf{where}\ d_1; \ldots; d_n : \tau} \ (\text{RHS-Simple})$$

$$\frac{\begin{array}{c} (\mathcal{M}', \mathcal{A}', \mathcal{T}_{\mathcal{C}}{}') = bga(\mathcal{M}, [\mathcal{B}_{new}, \mathcal{B}_1, \ldots, \mathcal{B}_m], \biguplus_{j=1}^{m} \Sigma_j) \\ \mathcal{B}_{new} = (\emptyset, \biguplus_{i=1}^{n} \mathcal{A}_i, \Lbag c_1 \lhd \mathcal{T}_{\mathcal{C}\,1}, \ldots, c_n \lhd \mathcal{T}_{\mathcal{C}\,n} \Rbag) \\ \left(\begin{array}{c} c_i = (\beta \equiv \tau_i) \\ \langle\mathcal{M}\rangle, \mathcal{A}_i, \mathcal{T}_{\mathcal{C}\,i} \vdash_{ge} ge_i : \tau_i \end{array} \right) 1 \leqslant i \leqslant n,\ n \geqslant 1 \\ \langle\mathcal{M}'\rangle, \mathcal{B}_j, \Sigma_j \vdash_d d_j \qquad 1 \leqslant j \leqslant m,\ m \geqslant 0 \end{array}}{\langle\mathcal{M}\rangle, \mathcal{A}', \mathcal{T}_{\mathcal{C}}{}' \vdash_{rhs} ge_1; \ldots; ge_n\ \mathbf{where}\ d_1; \ldots; d_m : \beta} \ (\text{RHS-Guarded})$$

$\boxed{\langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T}_{\mathcal{C}} \vdash_{ge} ge : \tau}$ *Guarded expression*

$$\frac{\begin{array}{c} c = (\tau_1 \equiv Bool) \\ \langle\mathcal{M}\rangle, \mathcal{A}_1, \mathcal{T}_{\mathcal{C}\,1} \vdash_e e_1 : \tau_1 \qquad \langle\mathcal{M}\rangle, \mathcal{A}_2, \mathcal{T}_{\mathcal{C}\,2} \vdash_e e_2 : \tau_2 \end{array}}{\langle\mathcal{M}\rangle, \mathcal{A}_1 + \!\!+ \mathcal{A}_2, \Lbag c \lhd \mathcal{T}_{\mathcal{C}\,1}, \mathcal{T}_{\mathcal{C}\,2} \Rbag \vdash_{ge} \mathbf{guard}\ e_1 = e_2 : \tau_2} \ (\text{GExpr})$$

Figure 6.3. Type rules for left-hand sides and right-hand sides

We proceed with the type rules for right-hand sides, which are also shown in Figure 6.3. A simple right-hand side contains an expression, and some local declarations defined in a where clause. Following the syntax of Haskell, the keyword where can be omitted if there are no local declarations. A guarded right-hand side contains a list of guarded expressions in combination with a where clause. A guarded expression is a pair of expressions, with the requirement that the first of these (the guard) is of type *Bool*.[1] We introduce typing judgements for right-hand sides and for guarded expressions: both have the same components as the typing judgement for expressions.

In the type rule (RHS-Guarded), we have a type for each guarded expression. These types must all be equivalent. Hence, we introduce a fresh type variable β, and create an equality constraint between β and each of the types. Observe how each of these constraints is added to the constraint tree in upward direction.

A polymorphic type (scheme) is assigned to the local definitions of a where clause, and these definitions can be used in the expression or guarded expressions. Moreover, these local definitions can be defined in terms of each other. We perform a binding group analysis to take care of this part. The function *bga* should be passed a set of monomorphic types ($\mathcal{M}$), a list of binding group triples, and a list of type signatures (Σ) to perform this analysis. A binding group triple consists of a pattern variable environment ($\mathcal{E}$), an assumption environment ($\mathcal{A}$), and a constraint tree ($\mathcal{T}_{\mathcal{C}}$). The function *bga* returns an extended set of monomorphic types, an assumption set with the remaining assertions, and a new constraint tree. In both type rules for a right-hand side, i.e., (RHS-Simple) and (RHS-Guarded), we introduce one new binding group triple $\mathcal{B}_{new}$ containing the assumption set and the constraint tree of the expression, or guarded expressions, respectively. The pattern variable environment component of this triple is empty, which indicates that this new binding group can use the functions defined in the other binding groups, although it does not add new functions that can be used by the others. We postpone presenting the definition of *bga* to Section 6.6.

6.5 Declarations

We use a judgement $\langle\mathcal{M}\rangle, \mathcal{B}, \Sigma \vdash_d d$ to type a declaration d. Such a judgement has a binding group triple $\mathcal{B}$ and a list of declared type signatures Σ, but it does not contain a type. We discuss the type rules for the three types of declarations as displayed in Figure 6.4.

A pattern binding (D-Pat) contains a pattern p and a right-hand side *rhs*. The types of p and *rhs* must be the same, which is expressed by constraint c. The binding group triple contains the pattern variable environment of p, the assumptions of *rhs*, and a constraint tree, which is composed of the two constraint subtrees and c.

The type rule for a declared function (D-FB) is probably the most difficult type rule of all. Such a declaration consists of a number of function bindings, each with a left-hand side and a right-hand side. Take a look at the type rule (FB)

[1] The keyword guard in (GExpr) is normally written as a bar.

$\boxed{\langle\mathcal{M}\rangle, \mathcal{B}, \Sigma \vdash_d d}$ *Declaration*

$$\frac{\begin{array}{c} c = (\tau_1 \equiv \tau_2) \qquad \mathcal{B} = (\mathcal{E}, \mathcal{A}, c \rhd \bullet\!\![\, \mathcal{T}_{\mathcal{C}\,1}, \mathcal{T}_{\mathcal{C}\,2} \,]\!\!\bullet) \\ \mathcal{E}, \mathcal{T}_{\mathcal{C}\,1} \vdash_p p : \tau_1 \qquad \langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T}_{\mathcal{C}\,2} \vdash_{rhs} rhs : \tau_2 \end{array}}{\langle\mathcal{M}\rangle, \mathcal{B}, \emptyset \vdash_d p = rhs} \ \text{(D-P}{\small\text{AT}}\text{)}$$

$$\frac{\begin{array}{c} \mathcal{B} = ([x : \beta], \biguplus_{i=1}^{n} \mathcal{A}_i, c \rhd \bullet\!\![\, \beta^\circ, \mathcal{C}_1 \trianglelefteq \mathcal{T}_{\mathcal{C}\,1}, \ldots, \mathcal{C}_n \trianglelefteq \mathcal{T}_{\mathcal{C}\,n} \,]\!\!\bullet) \\ c = (\beta \equiv \beta_1 \to \ldots \to \beta_m) \\ \left(\begin{array}{c} \mathcal{C}_i = [\beta_1 \equiv \tau_{(1,i)}, \ldots, \beta_m \equiv \tau_{(m,i)}] \\ \langle\mathcal{M}\rangle, x, \mathcal{A}_i, \mathcal{T}_{\mathcal{C}\,i} \vdash_{fb} fb_i : \{\!|\tau_{(1,i)}, \ldots, \tau_{(m,i)}|\!\} \end{array} \right) 1 \leqslant i \leqslant n,\ n \geqslant 1 \end{array}}{\langle\mathcal{M}\rangle, \mathcal{B}, \emptyset \vdash_d fb_1; \ldots; fb_n} \ \text{(D-FB)}$$

$$\frac{\mathcal{B} = (\emptyset, \emptyset, \bullet) \qquad \Sigma = [x_1 : \sigma, \ldots, x_n : \sigma] \qquad n \geqslant 1}{\langle\mathcal{M}\rangle, \mathcal{B}, \Sigma \vdash_d x_1, \ldots, x_n :: \sigma} \ \text{(D-T}{\small\text{YPE}}\text{)}$$

$\boxed{\langle\mathcal{M}\rangle, x, \mathcal{A}, \mathcal{T}_{\mathcal{C}} \vdash_{fb} fb : \{\!|\tau, \ldots, \tau|\!\}}$ *Function binding*

$$\frac{\begin{array}{c} \mathcal{C}_\ell = (\mathcal{E} \equiv \mathcal{A}) \\ x, \mathcal{E}, \mathcal{T}_{\mathcal{C}\,1} \vdash_{lhs} lhs : \{\!|\tau_1, \ldots, \tau_n|\!\} \qquad \langle\mathcal{M} +\!\!+ ftv(\mathcal{C}_\ell)\rangle, \mathcal{A}, \mathcal{T}_{\mathcal{C}\,2} \vdash_{rhs} rhs : \tau \end{array}}{\langle\mathcal{M}\rangle, x, \mathcal{A} \backslash dom(\mathcal{E}), \mathcal{C}_\ell \trianglerighteq\!\!\circ \bullet\!\![\, \mathcal{T}_{\mathcal{C}\,1}, \mathcal{T}_{\mathcal{C}\,2} \,]\!\!\bullet \vdash_{fb} lhs = rhs : \{\!|\tau_1, \ldots, \tau_n, \tau|\!\}} \ \text{(FB)}$$

Figure 6.4. Type rules for declarations

for a function binding (Figure 6.4). We introduce typing judgements for a function binding *fb*, which are of the form $\langle\mathcal{M}\rangle, x, \mathcal{A}, \mathcal{T}_{\mathcal{C}} \vdash_{fb} fb : \{\!|\ \tau_1, \ldots, \tau_n\ |\!\}$. The last component of this judgement is a list of types, which are collected for the formal parameters of the left-hand side, plus the type of the right-hand side. Observe that the variables in the patterns on the left-hand side bind free variables on the right-hand side. Hence, we create labeled equality constraints for the environment of the left-hand side ($\mathcal{E}$) and the assumption set of the right-hand side ($\mathcal{A}$). The set of monomorphic types is extended with the type variables occurring in these constraints.

We return to the type rule (D-FB). Assume that we have n function bindings ($n \geqslant 1$), and that each of these bindings has a type list of m elements. (All the function bindings must have the same number of arguments. Thus, we know that all type lists are of the same length.) Suppose that the left-hand sides define a function f, which has $m - 1$ patterns, and suppose that the right-hand sides are all simple (unguarded) expressions. Then we have the following situation.

$$\begin{array}{cccccc} & \beta_1 & \cdots & \beta_{(m-1)} & & \beta_m \\ f & p_{(1,1)} & \cdots & p_{(m-1,1)} & = & e_1 \\ & \vdots & & \vdots & & \vdots \\ f & p_{(1,n)} & \cdots & p_{(m-1,n)} & = & e_n \end{array}$$

We introduce m fresh type variables $(\beta_1 \ldots \beta_m)$, and use each to unify the types in the corresponding column. This approach gives us a matrix of equality constraints. Because the constraints should be attached to the constraint tree of their corresponding function binding and added in upward direction, we put the constraints of each row in a constraint set. This gives us n constraint sets $[\mathcal{C}_1, \ldots, \mathcal{C}_n]$ for the n function bindings. We introduce another fresh type variable (β), which represents the type of the function being defined. This type is a function type composed of the m fresh type variables introduced for the columns, as is formulated by constraint c. All the created constraints are assembled into a new constraint tree, which is part of the binding group triple $\mathcal{B}$ in the conclusion.

We could have devised simpler type rules to replace (D-FB) and (FB). A direct approach would be to assign a function type to each function binding, instead of creating a list of types, and constrain these function types to be equivalent. However, our approach is much more general, because we know for each equality constraint in the matrix the very specific reason why it was constructed. Hence, this approach is more suitable for creating precise error messages. Column-wise unification is also part of Yang's algorithm $\mathcal{H}$ [68], where this unification technique is called the two-dimensional pattern relation.

The third alternative for a declaration is the type rule for explicit type signatures (D-TYPE). Such a type signature ends up in the Σ component of the judgement. Furthermore, we return an empty binding group triple.

6.6 Binding group analysis

This section discusses the algorithm shown in Figure 6.5 to perform a binding group analysis (*bga*), which splits up a set of declarations into smaller sets. Input of the algorithm is a set of monomorphic types, a list of binding group triples, and a list of type signatures. The algorithm proceeds in two steps. First, the triples are organized into the actual binding groups. Then, these groups are processed, which results in an extended set of monomorphic types, a new assumption set, and a constraint tree. This constraint tree combines the subtrees in the triples and the newly created constraints.

In the first step, we combine the binding group triples of definitions that are mutually recursive. We do this by inspecting the defined variables (in the $\mathcal{E}$ component of the triple) and the used variables (in the $\mathcal{A}$ component of the triple). We ignore pattern variables for which we have a type signature, i.e., which are in $dom(\Sigma)$. Combining two binding group triples works as follows.

```
$bga\ (\mathcal{M}, [\mathcal{B}_1, \ldots, \mathcal{B}_n], \Sigma) = \mathcal{B}'_1 \ \oplus\ (\ldots\ \oplus\ (\mathcal{B}'_m\ \oplus\ initial))$
  where
    $[\mathcal{B}'_1, \ldots, \mathcal{B}'_m] = organizeTriples\ ([\mathcal{B}_1, \ldots, \mathcal{B}_n], \Sigma)$

    $initial = (\mathcal{M}, \emptyset, \bullet)$

    $(\mathcal{E}, \mathcal{A}_1, \mathcal{T}_{\mathcal{C}1})\ \oplus\ (\mathcal{M}_2, \mathcal{A}_2, \mathcal{T}_{\mathcal{C}2}) =$
      let
        $\mathcal{C}_{\ell 1} = \mathcal{A}_1 \preceq \Sigma;$                  $\mathcal{A}'_1\ = \mathcal{A}_1 \backslash dom(\Sigma)$
        $\mathcal{C}_{\ell 2} = \mathcal{E} \succeq_{\mathcal{M}} \Sigma;$                 $\mathcal{E}'\ \ = \mathcal{E} \backslash dom(\Sigma)$
        $\mathcal{C}_{\ell 3} = \mathcal{A}'_1 \equiv \mathcal{E}';$                  $\mathcal{A}''_1 = \mathcal{A}'_1 \backslash dom(\mathcal{E}')$

        $implicits = zip\ (dom(\mathcal{E}'))\ [\sigma_{v1}, \sigma_{v2}, \ldots]$   -- fresh type scheme vars

        $\mathcal{C}_4\ \ = [\ \sigma_v := \textsc{Gen}(\mathcal{M}, \tau) \mid (x, \sigma_v) \in implicits, (x, \tau) \in \mathcal{E}'\ ]$
        $\mathcal{C}_{\ell 5} = \mathcal{A}_2 \preceq implicits;$             $\mathcal{A}'_2 = \mathcal{A}_2 \backslash dom(\mathcal{E}')$
      in
        (   $\mathcal{M}_2 + \!\! + ftv(\mathcal{C}_{\ell 3})$
        ,   $\mathcal{A}''_1 + \!\! + \mathcal{A}'_2$
        ,   $(\mathcal{C}_{\ell 1}\ \underset{\circ}{\lll}\ \mathcal{C}_{\ell 2} \underset{\circ}{\rhd\!\lhd} \mathcal{C}_{\ell 3} \underset{\circ}{\rhd\!\lhd} \mathcal{T}_{\mathcal{C}1}) \ll \mathcal{C}_4{}^{\bullet} \ll (\mathcal{C}_{\ell 5}\ \underset{\circ}{\lll}\ \mathcal{T}_{\mathcal{C}2})$
        )
```

Figure 6.5. Binding group analysis

$$(\mathcal{E}_1, \mathcal{A}_1, \mathcal{T}_{\mathcal{C}1}) \uplus (\mathcal{E}_2, \mathcal{A}_2, \mathcal{T}_{\mathcal{C}2}) = (\mathcal{E}_1 + \!\! + \mathcal{E}_2, \mathcal{A}_1 + \!\! + \mathcal{A}_2, \{\!\!\{\, \mathcal{T}_{\mathcal{C}1}, \mathcal{T}_{\mathcal{C}2}\, \}\!\!\})$$

The list of combined triples is then ordered such that a triple that defines a variable for which we do not have a type signature is put before the triples using that particular variable. For this, we use a topological sort algorithm. We assume that the function *organizeTriples* takes care of this, thereby using $\uplus$ to combine triples.

Example 6.2. Consider the following five binding group triples, and the type signature environment Σ.

$$
\begin{aligned}
\mathcal{B}_1 &= (\ \emptyset\ ,\ [x : v_0]\ ,\ \mathcal{T}_{\mathcal{C}1}\ ) \\
\mathcal{B}_2 &= (\ [f : v_1]\ ,\ [y : v_2\ ,\ f : v_3]\ ,\ \mathcal{T}_{\mathcal{C}2}\ ) \\
\mathcal{B}_3 &= (\ [x : v_4]\ ,\ [g : v_5\ ,\ x : v_6\ ,\ y : v_7]\ ,\ \mathcal{T}_{\mathcal{C}3}\ ) \\
\mathcal{B}_4 &= (\ [y : v_8]\ ,\ [f : v_9\ ,\ x : v_{10}\ ,\ y : v_{11}\ ,\ z : v_{12}]\ ,\ \mathcal{T}_{\mathcal{C}4}\ ) \\
\mathcal{B}_5 &= (\ [g : v_{13}\ ,\ h : v_{14}]\ ,\ \emptyset\ ,\ \mathcal{T}_{\mathcal{C}5}\ ) \\
\\
\Sigma\ &= [f : \forall a.a \to a]
\end{aligned}
$$

$$\begin{tikzcd}
\mathcal{B}_1 & \mathcal{B}_2 \\
\mathcal{B}_3 \arrow[u, "(x)"] \arrow[r, bend left, "(x)"] & \mathcal{B}_4 \arrow[u, "(y)"'] \arrow[l, bend left, "(y)"] \\
\mathcal{B}_5 \arrow[u, "(g)"'] &
\end{tikzcd}$$

The dependencies between the five triples are shown in the figure on the right. An arrow goes from a triple which defines a variable to another triple that uses that variable. Dependencies from $\mathcal{B}_2$ labeled with f are absent, since f appears in Σ. We combine $\mathcal{B}_3$ and $\mathcal{B}_4$ since they are mutually recursive. The list $[\mathcal{B}_5, \mathcal{B}_{3+4}, \mathcal{B}_1, \mathcal{B}_2]$ is a valid ordering of the binding groups.

In the second step, we process the list of binding groups from right to left (*foldr*). We start with a triple containing $\mathcal{M}$, an empty assumption set, and an empty constraint tree. Suppose that we want to process a binding group $\mathcal{B} = (\mathcal{E}, \mathcal{A}_1, \mathcal{T}_{\mathcal{C}1})$. Furthermore, assume that we have a triple $(\mathcal{M}_2, \mathcal{A}_2, \mathcal{T}_{\mathcal{C}2})$, which is the current state. To deal with $\mathcal{B}$, we create five constraint sets (see the definition of *bga*).

$\mathcal{C}_{\ell 1}$: We create an instantiation constraint for assumptions in $\mathcal{A}_1$ for which we have a type scheme in Σ.

$\mathcal{C}_{\ell 2}$: A skolemization constraint is created for each variable in $\mathcal{E}$ with a type scheme in Σ. This constraint records the set of monomorphic types passed to *bga*, and it constrains the type of a definition to be more general than its declared type signature.

$\mathcal{C}_{\ell 3}$: We now consider the pattern variables in $\mathcal{E}$ without a type scheme in Σ. This environment $\mathcal{E}'$ contains the pairs for which no constraint was created in step 2. Similarly, we take the assumptions from $\mathcal{A}_1$ without a type scheme (say $\mathcal{A}_1'$), which are the leftovers from step 1. Equality constraints are generated between matching pairs of $\mathcal{E}'$ and $\mathcal{A}_1'$. Note that we create an equality constraint because these matching pairs correspond to (mutually) recursive calls of an implicitly typed definition. These calls are monomorphic with respect to the definition.

$\mathcal{C}_4$: The pattern variables in $\mathcal{E}'$ are implicitly typed, and we associate a fresh type scheme variable with each of these variables. Such a type scheme variable is a place-holder, and will eventually contain the inferred type of its corresponding pattern variable. Let *implicits* be this association list. A generalization constraint is constructed for each pattern variable x in $\mathcal{E}'$. This constraint relates the type of x in $\mathcal{E}'$ with the fresh type scheme variable of x in *implicits*. We generalize with respect to the set $\mathcal{M}$. Note that we do not spread these constraints. Hence, we have a constraint set $\mathcal{C}_4$ rather then $\mathcal{C}_{\ell 4}$.

$\mathcal{C}_{\ell 5}$: In $\mathcal{A}_2$ are assumptions for identifiers from other binding groups that are unbound (so far). For each identifier which is also in *dom*(*implicits*), we introduce an instantiation constraint.

After creating these five constraint sets, we return a new triple. The set of monomorphic types $\mathcal{M}_2$ is extended with the free type variables of $\mathcal{C}_{\ell 3}$. We combine the assumption sets $\mathcal{A}_1'$ and $\mathcal{A}_2$, but remove the assumptions for which a constraint was generated in step 3 or step 5. Furthermore, we build a new constraint tree consisting of $\mathcal{T}_{\mathcal{C}1}$, $\mathcal{T}_{\mathcal{C}2}$, and the five constraint sets. The first three constraint sets are spread downward into $\mathcal{T}_{\mathcal{C}1}$, and we spread $\mathcal{C}_{\ell 5}$ into $\mathcal{T}_{\mathcal{C}2}$.

The composed constraint tree imposes restrictions on the order of the type constraints. The generalization constraints of $\mathcal{C}_4$ have to be considered after all the constraints in $\mathcal{T}_{\mathcal{C}1}$ (including the constraints that are spread into this tree), but before the constraints of $\mathcal{C}_{\ell 5}$. To meet this requirement, a strict ordering is imposed ($\ll$). A more subtle requirement is that the instantiation constraints of $\mathcal{C}_{\ell 1}$ and $\mathcal{C}_{\ell 5}$ should not be considered too late in case we choose not to spread the constraints. For these labeled constraint sets, we use the strict variant ($\underline{\ll_\circ}$) to spread constraints. The next example illustrates why postponing instantiation constraints can be harmful.

Example 6.3. Consider the following expression, where *id* is the identity function from the Prelude, which has type $\forall a.a \to a$.

$$\textbf{let}\ f = id\ \textbf{in}\ f\ f$$

Assume that *id* is assigned the type variable v_0. Then, we have the constraint $v_0 := \textsc{Inst}(\forall a.a \to a)$ at top-level. This constraint has to be considered before we generalize the type of f, and assign this type scheme to the type scheme variable introduced for f. Spreading the instantiation constraint moves the constraint into the let definition, which circumvents the problem.

Example 6.2 (continued). We continue with the five binding groups and Σ presented at page 105. We start our binding group analysis with $\mathcal{B}_2$, which is the last triple in the ordered list. Two new labeled constraints are created using f's type signature in Σ, because f is in the pattern environment and in the assumption set. This gives us the following constraint tree.

$$\begin{aligned}
\mathcal{T}_{\mathcal{C}A} &= (\ell_{v_3}, v_3 := \textsc{Inst}(\forall a.a \to a)) \\
&\quad \lll\!\circ\, (\ell_{v_1}, v_1 := \textsc{Skol}(\mathcal{M}, \forall a.a \to a)) \rhd\!\circ\, \mathcal{T}_{\mathcal{C}2} \\
\mathcal{A}_A &= [\, y : v_2 \,]
\end{aligned}$$

The assumptions in $\mathcal{A}_A$ are the assumptions from $\mathcal{B}_2$ that are not yet dealt with. We continue with $\mathcal{B}_1$. The constraint tree $\mathcal{T}_{\mathcal{C}A}$ is extended with the constraint tree of $\mathcal{B}_2$, but no new constraints are added.

$$\mathcal{T}_{\mathcal{C}B} = \mathcal{T}_{\mathcal{C}1} \ll \mathcal{T}_{\mathcal{C}A} \qquad\qquad \mathcal{A}_B = [\, x : v_0, y : v_2 \,]$$

Next, we consider the combined triples of $\mathcal{B}_3$ and $\mathcal{B}_4$. At this point, various things happen. The type variable v_9 must be an instance of f's type signature, and an equality constraint is generated for all uses of x in $\mathcal{B}_3$ and $\mathcal{B}_4$, and also for y. Because x and y do not have an explicit type signature, we assign two fresh type scheme variables to them, and create two generalization constraints. Finally, the assumptions about x and y in $\mathcal{A}_B$ must be instances of the generalized types of x and y, respectively. Hence, we get the following constraint tree.

$$\begin{aligned}
\mathcal{T}_{\mathcal{C}C} &= ((\ell_{v_9}, v_9 := \textsc{Inst}(\forall a.a \to a)) \lll\!\circ \\
&\qquad\qquad [(\ell_{v_6}, v_4 \equiv v_6), (\ell_{v_{10}}, v_4 \equiv v_{10}), (\ell_{v_7}, v_8 \equiv v_7), (\ell_{v_{11}}, v_8 \equiv v_{11})]\ \underline{\rhd\!\circ} \\
&\qquad\qquad \lbrace\!\bullet\, \mathcal{T}_{\mathcal{C}3}, \mathcal{T}_{\mathcal{C}4} \,\bullet\!\rbrace) \\
&\quad \ll [(\ell_{v_4}, \sigma_0 := \textsc{Gen}(\mathcal{M}, v_4)), (\ell_{v_8}, \sigma_1 := \textsc{Gen}(\mathcal{M}, v_8))]^{\bullet} \\
&\quad \ll ([(\ell_{v_0}, v_0 := \textsc{Inst}(\sigma_0)), (\ell_{v_2}, v_2 := \textsc{Inst}(\sigma_1))]\ \underline{\lll\!\circ}\ \mathcal{T}_{\mathcal{C}B}) \\
\mathcal{A}_C &= [\, g : v_5, z : v_{12} \,]
\end{aligned}$$

Two assumptions from $\mathcal{B}_3$ and $\mathcal{B}_4$ can be found in $\mathcal{A}_C$. The last binding group to deal with is $\mathcal{B}_5$. This group defines g and h, which we assign the type scheme variables σ_2 and σ_3, respectively. We create an instantiation constraint for the assumption about g in $\mathcal{A}_C$. Thus, our final constraint tree is:

$$\begin{aligned}
\mathcal{T}_{\mathcal{C}D} &= \mathcal{T}_{\mathcal{C}5} \ll [(\ell_{v_{13}}, \sigma_2 := \textsc{Gen}(\mathcal{M}, v_{13})), (\ell_{v_{14}}, \sigma_3 := \textsc{Gen}(\mathcal{M}, v_{14}))]^{\bullet} \\
&\qquad\qquad \ll ((\ell_{v_5}, v_5 := \textsc{Inst}(\sigma_2)) \lll\!\circ\, \mathcal{T}_{\mathcal{C}C}) \\
\mathcal{A}_D &= [\, z : v_{12} \,]
\end{aligned}$$

These groups do not handle the assumption about z: this is done elsewhere.

$\boxed{\langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T_C} \vdash_e e : \tau}$ *Expression*

$$\frac{\begin{array}{c}(\mathcal{M}', \mathcal{A}', \mathcal{T_C}') = bga(\mathcal{M}, [\mathcal{B}_{new}, \mathcal{B}_1, \ldots, \mathcal{B}_n], \biguplus_{i=1}^{n} \Sigma_i) \\ \langle\mathcal{M}'\rangle, \mathcal{B}_i, \Sigma_i \vdash_d d_i \qquad 1 \leqslant i \leqslant n,\ n \geqslant 0 \\ \langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T_C} \vdash_e e : \tau \qquad \mathcal{B}_{new} = (\emptyset, \mathcal{A}, \mathcal{T_C}) \qquad c = (\beta \equiv \tau)\end{array}}{\langle\mathcal{M}\rangle, \mathcal{A}', c \rhd \mathcal{T_C}' \vdash_e \textbf{let } d_1; \ldots; d_n \textbf{ in } e : \beta}\ (\text{E-LET})$$

$$\frac{\begin{array}{c}c = (\beta_1 \equiv \tau_1 \to \tau_2 \to \beta_2) \\ \langle\mathcal{M}\rangle, \mathcal{A}_1, \mathcal{T_C}_1 \vdash_e e_1 : \tau_1 \qquad \langle\mathcal{M}\rangle, \mathcal{A}_2, \mathcal{T_C}_2 \vdash_e e_2 : \tau_2\end{array}}{\langle\mathcal{M}\rangle, \mathcal{A}_1 +\!\!+ [\otimes{:}\beta_1] +\!\!+ \mathcal{A}_2, c \rhd \bullet\![\, \mathcal{T_C}_1, \beta_1{}^{\circ}, \mathcal{T_C}_2 \,]\!\bullet \vdash_e e_1 \otimes e_2 : \beta_2}\ (\text{E-INFIX})$$

$$\frac{\begin{array}{c}c_1 = (\beta_1 \equiv \tau \to \beta_2 \to \beta_3) \qquad c_2 = (\beta_4 \equiv \beta_2 \to \beta_3) \\ \langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T_C} \vdash_e e : \tau\end{array}}{\langle\mathcal{M}\rangle, \mathcal{A} +\!\!+ [\otimes{:}\beta_1], c_1 \rhd c_2 \rhd \bullet\![\, \mathcal{T_C}, \beta_1{}^{\circ} \,]\!\bullet \vdash_e (e \otimes \cdot) : \beta_4}\ (\text{E-LSECT})$$

$$\frac{\begin{array}{c}c_1 = (\beta_1 \equiv \beta_2 \to \tau \to \beta_3) \qquad c_2 = (\beta_4 \equiv \beta_2 \to \beta_3) \\ \langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T_C} \vdash_e e : \tau\end{array}}{\langle\mathcal{M}\rangle, [\otimes{:}\beta_1] +\!\!+ \mathcal{A}, c_1 \rhd c_2 \rhd \bullet\![\, \beta_1{}^{\circ}, \mathcal{T_C} \,]\!\bullet \vdash_e (\cdot \otimes e) : \beta_4}\ (\text{E-RSECT})$$

Figure 6.6. Type rules for let expressions and infix applications

6.7 Expressions: part two

In this section, we present the expressions that we skipped in Section 6.3. We discuss the type rules for let expressions, infix applications, case expressions, enumerations, do expressions, and list comprehensions.

Let expressions

The type rule for a let expression is given in Figure 6.6. The binding group triples collected for the declarations, extended with one new triple for the body of the let expression, are used in the binding group analysis. We also pass the collected type signatures to the function *bga*.

Infix applications

The type rules for infix application, presented in Figure 6.6, are similar to the type rule (E-APPLY) for application. In addition to normal infix application (e.g., the expression $x + y$), we consider left sections $(5*)$ and right sections $(/2)$. In all three cases, multiple fresh type variables are introduced: for a section, we even get four new type variables. The type variable β_1 represents the type of the operator in the three rules. The pair $\otimes{:}\beta_1$ is included in the assumption set, and we add a receiver β_1° to the constraint tree. Furthermore, we assign a fresh type variable to the infix

$$\boxed{\langle \mathcal{M} \rangle, \mathcal{A}, \mathcal{T_C} \vdash_e e : \tau} \quad \textit{Expression}$$

$$\frac{\begin{array}{c} \mathcal{T_C}_{new} = \bullet\, c \lhd \mathcal{T_C}, \mathcal{C}_1 \unlhd \mathcal{T_C}_1, \ldots, \mathcal{C}_n \unlhd \mathcal{T_C}_n \,\bullet \\ c = (\beta_1 \equiv \tau) \qquad \langle \mathcal{M} \rangle, \mathcal{A}, \mathcal{T_C} \vdash_e e : \tau \\ \left(\begin{array}{c} \mathcal{C}_i = [\beta_1 \equiv \tau_{(1,i)}, \beta_2 \equiv \tau_{(2,i)}] \\ \langle \mathcal{M} \rangle, \mathcal{A}_i, \mathcal{T_C}_i \vdash_{alt} alt_i : \{\!|\tau_{(1,i)}, \tau_{(2,i)}|\!\} \end{array} \right) \quad 1 \leqslant i \leqslant n,\ n \geqslant 1 \end{array}}{\langle \mathcal{M} \rangle, \mathcal{A} +\!\!+ \biguplus_{i=1}^{n} \mathcal{A}_i, \mathcal{T_C}_{new} \vdash_e \textbf{case}\ e\ \textbf{of}\ alt_1; \ldots; alt_n : \beta_2} \quad (\text{E-Case})$$

$$\boxed{\langle \mathcal{M} \rangle, \mathcal{A}, \mathcal{T_C} \vdash_{alt} alt : \{\!|\tau, \tau|\!\}} \quad \textit{Alternative}$$

$$\frac{\begin{array}{cc} & \mathcal{C}_\ell = (\mathcal{E} \equiv \mathcal{A}) \\ \mathcal{E}, \mathcal{T_C}_1 \vdash_p p : \tau_1 & \langle \mathcal{M} +\!\!+ \mathit{ftv}(\mathcal{C}_\ell) \rangle, \mathcal{A}, \mathcal{T_C}_2 \vdash_{rhs} rhs : \tau_2 \end{array}}{\langle \mathcal{M} \rangle, \mathcal{A} \backslash dom(\mathcal{E}), \mathcal{C}_\ell \underline{\rhd\!\circ} \bullet\, \mathcal{T_C}_1, \mathcal{T_C}_2 \,\bullet \vdash_{alt} p \rightarrow rhs : \{\!|\tau_1, \tau_2|\!\}} \quad (\text{Alt})$$

Figure 6.7. Type rules for case expressions

application, and via type constraints, we relate this type variable with the type of the operator.

The type rule (E-LSect) requires that the operator is binary, conform the Haskell 98 Report [49]. Hugs, on the contrary, weakens this restriction, and allows left sections to have a non-function type. An operator for which both operands are omitted, such as (+), is typed as a variable.

Case expressions

A case expression consists of an expression (called the scrutinee), and a number of alternatives. Each alternative has a pattern and a right-hand side, and we use judgements of the form $\langle \mathcal{M} \rangle, \mathcal{A}, \mathcal{T_C} \vdash_{alt} alt : \{\!|\ \tau_1, \tau_2\ |\!\}$ to type an alternative *alt*. The type list contains exactly two types (the types of the pattern and the right-hand side, respectively). Figure 6.7 contains the type rule (Alt) for alternatives. The pattern variables in p bind the free identifiers in the right-hand side. Hence, we create equality constraints for matching pairs of identifiers with $(\mathcal{E} \equiv \mathcal{A})$, and we extend the set of monomorphic types.

We continue with a discussion on the type rule for a case expression (E-Case). For all alternatives of the case expression, we have two types. We introduce two fresh type variables: β_1 represents the type of all the patterns, and also the type of the scrutinee, and β_2 is used to equate the right-hand sides. The latter type variable is also the type assigned to the complete case expression. All the constraints created for a case expression are added to the constraint tree in upward direction.

$\boxed{\langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T_C} \vdash_e e : \tau}$ *Expression*

$$\frac{\begin{array}{c} \mathcal{T_C}_{new} = (c_4 \rhd \{\!\!\{ c_1 \lhd \mathcal{T_C}_1, c_2 \lhd \mathcal{T_C}_2, c_3 \lhd \mathcal{T_C}_3 \}\!\!\}) \\ c_1 = (\tau_1 \equiv Int) \qquad c_2 = (\tau_2 \equiv Int) \\ c_3 = (\tau_3 \equiv Int) \qquad c_4 = (\beta \equiv [Int]) \\ \langle\mathcal{M}\rangle, \mathcal{A}_1, \mathcal{T_C}_1 \vdash_e e : \tau_1 \\ \langle\mathcal{M}\rangle, \mathcal{A}_2, \mathcal{T_C}_2 \vdash_{me} me_1 : \tau_2 \qquad \langle\mathcal{M}\rangle, \mathcal{A}_3, \mathcal{T_C}_3 \vdash_{me} me_2 : \tau_3 \end{array}}{\langle\mathcal{M}\rangle, \mathcal{A}_1 +\!\!+ \mathcal{A}_2 +\!\!+ \mathcal{A}_3, \mathcal{T_C}_{new} \vdash_e [e, me_1 \mathinner{..} me_2] : \beta} \text{ (E-ENUM)}$$

$\boxed{\langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T_C} \vdash_{me} me : \tau}$ *Maybe expression*

$$\frac{\langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T_C} \vdash_e e : \tau}{\langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T_C} \vdash_{me} e : \tau} \text{ (ME-JUST)} \qquad \frac{}{\langle\mathcal{M}\rangle, \emptyset, \bullet \vdash_{me} \cdot : \beta} \text{ (ME-NOTHING)}$$

Figure 6.8. Type rules for enumerations

Enumerations

In Haskell 98, we can create enumerations for types in the *Enum* type class. The type rule (E-ENUM) for enumerations, which can be found in Figure 6.8, is limited to values of type *Int*. We consider four variants of an enumeration: examples are [1..], [1, 3..], [1..10], and [1, 3..10]. Instead of defining a type rule for each of the variants, we use *maybe expressions*, which are either just an expression, or nothing at all. The type rules for maybe expressions are straightforward: the typing judgements are similar to the judgements for expressions. In (ME-NOTHING), we assign a fresh type variable to the empty expression.[2] In (E-ENUM), the three subexpressions are all restricted to have type *Int*, and the complete expression, which is assigned a fresh type variable, is constrained to have type [*Int*]. The constraint c_2 has no effect if me_1 is empty (similarly for c_3 and me_2).

Do expressions

In our type system, the do expressions are restricted to the *IO* monad. Typing such an expression poses two challenges, which are, in combination, non-trivial to solve.

- The type of a do expression depends on the type of the last statement. This last statement must be an expression, and cannot be, for instance, a generator.
- We have to make sure that occurrences of identifiers are bound correctly. For example, consider the expression **do** $x \leftarrow e_1; e_2$. The pattern variable x can be used in e_2, but not in e_1.

[2] If me_1 in (E-ENUM) is the empty expression, then the comma is omitted as well.

$\boxed{\langle \mathcal{M} \rangle, \mathcal{A}, \mathcal{T}_{\mathcal{C}} \vdash_e e : \tau}$ *Expression*

$$\frac{c = (\beta \equiv IO\ \tau) \qquad \langle \mathcal{M}, \cdot \rangle, \mathcal{A}, \mathcal{T}_{\mathcal{C}} \vdash_{ms} ms : \tau}{\langle \mathcal{M} \rangle, \mathcal{A}, c \rhd \lBrace \mathcal{T}_{\mathcal{C}} \rBrace \vdash_e \mathbf{do}\ ms : \beta}\ (\text{E-Do})$$

$\boxed{\langle \mathcal{M}, \tau^? \rangle, \mathcal{A}, \mathcal{T}_{\mathcal{C}} \vdash_{ms} ms : \tau^?}$ *Sequence of statements*

$$\frac{}{\langle \mathcal{M}, \tau^? \rangle, \emptyset, \bullet \vdash_{ms} \cdot : \tau^?}\ (\text{M-Empty})$$

$$\frac{\begin{array}{c} c = (\tau \equiv IO\ \beta) \\ \langle \mathcal{M} \rangle, \mathcal{A}_1, \mathcal{T}_{\mathcal{C}1} \vdash_e e : \tau \qquad \langle \mathcal{M}, \beta \rangle, \mathcal{A}_2, \mathcal{T}_{\mathcal{C}2} \vdash_{ms} ms : \tau_2^? \end{array}}{\langle \mathcal{M}, \tau_1^? \rangle, \mathcal{A}_1 +\!\!+ \mathcal{A}_2, \lBrace c \lhd \mathcal{T}_{\mathcal{C}1}, \mathcal{T}_{\mathcal{C}2} \rBrace \vdash_{ms} e; ms : \tau_2^?}\ (\text{M-Expr})$$

$$\frac{\begin{array}{c} (\mathcal{M}', \mathcal{A}', \mathcal{T}_{\mathcal{C}}') = bga(\mathcal{M}, [\mathcal{B}_{new}, \mathcal{B}_1, \ldots, \mathcal{B}_n], \biguplus_{i=1}^{n} \Sigma_i) \\ \langle \mathcal{M}' \rangle, \mathcal{B}_i, \Sigma_i \vdash_d d_i \qquad 1 \leqslant i \leqslant n,\ n \geqslant 0 \\ \langle \mathcal{M}, \cdot \rangle, \mathcal{A}, \mathcal{T}_{\mathcal{C}} \vdash_{ms} ms : \tau_2^? \qquad \mathcal{B}_{new} = (\emptyset, \mathcal{A}, \mathcal{T}_{\mathcal{C}}) \end{array}}{\langle \mathcal{M}, \tau_1^? \rangle, \mathcal{A}', \mathcal{T}_{\mathcal{C}}' \vdash_{ms} \mathbf{let}\ d_1; \ldots; d_n; ms : \tau_2^?}\ (\text{M-Let})$$

$$\frac{\begin{array}{c} \mathcal{T}_{\mathcal{C}new} = (c \rhd \mathcal{C}_\ell\ \underline{\rhd\!\circ} \lBrace \mathcal{T}_{\mathcal{C}1}, \mathcal{T}_{\mathcal{C}2}, \mathcal{T}_{\mathcal{C}3} \rBrace) \\ c = (IO\ \tau_1 \equiv \tau_2) \qquad \mathcal{C}_\ell = (\mathcal{E} \equiv \mathcal{A}_3) \\ \mathcal{E}, \mathcal{T}_{\mathcal{C}1} \vdash_p p : \tau_1 \qquad \langle \mathcal{M} \rangle, \mathcal{A}_2, \mathcal{T}_{\mathcal{C}2} \vdash_e e : \tau_2 \\ \langle \mathcal{M} +\!\!+ ftv(\mathcal{C}_\ell), \cdot \rangle, \mathcal{A}_3, \mathcal{T}_{\mathcal{C}3} \vdash_{ms} ms : \tau_2^? \end{array}}{\langle \mathcal{M}, \tau_1^? \rangle, \mathcal{A}_2 +\!\!+ \mathcal{A}_3 \backslash dom(\mathcal{E}), \mathcal{T}_{\mathcal{C}new} \vdash_{ms} p \leftarrow e; ms : \tau_2^?}\ (\text{M-Gen})$$

Figure 6.9. Type rules for do expressions

The difficulty of dealing with these issues depends on how we choose to represent a sequence of statements in the abstract syntax tree. If we encode it as a list directly under the do node, then it is easy to determine the type of the do, but handling the scoping issue becomes complicated. We have chosen a different encoding that simplifies dealing with the scoping issue. We present a type rule for the empty sequence (M-Empty) in Figure 6.9, and three type rules for non-empty sequences, namely, sequences starting with a statement expression (M-Expr), a statement let (M-Let), or a generator (M-Gen). We use one inherited attribute and one synthesized attribute (a *maybe* type) to get the type of the last statement.

Let *ms* be a sequence of statements: we use a judgement $\langle \mathcal{M}, \tau_1^? \rangle, \mathcal{A}, \mathcal{T}_{\mathcal{C}} \vdash_{ms} ms : \tau_2^?$ to type *ms*. In this judgement, $\tau_1^?$ is an inherited maybe type (it is either a type, or it is nothing), and $\tau_2^?$ is a synthesized attribute. Take a look at the type rule for a statement expression (M-Expr). The inherited maybe type ($\tau_1^?$) is simply ignored since the statement associated with this type is not the last statement of the sequence. Instead, we pass the fresh type variable β to the judgement of *ms*. If *ms* is the empty sequence, then the synthesized maybe type ($\tau_2^?$) equals β. To see this, take a look at the type rule for the empty sequence (M-EMPTY). The type of the expression e must be equal to $IO\ \beta$.

The type rule for a monadic let is given by (M-LET). We create a new binding group for the assumptions and the constraint tree of the sequence of statements following the let. We perform a binding group analysis on this binding group, together with the binding groups from the declarations of the let. We use a *no type* ($\cdot$) in the judgements for *ms*, since a let cannot be the last statement of a sequence.

The pattern variables of a generator (M-GEN) bind the free identifiers in *ms* ($\mathcal{A}_3$). The free type variables of the created equality constraints are used to extend the monomorphic set with which we type *ms*. Observe how the constraint c relates the types of the pattern and the expression.

With these four type rules for sequences of statements, we present the type rule (E-DO) for a do expression. We pass a *no type* to the sequence of statements, and we get back a type τ. The do expression is assigned a fresh type variable β, which is constrained to be $IO\ \tau$ by constraint c.

List comprehensions

A list comprehension contains an expression, and a sequence of qualifiers. The type rules for such an expression are given in Figure 6.10. The type rules for a sequence of qualifiers are very similar to the rules for a sequence of statements. One difference is that the pattern variables of the qualifiers bind free identifiers in the expression of the list comprehension. Intuitively, the expression is considered after the qualifiers. We take care of this by passing the expression as an inherited attribute to the sequence of qualifiers, until we reach the end of the sequence.

The typing judgement for a sequence of qualifiers *qs* is $\langle \mathcal{M}, e \rangle, \mathcal{A}, \mathcal{T}_\mathcal{C} \vdash_{qs} qs : \tau$, where e is the inherited expression. Because the type rules for qualifiers and the type rules for statements are so alike, we only discuss the type rules (E-COMPR) and (Q-EMPTY).

In (E-COMPR), expression e is passed to the judgement of the qualifiers. The type τ mentioned in the judgement of *qs* represents the type of one element of the list comprehension. A fresh type variable β is assigned to the list comprehension, which must be equal to the type $[\tau]$. In the type rule for an empty sequence of qualifiers (Q-EMPTY), we deal with the expression from the list comprehension. Observe that the three type rules for non-empty sequences only propagate this expression, until it arrives at the empty sequence. The typing judgement for this expression provides all the information we need for the judgement of the empty sequence.

6.8 Modules

We present one more type rule which is to be applied at the top-level. This type rule is displayed in Figure 6.11, and assumes that we have a type inference environment

$\boxed{\langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T}_\mathcal{C} \vdash_e e : \tau}$ *Expression*

$$\frac{c = (\beta \equiv [\tau]) \qquad \langle\mathcal{M}, e\rangle, \mathcal{A}, \mathcal{T}_\mathcal{C} \vdash_{qs} qs : \tau}{\langle\mathcal{M}\rangle, \mathcal{A}, c \rhd \{\!\!\{ \mathcal{T}_\mathcal{C} \}\!\!\} \vdash_e [\ e \mid qs\ ] : \beta} \ (\textsc{E-Compr})$$

$\boxed{\langle\mathcal{M}, e\rangle, \mathcal{A}, \mathcal{T}_\mathcal{C} \vdash_{qs} qs : \tau}$ *Sequence of qualifiers*

$$\frac{\langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T}_\mathcal{C} \vdash_e e : \tau}{\langle\mathcal{M}, e\rangle, \mathcal{A}, \mathcal{T}_\mathcal{C} \vdash_{qs} \cdot : \tau} \ (\textsc{Q-Empty})$$

$$\frac{\begin{array}{c} c = (\tau_1 \equiv \mathit{Bool}) \\ \langle\mathcal{M}\rangle, \mathcal{A}_1, \mathcal{T}_{\mathcal{C}1} \vdash_e e : \tau_1 \qquad \langle\mathcal{M}, e'\rangle, \mathcal{A}_2, \mathcal{T}_{\mathcal{C}2} \vdash_{qs} qs : \tau_2 \end{array}}{\langle\mathcal{M}, e'\rangle, \mathcal{A}_1 +\!\!+ \mathcal{A}_2, \{\!\!\{ c \lhd \mathcal{T}_{\mathcal{C}1}, \mathcal{T}_{\mathcal{C}2} \}\!\!\} \vdash_{qs} e; qs : \tau_2} \ (\textsc{Q-Guard})$$

$$\frac{\begin{array}{c} (\mathcal{M}', \mathcal{A}', \mathcal{T}_\mathcal{C}') = \mathit{bga}(\mathcal{M}, [\mathcal{B}_{new}, \mathcal{B}_1, \ldots, \mathcal{B}_n], \biguplus_{i=1}^n \Sigma_i) \\ \langle\mathcal{M}'\rangle, \mathcal{B}_i, \Sigma_i \vdash_d d_i \qquad 1 \leqslant i \leqslant n,\ n \geqslant 0 \\ \langle\mathcal{M}, e'\rangle, \mathcal{A}, \mathcal{T}_\mathcal{C} \vdash_{qs} qs : \tau \qquad \mathcal{B}_{new} = (\emptyset, \mathcal{A}, \mathcal{T}_\mathcal{C}) \end{array}}{\langle\mathcal{M}, e'\rangle, \mathcal{A}', \mathcal{T}_\mathcal{C}' \vdash_{qs} \mathbf{let}\ d_1; \ldots; d_n; qs : \tau} \ (\textsc{Q-Let})$$

$$\frac{\begin{array}{c} \mathcal{T}_{\mathcal{C}new} = (c \rhd \mathcal{C}_\ell \mathrel{\underline{\rhd\!\circ}} \{\!\!\{ \mathcal{T}_{\mathcal{C}1}, \mathcal{T}_{\mathcal{C}2}, \mathcal{T}_{\mathcal{C}3} \}\!\!\}) \\ c = ([\tau_1] \equiv \tau_2) \qquad \mathcal{C}_\ell = (\mathcal{E} \equiv \mathcal{A}_3) \\ \mathcal{E}, \mathcal{T}_{\mathcal{C}1} \vdash_p p : \tau_1 \qquad \langle\mathcal{M}\rangle, \mathcal{A}_2, \mathcal{T}_{\mathcal{C}2} \vdash_e e : \tau_2 \\ \langle\mathcal{M} +\!\!+ \mathit{ftv}(\mathcal{C}_\ell), e'\rangle, \mathcal{A}_3, \mathcal{T}_{\mathcal{C}3} \vdash_{qs} qs : \tau_3 \end{array}}{\langle\mathcal{M}, e'\rangle, \mathcal{A}_2 +\!\!+ \mathcal{A}_3 \backslash \mathit{dom}(\mathcal{E}), \mathcal{T}_{\mathcal{C}new} \vdash_{qs} p \leftarrow e; qs : \tau_3} \ (\textsc{Q-Gen})$$

Figure 6.10. Type rules for list comprehensions

$\boxed{\langle\Gamma\rangle, \mathcal{T}_\mathcal{C} \vdash_{module} \mathit{module}}$ *Module*

$$\frac{\begin{array}{c} (\mathcal{M}, \mathcal{A}, \mathcal{T}_\mathcal{C}) = \mathit{bga}(\emptyset, [\mathcal{B}_1, \ldots, \mathcal{B}_n], \biguplus_{i=1}^n \Sigma_i) \qquad \mathcal{C}_\ell = (\mathcal{A} \preceq \Gamma) \\ \langle\mathcal{M}\rangle, \mathcal{B}_i, \Sigma_i \vdash_d d_i \qquad 1 \leqslant i \leqslant n,\ n \geqslant 0 \end{array}}{\langle\Gamma\rangle, \mathcal{C}_\ell \mathrel{\underline{\ll\!\circ}} \mathcal{T}_\mathcal{C} \vdash_{module} d_1; \ldots; d_n} \ (\textsc{Module})$$

Figure 6.11. Type rule for modules

Γ which is known a priori. Typically, this environment contains the types for all the imported functions, as well as the types for all data constructors. This environment is used to create instantiation constraints for the assumptions that are free after the binding group analysis, which is performed for the top-level declarations. This set of instantiation constraints is spread downwards into the constraint tree. We expect $\mathcal{A}\backslash dom(\Gamma)$ to be the empty set. If this is not the case, then there are unbound identifiers.

7

Type graphs

Overview. *A type graph is an advanced data structure for representing substitutions, which also keeps track of reasons for unifications. Type graphs allow us to solve a set of equality constraints in a more global way, thus significantly improving the quality of type error messages. Finally, we discuss how a type graph constraint solver can be combined with a more conventional solver to achieve a good overall performance.*

All inference algorithms for the Hindley-Milner type system rely on the unification of types. This holds for the two standard type inference algorithms $\mathcal{W}$ and $\mathcal{M}$ (Section 2.4), and also for the constraint-based algorithm INFER described in Chapter 4. These algorithms perform several unifications, and combine the (most general) unifiers into a single substitution. Knowledge about type variables is incorporated in this substitution, which is used during the rest of the inference process. However, the history of the type deductions is lost.

A number of proposals to improve type error messages suggest to modify algorithm $\mathcal{W}$, and to store a reason for each type deduction (Section 3.2). Although this idea seems promising, it is a rather ad-hoc extension to an algorithm that was not designed to provide good feedback in the first place. For example, the extended algorithm still proceeds in a predetermined order, which yields a bias in the stored justifications.

Instead of modifying an existing algorithm, we propose an advanced data structure (a type graph) to represent substitutions, in which we can store reasons in a completely unbiased way. A type graph is constructed out of a collection of equality constraints, and has the remarkable property that it is able to represent inconsistent states. Thus, we can also construct a type graph for an inconsistent constraint set. Using type graphs offers two advantages:

- For constructing high quality type error messages, it is crucial to have as much information as possible available. Type graphs store information about each unification in a natural way. In fact, with the right amount of information, type graphs easily generalize many of the approaches discussed in Chapter 3. Type graphs profit from compilers that hold on to valuable information (such as positional information), and that do not desugar the abstract syntax tree prior to type inference.
- Type graphs prevent the introduction of bias, because a set of equality constraints can be solved jointly. This is possible only because type graphs can

represent type inconsistencies. An additional property of this approach is that we can, as it were, inspect the complete program before we have to determine which site to blame. This gives us strictly more information in comparison to more traditional approaches.

Unfortunately, constructing and inspecting a type graph involves additional overhead, which considerably slows down the inference process. Clearly, there is a trade-off between efficiency and quality of the reported error messages, and one has to find the right balance. In a practical setting, we have experienced that the extra time spent on type inference does not hinder programming productivity. Besides, accurate error messages reduce the time programmers have to spend correcting their mistakes.

The type graphs presented in this chapter resemble the path graphs that were proposed by Port [53], and which can be used to find the cause of non-unifiability for a set of equations. However, we follow a more effective approach in dealing with derived equalities (i.e., equalities obtained by decomposing terms, and by taking the transitive closure). Besides, we have a special interest in type inference and type error messages, and formulate special-purpose heuristics.

This chapter is organized as follows. We start with an introduction to type graphs in a simplified setting, and explain the basic concepts (Section 7.1). In Section 7.2, we explain type graphs more fully, and generalize the ideas presented in the first section. Section 7.3 presents an implementation to construct and use a type graph. Finally, Section 7.4 explains how we can compose constraint solvers to create more complex solvers with nice properties.

7.1 Simple type graphs

We start by exploring type graphs for a simple type language in which a type τ is either a type variable or a type constant.

Simple type:

$$\begin{array}{lll} \tau ::= & v_0, v_1, v_2, \ldots & \textit{(type variable)} \\ \quad\ \mid & A, B, C, \ldots & \textit{(type constant)} \end{array}$$

Let us now consider a set of equality constraints between two simple types. Constraints that enforce equality on two type constants are not very interesting: such a constraint holds trivially ($A \equiv A$), or it cannot be satisfied in any way ($A \equiv B$). The other forms of equality constraints, such as $v_0 \equiv A$, $B \equiv v_1$, and $v_0 \equiv v1$, are of more interest to us.

A type graph is constructed from a set of equality constraints. This graph is used to discover combinations of constraints that are inconsistent, and it provides us with a solution (a substitution for the type variables) in case there is no inconsistency. A vertex is created for each type variable, and for each *occurrence* of a type constant. An (undirected) edge is inserted for each equality constraint, which connects the vertices of the two types of the constraint. The connected components that appear in the graph reflect the sets of vertices that should correspond to the same type.

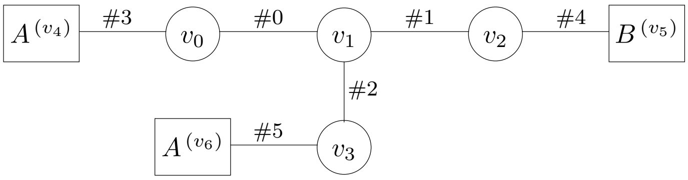


Figure 7.1. A simple type graph

Because each vertex corresponds to either a type variable or a type constant, we know which type variables and type constants should be equivalent. We call such a connected component an *equivalence group.*

Clearly, a consistent equivalence group can contain at most one type constant, which may have multiple occurrences. If an equivalence group has distinct constants, then this group is inconsistent, and so is the complete type graph. An inconsistency is witnessed by a path that connects two distinct constants, which we will refer to as an *error path*. Henceforth, we only consider paths that visit vertices at most once. For each path that we exclude, there is a shorter alternative.

Example 7.1. Consider the following constraint set.

$$\{\ v_0 \overset{\#0}{\equiv} v_1\ ,\ v_1 \overset{\#1}{\equiv} v_2\ ,\ v_1 \overset{\#2}{\equiv} v_3\ ,\ v_0 \overset{\#3}{\equiv} A\ ,\ v_2 \overset{\#4}{\equiv} B\ ,\ v_3 \overset{\#5}{\equiv} A\ \}$$

Each of the constraints is annotated with a label ($\#0, \#1, \#2, \ldots$). Figure 7.1 shows the type graph constructed from this constraint set. In this figure, the type constants are labeled with a variable so that we can refer to one particular type constant. The only equivalence group of this type graph contains two error paths between the constants A and B, namely $[\#3, \#0, \#1, \#4]$ and $[\#5, \#2, \#1, \#4]$.

An important property of a type graph is that it can be in an inconsistent state. We can get to a consistent state by removing edges, or, from a different perspective, by excluding equality constraints from the constraint set from which the type graph was constructed. Note that by repeatedly removing edges from the type graph one will eventually arrive at a consistent state. Moreover, the inconsistency is removed only if at least one edge of each error path is removed. In general, there may be many candidate edges to choose for removal. However, some edges seem to be more suitable than others. The following definitions make this idea more precise.

Definition 7.1 (Constant clash). *Two type constants clash if they are different, and present in the same equivalence group. In such a case we say that the type graph is in an inconsistent state. The inconsistency is witnessed by the error paths that connect the two constants.*

A maximal consistent subset is a set of constraints such that adding one constraint from the original set (one that is not yet present) would introduce an inconsistency.

Definition 7.2 (Maximal consistent subset). *$\mathcal{C}$ is a maximal consistent subset of $\mathcal{D}$ if and only if*

$$\mathcal{C} \subseteq \mathcal{D} \quad \wedge \quad \mathcal{C} \textit{ is consistent} \quad \wedge \quad \forall x \in (\mathcal{D} - \mathcal{C}) : \mathcal{C} \cup \{x\} \textit{ is inconsistent.}$$

A constraint set is minimal inconsistent if the removal of any constraint from this set leads to consistency.

Definition 7.3 (Minimal inconsistent set). *$\mathcal{C}$ is a minimal inconsistent set if and only if*

$$\mathcal{C} \textit{ is inconsistent} \quad \wedge \quad \forall x \in \mathcal{C} : \mathcal{C} - \{x\} \textit{ is consistent.}$$

The notions of maximal consistent subsets and minimal inconsistent sets are closely related. The set of constraints that is absent in a maximal consistent subset is of special interest. Removing all these constraints from a type graph restores consistency: the set of removed constraints contains at least one constraint from each minimal inconsistent set.

Lemma 7.1. *Let $\mathcal{C}$ be a maximal consistent subset of $\mathcal{D}$. Then it follows that*

$$\forall x \in (\mathcal{D} - \mathcal{C}) : (\exists X : x \in X \ \wedge \ X \textit{ is minimal inconsistent}).$$

Proof. The set $\mathcal{C} \cup \{x\}$ is inconsistent (by Definition 7.2), although it is not necessarily minimal inconsistent. Keep removing constraints from this set until it is minimal inconsistent: let this set be X. Because $\mathcal{C}$ is consistent, x must be present in X. □

Example 7.1 (continued). The constraint set used to build the type graph shown in Figure 7.1 has six maximal consistent subsets.

$$\begin{array}{lll} \{\#0,\#1,\#2,\#3,\#5\} & \{\#0,\#1,\#2,\#4\} & \{\#0,\#1,\#4,\#5\} \\ \{\#0,\#2,\#3,\#4,\#5\} & \{\#1,\#3,\#4,\#5\} & \{\#1,\#2,\#3,\#4\} \end{array}$$

These six sets are the alternatives that make the type graph consistent. If we leave out constraint #1 or #4, then the inconsistency disappears. Alternatively, we remove one of #2 and #5, and one of #0 and #3. In total, this gives us six options. Observe that the sizes of the maximal consistent subsets vary. This indicates that some alternatives to restore consistency require more corrections in the type graph than others. The two minimal inconsistent subsets of the original constraint set are the two error paths in the type graph.

7.2 Full type graphs

Conceptually, type graphs are as simple as sketched in Section 7.1. However, if we extend the type language to include composite types such as *Int* → *Bool* and [(*Char*, *Char*)], then we have to deal with two additional issues. Firstly, equality of two composite types is propagated to the subterms. For instance, $F\ v_0\ v_1 \equiv F\ v_2\ v_3$

implies that $v_0 \equiv v_2$ and $v_1 \equiv v_3$ (for some binary type constructor F). These two constraints obtained by propagation are not independent, since they both originate from the same source constraint. Note that equality can also be propagated indirectly as a result of a chain of equalities that equate two composite types. For instance, $G\ v_0 \equiv v_1$ and $v_1 \equiv G\ v_2$ together imply that $v_0 \equiv v_2$. Secondly, equalities between composite types can lead to infinite types. For example, the constraint $v_0 \equiv v_0 \rightarrow v_1$ can never be satisfied. We have to cope with such inconsistencies in our type graphs.

The simple type language of the previous section is extended with type application, which we assume to be left associative. (The new type language is the same as the one introduced on page 10.) Again, we only consider well-formed types, thereby completely ignoring kinding aspects such as the arity of a type constructor.

$$
\begin{array}{lll}
\multicolumn{2}{l}{\textit{Type:}} & \\
\tau ::= & v_0, v_1, v_2, \ldots & \textit{(type variable)} \\
\quad\mid & A, B, C, \ldots & \textit{(type constant)} \\
\quad\mid & \tau_1\ \tau_2 & \textit{(type application)}
\end{array}
$$

Before we explain how we construct a type graph for a given set of equality constraints, we present a brief overview of the structure of our type graphs.

- A *vertex* in the type graph corresponds to a type variable, a type constant, or a type application.
- A type graph contains three kinds of *edges*. *Child* edges are directed, and are used to express the parent-child relation between two vertices. Note that all child edges depart from a vertex that corresponds to a composite type. Furthermore, we have *initial* edges and *implied* edges, which are undirected. Both of these edges express equality. Each initial edge corresponds to a single equality constraint, and vice versa. Implied edges represent equality propagated for two composite types.
- The subgraph of a type graph that contains only the child edges, and not the initial edges nor the implied edges, is called the *term graph*.
- *Equivalence groups* are the connected components of a type graph when we take only initial and implied edges into account. The vertices of an equivalence group are supposed to correspond to the same type.
- An *equality path* is a path between two vertices of the same equivalence group such that this path contains only initial and implied edges.

We now discuss how to construct a type graph for a set of equality constraints. The following three steps are executed for each equality constraint that is to be included in the type graph.

1. A term graph is constructed for the left-hand side type of the equality constraint. Similarly, we construct a term graph for the type on the right-hand side. The term graph for a type variable is a single vertex: this vertex is shared by all occurrences of this type variable in the constraint set. The term graph for a type constant is a single vertex, which we annotate with the constant. For each occurrence of this constant in the constraint set, we introduce a new

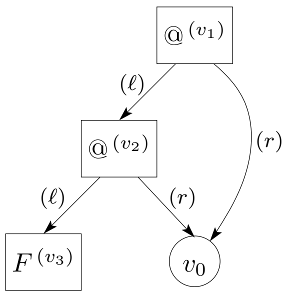


Figure 7.2. The term graph for $(F\ v_0\ v_0)$

vertex. In case of a composite type (application), we first construct term graphs for the two subterms. Then, we introduce a new vertex for the composite type, and we add directed edges (*child edges*) to indicate the parent-child relation between the vertices. These edges are labeled with (ℓ) or (r) for the left and the right subterm respectively. Note that our term graphs are directed acyclic graphs. Figure 7.2 shows the term graph that is constructed for $F\ v_0\ v_0$. Here, vertices labeled with (@) correspond to composite types.

2. An edge is inserted between the two vertices corresponding to the types in the operands of the equality constraint. These vertices were introduced in the previous step. We call such an edge an *initial edge*, since it represents type equality imposed directly by a single type constraint. Additional information that is supplied with a constraint is stored with the edge.
3. The last step is called *equality propagation*. The insertion of an initial edge in the previous step may have caused two equivalence groups to be merged. For all pairs of composite types that are in the same equivalence group, we have to propagate equality to the children. For example, suppose that the composite type $(\tau_0\ \tau_1)$ is present in equivalence group E_1, and that E_2 contains $(\tau_2\ \tau_3)$. As soon as E_1 and E_2 are combined into a single equivalence group, an (equality) edge is inserted between the children in pairwise fashion: between τ_0 and τ_2, and between τ_1 and τ_3. Such an edge will be called an *implied* or *derived* equality edge, and these edges are justified by the fact that the parents are part of the same group. Insertion of implied equality edges may cause other equivalence groups to be merged. This, again, may result in the insertion of new derived edges between different pairs of vertices, and so on.

Recall that equivalence groups are determined by considering initial edges and implied equality edges (child edges can be ignored at this point). Although propagation of equality may lead to the insertion of more and more derived edges, propagation will eventually stop. To see this, consider the number of equivalence groups in a type graph. Equality is propagated for composite types that become part of the same equivalence group. Hence, we propagate equality if and only if equivalence groups are merged. Equality propagation stops, because the number of groups strictly decreases.

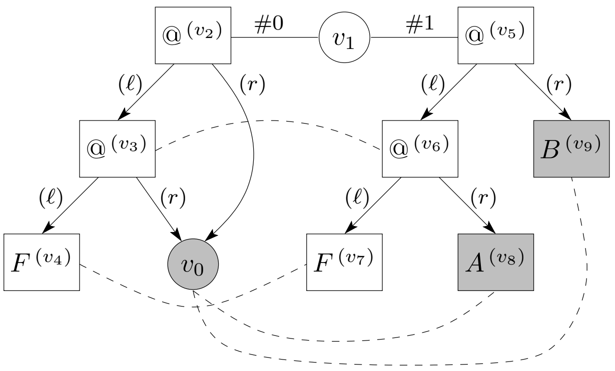


Figure 7.3. An inconsistent type graph

Example 7.2. Consider the following constraint set.

$$\{ \ v_1 \overset{\#0}{\equiv} F\ v_0\ v_0 \ \ , \ \ v_1 \overset{\#1}{\equiv} F\ A\ B \ \}$$

Figure 7.3 shows the type graph constructed for this constraint set. The two initial equality constraints are labeled with #0 and #1. The type graph has four implied equality edges (the dashed edges). The type graph contains four equivalence groups, including one that contains all the shaded vertices. This group contains both type constants A and B.

The interesting question is, of course, why two vertices end up in the same equivalence group. For example, if two distinct type constants are part of the same equivalence group, then we want to find out which equality constraints are (together) responsible for the inconsistency in the type graph. An equality path between the two constants is a combination of initial and implied edges, and witnesses the supposed equality. We want to put the blame of the inconsistency on equality constraints from which the type graph was constructed. Each initial edge corresponds directly to such an equality constraint, but for implied edges we have to trace why such an edge was inserted in the type graph. For this reason, we discuss expansion of equality paths.

Expanding a path entails replacing its implied edges by the equality paths between the two parent vertices (that are part of another equivalence group). Repeatedly replacing implied edges yields a path without implied edges. To denote an expanded equality path, we use the annotations $Up_i^{(\delta)}$ and $Down_i^{(\delta)}$, where δ is either ℓ (left child) or r (right child). The annotation *Up* corresponds to moving upwards in the term graph (from child to parent) by following a child edge, whereas *Down* corresponds to moving downwards (from parent to child). Each *Up* annotation in an equality path comes with a *Down* annotation at a later point, which we make explicit by assigning unique *Up-Down* pair numbers, written as subscript. The unique pair numbers emphasize the stack-like behavior of *Up-Down* pairs, and serve no other purpose.

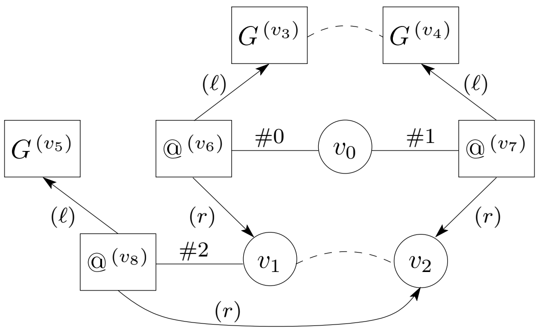


Figure 7.4. A type graph with an infinite path

Example 7.2 (continued). Consider Figure 7.3 again, and in particular the error path π from the type constant $A^{(v_8)}$ to the type constant $B^{(v_9)}$ (via the type variable v_0). Expanding the implied edge between $A^{(v_8)}$ and v_0 yields a path that contains the implied edge between $@^{(v_6)}$ and $@^{(v_3)}$. Expansion of this implied edge gives the path between $@^{(v_5)}$ and $@^{(v_2)}$, which consists of the two initial edges. Hence, we get the path $[\mathit{Up}_0^{(r)}, \mathit{Up}_1^{(\ell)}, \#1, \#0, \mathit{Down}_1^{(\ell)}, \mathit{Down}_0^{(r)}]$ after expanding the implied edge between $A^{(v_8)}$ and v_0. Similarly, we expand the implied edge between v_0 and $B^{(v_9)}$. The expanded error path π is now:

$$\pi = [\ \underbrace{\mathit{Up}_0^{(r)}, \mathit{Up}_1^{(\ell)}, \#1, \#0, \mathit{Down}_1^{(\ell)}, \mathit{Down}_0^{(r)}}\ ,\ \underbrace{\mathit{Up}_2^{(r)}, \#0, \#1, \mathit{Down}_2^{(r)}}\ ].$$

Both initial edges appear twice in π. Note that by following $\mathit{Up}_2^{(r)}$ from v_0, we can arrive at either v_2 or v_3. In general, *Up* annotations do not uniquely determine a target vertex. This ambiguity can be circumvented straightforwardly by including a target vertex in each *Up* annotation.

To make a type graph consistent, we first determine all (expanded) error paths, and then remove at least one initial edge from each path. When we remove an initial edge from the type graph, all implied edges that rely on this initial edge are removed as well. To determine which implied edges have to be removed as a result of removing an initial edge, we use a technique similar to equality propagation: we search for composite types that are no longer part of the same equivalence group, and we check the equivalence groups of the children.

Besides the error paths that connect different type constants in the same equivalence group, we also consider error paths between type constants and type applications that are part of the same equivalence group. There is a third category of error paths, which are closely related to the occurs check found in unification algorithms. Such a path starts and ends in the same vertex v, and may contain any number of equality edges (both initial and implied), and at least one edge from parent to child (without ever going back in the opposite direction). This path is a

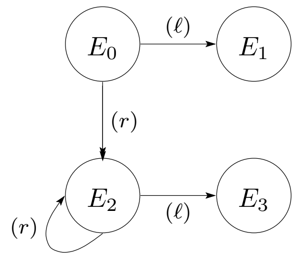


Figure 7.5. Parent-child dependencies between equivalence groups

proof that v represents an infinite type.[1] We will refer to such a path as an *infinite path*. The following example illustrates the concept of infinite paths.

Example 7.3. Consider the following set of type constraints.

$$\{ \ v_0 \overset{\#0}{\equiv} G\ v_1 \ \ , \ \ v_0 \overset{\#1}{\equiv} G\ v_2 \ \ , \ \ v_1 \overset{\#2}{\equiv} G\ v_2 \ \}$$

From the first two constraints (#0 and #1) we conclude that v_1 and v_2 should be the same type, but the third constraint (#2) contradicts this conclusion. This contradiction becomes apparent in the type graph for this constraint set, which is shown in Figure 7.4. Starting in v_1, we follow edge #2 and arrive at v_2 by taking the right-child edge of the application vertex. The implied equality edge brings us back to our starting point v_1. After expansion of this implied edge, we get the following error path π.

$$\pi = [\#2, Down_{\infty}^{(r)}, Up_0^{(r)}, \#1, \#0, Down_0^{(r)}]$$

The one child edge followed downwards with no matching upward child edge is annotated with ∞. The removal of any of the three constraints would make this path disappear.

Infinite paths can be found by analyzing the parent-child dependencies between the equivalence groups of a type graph. This dependency graph should be acyclic: a cycle indicates that the graph has an infinite path.

Example 7.3 (continued). Take another look at the type graph of Figure 7.4, and consider the four equivalence groups $E_0 \ldots E_3$.

$$E_0 = \{v_0, v_6, v_7\} \ \ , \ \ E_1 = \{v_3, v_4\} \ \ , \ \ E_2 = \{v_1, v_2, v_8\} \ \ , \ \ E_3 = \{v_5\}$$

Figure 7.5 displays the parent-child dependencies between these equivalence groups. The directed edge from E_2 to E_2 reveals the presence of an infinite path.

[1] If the type language can represent recursive types, for instance via a recursion operator, then there is no need to consider these infinite paths as problematic.

7.2.1 Cliques

The number of vertices in an equivalence group can become large, and also the number of application vertices in a group can become arbitrarily large. Equality must be propagated to the children of each pair of application vertices. Hence, the number of implied equality edges is quadratic in the number of application vertices that are present in a single equivalence group.[2] A simple observation is that this leads to a clique of vertices: all vertices are connected to each other by an implied edge, and all have a parent that is part of the same equivalence group. Hence, we choose a different (and more efficient) representation for our implementation of a type graph. This optimization gives us two advantages.

- It reduces the number of implied equality edges in a type graph. The number of implied equality cliques that can appear in a type graph is limited to the number of application vertices, whereas the number of implied equality edges is quadratic in the number of application vertices.
- Some of the equality paths that contain multiple implied equality edges from the same clique can be excluded in our search for the error paths of a type graph. This reduces the number of error paths.

Note that this optimization does not affect the basic principles of a type graph.

Definition 7.4 (Detour equality path). *A detour is a path that consists of two (consecutive) implied edges from the same clique. A detour equality path is an equality path that contains at least one detour. A shortcut of a detour equality path is the path in which we remove the two implied edges that form a detour, say from v_0 to v_1 and from v_1 to v_2, and replace it by the implied edge from v_0 to v_2.*

The idea is that error paths with a detour can be completely ignored for the following reason: the shortcut of such a path is also an error path, and removing initial edges from the type graph such that the shortcut disappears implies that the detour equality path has disappeared as well. This result is formulated in the next lemma.

Lemma 7.2 (Redundancy of detour equality paths). *Let π be a detour equality path, and let π' be a shortcut for π. We can get rid of π' by removing initial edges from the type graph. As soon as π' is no longer present in the type graph, the detour equality path π has been removed too.*

Proof. Suppose that the source of π (and π') is v_s, the target is v_t, and assume π contains the detour from v_0 to v_1 to v_2. We have three options to remove π' from the type graph: we break the path from v_s to v_0, from v_0 to v_2, or the path from v_2 to v_t. The first and the last option immediately remove π as well. We work out the case of breaking the implied edge from v_0 to v_2 in detail. Recall that implied edges cannot be removed from the type graph: instead, implied edges may disappear as a result of removing initial equality edges.

[2] To be precise: if an equivalence group contains n application vertices, then $n^2 - n$ implied equality edges will be inserted in the type graph, half for the left children, half for the right children.

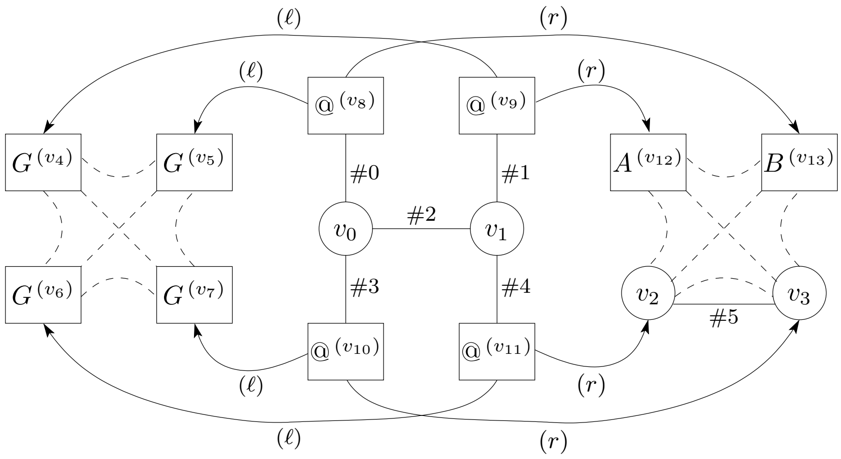


Figure 7.6. A type graph with two implied equality cliques

The clique of the detour was introduced because the parents of v_0, v_1, and v_2 (say v_0', v_1', and v_2', respectively) are in the same equivalence group. The implied equality edge between v_0 and v_2 disappears only if v_0' and v_2' are no longer part of the same equivalence group. If there is no equality path between v_0' and v_2', then certainly no equality path exists from v_0' to v_2' via v_1'. Hence, at least one of the two implied equality edges of π's detour has disappeared, which implies that the path π is no longer present in the type graph. □

The intuition is that an equality path should never have two consecutive implied equality edges from the same clique. Note, however, that a non-consecutive (second) visit of an implied equality clique is not necessarily a detour. The next example illustrates this idea.

Example 7.4. Consider the following set of type constraints.

$$\mathcal{C} = \{\ v_0 \overset{\#0}{\equiv} G\ B\ ,\ v_1 \overset{\#1}{\equiv} G\ A\ ,\ v_0 \overset{\#2}{\equiv} v_1\ ,\ v_0 \overset{\#3}{\equiv} G\ v_3\ ,\ v_1 \overset{\#4}{\equiv} G\ v_2\ ,\ v_2 \overset{\#5}{\equiv} v_3\ \}$$

The type graph constructed for this constraint set is shown in Figure 7.6. This type graph consists of three equivalence groups – two equivalence groups have an implied equality clique of four vertices. What are the error paths in this type graph that are not a detour?

The only irregularity in the type graph is that the constants A and B are part of the same equivalence group. The most obvious equality path is the implied edge from the vertex $A^{(v_{12})}$ to $B^{(v_{13})}$. Expansion of this edge gives us a path between the application vertices v_9 and v_8. Hence, our first error path is

$$\pi_0 = [\mathit{Up}_0^{(r)}, \#1, \#2, \#0, \mathit{Down}_0^{(r)}].$$

But this is not the only error path – if that were the case, then removing #0, #1, or #2 would make the constraint set consistent. For the first two constraints this is true, but removing #2 from $\mathcal{C}$ leaves the constraint set inconsistent.

One could propose to consider the error path from $A^{(v_{12})}$ to v_3 to $B^{(v_{13})}$. However, Lemma 7.2 tells us that this equality path is a detour (and is thus redundant). And indeed, expanding this path gives us

$$[\mathit{Up}_0^{(r)}, \#1, \#2, \#3, \mathit{Down}_0^{(r)}, \mathit{Up}_1^{(r)}, \#3, \#0, \mathit{Down}_1^{(r)}],$$

which is a detour of π_0.

The additional error path we should consider is from $A^{(v_{12})}$ to v_2 by following the implied edge, then taking the initial equality edge #5 to vertex v_3, and from there to $B^{(v_{13})}$ with an implied edge. This path visits the implied equality clique twice, but there are no two consecutive steps within the same clique. A third path exists, going first to v_3, then to v_2, before ending in $B^{(v_{13})}$.

$$\pi_1 = [\mathit{Up}_0^{(r)}, \#1, \#4, \mathit{Down}_0^{(r)}, \#5, \mathit{Up}_1^{(r)}, \#3, \#0, \mathit{Down}_1^{(r)}]$$

$$\pi_2 = [\mathit{Up}_0^{(r)}, \#1, \#2, \#3, \mathit{Down}_0^{(r)}, \#5, \mathit{Up}_1^{(r)}, \#4, \#2, \#0, \mathit{Down}_1^{(r)}]$$

The error path π_1 remains after the removal of #2, which confirms that only removing #2 does not make the type graph consistent. There are five “minimal” sets, and removing all the constraints of one of these sets makes the type graph consistent.

$$\{\{\#0\}, \{\#1\}, \{\#2, \#3\}, \{\#2, \#4\}, \{\#2, \#5\}\}$$

Note that the complements of these sets are maximal consistent subsets of the original constraint set.

7.2.2 A type graph as a substitution

A consistent type graph represents a substitution. Given a vertex v in an equivalence group E, the type (or type variable) associated with v can be determined as follows.

- If E has exactly one type constant and no application vertices, then this type constant is the type we assign to v. Of course, this type constant may appear several times in E.
- If E has no type constants, but there is at least one application vertex, then the type associated with v is a composite type. Choose one left child of one of the application vertices in E (say v_0) and one right child (say v_1). Now, assign to v the application of the type associated with v_0 and the type associated with v_1.
- If E has no type constants and there is no application vertex in E, then a type variable is chosen to represent all vertices of E. This can be one of the vertices of E, or else, a fresh type variable that does not appear elsewhere.

Note that these three conditions fully partition all the equivalence groups appearing in a consistent type graphs. In particular, the absence of an infinite path in a type graph guarantees that the recursion in the second case leads to a finite computation.

We define a new constraint solver which uses type graphs to represent substitutions. This solver is interchangeable with the greedy constraint solver (see Definition 5.4 on page 78).

Definition 7.5 (Type graph constraint solver). *A type graph solver is a constraint solver that uses a type graph to implement the substitution state.*

7.2.3 Type synonyms

Interestingly, type graphs have an additional benefit when it comes to type synonyms. Type synonyms let one introduce new type constants to abbreviate types that are already present in the language. This serves two purposes: complex types become easier to write and read, and one can introduce intuitive names for a type. One advantage type synonyms have over introducing new datatypes is that standard functions on the representation can still be used with the type synonym. A type synonym can have type arguments, although partial application of a type synonym is not allowed. Examples of type synonyms include:

$$\begin{array}{ll}\textbf{type}\ \mathit{Number} & = \mathit{Int} \\ \textbf{type}\ \mathit{Telephone} & = \mathit{Number} \\ \textbf{type}\ \mathit{String} & = [\mathit{Char}] \\ \textbf{type}\ \mathit{Parser}\ s\ a & = [s] \to [(a, [s])]\end{array}$$

Dealing with type synonyms in a type inference algorithm is generally considered a straightforward extension, especially since completely unfolding the type synonyms solves the problem. However, unfolding type synonyms is not satisfactory if we want to report type error messages (or present inferred types) in terms of the original program. Therefore, a conservative unfolding policy is to be preferred.

We extend our type graphs, and associate type synonym information with each vertex. Recall how a term graph is built for each type in the constraint set. Let τ be a type for which we want to construct a term graph. We unfold τ until the top-level type constructor is not a type synonym. For this unfolded type we build a term graph, and we remember the original type τ in the top-level vertex. Likewise, type synonyms inside τ are handled.

When we determine the type associated with some equivalence group E, we take into account the original types stored in the vertices of E. For instance, if E has two type constants *Int*, and both are annotated with the type synonym *Telephone*, then the latter constant is to be used to represent the type of E since this is the more informative type. If the constants *Int*, *Number*, and *Telephone* are the original types in an equivalence group, then it is unclear which of the three should be appointed representative. The type constant *Int* is the safest alternative, since this is the common unfolded type. However, one could also opt for *Telephone* or *Number*, as these types, being more specific, may be more intuitive.

7.3 Implementation

This section presents a monadic implementation of a type graph in Haskell. Code is given for the four elementary operations: constructing a term graph, adding and removing an equality constraint, and making a substitution from a type graph. Parts of the implementation are left unspecified, e.g. the representation of the monad.

7.3.1 The basics

The data type *Tp* is used to represent types.

```
data Tp = TVar Int | TCon String | TApp Tp Tp
```

The three alternatives are for type variables, type constants, and binary type applications. We assume that *TypeGraph* is an instance of the *Monad* type class

```
data TypeGraph info a   -- abstract data type
instance Monad (TypeGraph info)
```

Observe how we parameterize over the type of additional information that is stored in the type graph. Because the type of information stored in a type graph is a parameter, we can abstract from which specific information is to be stored. In the monad we can also maintain the set of known type synonyms, and a counter for creating unique vertices.

```
getTypeSynonyms :: TypeGraph info TypeSynonyms
nextUnique      :: TypeGraph info VertexId
```

The following data types are introduced for vertices.

```
data VertexId   = VertexId Int deriving Eq
type VertexInfo = (VertexTp, Maybe Tp)
data VertexTp   = VVar | VCon String | VApp VertexId VertexId
```

Each vertex in the type graph gets a unique number (*VertexId*), and some extra information about the type that the vertex represents (*VertexInfo*), which is a pair. The first component describes the type in which type synonyms have been unfolded completely. It's type, *VertexTp*, is very similar to *Tp*, except that no *int* value is stored with a variable, and that for each application we have the two vertices that represent the types that are combined. The second component of the pair is optional, and contains the original type. *Nothing* indicates that no type synonyms were used.

The type *EdgeId* is used to represent the edges of the type graph.

```
data EdgeId = EdgeId VertexId VertexId
```

In our implementation, a type graph is a list of equivalence groups: an equivalence group is a collection of vertices, edges, and (implied) cliques.

```
data EQGroup info =
      EQGroup{ vertices :: [(VertexId, VertexInfo)]
             , edges    :: [(EdgeId, info)]
             , cliques  :: [Clique]
             }
```

We assume that there is an equality path between each pair of vertices in an equivalence group. Figure 7.7 contains a list of functions to create and manipulate equivalence groups and type graphs. We will use these functions, but leave them unspecified.

```
  -- Equivalence group
emptyGroup     :: EQGroup info
splitGroup     :: EQGroup info → [EQGroup info]
insertVertex   :: VertexId → VertexInfo → EQGroup info → EQGroup info
insertEdge     :: EdgeId   → info       → EQGroup info → EQGroup info
insertClique   :: Clique                → EQGroup info → EQGroup info
removeEdge     :: EdgeId                → EQGroup info → EQGroup info
removeClique   :: Clique                → EQGroup info → EQGroup info
combineGroups  :: EQGroup info          → EQGroup info → EQGroup info

  -- Type graph
createGroup         :: EQGroup info → TypeGraph info ()
updateGroupOf       :: VertexId → (EQGroup info → EQGroup info)
                         → TypeGraph info ()
equivalenceGroupOf  :: VertexId   → TypeGraph info (EQGroup info)
combineClasses      :: [VertexId] → TypeGraph info ()
splitClass          :: VertexId   → TypeGraph info [VertexId]
```

Figure 7.7. Interfaces for equivalence groups and type graphs

7.3.2 Constructing a term graph

We start with a function to construct a term graph for a given type. The *VertexId* associated with the type is returned.

```
addTermGraph :: Tp → TypeGraph info VertexId
addTermGraph tp =
  do synonyms ← getTypeSynonyms
     let (newtp, original) =
            case expandToplevelTC synonyms tp of
               Nothing → (tp, Nothing)
               Just x  → (x, Just tp)
     case newtp of
        TVar i     → return (VertexId i)
        TCon s     → makeNewVertex (VCon s, original)
        TApp t1 t2 →
          do v1 ← addTermGraph t1
             v2 ← addTermGraph t2
             makeNewVertex (VApp v1 v2, original)
```

The type synonyms are used to expand the top-level type constructor of the type *tp*. Inner type synonyms are left unchanged. Note that *newtp* is the expanded type, and *original* is the original type (or *Nothing* if no type synonym was used). Then we proceed by the case of *newtp*. For a type variable, we return the corresponding *VertexId* (recall that we use one vertex for all occurrences for the same type variable). For a type constant, we create a new vertex with the function *makeNewVertex*, to which we pass information about the original type. If *newtp* is a type application, then we recursively create the term graphs for the children, and create a new vertex to represent the type application.

```
makeNewVertex :: VertexInfo → TypeGraph info VertexId
makeNewVertex info =
  do vid ← nextUnique
     createGroup (insertVertex vid info emptyGroup)
     return vid
```

In order to make a new vertex, we request a fresh *VertexId*. Next, a new equivalence group is created that only contains this single vertex, together with the information that is passed as an argument. The types of *createGroup*, *insertVertex*, and *emptyGroup* can be found in Figure 7.7.

7.3.3 Adding an equality constraint

Now that we can create a term graph for a given type, the next step is to handle the insertion of equality constraints. Or, by taking another viewpoint, to add an initial edge to the type graph. First, we create two term graphs for the types involved.

```
unifyTypes :: info → Tp → Tp → TypeGraph info ()
unifyTypes info t1 t2 =
  do v1 ← addTermGraph t1
     v2 ← addTermGraph t2
     addEdge (EdgeId v1 v2) info
```

We add an (initial) equality edge between the two *VertexId*s and store the additional information. Adding an initial edge to the type graph is defined as follows.

```
addEdge :: EdgeId → info → TypeGraph info ()
addEdge edge@(EdgeId v1 v2) info =
  do combineClasses [v1, v2]
     updateGroupOf v1 (insertEdge edge info)
     propagateEquality v1
```

Clearly, the vertices *v1* and *v2* should, eventually, end up in the same equivalence group. To achieve this, we call *combineClasses* with a list containing the two vertices. This is just to record that the two equivalence groups have been combined.[3] Next, we insert the edge in the equivalence group of *v1* (and thus also the group of *v2*). At this point, the only thing left to do is to propagate equality caused by the merging of the equivalence groups. But before we give an implementation of *propagateEquality*, take a look at the following type definitions.

[3] Depending on the implementation of the type graph monad, this call to *combineClasses* can be omitted altogether.

```
data Clique      = Clique [ParentChild]
data ChildSide   = LeftChild | RightChild
data ParentChild =
     ParentChild { parent    :: VertexId
                 , child     :: VertexId
                 , childSide :: ChildSide
                 }
```

A *Clique* is a list containing at least two *ParentChild* values. A *ParentChild* record consists of a parent vertex, a child vertex, and whether it is a left or a right child. All the parents of a clique must be part of the same equivalence group. Furthermore, the value of *childSide* must be the same for all the elements of a clique. We define two helper-functions on cliques.

```
childrenInClique :: Clique → [VertexId]
childrenInClique (Clique xs) = map child xs

cliqueRepresentative :: Clique → VertexId
cliqueRepresentative (Clique [])      = error "empty list"
cliqueRepresentative (Clique (x : _)) = child x
```

We nominate the first vertex as representative of the clique. The same criterion is used to obtain the representative of an equivalence group containing a given vertex.

```
representative :: VertexId → TypeGraph info VertexId
representative v =
  do eqgroup ← equivalenceGroupOf v
     case vertices eqgroup of
       []         → error "empty list"
       (v, _) : _ → return v
```

Given a vertex *v*, *childrenInGroupOf* determines the left and right children in the equivalence group that contains *v*.

```
childrenInGroupOf :: VertexId →
                         TypeGraph info ([ParentChild], [ParentChild])
childrenInGroupOf v =
  do eqgroup ← equivalenceGroupOf v
     return $ unzip
       [ (ParentChild p v1 LeftChild, ParentChild p v2 RightChild)
       | (p, (VApp v1 v2, _)) ← vertices eqgroup
       ]
```

We continue with the definition for *propagateEquality*.

```
propagateEquality :: VertexId → TypeGraph info ()
propagateEquality v =
  do (listLeft, listRight) ← childrenInGroupOf v
     left  ← mapM (representative ∘ child) listLeft
     right ← mapM (representative ∘ child) listRight

     when (length listLeft > 1) $
       do addClique (Clique listLeft)
          addClique (Clique listRight)

          when (length (nub left) > 1) $
            propagateEquality (head left)

          when (length (nub right) > 1) $
            propagateEquality (head right)
```

The left and right children of v's equivalence group are computed. This results in a list of *ParentChild* values for the left children, and a list for the right children. Of course, both lists have the same length. Next, the representative of the equivalence group is determined for each left child. For the moment, we leave this list (*left*) with values of type *VertexId* for what it is. We do the same for the right children. In case *listLeft* has at least two elements, then we add a clique for the left children, and also for the right children. How to add a clique is discussed later on. After the two cliques have been added to the type graph, all the left children are in the same equivalence group. If necessary, we propagate equality for this equivalence group. Likewise, we propagate equality for the right children.

```
addClique :: Clique → TypeGraph info ()
addClique clique =
  do combineClasses (childrenInClique clique)
     updateGroupOf (cliqueRepresentative clique) (insertClique clique)
```

First, the equivalence groups of the clique's children are merged into a single group. Then a clique is inserted in this group.

7.3.4 Removing an equality constraint

Removing an edge from a type graph is the reverse process. The following function defines how to delete an initial edge.

```
deleteEdge :: EdgeId → TypeGraph info ()
deleteEdge edge@(EdgeId v1 _) =
  do updateGroupOf v1 (removeEdge edge)
     propagateRemoval v1
```

The edge is removed from its equivalence group. This removal may cause the equivalence group to be split in two (although not always). Hence, we have to propagate the removal. Take a look at the following definition.

```
propagateRemoval :: VertexId → TypeGraph info ()
propagateRemoval v =
  do vs     ← splitClass v
     tuples ← mapM childrenInGroupOf vs

     let (leftList, rightList) = unzip tuples
         cliqueLeft  = Clique (concat leftList)
         cliqueRight = Clique (concat rightList)
         children    = [child pc | pc : _ ← leftList ++ rightList]
         newCliques  = [Clique list | list ← leftList ++ rightList,
                                      length list > 1]

     when (length (filter (not ∘ null) leftList) > 1) $
       do deleteClique cliqueLeft
          deleteClique cliqueRight
          mapM addClique newCliques
          mapM propagateRemoval children
          return ()
```

We use *splitClass* to lookup the equivalence group that may have to be split, and we determine the new equivalence groups. Then we determine the children for each of those new groups. Both *cliqueLeft* and *cliqueRight* are part of the type graph, but since the equivalence group is split, and vertices may no longer be part of the same group, we have to remove those cliques. If *leftList* has more than one non-empty list, then we delete *cliqueLeft* and *cliqueRight* from our type graph. After this, we insert (smaller) cliques that are still valid: those are subsets of one of the two original cliques. Note that splitting up the cliques may also imply that some vertices are no longer part of the same equivalence group. Hence, we propagate this removal too by a number of recursive calls to *propagateRemoval*.

7.3.5 Finding a substitution

Before we define which type is assigned to a type variable, we first define the helper-function *typeOfGroup* to query the type of an equivalence group. For this, we follow the procedure described on page 126.

```
typeOfGroup :: TypeSynonyms → EQGroup info → Maybe Tp
typeOfGroup synonyms eqgroup

  | length constants > 1                    = Nothing
  | not (null constants) ∧ not (null applies) = Nothing

  | not (null originals)  = Just (theBestType synonyms originals)
  | not (null constants)  = Just (TCon (head constants))
  | not (null applies)    = let (VertexId l, VertexId r) = head applies
                            in Just (TApp (TVar l) (TVar r))
  | otherwise             = Just (TVar (head variables))
```

```
    where
      variables  =     [i      | (VertexId i, _)      ← vertices eqgroup]
      constants  = nub [s      | (_, (VCon s, _))     ← vertices eqgroup]
      applies    =     [(l, r) | (_, (VApp l r, _))   ← vertices eqgroup]
      originals  =     [tp     | (_, (_, Just tp))    ← vertices eqgroup]
```

The function *typeOfGroup* returns a value of type *Maybe Tp*: *Nothing* is returned for an inconsistent equivalence group. First, we list all variables that are in the equivalence group, all the type constants (we remove duplicates), and all the type applications. The list *originals* contains the original types (where the type synonyms are not expanded) stored in this equivalence group. In the first two cases, *Nothing* is returned, reflecting that this equivalence group is not consistent. The third case handles a consistent group that contains at least one original type. A function *theBestType* is used to select one of the possible types that are equivalent under the list of type synonyms (see the discussion on type synonyms of Section 7.2). The final case returns a type variable (the representative of this equivalence group) if there are no type constants or type applications present. Note that in the case of a type application, we return an application of two type variables and do not recurse.

We continue with the definition of *substituteVariable*, which returns, if possible, the type of a vertex in our *TypeGraph* monad.

```
substituteVariable :: VertexId → TypeGraph info (Maybe Tp)
substituteVariable (VertexId v) = rec [] (TVar v)
  where
    rec history (TVar i)
      | i ∈ history = return Nothing
      | otherwise   =
          do synonyms ← getTypeSynonyms
             eqgroup  ← equivalenceGroupOf (VertexId i)
             case typeOfGroup synonyms eqgroup of
               Just (TVar j) → return (Just (TVar j))
               Just newtp    → rec (i : history) newtp
               Nothing       → return Nothing

    rec history tp@(TCon _) =
      return (Just tp)

    rec history (TApp l r) =
      do ml ← rec history l
         mr ← rec history r
         case (ml, mr) of
           (Just l′, Just r′) → return (Just (TApp l′ r′))
           _                  → return Nothing
```

We have to be careful not to end up in an infinite computation caused by an infinite type. We keep a list of already inspected type variables (a *history*) to prevent this. The recursive function *rec* is defined by three cases. For a type variable, we check the history, and return *Nothing* if this type variable was encountered before. Otherwise, we use *typeOfGroup* to determine the type of the equivalence group of the type variable. Depending on the result, we are done, we recurse (and extend our *history*), or we fail. The case for a type constant is straightforward, as it is this type that is to be returned. In the final case, a type application, we determine the type of both constituents, and combine these to return a composite type.

7.4 Composing constraint solvers

Type graphs can be used to improve the quality of type error messages, but they introduce a considerable overhead, which comes at the expense of run-time performance. Trading compilation speed for better error messages seems to be a fair deal, especially since our framework allows programmers to decide themselves which constraint solver to use. However, it would be more satisfactory if type inference would proceed quickly for well-typed programs (without having the overhead that makes the process so time-consuming), while still producing high quality error messages. Ideally, a type graph is only constructed if there really is an inconsistency. To accomplish this, we discuss constraint solver combinators to combine our existing constraint solvers.

7.4.1 The switch combinator

With the first combinator, we can switch from one constraint solver to another. We switch to the second constraint solver if a condition is met on the result of solving the constraints with the first solver. Given two constraint solvers s_1 and s_2, and a predicate p, we write $s_1 \oslash_p s_2$ to denote the composition of s_1 and s_2 under p with the switch combinator.

Definition 7.6 (Switch combinator). *Let s_1 and s_2 be constraint solvers, and let p be a predicate. Solving a constraint set $\mathcal{C}$ with the composite solver $(s_1 \oslash_p s_2)$ is defined by:*

$$(s_1 \oslash_p s_2)\ \mathcal{C} \ =_{def} \begin{cases} s_2\ \mathcal{C} & \text{if } p\ (s_1\ \mathcal{C}) \\ s_1\ \mathcal{C} & \text{otherwise} \end{cases}$$

This definition does not imply that s_1 has to finish solving $\mathcal{C}$ completely before we can switch to s_2. If the condition p can be verified during the process of solving constraints with s_1, then we can switch immediately.

Example 7.5. Let us consider a “smart” constraint solver that follows a greedy strategy (Definition 5.4, page 78), but as soon as it runs into an inconsistency, it switches to the type graph solver (Definition 7.5, page 126) to get more precise error messages. This composite solver is fast in solving consistent constraint sets, and accurate for inconsistent sets. We define this solver as

$$\mathit{switching} = \mathit{greedy} \oslash_p \mathit{typegraph},$$

where condition p indicates the presence of errors. Clearly, we want to switch as soon as we find the first error. The other way around (that is, *typegraph* $\oslash_p$ *greedy*) would give us an ineffective solver for the programmer with patience, and not in need of any help.

7.4.2 The partition and fusion combinators

Thus far, we have only considered solving a complete constraint set at once. It is plausible, however, that not all constraints in a given set relate to each other: not even indirectly via other constraints. Therefore it may be worthwhile to unravel a constraint set, and to search for subsets that can be solved independently. We first define when two constraint sets are independent of each other.

Definition 7.7 (Independence). *The constraint sets $\mathcal{C}_1$ and $\mathcal{C}_2$ are independent if and only if no type variable and no type scheme variable is shared by the two sets.*

Let INDEPENDENT be an algorithm that partitions a constraint set into the maximum number of independent sets. This algorithm can be defined in a straightforward way, as is illustrated by the next example.

Example 7.6. Consider the following set of type constraints.

$$\begin{aligned}\mathcal{C} = \{\ & v_0 \equiv v_1\ ,\ \sigma_0 := \textsc{Gen}(\emptyset, v_0 \to v_1)\ ,\ v_2 := \textsc{Inst}(\sigma_0)\ ,\ v_2 \equiv \mathit{Int} \to \mathit{Int} \\ ,\ & v_3 := \textsc{Skol}(\emptyset, \sigma_0)\ ,\ v_4 \equiv v_5 \to v_6\ ,\ v_5 \equiv v_6\ ,\ v_7 \equiv \mathit{Bool}\ \}\end{aligned}$$

Applying algorithm INDEPENDENT to $\mathcal{C}$ partitions the constraint set in the following three parts.

$$\begin{aligned}\mathcal{C}_1 &= \{\ v_0 \equiv v_1\ ,\ \sigma_0 := \textsc{Gen}(\emptyset, v_0 \to v_1)\ ,\ v_2 := \textsc{Inst}(\sigma_0) \\ &\quad ,\ v_2 \equiv \mathit{Int} \to \mathit{Int}\ ,\ v_3 := \textsc{Skol}(\emptyset, \sigma_0)\ \} \\ \mathcal{C}_2 &= \{\ v_4 \equiv v_5 \to v_6\ ,\ v_5 \equiv v_6\ \} \\ \mathcal{C}_3 &= \{\ v_7 \equiv \mathit{Bool}\ \}\end{aligned}$$

The following lemma formulates that if two constraint sets are independent, then their solutions are also unrelated (w.r.t. the substitutions and the type scheme substitutions).

Lemma 7.3. *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be independent constraint sets, and let Θ_1 and Θ_2 be solutions found for $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. Then*

$$\mathit{dom}(S_{\Theta_1}) \cap \mathit{dom}(S_{\Theta_2}) = \emptyset \quad \wedge \quad \mathit{dom}(\Sigma_{\Theta_1}) \cap \mathit{dom}(\Sigma_{\Theta_2}) = \emptyset.$$

Proof. Only type variables that are free in the constraint set that is solved can appear in the domain of the substitution, plus some fresh type variables that are introduced during solving (Lemma 4.7). Under the assumption that we use different sets of fresh type variables to solve $\mathcal{C}_1$ and $\mathcal{C}_2$, the domains of S_{Θ_1} and S_{Θ_2} are disjoint. Likewise, the domain of a type scheme substitution is limited to the type scheme variables that appear in the constraint set. □

To solve a constraint set in parts, we prefer a commutative and associative operator to combine the results. Let $\odot$ be such an operator for our constraint solvers. Lemma 7.3 indicates that defining $\odot$ is easy, since the substitutions and the type scheme substitutions in the solutions are unrelated.

We introduce a fusion combinator, which solves two constraint sets separately with a given constraint solver, and then combines the two results using $\odot$.

Definition 7.8 (Fusion). *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two independent constraint sets, and let s be a constraint solver. Fusing the process of solving $\mathcal{C}_1$ and $\mathcal{C}_2$ is defined by:*

$$\mathcal{C}_1 \;\oslash_s\; \mathcal{C}_2 \;\;=_{def}\;\; s\;\mathcal{C}_1 \odot s\;\mathcal{C}_2$$

Of course, it is to be expected that fusing two independent constraint sets yields the same result as solving the two sets in one run. The next proposition states that we can safely solve independent constraint sets separately.

Proposition 7.1 (Soundness of fusion). *If $\mathcal{C}_1$ and $\mathcal{C}_2$ are independent, then $\mathcal{C}_1 \;\oslash_s\; \mathcal{C}_2 = s\;(\mathcal{C}_1 \cup \mathcal{C}_2)$ up to alpha conversion.*

Example 7.7. We define a new constraint solver, which uses the *switching* solver from Example 7.5. Let $\mathcal{C}$ be the constraint set to be solved. Our solver yields $\mathcal{C}_1 \;\oslash_{switching}\; \cdots \;\oslash_{switching}\; \mathcal{C}_n$ with $\{\mathcal{C}_1, \ldots, \mathcal{C}_n\} = \textsc{Independent}(\mathcal{C})$.

If we take a closer look at the independent constraint sets of Example 7.6 (and $\mathcal{C}_1$ in particular), then we see that our algorithm for partitioning a constraint set is too coarse. Three constraints in $\mathcal{C}_1$ contain σ_0: the generalization constraint associates a type scheme with σ_0, whereas the instantiation constraint and the skolemization constraint use this type scheme. It is, however, not necessary to solve these three constraints within the same constraint set as long as the constraint set containing the generalization constraint is taken into account before the constraint sets that use σ_0. Hence, we define:

Definition 7.9 (Before relation). *$\mathcal{C}_1$ must be solved before $\mathcal{C}_2$ if and only if at least one of the type scheme variables that are assigned a type scheme in $\mathcal{C}_1$ (i.e., by a generalization constraint) is used in $\mathcal{C}_2$.*

In conjunction with this definition, we weaken the notion of independent constraint sets, and only take the free type variables into account (and not the type scheme variables). Note that Lemma 7.3 still holds under the weakened notion: if Θ is the solution for $\mathcal{C}$, then $dom(\Sigma_\Theta)$ contains exactly the type scheme variables that are defined in $\mathcal{C}$. We introduce a new algorithm $\textsc{Topological}$ that partitions a constraint set, and orders the parts, such that the free type variables of the constraint sets are disjoint, and such that the ordering is consistent with the before relation.

Example 7.6 (continued). If we use Topological to partition $\mathcal{C}$, we obtain (for example) the list $[\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5]$, where

$$\begin{aligned}
\mathcal{C}_1 &= \{\ v_0 \equiv v_1\ ,\ \sigma_0 := \textsc{Gen}(\emptyset, v_0 \to v_1)\ \}\\
\mathcal{C}_2 &= \{\ v_2 := \textsc{Inst}(\sigma_0)\ ,\ v_2 \equiv \mathit{Int} \to \mathit{Int}\ \}\\
\mathcal{C}_3 &= \{\ v_3 := \textsc{Skol}(\emptyset, \sigma_0)\ \}\\
\mathcal{C}_4 &= \{\ v_4 \equiv v_5 \to v_6\ ,\ v_5 \equiv v_6\ \}\\
\mathcal{C}_5 &= \{\ v_7 \equiv \mathit{Bool}\ \}.
\end{aligned}$$

Note that the order of these sets is now important. Other orderings exist: the only restrictions we have on the order of the constraint sets is that $\mathcal{C}_1$ must be taken into account before $\mathcal{C}_2$ and $\mathcal{C}_3$, because of the type scheme variable σ_0.

We have to refine Definition 7.8 for fusing two solutions. Suppose that $\mathcal{C}_1$ must be solved before $\mathcal{C}_2$. The idea is that the type scheme substitution in the solution found for $\mathcal{C}_1$ is applied to $\mathcal{C}_2$, after which we solve this substituted set.

Definition 7.10 (Fusion refined). *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be constraint sets such that $\mathit{ftv}(\mathcal{C}_1) \cap \mathit{ftv}(\mathcal{C}_2) = \emptyset$, and $\mathcal{C}_1$ must be solved before $\mathcal{C}_2$. Fusing the process of solving $\mathcal{C}_1$ and $\mathcal{C}_2$ is defined as follows.*

$$\mathcal{C}_1 \ \oslash_s\ \mathcal{C}_2 \ =_{\mathit{def}}\ \Theta_1 \odot s\ (\Sigma_{\Theta_1}(\mathcal{C}_2)) \qquad \text{where } \Theta_1 = s\ \mathcal{C}_1$$

The following proposition states that the refined fusion combinator is still sound.

Proposition 7.2. *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be constraint sets such that $\mathit{ftv}(\mathcal{C}_1) \cap \mathit{ftv}(\mathcal{C}_2) = \emptyset$, and $\mathcal{C}_1$ must be solved before $\mathcal{C}_2$. Then $\mathcal{C}_1 \ \oslash_s\ \mathcal{C}_2$ and $s\ (\mathcal{C}_1 \cup \mathcal{C}_2)$ are equal up to alpha conversion.*

7.4.3 A different approach: using the abstract syntax tree

Instead of using algorithm Topological, Helium determines the partitioned constraint set directly from the abstract syntax tree during the constraint collecting phase. The main advantage of this approach is that it becomes more efficient to obtain the partitioned constraint set. However, this technique is less precise, and it may find fewer parts compared to algorithm Topological. We use the refined fusion combinator (Definition 7.10) to combine the solving processes of the parts.

Roughly speaking, for each definition in a program we create a constraint set that is only sparsely related to the constraints generated for the rest of the program. This makes sense because changing a definition at one place does not necessarily change all the types in a program.[4] In most cases, changes remain local.

Writing type signatures helps to split the typing problem into subproblems. An explicit type signature provides an interface between the definition of a function and the uses of that function. Type signatures reduce type dependencies in programs, and increase the number of constraint sets that can be solved independently. Functions without an explicit type are assigned an inferred type during constraint

[4] Note, however, the fact that a small change in one definition can change the type of *every* function in a program.

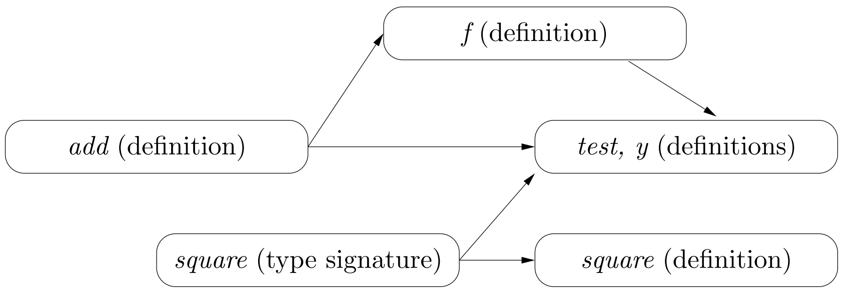


Figure 7.8. Constraint set dependencies

solving (by a generalization constraint), which imposes the *before* ordering on constraint sets.

A monomorphic variable (i.e., an identifier that has just one type, for instance introduced by a lambda abstraction) opposes solving constraint sets independently. In fact, the constraint sets of functions that share one or more monomorphic variables must be solved jointly. In practice, this is not a serious restriction since the top-level functions are free of monomorphic variables.

The following example sketches how we determine the constraint sets that are solved in topological order.

Example 7.8. Consider the following code fragments.

$$add\ x\ y = x + y$$

The function *add* only depends on the predefined function for addition. Because no type signature is supplied for *add*, we can infer its type independently of the rest of the program.

$$\begin{array}{l} square :: Int \to Int \\ square\ x = x * x \end{array}$$

The function *square* depends on the predefined function for multiplication. An explicit type signature is given, and thus we have to verify that the declared type is in accordance with the type for *square*'s definition. A type for *square* can be inferred regardless of the rest of the program.

$$\begin{array}{l} test\ x = \textbf{let}\ y = square\ x \\ \qquad\qquad\quad\ f = add\ 2 \\ \qquad\quad\ \ \textbf{in}\ add\ y\ (f\ y) \end{array}$$

The type of *test* depends on the type inferred for *add*, and the types assigned to the local definitions y and f. The definition for y uses *square* and x. Because we have a type signature for *square*, the type of y does not depend on the definition of *square*. Moreover, the constraint sets for (the definition of) *square* and y can be solved independent of each other, although both depend on *square*'s explicit type.

However, y's type also depends on the type of x, which is monomorphic in y's definition. Since x is introduced in *test*, the types of *test* and y are related, and, as a result, the constraints collected for *test* and y must be solved simultaneously. The definition of f, on the other hand, only requires an inferred type for *add*.

Figure 7.8 shows the dependencies between the constraint sets of the definitions and the type signatures. Observe that the constraint sets for *test* and y are combined. Furthermore, the type signature for *square* ensures that y does not depend on *square*'s definition. Note that partitioning the constraint set is closely related to performing binding group analysis.

Solving constraint sets in this way is a step towards incremental type inference: making a small change to a program should result in a new typed program without the need to recompute all types. Of course, changes may introduce type errors in a well-typed program, and vice versa. Type dependencies, such as the dependencies in Figure 7.8, encode exactly in which parts of a program types may change. Hence, we only have to solve the constraints for the parts of a program that are influenced by an edit operation. Advanced editors, such as Proxima [56], could benefit from incremental type inference as they can offer online type inference during program development. Without support in the type inference process, online inference is hardly advantageous, since full type inference for a large program is likely to cost too much time.

But we can go one step further. An editor for structured documents is aware of the program's abstract syntax tree. Changes on the structure of such a tree can be translated into changes in the constraint set. Because constraints can be added to and removed from a type graph, it is relatively simple to incorporate small changes of a constraint set in a type graph without building the type graph from scratch.

8

Heuristics

Overview. *In this chapter, we propose a number of heuristics that operate on type graphs. Heuristics select which location is reported for a type inconsistency, and may change the content of a type error message. For all our heuristics, we present some concrete examples of the type error messages that are reported.*

The previous chapter explains how type errors in a program show up as error paths in the program's type graph. The last step in the type inference process is to extract information from the type graph, and use this to notify the programmer about his mistakes.

We take the following approach: we keep on removing initial edges from the type graph until the graph is consistent. Because each initial edge corresponds directly to a specific equality constraint, we basically remove constraints from our constraint set. For each edge removed from the type graph, we create one type error message, thereby using the constraint information stored with that edge. The approach naturally leads to multiple type error messages being reported. The important question is, of course, which constraints (or edges) to select for removal.

In principle, all the constraints that are on an error path are candidates. However, we feel that some constraints are better candidates for removal than others. To select the "best" candidate for removal, we consult a number of type graph heuristics. These heuristics are mostly based on common techniques used by experts to explain type errors. In addition to selecting what is reported, heuristics can specialize error messages, for instance by including hints and probable fixes.

Because our system of heuristics consists of several independent type graph heuristics, we also need some facility to coordinate the interaction between the heuristics. Although this is a crucial part of the system, we will not go into details. We present a number of heuristics. Of course, one can extend and fine-tune this collection. In fact, the next chapter introduces type inference directives, which allow a user to define his own heuristics.

A final consideration is how to present the errors to a user, taking into consideration the limitations imposed by the used output format. In this chapter we consider reporting textual error messages, which is a common approach taken by (command-line) compilers and interpreters. Other techniques to present information available in a type graph (e.g., an interactive session, the highlighting of all contributing program sites, or an explanation in English) are also within reach.

8.1 Share in error paths

Our first heuristic applies some common sense reasoning: if a constraint is involved in more than one error path, then it is a better candidate for removal than a constraint appearing in just one error path. The set of candidates is reduced to the constraints that occur most often in the error paths. Note that this heuristic helps to decrease the number of reported error messages, as multiple error paths disappear by removing one constraint. However, repeatedly removing the constraint on the largest number of error paths is not always optimal, and does not necessarily lead to the minimum number of error messages. To achieve this, we need to select the best set of constraints to be removed, instead of repeatedly selecting one constraint. We show this by example.

Example 8.1. Consider the set P consisting of six error paths.

$$P = \{ \ \{\#1, \#2\} \ , \ \{\#1, \#3\} \ , \ \{\#1, \#4\} \ , \ \{\#2\} \ , \ \{\#3\} \ , \ \{\#4\} \ \}$$

For each set in P, we have to remove at least one of its constraints from the type graph. At first sight, the constraint #1 is the best candidate for removal because it is present in three error paths. Unfortunately, after removing #1, we still have to remove #2, #3, and #4. This confirms that selecting the constraint which appears in most error paths not necessarily yields the least number of corrections: a better alternative would be to remove $\{\#2, \#3, \#4\}$ Although this example is contrived (for instance, because $\{\#1, \#2\} \supseteq \{\#2\}$), comparable situations can arise from real programs.

The share-in-error-path heuristic helps to report the most likely source of an error, and implements the approach suggested by Johnson and Walz [30]: if we have three pieces of evidence that a value should have type *Int*, and only one for type *Bool*, then we should focus on the latter. In general, this heuristic does not reduce the set of candidates to a singleton. As such, it only serves to guide the selection process.

Example 8.2. Consider the following definition, which contains a type error.

```
showInt :: Int → String
showInt x =
  case x of
    0 → False
    1 → "one"
    2 → "two"
    _ → "many"
```

The right-hand sides of the case alternatives have different types: in the first alternative, the right-hand side is of type *Bool*, but in the other alternatives it is of type *String*. Also the type signature supports the theory that the right-hand side should have type *String*. As a result, our heuristic will suggest to report the expression

False as the anomaly. To give an idea, we could report the following error message for the definition of *showInt*.

```
(4,14): Type error in right-hand side
expression      : False
  type          : Bool
  does not match : String
```

8.2 Constraint number heuristic

The next heuristic we present can be use as a final tie-breaker: this heuristic reduces the number of candidate constraints to one. This is an important task: without such a selection criteria, it would be unclear (even worse: arbitrary) what is reported. We propose a tie-breaker heuristic which considers the position of a constraint in the constraint list.

Section 5.3 contains functions that flatten a constraint tree (created for a program). Flattening a constraint tree results in a list, and for each constraint in this list we record its position in the list. Although the order of the constraints is irrelevant in constructing the type graph, we store it in the constraint information, and use it for this particular heuristic.

For each error path, we take the constraint which completes the path, that is, which has the highest constraint number. This results in a list of constraints that complete an error path, and out of these constraints we pick the one with the lowest constraint number.

A nice property of this heuristic is that, if no other heuristics are considered, we end up reporting the same constraints as a greedy solver would have done.

Example 8.3. Suppose that we have three error paths, and assume that the labels of the constraints correspond to the order of the constraints.

$$\pi_0 = [\#1, \#2, \#3] \qquad \pi_1 = [\#2, \#4, \#5] \qquad \pi_2 = [\#2, \#5, \#6, \#7]$$

The first constraint which is selected for removal is #3, which completes π_0. The error paths π_1 and π_2 will remain after removal of #3. In the second round, constraint #5 is selected, which resolves the other two inconsistencies. Note that this heuristic does not select #2, although this constraint is present in all error paths.

8.3 Trust factor heuristic

The two previous heuristics are generally applicable, and do not depend on the program for which the constraint set was generated. Hence, they can be used for all type graphs. However, it is beneficial to use information about the program in our heuristics. We demonstrate this in the remaining heuristics.

The trust factor heuristic computes a trust factor for each constraint, which reflects the level of trust we have in the validity of a constraint. Constraints with a high trust factor will, if possible, not be used to create an error message. Instead, we choose to report constraints with a low trust factor. The trust factor of a constraint can be a combination of many considerations, and we discuss a number of cases. The list can be extended to further enhance error messaging.

- Some constraints are introduced *pro forma*: they seem to hold trivially. An example is the constraint expressing that the type of a let expression equals the type of the body (see (E-Let) on page 103). Reporting such a constraint as incorrect would be highly inappropriate. Thus, we make this constraint highly trusted by assigning it the highest trust factor possible. As a result, this constraint will not be reported.

  Example 8.4. The following definition is ill-typed because the declared type signature does not match with the type of the body of the let expression.

  $$\begin{aligned}&\mathit{squares} :: \mathit{Int}\\ &\mathit{squares} = \textbf{let}\ f\ i = i * i\\ &\qquad\qquad\ \ \textbf{in}\ \mathit{map}\ f\ [1\,..\,10]\end{aligned}$$

  Dropping the constraint that the type of the let expression equals the type of the body would remove the type inconsistency. However, the high trust factor of this constraint prevents us from doing so. In this case, we select a different constraint, and report, for instance, the incompatibility between the type of *squares* and its right-hand side.

- Some of the constraints restrict the type of a subterm (e.g., the condition of an if-then-else expression must be of type *Bool*), whereas others constrain the type of the complete expression at hand (e.g., the type of a pair is a tuple type). These two classes of constraints correspond very neatly to the unifications (error messages) that are performed (reported) by algorithm $\mathcal{W}$ and algorithm $\mathcal{M}$ respectively. We refer to constraints corresponding to $\mathcal{M}$ as *folklore* constraints. Often, we can choose between two constraints – one which is folklore, and one which is not.

  Example 8.5. In the following definition, the condition should be of type *Bool*, but is of type *String*.

  $$\begin{aligned}&\mathit{test} :: \mathit{Bool} \rightarrow \mathit{String}\\ &\mathit{test}\ b = \textbf{if}\ \texttt{"b"}\ \textbf{then}\ \texttt{"yes!"}\ \textbf{else}\ \texttt{"no!"}\end{aligned}$$

  Algorithm $\mathcal{W}$ detects the inconsistency at the conditional, when the type inferred for `"b"` is unified with *Bool*. This would result in the following error message.

  ```
  (2,13): Type error in conditional
  expression       : if "b" then "yes!" else "no!"
  term             : "b"
    type           : String
    does not match : Bool
  ```

Algorithm $\mathcal{M}$, on the other hand, pushes down the expected type *Bool* to the literal `"b"`, which leads to the following error report.

```
(2,13): Type error in literal
expression      : "b"
  type          : String
  expected type : Bool
```

In our constraint-based system, both error messages are available, and the latter corresponds to a folklore constraint. In our experience, folklore constraints should (preferably) not be reported, since they lack contextual information and are usually harder to understand. Each folklore constraint is assigned a negative trust factor.

- The type of a function imported from the Prelude should not be questioned. At most, such a function is used incorrectly.

  Example 8.6. The arguments of *foldr* should have been supplied in a different order.

  $sumInt :: [Int] \to Int$
  $sumInt\ is = foldr\ 0\ (+)\ is$

  The type of *foldr* in this definition must be an instance of *foldr*'s type scheme, which is $\forall ab.(a \to b \to b) \to b \to [a] \to b$. Relaxing this constraint resolves the type inconsistency. However, we assign a high trust value to this constraint, and thus report a different constraint as the source of error.

- Although not mandatory, type annotations provided by a programmer can guide the type inference process. In particular, they can play an important role in the reporting of error messages. These type annotations reflect the types expected by a programmer, and are a significant clue where the actual types of a program differ from his perception.
  We can decide to trust the types that are provided by a user. In this way, we can mimic a type inference algorithm that pushes a type signature into its definition. Practice shows, however, that one should not rely too much on type information supplied by a novice programmer: these annotations are frequently in error themselves.
- A final consideration for the trust factor of a constraint is in which part of the program the error is reported. Not only types of expressions are constrained, but errors can also occur in patterns, declarations, and so on. Hence, also patterns and declarations can be reported as the source of a type conflict. Whenever possible, we report an error for an expression.

  Example 8.7. In the definition of *increment*, the pattern $(\_ : x)$ (x must be a list) contradicts with the expression $x + 1$ (x must be of type *Int*).

  $increment\ (\_ : x) = x + 1$

  We prefer to report the expression, and not the pattern. If a type signature supports the assumption that x must be of type *Int*, then the pattern can still be reported as being erroneous.

8.4 Program correcting heuristics

A different direction in error reporting is trying to discover what a user was trying to express, and how the program could be corrected accordingly. Given a number of possible edit actions, we can start searching for the closest well-typed program. An advantage of this approach is that we can report locations with more confidence that this is really the position where a correction is required. Additionally, we can equip our error messages with hints how the program might be corrected. However, this approach has a disadvantage too: suggesting program fixes is potentially harmful since there is no guarantee that the proposed correction is the right one (although we can guarantee that the correction will result in a well-typed program). Furthermore, it is not clear when to stop searching for a correction, nor how we could present a complicated correction to a programmer. We present a small overview with some of the possibilities. Later chapters will discuss more program correcting heuristics.

An interesting approach to automatically correcting ill-typed programs is based on a theory of type isomorphisms [40] (see Section 3.3, page 26). Two types are considered isomorphic if they are equivalent under (un)currying and permutation of arguments. Such an isomorphism is witnessed by two morphisms: these are expressions that transform a function of one type to a function of the other type, in both directions. For each ill-typed application, we search for an isomorphism between the type of the function and the type expected by the arguments and the context of that function. We illustrate this idea with a simple example.

Example 8.8. The definition of *squareList* is not well-typed, although we supply a list to *map*, and a function that works on the elements of this list.

$$\begin{array}{l} \mathit{square} :: \mathit{Int} \to \mathit{Int} \\ \mathit{square}\ i = i * i \\[1ex] \mathit{squareList} :: \mathit{Int} \to [\mathit{Int}] \\ \mathit{squareList}\ n = \mathit{map}\ ([1\,..\,n], \mathit{square}) \end{array}$$

To correct the application $\mathit{map}\ ([1\,..\,n], \mathit{square})$, we search for an adapted version of *map* which expects its arguments paired and in reversed order. Consider the following two morphisms between the real type of *map* (on the left), and its expected type (on the right).

$$(a \to b) \to [a] \to [b] \quad \overset{\mu_1 = \lambda f\ (xs, g) \to f\ g\ xs}{\underset{\mu_2 = \lambda f\ g\ xs \to f\ (xs, g)}{\rightleftarrows}} \quad ([a], a \to b) \to [b]$$

In this case, the type variables a and b are both *Int*. Note that applying μ_1 to *map* corrects the type error.

$$\mathit{squareList}\ n = (\mu_1\ \mathit{map})\ ([1\,..\,n], \mathit{square})$$

We can eliminate μ_1 from this definition by beta reduction.

$$\mathit{squareList}\ n = \mathit{map}\ \mathit{square}\ [1\mathinner{\ldotp\ldotp} n]$$

Hence, we can suggest to change the expression to $\mathit{map}\ \mathit{square}\ [1\mathinner{\ldotp\ldotp} n]$.

Another edit operation is to change the structure of an abstract syntax tree by inserting or removing parentheses. Beginning Haskell programmers have a hard time understanding the exact priority and associativity rules for operators and applications. As a result, lots of superfluous parentheses are written, and, every now and then, a pair of parentheses is missing, which often results in a type error. It seems appealing to search in this direction for slightly modified well-typed abstract syntax trees.

Example 8.9. The following definition is not type correct.

$$\begin{array}{l}\mathit{isZero} :: \mathit{Int} \to \mathit{Bool}\\ \mathit{isZero}\ i = \mathit{not}\ i == 0\end{array}$$

In this definition, not of type $\mathit{Bool} \to \mathit{Bool}$ is applied to i, which is (probably) of type Int, because of the declared type. By looking at the constituents of $\mathit{not}\ i == 0$, we learn that $\mathit{not}\ (i == 0)$ is the only well-typed arrangement. Hence, we can suggest the programmer to insert parentheses at these locations. Note that the type signature supports this rearrangement, which increases the confidence that this is the correction we want.

Example 8.10. It is important to realize that not all missing parentheses can be noticed and reported. The suggested corrections are based on types inferred for parts of the program, and do not take semantics into account. We illustrate this with our next example, which attempts to define the faculty function.

$$\begin{array}{l}\mathit{fac} :: \mathit{Int} \to \mathit{Int}\\ \mathit{fac}\ 0 = 1\\ \mathit{fac}\ n = n * \mathit{fac}\ n - 1\end{array}$$

Note that the recursive call should have been $\mathit{fac}\ (n - 1)$: the parentheses correct the definition. Although this function does not compute the faculty of an integer, it is still well-typed. This semantic error will not be detected.

Up to this point, we have seen two program correcting heuristics: the first rearranges the types of functions, the second rearranges the abstract syntax tree of expressions by inserting and removing parentheses. Clearly, a combination of the two would allow us to suggest more complex sequences of edit operations. However, the more complicated our suggestions become, the less likely it is that it makes sense to the programmer.

We conclude this discussion with a last program correcting heuristic. Since the numeric literals are not overloaded, the constants 0 and 0.0 cannot be interchanged – the first is of type *Int*, the second has type *Float*. It may be worthwhile to consider the other variant if such a literal is involved in a type inconsistency.

Example 8.11. In the following definition, the initial value supplied to *foldr* is of the wrong type.

$$\begin{array}{l} \mathit{sumFloat} :: [\mathit{Float}] \to \mathit{Float} \\ \mathit{sumFloat} = \mathit{foldr}\ (+.)\ 0 \end{array}$$

Here, (+.) is an operator which adds two floats. Changing the literal solves the problem. Thus, we report the following.

```
(2,23): Type error in literal
expression      : 0
  type          : Int
  expected type : Float
probable fix    : use the float literal 0.0 instead
```

8.5 Application heuristic

Function applications are often involved in type inconsistencies. Hence, we introduce a special heuristic to improve error messages involving (infix) applications. It is advantageous to have all the arguments of a function available when analyzing a type inconsistency. This contrasts with using an abstract syntax with binary applications. Although choosing for binary applications simplifies type inference, it does not correspond to the way most programmers view their programs.

The heuristic applies to all applications that are involved in a type inconsistency. Note that this is also the point where we could consider some of the other program correcting heuristics, such as permutations of arguments by considering type isomorphisms, or the insertion of parentheses.

The heuristic behaves as follows. First, we try to determine the type of the function. We can do this by inspecting the type graph after having removed the constraint created for the application. In some cases, we can determine the maximum number of arguments that a function can consume. However, if the function is polymorphic in its result, then it can receive infinitely many arguments (since a type variable can always be instantiated to a function type). Here are some functions, and their maximum number of arguments.

expression	*type*	*max arguments*
1	*Int*	0
map	$(a \to b) \to [a] \to [b]$	2
foldr	$(a \to b \to b) \to b \to [a] \to b$	∞

If the number of arguments passed to a function exceeds the maximum, then we can report that too many arguments are given – even without considering the types of the arguments. In the special case that the maximum number of arguments is zero, we report that *it is not a function.*

To conclude the opposite, namely that not enough arguments have been supplied, we do not only need the type of the function, but also the type expected by the context of the application. Using a type graph, this type is easily determined, but for algorithm $\mathcal{W}$, for instance, this is an impossible task. Unless this type is unconstrained, we can determine the exact number of arguments that should be given to the function, as is shown in the next example.

Example 8.12. The following definition is ill-typed: *map* should be given more arguments (or *xs* should be removed from the left-hand side).

$$\begin{array}{l} \mathit{incrementList} :: [\mathit{Int}] \to [\mathit{Int}] \\ \mathit{incrementList}\ \mathit{xs} = \mathit{map}\ (+1) \end{array}$$

At most two arguments can be given to *map*: only one is supplied. The type signature for *incrementList* provides an expected type for the result of the application, which is [*Int*]. Note that the first [*Int*] from the type signature belongs to the left-hand side pattern *xs*. Hence, we can conclude that *map* should be applied to exactly two arguments. We may report that not enough arguments are supplied to *map*, but we can do even better. If we are able to determine the types inferred for the arguments (this is not always the case), then we can determine at which position we have to insert an argument, or which argument should be removed. We achieve this by unification with *holes*.

First, we have to establish the type of *map*'s only argument: $(+1)$ has type $\mathit{Int} \to \mathit{Int}$. Because we are one argument short, we insert one hole (•) to indicate a forgotten argument. (Similarly, for each superfluous argument, we would insert one hole in the function type.) This gives us two cases to consider.

$$\begin{array}{lccccc} \textbf{configuration 1}: & & & & & \\ \quad \mathit{function} & (a \to b) & \to & [a] & \to & [b] \\ \quad \mathit{arguments} + \mathit{context} & \bullet & \to & (\mathit{Int} \to \mathit{Int}) & \to & [\mathit{Int}] \\ & & & & & \\ \textbf{configuration 2}: & & & & & \\ \quad \mathit{function} & (a \to b) & \to & [a] & \to & [b] \\ \quad \mathit{arguments} + \mathit{context} & (\mathit{Int} \to \mathit{Int}) & \to & \bullet & \to & [\mathit{Int}] \end{array}$$

Configuration 1 does not work out, since column-wise unification fails. The second configuration, on the other hand, gives us the substitution $S = [a := \mathit{Int}, b := \mathit{Int}]$. This informs us that our function (*map*) requires a second argument, and that this argument should be of type $S([a]) = [\mathit{Int}]$.

The final technique we discuss attempts to blame one argument of a function application in particular. If such an argument exists, then we put extra emphasis on this argument in the reported error message.

Example 8.13. The expression (-1) is of type *Int*, and can therefore not be used as the first argument of *map*.

$$\begin{array}{l} \mathit{decrementList} :: [\mathit{Int}] \to [\mathit{Int}] \\ \mathit{decrementList}\ \mathit{xs} = \mathit{map}\ (-1)\ \mathit{xs} \end{array}$$

We report the following error message, which focuses on the first argument of *map*.

```
(2,25): Type error in application
expression      : map (−1) xs
function        : map
  type          : (a → b) → [a] → [b]
1st argument    : − 1
  type          : Int
  does not match : a → b
```

8.6 Unifier heuristic

At this point, the reader may have the impression that heuristics always put the blame on a single location. If we have only two locations that contradict, however, then preferring one over another introduces a bias. Our last heuristic illustrates that we can also design heuristics to restore balance and symmetry in error messages. We consider a heuristic which reports multiple program locations with contradicting types. This technique is comparable to the approach suggested by Yang (see Section 3.2 on related work, page 23).

The design of the constraint collecting type rules of Chapter 6 accommodates such a heuristic: at several locations, a fresh type variable is introduced to unify two or more types. We call such a type variable a *unifier*. Examples of unifiers in the type rules include:

- The two branches of a conditional: unifier β in (E-COND).
- All the elements of a list: unifier β_1 in (E-LIST).
- Identifiers bound by the same pattern variable of a lambda abstraction: the unifiers are the type variables from $\mathcal{E}$ in (E-ABS).
- The i^{th} pattern of each function binding ($1 \leqslant i \leqslant m$): unifier β_i in (D-FB).

Our heuristic deals with unifiers in the following way. First, we remove the edges from and to a unifier type variable. Then, we try to determine the types of the program fragments that were equated via this unifier. With these types we create a specialized error message.

Example 8.14. In the following definition, the branches have different types.

$$\textit{test}\ c = \textbf{if}\ c\ \textbf{then}\ [1\,..\,10]\ \textbf{else}\ \texttt{"abc"}$$

Neither of the two branches is more likely to contain the error. (A type signature could change this.) Therefore, we report both branches without indicating which of the two should be changed.

```
(1,10): Type error in branches of conditional
expression  : if c then [1..10] else "abc"
then branch : [1..10]
  type      : [Int]
else branch : "abc"
  type      : String
```

This type error message is unbiased.

Example 8.15. All the elements of a list should be of the same type, which is not the case in f's definition.

$$f\ x\ y = [x, y, id, \texttt{"\textbackslash n"}]$$

In the absence of a type signature for f, we choose to ignore the elements x and y in the error message. By considering how f is applied in the program, we could obtain information about the types of x and y. In our system, however, we never let the type of a function depend on the way it is used. We report that id, which has a function type, cannot appear in the same list as the string `"\n"`.

In the two examples above, the type of the context is also a determining factor, in addition to the branches of a conditional and the elements of a list, respectively. Our last example shows that even if we want to put blame on one of the cases, we can use the other cases for justification.

Example 8.16. The following definition contains a type error.

$$\begin{array}{ll} \mathit{maxOfList} :: [\mathit{Int}] \to \mathit{Int} & \\ \mathit{maxOfList}\ [\,] & = \mathit{error}\ \texttt{"empty list"} \\ \mathit{maxOfList}\ [x] & = x \\ \mathit{maxOfList}\ (x, \mathit{xs}) & = x\ \lq\mathit{max}\lq\ \mathit{maxOfList}\ \mathit{xs} \end{array}$$

A considerable amount of evidence supports the assumption that the pattern (x, xs) in *maxOfList*'s third function binding is in error: the first two bindings both have a list as their first argument, and the explicit type expresses that the first argument of *maxOfList* should be of type $[Int]$. In a special hint we enumerate the locations that support this assumption. In our message, we focus on the pattern (x, xs).

```
(4,11): Type error in pattern of maxOfList
pattern          : (x, xs)
  type           : (Int, [Int])
  does not match : [Int]
  Hint: The first pattern of maxOfList should be of type [Int],
        according to (1,14), (2,11), (3,11).
```

Lastly, observe that the type variables used to instantiate a type scheme serve the same purpose as a unifier. Hence, we could apply the same techniques to improve error reporting for a polymorphic function. For instance, consider the operator $+\!\!+$, which has type $\forall a.[a] \to [a] \to [a]$. If two operands of $+\!\!+$ cannot agree upon the type of the elements of the list, we could report $+\!\!+$ and its type, together with the two candidate types for the type variable a.

9 Directives

Overview. *We propose techniques to influence the behavior of type inference processes, including the error message reporting facility. These techniques take the form of externally supplied type inference directives, precluding the need to make any changes to the compiler. A second advantage is that the directives are automatically checked for soundness with respect to the underlying type system. We show how the techniques can be used to improve the type error messages reported for a specific combinator library. More specifically, how directives can help to generate error messages which are conceptually closer to the domain for which the library was developed. This chapter is based on "Scripting the type inference process" [23].*

Clarity and concision are often lacking in type errors reported by modern compilers. This problem is exacerbated in the case of combinator languages. Combinator languages are a means of defining domain specific languages embedded within an existing programming language, using the abstraction facilities present in the latter. However, since the domain specific extensions are expressed in terms of constructs present in the underlying language, all type errors are reported in terms of the host language, and not in terms of concepts from the domain specific language. In addition, the developer of a combinator library may be aware of various mistakes which users of the library are likely to make: something which can be explained in the documentation for the library, but which cannot be made part of the library itself.

We have identified the following problems that are inherent to commonly used type inference algorithms.

1. *A fixed order of unification.* Typically, the type inferencer traverses a program in a fixed order, and this order strongly influences the reported error site. Moreover, a type inferencer checks type correctness for a given construct, say function application, in a uniform way. However, for some function applications it might be worthwhile to be able to depart from this fixed order. To overcome this problem, it should be possible to override the order in which types are inferred by delaying certain unifications, or by changing the order in which subexpressions are visited.
2. *The size of the mentioned types.* Often, a substantial part of the types shown in a type error message is not relevant to the actual problem. Instead, it only distracts the programmer and makes the message unnecessarily verbose. Preservation of type synonyms where possible reduces the impact of this problem.

3. *The standard format of type error messages.* Because of the general format of type error messages, the content is often not very poignant. Specialized explanations for type errors arising from specific functions would be an improvement for the following reasons. Firstly, domain specific terms can be used in explanations, increasing the level of abstraction. Secondly, depending on the complexity of the problem and the expected skills of the intended users, one could vary the verbosity of the error message.
4. *No anticipation for common mistakes.* Often, the designer of a library is aware of the common mistakes and pitfalls that occur when using the library. The inability to anticipate these pitfalls is regretful. Such anticipation might take the form of providing additional hints, remarks, and suggested fixes that come with a type error.

By way of example, consider the set of parser combinators designed by Swierstra [61], which we believe is representative for (combinator) libraries. Figure 9.1 (a) contains a type incorrect program fragment for parsing a lambda abstraction (see Section 9.1 for a description of the parser combinators). In the example, an expression is either an expression with operators which have the same priority as the boolean *and* operator (*pAndPrioExpr*), or the expression is a lambda abstraction which may abstract a number of variables.[1] The most likely error is indicated in comments in the example itself: the subexpression $<\!\!*$ *pExpr* indicates that an expression, the body of the lambda, should be parsed at this point, but that the result (in this case an *Expr*) should immediately be discarded (as is the case with `"\\"` and `"->"`). As a result, the constructor *Lambda* is only applied to a list of patterns, so that the second alternative in *pExpr* has *Expr* → *Expr* as result type. However, the first alternative of *pExpr* yields a parser with result type *Expr* and here the conflict surfaces for this program.

Consider the type errors reported by Hugs in Figure 9.1 (b), and by GHC in (c). Comparing the two messages with the third message in (d) which was generated using our techniques, we note the following.

- The third message refers to parsers and not to more general terms such as functions or applications. It has become domain specific and can solve problem (3) to a large extent.
- It only refers to the incompatible result types.
- It includes precise location information.

In addition, Hugs displays the unfolded non-matching type, but it does not unfold the type of *pAndPrioExpr*, which makes the difference between the two seem to be even larger. This is an instance of problem (2). Note that if the *Parser* type had been defined using a newtype (or data) declaration, then this problem would not have occurred. However, a consequence of such a definition is that wherever parsers are used, the programmer has to pack and unpack them. This effectively puts the burden on him and not on the compiler.

[1] We assume here that we are dealing with a list of tokens, and not characters, but this is no essential difference.

(a) Type incorrect program

```
data Expr       = Lambda Patterns Expr   -- can contain more alternatives
type Patterns = [Pattern]
type Pattern   = String

pExpr :: Parser Token Expr
pExpr  =  pAndPrioExpr
      <|> Lambda <$ pKey "\\"
                 <*> many pVarid
                  <* pKey "->"
                  <* pExpr   -- wrong combinator is used
```

(b) Hugs, version November 2002

```
ERROR "Example.hs":7 - Type error in application
*** Expression      : pAndPrioExpr <|> Lambda <$ pKey "\\" <*> many pVarid <*
pKey "->" <* pExpr
*** Term            : pAndPrioExpr
*** Type            : Parser Token Expr
*** Does not match  : [Token] → [(Expr → Expr, [Token])]
```

(c) The Glasgow Haskell Compiler, version 5.04.3

```
Example.hs:7:
  Couldn't match Expr against Expr → Expr
    Expected type: [Token] → [(Expr, [Token])]
    Inferred type: [Token] → [(Expr → Expr, [Token])]
  In the expression:
    (((Lambda <$ (pKey "\\")) <*> (many pVarid)) <* (pKey "->")) <* pExpr
  In the second argument of (<|>), namely
    (((Lambda <$ (pKey "\\")) <*> (many pVarid)) <* (pKey "->")) <* pExpr
```

(d) Helium, version 1.1 (type rules extension)

```
(7,6): The result types of the parsers in the operands of <|> don't match
  left parser   : pAndPrioExpr
    result type : Expr
  right parser  : Lambda <$ pKey "\\" <*> many pVarid <* pKey "->" <* pExpr
    result type : Expr → Expr
```

(e) Helium, version 1.1 (type rules extension and sibling functions)

```
(11,13): Type error in the operator <*
  probable fix: use <*> instead
```

Figure 9.1. A program and four type error messages

In the case of GHC, the error message explicitly states what the non-matching parts in the types are. On the other hand, it is not evident that the expected type originates from the explicit type signature for *pExpr*. The expressions in the error message include parentheses which are not part of the original source. It is striking that the same long expression is listed twice, which makes the message more verbose without adding any information.

Note that if, instead of applying the constructor *Lambda* to the result of the parser, we immediately apply an arbitrary semantic function, then the messages generated by Hugs and GHC become more complex. Again, we see an instance of problem (2).

In Figure 9.1 (e) we have the absolute extreme of concision: when our facility for specifying so-called sibling functions is used, the inferencer discovers that replacing <* by the related combinator <*> yields a type correct program. The fact that <*> and <* are siblings is specified by the programmer of the library, usually because his experiences are that these two are often mixed up. This kind of facility helps to alleviate problem (4). Please note that it is better to generate a standard type error here, and to give the "probable fix" as a hint. There is always the possibility that the probable fix is not the correct one, and then the user needs more information.

In the light of the problems just described, we present an approach that can be used to overcome the problems with type error messages for a given library. An important feature of our approach is that the programmer of a such a library need not be familiar with the type inference process as it is implemented in the compiler: everything can be specified by means of a collection of type inference directives, which can then be distributed as part of the combinator library. If necessary, a user of such a library may adapt the directives to his own liking. An additional benefit is that the type inference directives can be automatically checked for soundness with respect to the type inference algorithm present in the compiler. All directives proposed in this chapter are implemented in the Helium compiler [26].

This chapter is organized as follows. After a minimal introduction to the parser combinator library in Section 9.1, we propose solutions for the four problems just identified (Section 9.2), and describe how to specify the necessary type inference directives. In Section 9.3, we explain some technical details that are essential in making this work. Finally, Section 9.4 summarizes the contributions of the directives that are proposed in this chapter.

9.1 Parser combinators

In this section we briefly describe the parser combinators which we use in our examples. Whenever necessary, we explain why the combinators are defined the way they are, especially where this departs from what might be the more intuitive way of defining them. The parser combinators [61] were defined to correspond as closely as possible to (E)BNF grammars, although the complete library provides combinators for many other often occurring patterns.

Consider the Haskell declarations listed in Figure 9.2. The type *Parser s a* describes a parser which takes a list of symbols of type *s* and returns a list of

```
infixl 7 <$> , <$
infixl 6 <*>  , <*
infixr 4 <|>

type Parser s a = [s] → [(a, [s])]

(<$>) :: (a → b)            → Parser s a → Parser s b
(<*>) :: Parser s (a → b) → Parser s a → Parser s b
(<|>) :: Parser s a         → Parser s a → Parser s a
(<$)  :: a                  → Parser s b → Parser s a
(<*)  :: Parser s a         → Parser s b → Parser s a

sym    :: (s → s → Bool) → s   → Parser s s
tok    :: (s → s → Bool) → [s] → Parser s [s]
option :: Parser s a → a → Parser s a
many   :: Parser s a → Parser s [a]

symbol :: Char   → Parser Char Char
token  :: String → Parser Char String
```

Figure 9.2. The parser combinators

possible results (to cope with failing and ambiguous parsings). A result consists of (the semantics of) whatever was parsed at this point, which is of type a, and the remainder of the input.

The main combinators in our language are the operators <*>, <|>, and <$>, and the parser *sym*. The first of these is the sequential composition of two parsers, the second denotes the choice between two parsers. We may recognize terminal symbols by means of the *sym* parser, which takes the symbol to recognize as its parameter. To be able to parse a symbol, we have to be able to test for equality. In Haskell this is usually done by way of type classes, but we have chosen to include the predicate explicitly. In forthcoming chapters we deal with type classes, and explain how these type classes influence the directives proposed in this chapter. We introduce *symbol* for notational convenience, which works on characters.

We now have all we need to implement BNF grammars. For instance, the production $P \to QR \mid a$ might be transformed to the Haskell expression

$$pP = pQ \mathbin{<\!\!*\!\!>} pR \mathbin{<\!|\!>} \mathit{symbol}\ \texttt{'a'}.$$

The type of each of the parsers *pQ* <*> *pR* and *symbol* `'a'` must of course be the same, as evidenced by the type of <|>. The type of the combinator <*> specifies that the first of the two parsers in the composition returns a function which is applied to the result of the second parser. An alternative would be to return the paired results, but this has the drawback that, with a longer sequence of parsers, we obtain deeply nested pairs which the semantic functions have to unpack.

Usually, one uses the <$> combinator to deal with the results of a sequence of parsers:

$$pP = f \mathbin{<\$>} pQ \mathbin{<*>} pR \mathbin{<|>} \mathit{symbol}\ \texttt{'a'},$$

where f is a function which takes a value of the result type of pQ and a second parameter which has the result type of pR, returning a value which should have type *Char* (because of the *symbol* `'a'` parser). The operator <\$> has a higher priority than <*>, which means that the first alternative for <|> should be read as $((f \mathbin{<\$>} pQ) \mathbin{<*>} pR)$. A basic property of <\$> is that it behaves like function application, consuming its arguments one by one. However, this is generally not the way parser builders read their parsers. Usually, the parser is constructed first, and the semantic functions are added afterwards. This difference in perception is one of the sources of confusion when people use the parser combinators. In the following section we describe how our techniques can help to alleviate this difference in perception.

For <\$> and <*>, we have variants <\$ and <*, which discard the result of their right operand. This is useful when what is parsed needs only to be recognized, but does not bear a meaning. An example of this can be found in the code of Figure 9.1 (a), which throws away whatever comes out of either *pKey* application. This means that we can simply apply the constructor *Lambda*, instead of a function which takes four parameters and throws two of them away.

In the same example, we see an application of the *many* combinator which recognizes a list of things, in this case *pVarid*s. Here, *pVarid* is a predefined parser which recognizes an identifier. The *pKey* combinator is used to recognize keywords and reserved operators (which have been tokenized already so that we do not have to deal with whitespace). Finally, the *token* combinator is a combination of *many* and *symbol*, and the *option* combinator is used when a parser can recognize the empty word. For example,

$$\mathit{option}\ (\mathit{token}\ \texttt{"hello!"})\ \texttt{""}$$

either recognizes the string `"hello!"` and returns it, or it succeeds without consuming any input. In the latter case, the *option* parser returns the empty string instead.

9.2 Type inference directives

Type inference directives can be used to easily modify the behavior of the type inference process, and, in particular, the reported type error messages. For a given module `X.hs`, the directives can be found in a file called `X.type`. Although it is not essential that the directives are provided in a different file (they could equally well be written next to the functions and definitions of the Haskell source file), this separation is a reminder that the directives operate at a meta-level: their primary concern is to change the error messages reported by the compiler. If `X.hs` imports a module `Y.hs`, then all the directives in `Y.type` are included as well. This applies transitively to the modules that `Y.hs` imports, which we achieve by storing directive information in object files. The use of compiler directives has a number of advantages.

- They are not part of the programming language, and can be easily switched off.
- The directives function as a high-level specification language for parts of the type inference process, precluding the need to know anything about how type inferencing is implemented in the compiler. In tandem with our automatic soundness and sanity checks for each directive, this makes them relatively easy to use.
- It should be straightforward for other compilers to support some of our directives, although some of the directives benefit from a constraint-based formulation of the type inference process.

We continue with a description of five type inference directives. To start with, we present a notation to define your own type rules with their specialized type error messages. Next, we explain why flexibility in the order of unification is absolutely necessary for getting appropriate messages. We conclude with examples of common mistakes, and discuss how to anticipate these known pitfalls.

9.2.1 Specialized type rules

This section describes how to write specialized type rules, and explains how this influences the error reporting in case a type rule fails. There are serious disadvantages to incorporating these rules directly in a type inferencer. It requires training and experience to extend an existing type inferencer, and it may induce a loss of compositionality and maintainability of the implemented type rule. Since the correctness of a type inferencer is quite a subtle issue, it is no longer possible to guarantee that the underlying type system remains unchanged. Instead, we follow a different approach in which the type rules can only be specified externally. This makes it relatively easy to specify a type rule and to experiment with it without having to change any code of the type inference engine.

Specializing a type rule
Let us take a closer look at a traditional type inference rule for infix application. An infix application is type correct if the types of the two operands fit the type of the operator. Clearly, infix application is nothing but syntactic sugar for normal prefix function application. Applying the type rule for function application twice in succession results in the following.

$$\frac{\Gamma \vdash_{\text{HM}} op : \tau_1 \to \tau_2 \to \tau_3 \qquad \Gamma \vdash_{\text{HM}} x : \tau_1 \qquad \Gamma \vdash_{\text{HM}} y : \tau_2}{\Gamma \vdash_{\text{HM}} x \text{ `}op\text{` } y : \tau_3}$$

Here, $\Gamma \vdash_{\text{HM}} e : \tau$ expresses that under the type environment Γ we can assign type τ to expression e [11]. Instead of using this general type rule to deal with infix applications, one could come up with a more specific type rule for a particular operator, for instance <\$>. Let <\$> be part of the type environment Γ, and have type $(a \to b) \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ b$. Then we can introduce the following specialized type rule.

$$\frac{\Gamma \vdash_{\text{HM}} x : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\text{HM}} y : \mathit{Parser}\ \tau_3\ \tau_1}{\Gamma \vdash_{\text{HM}} x \mathbin{<\$>} y : \mathit{Parser}\ \tau_3\ \tau_2}$$

If we encounter a local definition of <$>, then the above type rule will not be used within the scope of that local definition, even if the new definition of <$> has the same type as the old one. To avoid confusion, we only want to use the type rule if the very same operator, here <$>, is used.

The type rule does not adjust the scope, as can be concluded from the fact that the same type environment Γ is used above and below the line. In the rest of this paper we will only consider specialized type rules with this property. This limitation is necessary to avoid complications with monomorphic and polymorphic types. Since the type environment remains unchanged, we will omit it from now on.

In general, a type rule contains a number of constraints, for each of which a type inferencer may fail. For instance, the inferred types for the two operands of <$> are restricted to have a specific shape (a function type and a *Parser* type), the relations between the three type variables constrain the inferred types even further and, lastly, the type in the conclusion must fit into its context. To obtain a better understanding why some inferred types may be inconsistent with this type rule, let us reformulate the type rule to make the type constraints more explicit.

$$\frac{x : \tau_1 \qquad y : \tau_2}{x \mathbin{<\$>} y : \tau_3} \qquad \begin{cases} \tau_1 \equiv a \rightarrow b \\ \tau_2 \equiv \mathit{Parser}\ s\ a \\ \tau_3 \equiv \mathit{Parser}\ s\ b \end{cases}$$

In addition to the type variables introduced in the type rule, three more type variables are introduced in the constraint set, namely a, b, and s. The order in which the constraints are solved is irrelevant for the success or failure of the type inference process. However, the order chosen does determine where the type inferencer first notices an inconsistency. Typically, the order is determined by the type inference algorithm that one prefers, e.g., the standard bottom-up algorithm $\mathcal{W}$ [11] or the folklore top-down algorithm $\mathcal{M}$ [36]. To acquire additional information, we split up each constraint in a number of more basic type constraints. The idea of these small unification-steps is the following: for a type constraint that cannot be satisfied, the compiler can produce a more specific and detailed error message. The example now becomes

$$\frac{x : \tau_1 \qquad y : \tau_2}{x \mathbin{<\$>} y : \tau_3} \qquad \begin{cases} \tau_1 \equiv a_1 \rightarrow b_1 & s_1 \equiv s_2 \\ \tau_2 \equiv \mathit{Parser}\ s_1\ a_2 & a_1 \equiv a_2 \\ \tau_3 \equiv \mathit{Parser}\ s_2\ b_2 & b_1 \equiv b_2 \end{cases}$$

The definition of a type rule, as included in a `.type` file, consists of two parts, namely a deduction rule and a list of constraints. The deduction rule consists of premises, which occur above the line, and a conclusion below the line. A premise consists of a single meta-variable, which matches with every possible expression, and a type. On the other hand, the conclusion may contain an arbitrary expression, except that lambda abstractions and let expressions are not allowed, because they modify the type environment. There is no restriction on the types in the premises and the conclusion. Below the deduction rule, the programmer can list a number of equality constraints between types. Each of these is followed by a corresponding error message.

Example 9.1. We present a special type rule for the <\$> combinator. Each of the constraints is specified with an error message that is reported if the constraint cannot be satisfied. The order in which the constraints are listed determines the order in which they shall be considered during the type inference process. By convention we write all type inference directives on a light gray background.

$$\frac{x :: t_1; \quad y :: t_2;}{x \mathbin{<\$>} y :: t_3;}$$

$t_1 \equiv a_1 \rightarrow b_1$	: `left operand is not a function`
$t_2 \equiv \mathit{Parser}\ s_1\ a_2$	: `right operand is not a parser`
$t_3 \equiv \mathit{Parser}\ s_2\ b_2$	: `context does not accept a parser`
$s_1 \equiv s_2$	: `parser has an incorrect symbol type`
$a_1 \equiv a_2$	: `function cannot be applied to parser's result`
$b_1 \equiv b_2$	: `parser has an incorrect result type`

Now take a look at the following function definition, which is clearly ill-typed.

```
test :: Parser Char String
test = map toUpper <$> "hello, world!"
```

Because it is pretty obvious which of the six constraints is violated here (the right operand of <\$> is not a parser, hence, $t_2 \equiv \mathit{Parser}\ s_1\ a_2$ cannot be satisfied), the following error is reported.

```
Type error: right operand is not a parser
```

Note that this type error message is still not too helpful since important context specific information is missing, such as the location of the error, pretty-printed parts of the program, and conflicting types. To overcome this problem, we introduce attributes in the specification of error messages.

Error message attributes

A fixed error message for each constraint is too simplistic. The main focus of a message should be the contradicting types that caused the unification algorithm to fail. To construct a clear and concise message, one typically needs the following information.

- *The inferred types of the subexpressions.* One should be able to refer to the actual type of a type variable that is mentioned in either the type rule or the constraint set. In the special case that a subexpression is a single identifier which is assigned a polymorphic type, we prefer to display this generalized type instead of simply using the instantiated type.
- *A pretty-printed version of the expression and its subexpressions.* This should resemble the actual program text as closely as possible, and should (preferably) fit on a single line.

- *Position and range information.* This also includes the name of and the location in the source file at hand.

Example 9.2. To improve the error message of Example 9.1, we replace the annotation of the type constraint

$t_2 \equiv \mathit{Parser}\ s_1\ a_2$: `right operand is not a parser`

by the following error message, which contains attributes.

$t_2 \equiv \mathit{Parser}\ s_1\ a_2$:

```
@expr.pos@: The right operand of <$> should be a parser
  expression       : @expr.pp@
  right operand    : @y.pp@
    type           : @t2@
    does not match : Parser @s1@ @a2@
```

In the error message, the expression in the conclusion is called `expr`. We can access its attributes by using the familiar dot notation, and surrounding it by `@` signs. For example, `@expr.pos@` refers to the position of *expr* in the program source. Similarly, `pp` gives a pretty printed version of the code.

The specification of a type constraint and its type error message is layout-sensitive: the first character of the error report (which is an `@` in the example above) determines the level of indentation. The definition of the error report stops at the first line which is indented less. As a result, the error report for the definition of *test* in Example 9.1 now becomes:

```
(2,21): The right operand of <$> should be a parser
expression       : map toUpper <$> "hello, world!"
right operand    : "hello, world!"
  type           : String
  does not match : Parser Char String
```

For a given expression (occurring in the conclusion of a type rule), the number of type constraints can be quite large. We do not want to force the library designer to write out all these constraints and provide corresponding type error messages. For this reason, the library designer is allowed to move some constraints from the list below the type rule to the type rule itself, as we illustrate in the next example.

Example 9.1 (continued). Because we prefer not to give special error messages for the case that the context does not accept a parser, we may as well give the following type rule.

$$\frac{x :: t_1; \qquad y :: t_2;}{x \mathbin{<\$>} y :: \mathit{Parser}\ s\ b;}$$

$t_1 \equiv a_1 \to b$: `left operand is not a function`
$t_2 \equiv \mathit{Parser}\ s\ a_2$: `right operand is not a parser`
$a_1 \equiv a_2$: `function cannot be applied to parser's result`

At this point, only three of the original type constraints remain. If any of the "removed" constraints contributes to an inconsistency, then a standard error message will be generated. These constraints will be considered before the explicitly listed constraints.

Order of the type constraints

In the type rule specifications we have so far only listed the constraints for that rule and the order in which they should be considered. In principle, we do not assume that we know anything about how the type inferencer solves the constraints. The only thing a type rule specifies is that an error report will be generated for the first constraint of the specialized type rule that cannot be satisfied. Note that we take advantage of this property: an error message for a type constraint is only reported if all the constraints listed before that particular constraint are satisfied.

The situation is not as simple as this. Each of the meta-variables in the rule corresponds to a subtree of the abstract syntax tree for which sets of constraints are generated. How should the constraints of the current type rule be ordered with respect to these constraints?

Example 9.1 (continued). If we want the constraints generated by the subexpression y to be considered after the constraint $t_1 \equiv a_1 \rightarrow b$, then we should change the type rule to:

$$\frac{x :: t_1; \qquad y :: t_2;}{x \text{ <\$> } y :: Parser\ s\ b;}$$

constraints x

$t_1 \equiv a_1 \rightarrow b$: `left operand is not a function`

constraints y

$t_2 \equiv Parser\ s\ a_2$: `right operand is not a parser`

$a_1 \equiv a_2$: `function cannot be applied to parser's result`

Note that in this rule we have now explicitly stated at which point to consider the constraints of x and y. By default, the sets of constraints are considered in the order of the corresponding meta-variables in the type rule, to be followed afterwards by the constraints listed below the type rule. Hence, we could have omitted **constraints** x.

By supplying type rules to the type inferencer we can adapt the behavior of the type inference process. It is fair to assume that the extra type rules should not have an effect on the underlying type system, especially since an error in the specification of a type rule is easily made. We have made sure that user defined type rules that conflict with the default type rules are automatically rejected at compile-time. A more elaborate discussion on this subject can be found in Section 9.3.2.

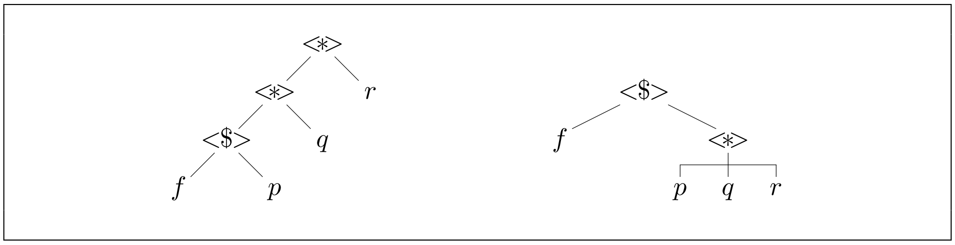


Figure 9.3. Abstract syntax tree (left) compared with the conceptual structure (right)

9.2.2 Phasing

Recall the motivation for the chosen priority and associativity of the <$> and <*> combinators: it allows us to combine the results of arbitrary many parsers with a single function in a way that feels natural for functional programmers, and such that the number of parentheses is minimized in a practical situation. However, the abstract syntax tree that is a consequence of this design principle differs considerably from the view that we expect many users to have of such an expression. Unfortunately, the shape of the abstract syntax tree strongly influences the type inference process. As a consequence, the reported site of error for an ill-typed expression can be counter-intuitive and misleading. Ideally, the type inferencer should follow the conceptual perception rather than the view according to the abstract syntax tree.

Phasing by example
Let f be a function, and let p, q, and r be parsers. Consider the following expression.

$$f \mathbin{<\$>} p \mathbin{<*>} q \mathbin{<*>} r$$

Figure 9.3 illustrates the abstract syntax tree of this expression and its conceptual view. How can we let the type inferencer behave according to the conceptual structure? A reasonable choice would be to treat it as n-ary function application: first we infer a type for the function and all its arguments, and then we unify the function and argument types. We can identify four steps if we apply the same idea to the parser combinators. Note that the four step process applies to the program as a whole.

1. Infer the types of the expressions between the parser combinators.
2. Check whether the types inferred for the parser subexpressions are indeed *Parser* types.
3. Verify that the parser types do agree upon a common symbol type.
4. Determine whether the result types of the parser fit the semantic functions that appear as left operands of <$>.

One way to view the four step approach is that all parser related unifications are delayed. Consequently, if a parser related constraint is inconsistent with another constraint, then the former will be blamed.

Example 9.3. The following example presents a type incorrect attempt to parse a string followed by an exclamation mark.

```
test :: Parser Char String
test = (++) <$> token "hello world"
            <*> symbol '!'
```

The type error message of `Hugs` is not too helpful here.

```
ERROR "Phase1.hs":4 - Type error in application
*** Expression      : (++) <$> token "hello world" <*> symbol '!'
*** Term            : (++) <$> token "hello world"
*** Type            : [Char] → [([Char] → [Char], [Char])]
*** Does not match  : [Char] → [(Char → [Char], [Char])]
```

The four step approach might yield:

```
(1,7): The function argument of <$> does not work for the
  result types of the parser(s)
function          : (++)
  type            : [a] → [a] → [a]
  does not match  : String → Char → String
```

Observe the two major improvements. First of all, it focuses on the problematic function, instead of mentioning an application. Secondly, the types do not involve the complex expanded *Parser* type synonym, nor do they contain the symbol type of the parsers, which in this example is irrelevant information.

Assigning phase numbers

In order to further specify the order in which constraints are to be taken into account, we introduce phase number annotations. The constraints in phase number i are solved before the constraint solver continues with the constraints of phase $i + 1$. Consequently, phasing has a global effect on the type inference process.

Adding the keyword phase, followed by a phase number, will assign the constraints after this directive to this phase. By default, constraints are assigned to phase 5, leaving space to introduce new phases. Of course, the constraints of a single type rule can be assigned to different phases.

Example 9.4. We introduce phases numbered from 6 to 8 for the steps *2*, *3*, and *4* respectively. We assign those phase numbers to the constraints in the specialized type rule for <$>. Note that step *1* takes place in phase 5, which is the default. No constraint generated by the following type rule will be solved in phase 5.

$$\frac{x :: t_1; \qquad y :: t_2;}{x \mathbin{<\$>} y :: t_3;}$$

phase 6
$t_2 \equiv \mathit{Parser}\ s_1\ a_2$: `right operand is not a parser`
$t_3 \equiv \mathit{Parser}\ s_2\ b_2$: `context does not accept a parser`
phase 7
$s_1 \equiv s_2$: `parser has an incorrect symbol type`
phase 8
$t_1 \equiv a_1 \rightarrow b_1$: `left operand is not a function`
$a_1 \equiv a_2$: `function cannot be applied to parser's result`
$b_1 \equiv b_2$: `parser has an incorrect result type`

One may wonder what happens when the sets **constraints** x and **constraints** y are included among the listed constraints. Because phasing is a global operation, the constraints in these sets keep their own assigned phase number.

Sometimes the opposite approach is desired: to verify the correctness of the parser related unifications before continuing with the rest of the program. This technique is similar to pushing down the type of a type declaration as an expected type, a useful technique applied by, for instance, the GHC compiler.

Example 9.5. Let us take another look at the ill-typed function definition *test* from Example 9.1.

```
test :: Parser Char String
test = map toUpper <$> "hello, world!"
```

If the constraints introduced by the type rule for <\$> are assigned to an early phase, e.g. 3, then, effectively, the right operand is imposed to have a *Parser* type. Since `"hello, world!"` is of type *String*, it is at the location of this literal that we report that a different type was expected by the enclosing context. By modifying the `.type` file along these lines, we may obtain the following error message.

```
(2,21): Type error in string literal
expression      : "hello, world!"
  type          : String
  expected type : Parser Char String
```

9.2.3 Sibling functions

The following three sections deal with anticipating common mistakes. Although some mistakes are made over and over again, the quality of the produced error reports can be unsatisfactory. In some cases it is possible to detect that a known pitfall resulted in a type error message. If so, a more specific message should be presented, preferably with hints on how to solve the problem.

One typical mistake that leads to a type error is mixing two functions that are somehow related. For example, novice functional programmers have a hard time remembering the difference between inserting an element in front of a list (:), and concatenating two lists (++). Even experienced programmers may mix up the types of *curry* and *uncurry* now and then. Similar mistakes are likely to occur in the context of a combinator language. We will refer to such a pair of related functions as *siblings*. The idea is to suggest the replacement of a function with one of its sibling functions if this resolves the type error. The types of all siblings should be distinct, since we cannot distinguish the differences based on their semantics.

Example 9.6. Consider the parser combinators from Section 9.1, and, in particular, the special variants that ignore the result of the right operand parser. These combinators are clearly siblings of their basic combinator. A closer look to the program in Figure 9.1 (a) tells us that the most likely source of error is the confusion over the combinators <*> and <*. The observation that replacing one <* combinator by <*> results in a type correct program paves the way for a more appropriate and considerably simpler error message.

A function can have multiple sibling functions, but the sibling relation is not necessarily transitive. Furthermore, a sibling function should only be suggested if replacement completely resolves a type error. Moreover, the suggested function should not only match with its arguments, but it should also fit the context to prevent misleading hints. Implementing this in a traditional type inference algorithm can be quite a challenge. A practical concern is the runtime behavior of the type inferencer in the presence of sibling functions. Ideally, the presence of sibling functions should not affect the type inference process for type correct programs: we only perform some extra computation for type incorrect programs, and only for those operators that contribute directly to the type error. With type graphs, introduced in Chapter 7, we can handle sibling functions in an elegant way: to test whether replacing a function with one of its siblings removes a type inconsistency, we first remove the constraint associated with the function (which corresponds to one edge in the type graph, and all edges derived from it). Subsequently, we test whether adding the constraint associated to the sibling results in a consistent state.

A set of sibling functions can be declared in the file containing the type inference directives by giving a comma separated list of functions.

```
siblings <$> , <$
siblings <*> , <*
```

The type error that is constructed for the program in Figure 9.1 (a) can be found in Figure 9.1 (e). A more conservative approach would be to show a standard type error message, and add the *probable fix* as a hint.

The idea of identifying common programming mistakes was also the motivation for the program correcting heuristics introduced in Section 8.4 (page 146). The fundamental difference between these heuristics and the sibling directives is that the former captures only general mistakes, whereas sibling functions (and directives in general) help to catch mistakes based on domain-specific knowledge.

9.2.4 Permuted arguments

Another class of problems is the improper use of a function, such as supplying the arguments in a wrong order, or mistakenly pairing arguments. McAdam discusses the implementation of a system that tackles these problems by unifying types modulo linear isomorphism [40] (see related work on page 26). Although we are confident that these techniques can be incorporated into our own system, we limit ourselves to a small subset, that is, permuting the arguments of a function.

Example 9.7. The function *option* expects a parser and a result that should be returned if no input is consumed. But in which order should the arguments be given? Consider the following program and its type error message.

```
test :: Parser Char String
test = option "" (token "hello!")
```

```
ERROR "Swapping.hs":2 - Type error in application
*** Expression     : option "" (token "hello!")
*** Term           : ""
*** Type           : String
*** Does not match : [a] → [([Char] → [(String, [Char])], [a])]
```

The error message does not guide the programmer in fixing his program. In fact, it assumes the user knows that the non-matching type is equal to *Parser a (Parser Char String).*

We can design directives to specify for each function whether the type inferencer should attempt to resolve an inconsistency by permuting the arguments to the function. However, we have chosen to include permutation of arguments as default behavior. A conservative type error message for the program of Example 9.7 would now be:

```
(2,8): Type error in application
expression       : option "" (token "hello!")
term             : option
  type           : Parser a b → b → Parser a b
  does not match : String → Parser Char String → c
probable fix     : flip the arguments
```

If, for a given type error, both the method of sibling functions and permuted arguments can be applied, then preference is given to the former.

In a class room setting, we have seen that the permuted arguments facility gives useful hints in many cases. However, we are aware that sometimes it may result in misleading information from the compiler. During a functional programming course we have collected information to determine how often this occurs in a practical setting. The data remains to be analyzed.

9.2.5 Repair directive

Declaring sibling functions and considering permutations of function arguments both serve the same purpose: if an expression is ill-typed, we have a collection of possibilities to try to correct the problem. Clearly, one can think of various other program corrections besides sibling functions and permuted arguments. We present a *repair* directive to declare suggestions how to correct an ill-typed expression. This directive is a generalization of the sibling and permutation directives.

Example 9.8. Take a look at the following *repair* directives.

```
repair p <*> q ⇒ p <* q        : use <* instead
repair f a b   ⇒ f b a         : flip the arguments
repair f a b   ⇒ f (a, b)      : pair the arguments
```

Each directive consists of two expressions (separated by an $\Rightarrow$) and a message, which is included as a hint if the directive removes the type inconsistency. The first directive is an alternative way to formulate that <*> and <* are sibling functions: except that this directive works one-way only, and is considered only if the combinator <*> is applied to two arguments. In this directive, p and q are meta-variables, which represent arbitrary expressions. The second repair directive suggests to repair a type error by flipping the arguments of a binary function. The last directive considers pairing two arguments of a function, i.e., currying a function.

Many useful program corrections can be captured by this directive, ranging from general transformations (such as flipping arguments and currying functions) to domain-specific modifications (such as declaring sibling functions). Although many small mistakes are covered by the repair directives, there are two enhancements to increase the expressiveness of these directives. Firstly, we consider composing declared repair directives to obtain complex program transformations to correct a program. This is an attractive extension since many more ill-typed programs can be corrected automatically. For instance, composing the second and third directive from Example 9.8 gives us the directive **repair** $f\ a\ b \Rightarrow f\ (b, a)$ for free. On the other hand, the more complex corrections become, the more unlikely it is that such a correction makes sense for the programmer. Furthermore, more than one correction may be available, and it is unclear how to explain composed program transformations to a user. A second enhancement of the repair directives is to rely on types instead of syntax. For example, the expressions $(f\ a\ b)$ and $(f\ a\ \$\ b)$ are closely related, but the abstract syntax trees differ considerably. As a result, directives for these two expressions are not interchangeable, although this would be desirable.

9.3 Implementation

This section focuses on some technical details that are required to implement the proposed type inference directives in a constraint-based setting. We briefly discuss

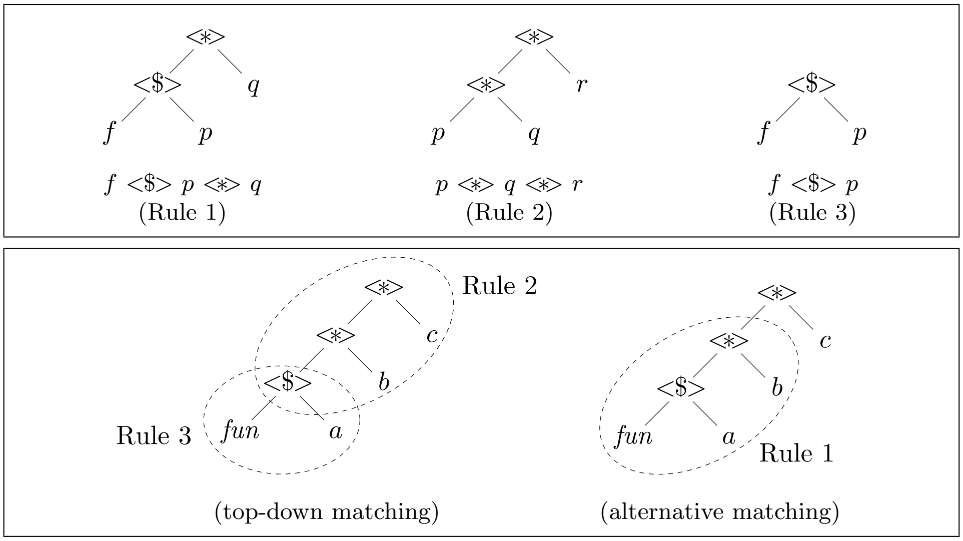


Figure 9.4. Applying specialized type rules

in which way the specialized type rules are applied, and how we check soundness for these rules. We continue with a description of the machinery we use for our program correcting directives (i.e., siblings, permutations, and repair).

9.3.1 Applying specialized type rules

In Section 9.2.1 we introduced notation to define specialized type rules for combinator libraries. Typically, a set of type rules is given to cover the existing combinators and possibly some more complex combinations of these. Since we do not want to forbid overlapping patterns in the conclusions of the type rules, we have to be more specific about the way we apply type rules to a program.

The abstract syntax tree of the program is used to find matching patterns. We look for fitting patterns starting at the root of this tree, and continue in a top-down fashion. In case more than one pattern can be applied at a given node, we select the type rule which occurs first in the `.type` file. Consequently, nested patterns should be given before patterns that are more general. This first-come first-served way of dealing with type rules also takes place when two combinator libraries are imported: the type rules of the combinator library which is imported first have precedence.

Example 9.9. Matching the patterns on the abstract syntax tree of the program involves one subtle issue. Consider type rules for the expressions f <$> p <*> q, p <*> q <*> r, and f <$> p, where f, p, q, and r are meta-variables. These three type rules correspond to Rule 1, Rule 2, and Rule 3, respectively, which are listed in Figure 9.4. What happens if we apply these rules to the code fragment

$$\mathit{test} = \mathit{fun} \mathbin{<\$>} a \mathbin{<\!\ast\!>} b \mathbin{<\!\ast\!>} c.$$

At first sight, Rule 1 seems to be a possible candidate to match the right-hand side of *test*. However, following the chosen priority and associativity of the operators, the second rule matches the top node of the abstract syntax tree, which is the rightmost <*>. Meta-variable p in Rule 2 matches with the expression *fun* <\$> *a*. Since this subexpression matches Rule 3, we apply another type rule. Figure 9.4 shows that our top-down matching policy leads to applying the second and third specialized type rule. Note that a different matching strategy could apply Rule 1, as indicated in Figure 9.4 (on the right-hand side).

9.3.2 Soundness of specialized type rules

Chapter 6 presents rules to collect type constraints. The type rules that form the core of this type system were proven correct in Chapter 4. To preserve correctness of the type system, we verify that specialized type rules do not change the underlying type system before they are used. Such a feature is essential, because a mistake is easily made. We do this by ensuring that for a given expression in the deduction rule, the constraints generated by the type system and the constraints generated from the specialized type rule are equivalent (under entailment).

A specialized type rule allows us to influence the order in which constraints are solved. The order of solving constraints is irrelevant except that the constraints for let definitions should be solved before continuing with the body. If we make sure that all our specialized type rules respect this fact, then the correctness of the new type rules together with the underlying type system is guaranteed.

Procedure to validate a specialized type rule

A type rule is validated in two steps. In the first step, a type rule is checked for various restrictions. The (expression) variables that occur in the conclusion can be divided into two classes: the variables that are present in a premise, which are the meta-variables, and the variables that solely occur in the conclusion. Each meta-variable should occur exactly once in the conclusion, and it should not be part of more than one premise. Every other variable in the conclusion should correspond to a top-level function inside the scope of the type rule's module. The type signature of such a function should be known a priori.

If none of the restrictions above is violated, then we continue with the second step. Here, we test if the type rule is a specialized version of the standard type rules present in the type system. Two types are computed: one for which the type rule is completely ignored, and one type according to the type rule. The type rule will be added to the type system if and only if these types are equivalent (up to the renaming of type variables). Before we discuss how to check the soundness of a type rule, we present an example of an invalid type rule.

Example 9.10. Take a look at the following type rule.

$$\frac{x :: t_1; \qquad y :: t_2;}{x \mathbin{<\$>} y :: \mathit{Parser}\ s\ b;}$$

$t_1 \equiv a_1 \to b$: `left operand is not a function`
$t_2 \equiv \mathit{Parser}\ s\ a_2$: `right operand is not a parser`

Because the programmer forgot to specify that x should work on the result type of y, the type rule above is not restrictive enough. Thus, it is rejected by the type system with the following error message.

```
The type rule for x <$> y is not correct
  the type according to the type rule is
    (a → b, Parser c d, Parser c b)
  whereas the standard type rules infer the type
    (a → b, Parser c a, Parser c b)
```

Note that the 3-tuple in this error message lists the type of x, y, and $x \mathbin{<\$>} y$, which reflects the order in which they occur in the type rule.

To check the soundness of a specialized type rule, we first ignore the type rule, and use the default type inference algorithm. Let Γ be the current type environment containing the types of "known" functions, such as <\$>. All meta-variables from the type rule are added to Γ, each paired with a fresh type variable. For the expression e in the conclusion, the type inference algorithm returns a type τ and a substitution S such that $S\Gamma \vdash_{\mathrm{HM}} e : S\tau$. Let ϕ be $(S\beta_1, \ldots, S\beta_n, S\tau)$, where β_i is the fresh type variable of the ith meta-variable.

Example 9.10 (continued). Let $e = x \mathbin{<\$>} y$, and construct a type environment $\Gamma = \{<\$> : \forall abs.(a \to b) \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ b,\ x : \beta_1,\ y : \beta_2\}$. Given Γ and e, the default type inference algorithm returns a most general unifier S and a type τ.

$$S = [\beta_1 := \beta_3 \to \beta_4, \beta_2 := \mathit{Parser}\ \beta_5\ \beta_3] \qquad \tau = \mathit{Parser}\ \beta_5\ \beta_4$$

As a result we find that $\phi = (\beta_3 \to \beta_4, \mathit{Parser}\ \beta_5\ \beta_3, \mathit{Parser}\ \beta_5\ \beta_4)$.

In the next step, we use the specialized type rule (and ignore the standard type system). Let Γ' be the type environment containing all top-level definitions of which the type is known at this point, and let $\mathcal{C}$ be the set of type constraints declared in the type rule. Compute a most general substitution S that satisfies $\mathcal{C}$, and let ψ be $(S\tau_1, \ldots, S\tau_n, S\tau)$, where τ_i is the type of the i^{th} meta-variable, and τ is the type of the conclusion. The specialized type rule is consistent with the default type system if and only if ϕ and ψ are equivalent up to variable renaming.

Example 9.10 (continued). Let Γ' be a type environment that contains <\$> with its type scheme, and let $\mathcal{C}$ be the two constraints specified in the type rule. We compute $S = [t_1 := a_1 \to b, t_2 := \mathit{Parser}\ s\ a_2]$, and, consequently, this gives us

$\psi = (a_1 \to b, \mathit{Parser}\ s\ a_2, \mathit{Parser}\ s\ b)$. Because $\phi \neq_\alpha \psi$ (β_3 in ϕ corresponds to both a_1 and a_2 in ψ), we reject the examined type rule.

In our implementation, the constraints implied by a specialized type rule "replace" the constraints that would have been collected by the standard type system. In theory, there is no need to reject a type rule that is more specific than the default type rules, as this will not make the type system unsound. Rejecting more restrictive type rules has the advantage that the same set of programs is accepted when using specialized type rules: we only modify the type error messages that are reported for ill-typed programs. Alternatively, we could accept more restrictive type rules, which leads to the rejection of more programs.

Example 9.11. Although the following specialized type rule is sound, it is rejected because it is too restrictive in the symbol type. It is important to realize that applying the type rule to expressions of the form x <\$> y is not influenced by the type mentioned in the type rule, here *Parser Char b*. In this example, the set of type constraints is empty.

$$\frac{x :: a \to b \qquad y :: \mathit{Parser}\ \mathit{Char}\ a;}{x \mathbin{<\$>} y :: \mathit{Parser}\ \mathit{Char}\ b;}$$

Phasing and let expressions

Phasing the type inference process gives a great degree of freedom to order the constraints, because it is a global operation. However, this freedom is restricted by the correct treatment of let-polymorphism. The type scheme of an identifier defined in a let should be computed before it can be instantiated for occurrences of that identifier in the body. This imposes a restriction on the order since the constraints corresponding to the let-definition should be solved earlier than the constraints that originate from the body. Recall that in the TOP framework, this restriction is enforced by introducing strict nodes in the constraint trees. Moreover, the assignment of phase numbers to constraints can be encoded directly into the constraint tree using *Phase i* $\mathcal{T}_\mathcal{C}$ nodes (see page 86).

Example 9.12. Consider the following function.

```
maybeTwice =
   let p = map toUpper <$> token "hello"
   in option ((++) <$> p <*> p) []
```

A type scheme should be inferred for p, before we can infer a type for *maybeTwice*. Therefore, all type constraints that are collected for the right-hand side of p are considered before the constraints of the body of the let, thereby ignoring assigned phase numbers. Of course, phasing still has an effect inside the local definition as well as inside the body. Note that if we provide an explicit type declaration for p, then we do not have to separate the constraint sets of p and the body of the let.

9.3.3 Implementation of repair directives

To implement the repair directives effectively, we rely on the type graphs discussed in Chapter 7. To test whether a directive **repair** $e_1 \Rightarrow e_2$ removes a type inconsistency in a given expression e, we start by identifying locations in e that match with e_1 (in a way similar to matching specialized type rules). The (equality) constraints generated for the part of e that matches with e_1 are removed temporarily, except for those constraints corresponding to a meta-variable of e_1. We insert equality constraints in the type graph corresponding to e_2, and we inspect the type graph. If the type graph is in a consistent state, then the repair directive can successfully remove the type error. We give an example to illustrate this procedure.

Example 9.13. Let *square* have type $Int \to Int$. The following expression is ill-typed since the two arguments of *map* are supplied in the wrong order.

$$map\ [1\,..\,5]\ ((>10) \circ square)$$

We use the type rules given in Chapter 6 to collect type constraints for this expression. The following table shows the constraints collected for this expression and three of its subexpressions (under the assumption that we spread the constraints for the standard functions, such as *map*, downwards before we flatten the constraint tree).

expression	*type*	*constraints*
map	v_1	$\mathcal{C}_1 = \{v_1 \preceq \forall ab.(a \to b) \to [a] \to [b]\}$
$[1\,..\,5]$	v_4	$\mathcal{C}_2 = \{v_4 \equiv [Int], \ldots\}$
$(>10) \circ square$	v_{12}	$\mathcal{C}_3 = \{\ldots\}$
$map\ [1\,..\,5]\ ((>10) \circ square)$	v_{13}	$\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{v_1 \equiv v_4 \to v_{12} \to v_{13}\}$

The constraints in $\mathcal{C}_3$ assign the type $Int \to Bool$ to v_{12}. The complete set of constraints is inconsistent.

Consider the directive **repair** $f\ a\ b \Rightarrow f\ b\ a$. This directive matches with our ill-typed expression: we associate f with *map*, a with $[1\,..\,5]$, and b with $(>10) \circ square$. First, we remove $v_1 \equiv v_4 \to v_{12} \to v_{13}$ because this constraint was generated for the application that matches the repair directive. Next, we collect constraints for $f\ b\ a$ (the right-hand side of the repair directive), under the assumption that f, a, and b have types v_1, v_4, and v_{12}, respectively. These assumptions arise from the types assigned to the expressions that match our meta-variables. Moreover, we take care that v_{13} is the type assigned to $f\ b\ a$. This yields the constraint $v_1 \equiv v_{12} \to v_4 \to v_{13}$, which we add to the type graph. Because the modifications make the type graph consistent, we conclude that this repair directive resolves the type inconsistency.

But how can we determine which repair directives to consider, and at which locations in the source program? General repair directives, such as **repair** $f\ a\ b \Rightarrow f\ b\ a$, are likely to match at many positions in the abstract syntax tree, and analyzing all these potential error locations can become computationally expensive. One solution is to analyze where the directives match the shape of the abstract syntax tree before the type constraints are collected. The constraint information carried by each of the type constraints can be used to store which patterns are applicable at the location where the constraint is generated. The error path of a type graph witnesses a type inconsistency, and only these constraints contribute directly to the inconsistency. Hence, we only have to consider the directives stored in these constraints.

9.3.4 Awareness of skolem variables

We have to be conservative when it comes to suggesting program fixes to a user: misleading a programmer is worse than giving no hint at all. Because we cannot be aware of the intentions of a programmer, there is always a possibility of reporting hints that distract the programmer's attention from the actual problem. However, it is fair to expect that following a hint does indeed remove a type inconsistency.

In Section 5.2.5, we proposed a technique to fake skolemization of type schemes. We have to make sure this technique does not interfere with our program correcting heuristics. A heuristic corrects a type problem only if no deductions about skolem type variables are required. The next example illustrates the problem.

Example 9.14. The operator (!!) has the type $[a] \to \mathit{Int} \to a$, and, as a result, f's definition is not well-typed.

$f :: [a] \to \mathit{Int}$
$f\ \mathit{xs} = 0\ !!\ \mathit{xs}$

Consider swapping the operands of (!!) to correct the problem. This transformation seems to be a step in the right direction, but it does not solve the problem. Because f is expected to return a value of type *Int*, the list *xs* should have type $[\mathit{Int}]$. Hence, this "correction" leads to a type more specific than the signature of f, and is thus invalid. If we fake skolemization and introduce a type variable for a in f's type signature, we have to recognize that this type variable is special (it is a skolem constant), and cannot be unified with *Int*.

In the next chapter, we extend the framework to include type class predicates for overloading, and a similar problem arises because of this extension. Suppose we have assigned f the type $v_0 \to \mathit{Int} \to \mathit{Bool}$, and that this particular f appears in the application f 42 *True*. Clearly, this application is not well-typed. Swapping the arguments 42 and *True* removes the inconsistency. However, if we have the type class predicate $(\mathit{Num}\ v_0)$, then changing the order of arguments leads to $(\mathit{Num}\ \mathit{Bool})$. Under the assumption that booleans are not in the *Num* type class, this does not remove the problem at all. Hence, suggesting this program correction would be inappropriate.

9.4 Summary

This chapter proposes type inference directives to externally modify the behavior of the type inference process, and to improve the quality of type error messages in particular. The major advantages of our directives are the following.

- Type directives are supplied externally. As a result, no detailed knowledge of the implementation of the type inference process is required.
- Type directives can be concisely and easily specified by anyone familiar with type systems. Consequently, experimenting effectively with the type inference process becomes possible.
- For combinator libraries in particular, it becomes possible to report error messages which correspond more closely to the conceptual domain for which the library was developed.
- The specialized type rules are automatically checked for soundness. The major advantage here is that the underlying type system remains unchanged, thus providing a firm basis for the extensions.
- Repair directives help a compiler to suggest fixes for ill-typed programs. These directives can describe general program transformations, such as permuting function arguments, as well as domain-specific corrections.

We have shown how to write type inference directives for a parser combinator library. Of course, our techniques can be applied equally well to other libraries, including the standard Haskell Prelude. Constructing type inference directives for functions from the Prelude may assist beginners in using higher-order polymorphic functions, such as *map*, *filter*, and *foldr*.

10

Overloading and type classes

Overview. *We extend the framework* Top *with type classes to support overloading of identifiers. After a general introduction to qualified types, we add new constraints to the framework. New type rules are presented to collect type constraints for overloaded language constructs, and we explain how overloading can be resolved on a global level. We conclude with a discussion on improving substitutions, and dependency qualifiers are used to illustrate this concept. Dependency qualifiers provide an alternative solution for overloading.*

A polymorphic function is defined only once, and works on values of different types in a uniform manner. An overloaded function, on the contrary, can be used for a limited set of types, but its behavior is defined uniquely for each of the types in this set. The way to overload identifiers in Haskell is to make use of type classes: all member types of such a class provide an implementation for the overloaded member functions.

For instance, consider the operator (==) to test for equality. Assigning it the type $\forall a.a \to a \to \mathit{Bool}$ would be inappropriate, and a meaningful implementation with this general type cannot be given. Besides, this type suggests that we can use it to test two functions for equality: if this is possible at all, then it is unlikely to be useful. Therefore, we assign the type $\forall a.\mathit{Eq}\ a \Rightarrow a \to a \to \mathit{Bool}$ to the equality operator, to express that the type we choose for a must be a member of the *Eq* type class. In this type scheme, $\mathit{Eq}\ a$ is a qualifier, which restricts polymorphism for the equality function. We can make *Int* a member of the *Eq* type class by providing a function which tests two integer values for equality (e.g., using the primitive function *primEqInt* of type $\mathit{Int} \to \mathit{Int} \to \mathit{Bool}$). We can also add *all* list types to *Eq*, provided that the type of the elements is also member of *Eq*.[1] Note that because of these instance declarations not only *Int* and [*Int*] are in *Eq*, but also [[*Int*]] (etcetera).

In Haskell, type classes are defined via class declarations and instance declarations. For each class declaration, an implicit new datatype is created, which contains the member functions of the type class. Such a datatype is called a dictionary. Each instance declaration is translated into such a dictionary, or into a function from dictionaries to a dictionary. Overloaded functions are passed dictionaries as implicit extra arguments, which are provided by the compiler. Type information

[1] This requires a function of the type $\forall a.(a \to a \to \mathit{Bool}) \to [a] \to [a] \to \mathit{Bool}$.

from the type inference process is used to resolve overloading by determining which dictionary to insert or to construct, and at which point.

Type classes have proven to be extraordinary convenient in practice, and their effects are well-understood nowadays [3, 48, 33]. Various implementations follow the Haskell 98 standard [49], which is a rather conservative design. A survey paper by Peyton-Jones, Jones, and Meijer [50] explores various useful extensions to the standard, for example, whether or not to allow overlapping instances. Another proposed extension is to allow multi-parameter type classes. Although this is useful in practice, it often leads to ambiguities. To cope with this, functional dependencies (known from relational databases) were introduced by Jones [34]. Duck et al. [13] have shown that type inference with functional dependencies is sound and decidable under certain conditions. In this chapter, however, we limit ourselves to type classes conform the Haskell 98 standard.

This chapter is organized as follows. We start with an introduction to qualified types (and type class predicates in particular). New type constraints are introduced in Section 10.3 for describing qualifiers. In Section 10.4, we extend the type inference framework to handle these new constraints. Next, we adjust the type rules that collect type constraints for the overloaded language constructs (Section 10.5). We conclude this chapter with improving substitutions for qualifiers, and show how this works out for dependency qualifiers.

10.1 Qualified types

A qualified type is constructed from a type and a context containing predicates (or qualifiers). This context places restrictions on certain type variables. For instance, the qualified type scheme $\forall a.Eq\ a \Rightarrow [a] \to [a]$ restricts a to the members of the type class Eq. Similarly, $\forall ab.(a \equiv b) \Rightarrow a \to b$ could be an alternative formulation for the type of the identity function. We extend the type language to include contexts of qualifiers. The symbol π denotes a predicate.

$$\begin{array}{lll} P & ::= (\pi_1, \ldots, \pi_n) & \textit{(qualifier context)} \\ \sigma & ::= \forall \overline{a}.P \Rightarrow \tau & \textit{(qualified type scheme)} \end{array}$$

We write $\forall \overline{a}.\tau$ as a short-hand notation for $\forall \overline{a}.() \Rightarrow \tau$, and we omit parentheses for a singleton qualifier context. A context should be interpreted as a set of predicates: we use parentheses instead of set-notation to stay as close as possible to the concrete syntax of a type class context in Haskell.

We define an entailment relation on sets of qualifiers: P entails Q ($P \Vdash_e Q$) implies that if we have the predicates in P, we can infer those in Q. For convenience, we write $\pi \Vdash_e Q$ instead of $\{\pi\} \Vdash_e Q$ (similar notation is used if Q is a singleton), and $\Vdash_e P$ for $\emptyset \Vdash_e P$. Although the entailment relation depends on the kind of qualifiers we have at hand, three properties hold for qualifiers in general. Rules for these properties are listed in Figure 10.1. The entailment relation is monotonic (Mono). Hence, we also have $P \Vdash_e \emptyset$, $P \Vdash_e P$, and $P \Vdash_e \pi$ if $\pi \in P$. Furthermore, the entailment relation is transitive (Trans), and closed under substitution (Closed).

$\boxed{P \Vdash_e P}$ *Entailment relation*

$$\frac{P \supseteq Q}{P \Vdash_e Q}\ (\textsc{Mono}) \qquad \frac{P \Vdash_e Q \quad Q \Vdash_e R}{P \Vdash_e R}\ (\textsc{Trans}) \qquad \frac{P \Vdash_e Q}{SP \Vdash_e SQ}\ (\textsc{Closed})$$

Figure 10.1. Basic rules for qualifier entailment

$\boxed{\sigma < \sigma}$ *Instance-of relation*

Type rules of Figure 2.2 (page 13)

$$\frac{P \Vdash_e Q}{(P \Rightarrow \tau) < (Q \Rightarrow \tau)}\ (\textsc{Sub-Qual})$$

Figure 10.2. Instance-of relation on qualified type schemes

The instance-of relation is lifted to work on qualified type schemes, and for this we use the entailment relation presented earlier. The rule (Sub-Qual), shown in Figure 10.2, is added to the three rules. Because $P \Vdash_e Q$ if $P \supseteq Q$, it follows that adding more qualifiers to a context results in a more specific type. Because of this new type rule, (Sub-Mono) has become superfluous (by taking P and Q empty).

Example 10.1. Assume that $\mathit{Ord}\ a \Vdash_e \mathit{Eq}\ a$ and $\Vdash_e \mathit{Eq}\ \mathit{Int}$. Then we can order the following qualified type schemes according to the instance-of relation.

$$\mathit{Int} \to \mathit{Int} \quad < \quad \forall a.\mathit{Ord}\ a \Rightarrow a \to a \quad < \quad \forall a.\mathit{Eq}\ a \Rightarrow a \to a \quad < \quad \forall a.a \to a$$

10.2 Type class qualifiers

In this section, we introduce type class qualifiers, and fit these into our qualifier framework. We limit ourselves to single parameter type classes as in the Haskell 98 standard. The type class qualifiers are of the following form (where C denotes the name of a type class).

$$\pi ::= C\ \tau \qquad \textit{(type class qualifier)}$$

A type class is defined by one class declaration, and a number of instance declarations. We introduce some notation to denote these declarations.

$$\mathit{Class}\ P \Rightarrow \pi \qquad \mathit{Inst}\ P \Rightarrow \pi$$

All superclasses of a class are listed in its class declaration. The class declaration $\mathit{Class}\ \mathit{Eq}\ a \Rightarrow \mathit{Ord}\ a$, for instance, states that *Eq* is the only superclass of *Ord*, which means that each type in *Ord* must also be present in *Eq*. (The arrow in a class declaration should definitely not be interpreted as implication.) With instance

$\boxed{P \Vdash_e P}$ *Entailment relation*

$$\frac{P \Vdash_e \pi_1 \qquad \pi_2 \in Q \qquad (\mathit{Class}\ Q \Rightarrow \pi_1) \in \Gamma}{P \Vdash_e \pi_2}\ (\textsc{Super}) \qquad \frac{P \Vdash_e Q \qquad (\mathit{Inst}\ Q \Rightarrow \pi) \in \Gamma}{P \Vdash_e \pi}\ (\textsc{Inst})$$

Figure 10.3. Additional rules for entailment relation of type class qualifiers

declarations, types are added to a type class. The declaration *Inst* (*Eq Int*) makes *Int* a member of *Eq*. But an instance declaration can also be constrained. For example, $[a]$ is a member of *Eq* (a can be any type) provided that a is a member of *Eq*.

Class declarations and instance declarations determine the entailment relation for type class qualifiers. In fact, we parameterize the entailment relation with the declarations present in the initial (type) environment, denoted by Γ. Although the set of declarations in Γ is arbitrary, it is fixed during type inference. Two new rules are given in Figure 10.3 to enhance the entailment relation for type class qualifiers. The rule (Super) corresponds to entailment because of the superclass hierarchy, and (Inst) describes entailment because of the instance declarations in Γ.

We present a small type class environment which we use in the examples of this chapter. All of these declarations are defined in the Haskell Prelude.

Definition 10.1 (Type class environment). *Let $C = \{Eq, Ord, Show\}$ be a set of type classes, and let $T = \{Int, Float, Bool, Char\}$ be a set of types. The initial type environment Γ contains at least the following declarations.*

$$\begin{array}{rl}
(\mathit{Class}\ (\mathit{Eq}\ a) \Rightarrow \mathit{Ord}\ a) \in \Gamma & \text{superclass of } \mathit{Ord} \\
(\mathit{Class}\ (\mathit{Eq}\ a, \mathit{Show}\ a) \Rightarrow \mathit{Num}\ a) \in \Gamma & \text{superclasses of } \mathit{Num} \\
(\mathit{Inst}\ (\mathit{Num}\ \mathit{Int})) \in \Gamma & \text{instance for } \mathit{Num} \\
(\mathit{Inst}\ (\mathit{Num}\ \mathit{Float})) \in \Gamma & \text{instance for } \mathit{Num} \\
\{\mathit{Inst}\ (c\ \tau) \mid c \leftarrow C, \tau \leftarrow T\} \subseteq \Gamma & \\
\{\mathit{Inst}\ (c\ a) \Rightarrow c\ [a] \mid c \leftarrow C\} \subseteq \Gamma & \text{list instances} \\
\{\mathit{Inst}\ (c\ a, c\ b) \Rightarrow c\ (a, b) \mid c \leftarrow C\} \subseteq \Gamma & \text{tuple instances}
\end{array}$$

Example 10.2. We present a derivation for $\Vdash_e$ *Eq* $[Int]$ to illustrate the entailment relation for type class predicates. We use the default type class environment given in Definition 10.1.

$$\frac{\dfrac{\mathit{Inst}\ (\mathit{Eq}\ \mathit{Int}) \in \Gamma}{\Vdash_e \mathit{Eq}\ \mathit{Int}}\ (\textsc{Inst}) \qquad \dfrac{\dfrac{(\mathit{Inst}\ (\mathit{Eq}\ a) \Rightarrow \mathit{Eq}\ [a]) \in \Gamma \qquad \dfrac{}{\mathit{Eq}\ a \Vdash_e \mathit{Eq}\ a}\ (\textsc{Mono})}{\mathit{Eq}\ a \Vdash_e \mathit{Eq}\ [a]}\ (\textsc{Inst})}{\mathit{Eq}\ \mathit{Int} \Vdash_e \mathit{Eq}\ [\mathit{Int}]}\ (\textsc{Closed})}{\Vdash_e \mathit{Eq}\ [\mathit{Int}]}\ (\textsc{Trans})$$

The Hindley-Milner type rules are extended to support overloading. We use judgements of the form $P \mid \Gamma \vdash_\pi e : \tau$, where P is a set of type class predicates. The

$$\boxed{P \mid \Gamma \vdash_\pi e : \tau} \quad \textit{Typing judgement}$$

$$\frac{(P \Rightarrow \tau) < \Gamma(x)}{P \mid \Gamma \vdash_\pi x : \tau} \ (\text{O-Var})$$

$$\frac{P \mid \Gamma \vdash_\pi e_1 : \tau_1 \to \tau_2 \qquad P \mid \Gamma \vdash_\pi e_2 : \tau_1}{P \mid \Gamma \vdash_\pi e_1\ e_2 : \tau_2} \ (\text{O-App})$$

$$\frac{P \mid \Gamma \backslash x \cup \{x : \tau_1\} \vdash_\pi e : \tau_2}{P \mid \Gamma \vdash_\pi \lambda x \to e : (\tau_1 \to \tau_2)} \ (\text{O-Abs})$$

$$\frac{P_1 \mid \Gamma \vdash_\pi e_1 : \tau_1 \qquad P_2 \mid \Gamma \backslash x \cup \{x : \mathit{generalize}(\Gamma, P_1 \Rightarrow \tau_1)\} \vdash_\pi e_2 : \tau_2}{P_2 \mid \Gamma \vdash_\pi \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2} \ (\text{O-Let})$$

Figure 10.4. Type rules for overloaded expressions

new type rules can be found in Figure 10.4. The type rules are syntax-directed, and they can be applied equally well for qualifiers other than type class predicates. For an in-depth study of a general framework for qualified types, we recommend Jones' *"Qualified Types"* [31] for further reading.

In the type rule (O-Var) for a variable, we retrieve the type scheme of that variable from the type environment Γ. This variable can be assigned any instance of the type scheme, provided that the instantiated type class predicates are part of the typing judgement. In the rules for application and lambda abstraction, the set of type class predicates P in the conclusion is the same as in the premises. In the type rule for a let expression (O-Let), the type assigned to the local definition is qualified with the set of predicates for this definition, and this qualified type is generalized. The resulting type scheme is associated with x in the type environment to type the body of the let expression.

The Hindley-Milner type inference algorithms can be extended accordingly. The set of type class predicates is collected in a bottom-up fashion. These predicates arise from instantiating type schemes assigned to variables. Before we generalize the type inferred for the local definition of a let expression, we include all the collected type class predicates. The predicates for the complete let expression are only those collected for the body.

We equip the inference algorithm with a technique to simplify the type class context of a type scheme. Take a look at the following two type schemes.

$$\sigma_1 = \forall a.(\mathit{Eq}\ a, \mathit{Ord}\ a, \mathit{Eq}\ a, \mathit{Eq}\ \mathit{Int}) \Rightarrow a \to a \qquad \sigma_2 = \forall a.\mathit{Ord}\ a \Rightarrow a \to a$$

These type schemes are equivalent since the predicates of the two type schemes are equivalent under the entailment relation. However, σ_2 is more concise, and we prefer this simpler type scheme. The process of simplifying a type class context is called *context reduction*, which consists of three steps.

1. *Simplification using instances.* The instance declarations are used to simplify the predicates. For instance, the predicate $Eq\ (a, b)$ is simplified to $Eq\ a$ and $Eq\ b$, and the predicate $Eq\ Int$ is removed altogether. At this point, we might also discover predicates that require a missing instance declaration. For these predicates (e.g., $Num\ Bool$), we generate an error message. Predicates conform the Haskell 98 standard can always be simplified to predicates in head-normal form, i.e., of the form $C\ (a\ \tau_1 \ldots \tau_n)$. If all type variables are of kind $*$, then n must be zero.
2. *Removal of superclasses.* Because Eq is the superclass of Ord, we have $Ord\ a \Vdash_e Eq\ a$. Thus, the predicate concerning the superclass can be safely removed if the other predicate is present.
3. *Removal of duplicates.* Duplicate class predicates can be removed.

Context reduction takes place at every generalization point. Simplification yields a set of predicates that is equivalent to the original set under the entailment relation.

The type at hand plus the simplified predicate set are generalized after we have performed context reduction. As a final step, we inspect the type scheme to detect ambiguities. Consider the type scheme $\forall a.Eq\ a \Rightarrow Int \rightarrow Int$. Although this type scheme may appear to be all right, it introduces problems as soon as we want to use it. Because the type variable a only appears in the predicate and not in the type, there is no way we can ever find out which dictionary to insert. Therefore, we report such a predicate as ambiguous.

10.3 Type class constraints

In our type inference framework, we have a clear separation between collecting constraints and solving constraints. To maintain this separation, we have decided to perform context reduction during the solving phase. Pursuing this approach has major consequences for the way we collect type class qualifiers in our framework. The standard approach is to collect these qualifiers in a bottom-up fashion, and to use these *local* sets at the generalization points. We present a different approach, which uses two *global* sets of qualifiers, that is, for the program as a whole.

The first set contains qualifiers for which we have to prove that they hold. This set contains exactly the qualifiers collected by a conventional type inference algorithm (such as the type rules presented in Figure 10.4). For example, if we use the overloaded equality function, which has type $\forall a.Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$, then this type scheme is instantiated with a fresh type variable (say v_{15}), and the instantiated type class qualifier $Eq\ v_{15}$ is added to the qualifier set. In our constraint-based framework, instantiation of type schemes is taken care of by INST constraints (page 69). While we are solving constraints, we may learn more about the type variable v_{15}: if v_{15} turns out to be Int, for instance, then the predicate $Eq\ v_{15}$ holds ($\Vdash_e Eq\ v_{15}$ because $Inst\ (Eq\ Int) \in \Gamma$). But not all type class qualifiers are resolved in this way.

Certain instantiated type class qualifiers contain type variables that stay polymorphic. Suppose that the equality function for which we introduced $\mathit{Eq}\ v_{15}$ is used to define a function f. If we generalize the type of f, and quantify over the type variable v_{15}, then we have to add the qualifier $\mathit{Eq}\ v_{15}$ to the context of the type scheme. More precisely, before we generalize a type, we perform context reduction on the collected type class qualifiers. All qualifiers in the reduced set that contain a type variable over which we quantify are included in the type scheme. The fact that a qualifier has become part of a type scheme's context immediately ensures that the qualifier is now validated. To record this fact, we copy these qualifiers to a second set of qualifiers with the opposite meaning: the predicates in this set are assumed to hold.

In short, we maintain two sets of qualifiers: one set contains qualifiers we have to prove, the other contains qualifiers we assume to hold. The idea is that at the end of the constraint solving phase, the former set of qualifiers must be entailed by the latter. Qualifiers that are not entailed by the set of assumed predicates are reported as ambiguous.

The two set approach also allows us to deal with functions that have an explicit type signature. To handle these signatures, we use skolemization constraints in our framework. Suppose that a function f has the type signature $\forall a.\mathit{Ord}\ a \Rightarrow a \rightarrow \mathit{Bool}$. Skolemization of this type scheme yields $\mathit{Ord}\ v_{21}$ together with $v_{21} \rightarrow \mathit{Bool}$, where v_{21} is a fresh skolem variable. Each qualifier that is obtained by skolemizing a type scheme (i.e., $\mathit{Ord}\ v_{21}$) is added to the list of assumed qualifiers.

At first sight, our approach with two global sets of qualifiers may seem totally unrelated to the standard algorithm. However, it is the limited scope in which type variables can appear that makes this approach work. For example, suppose that we generalize over the type variable v_0 at a generalization point that corresponds to a let declaration. Because we only use fresh type variables (also for instantiating a type scheme), we know that this type variable was only introduced somewhere inside this declaration. Furthermore, v_0 is not influenced by anything outside this declaration, because it is polymorphic. All type class predicates that contain this type variable are included in the type scheme of the let declaration, and arise from an overloaded identifier that is used in the declaration itself.

We illustrate our approach with three examples.

Example 10.3. Assume that $(==) :: \forall a.\mathit{Eq}\ a \Rightarrow a \rightarrow a \rightarrow \mathit{Bool}$, and that $(<)$ and $(>)$ are of type $\forall a.\mathit{Ord}\ a \Rightarrow a \rightarrow a \rightarrow \mathit{Bool}$. Consider the following two definitions.

```
main = merge ['a','b','c'] ['a','c','d','e']

merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x : xs) (y : ys) | x == y = merge (x : xs) ys
                        | x < y  = x : merge xs (y : ys)
                        | x > y  = y : merge (x : xs) ys
```

First, we take a look at *merge*. This function has a type signature, which is skolemized and matched with the type we find for the definition. Say we introduce the skolem type variable v_0. From skolemization, we get the predicate $Ord\ v_0$, which is now assumed to hold. Six overloaded functions are used in the last clause of *merge*: three comparison functions in the guards, and three recursive calls. The recursive calls are overloaded because of *merge*'s type signature. Instantiating the overloaded functions leads to six predicates that must be validated: five concerning the *Ord* type class, and one predicate for *Eq*. The equality constraints collected for *merge* take care that the type variables in these predicates are mapped all to v_0 (or the same type variable to which v_0 is mapped). Hence, all six predicates are entailed by the single predicate we obtained from skolemization.

In the definition of *main*, one more predicate arises from using *merge* (say $Ord\ v_1$). Because *merge* is applied to two lists of characters, we infer that v_1 equals *Char*. Overloading is fully resolved, because $\Vdash_e Ord\ Char$ holds in our initial type environment.

Example 10.4. Suppose we change the type signature of *merge* (Example 10.3) into $Eq\ a \Rightarrow [a] \to [a] \to [a]$. Skolemization yields the predicate $Eq\ v_0$, which we assume to hold. After handling the equality constraints for *merge*, this predicate entails the predicate arising from (==), and the three predicates from the recursive calls. However, the two predicates from the uses of (<) and (>) remain, and these are reported. In this scenario, overloading cannot be resolved.

Example 10.5. Take another look at Example 10.3. This time, we omit the type signature for *merge*. Because we do not have a type for *merge* yet, we only have to deal with the three overloaded functions in the guards. They result in the predicates $Eq\ v_0$, $Ord\ v_1$, and $Ord\ v_2$, respectively. Handling the equality constraints collected for *merge*, we deduce that v_0, v_1, and v_2 are in fact equivalent. Suppose they are all mapped to v_0. Context reduction simplifies the predicates such that only one predicate (i.e., $Ord\ v_0$) remains. The type we inferred for *merge* is $[v_0] \to [v_0] \to [v_0]$. This type is generalized, and because we quantify over v_0, $Ord\ v_0$ is included in the type scheme's context. Moreover, this predicate is moved to the set of predicates we assume to hold. The overloading in this example can be resolved completely.

Type class predicates place extra restrictions on types and type variables, and are thus a new sort of type constraint we have to consider. Two new alternatives are added to the constraint language.

Qualifier constraints:

$$\begin{array}{lll} c_\pi ::= & Prove\ (\pi) & \textit{(prove qualifier)} \\ \quad\ \mid & Assume\ (\pi) & \textit{(assume qualifier)} \end{array}$$

A constraint *Prove* (π) means that we have to validate the predicate π. For instance, the type constraint *Prove* (*Eq Int*) holds trivially in our default type class environment, *Prove* (*Num Bool*) cannot be satisfied, and *Prove* ($Ord\ v_0$) somehow limits the types that the type variable v_0 can take. In addition, we introduce type constraints to assume the validity of a predicate.

We have to extend a solution Θ to define semantics for the new constraints. The type constraints in Chapter 5 require Θ to have at least two components: a substitution S_Θ, and a type scheme map Σ_Θ. We add a third component Π_Θ, which is a set of predicates that are assumed to hold. Furthermore, we assume that $\Pi_\Theta = \Theta(\Pi_\Theta)$. Satisfaction of the qualifier constraints is defined by:

$$\begin{array}{lll} \Theta \vdash_s \mathit{Prove}\ (\pi) & =_{\mathit{def}} & \Pi_\Theta \Vdash_e \Theta(\pi) \\ \Theta \vdash_s \mathit{Assume}\ (\pi) & =_{\mathit{def}} & \Theta(\pi) \in \Pi_\Theta \end{array}$$

The predicate of a *prove* constraint must be entailed by the predicates in Θ, whereas the *assume* predicates must be part of Π_Θ. We illustrate this definition with a number of examples.

Example 10.6. Consider triples of the form (S, Σ, Π) as solutions. Then,

$$\begin{array}{rl} (\mathit{id}, \emptyset, \emptyset) & \not\vdash_s \mathit{Prove}\ (\mathit{Eq}\ v_0) \\ (\mathit{id}, \emptyset, \emptyset) & \not\vdash_s \{\mathit{Prove}\ (\mathit{Eq}\ v_0), \mathit{Assume}\ (\mathit{Eq}\ v_0)\} \\ (\mathit{id}, \emptyset, \{\mathit{Eq}\ v_0\}) & \vdash_s \mathit{Prove}\ (\mathit{Eq}\ v_0) \\ (\mathit{id}, \emptyset, \{\mathit{Eq}\ v_0\}) & \vdash_s \{\mathit{Prove}\ (\mathit{Eq}\ v_0), \mathit{Assume}\ (\mathit{Eq}\ v_0)\} \\ ([v_0 := \mathit{Int}], \emptyset, \emptyset) & \vdash_s \mathit{Prove}\ (\mathit{Eq}\ v_0) \\ ([v_0 := \mathit{Int}], \emptyset, \emptyset) & \vdash_s \{\mathit{Prove}\ (\mathit{Eq}\ v_0), v_0 \equiv \mathit{Int}\}. \end{array}$$

In the last two cases, we use that $\mathit{Inst}\ (\mathit{Eq}\ \mathit{Int}) \in \Gamma$.

Because we introduced qualifiers, we have to reconsider our constraints to handle polymorphism. Suppose we are about to generalize the type $v_0 \to v_0$ to $\forall a.a \to a$. If we have $\mathit{Eq}\ v_0$, then this predicate must be included in the type scheme, which results in $\forall a.\mathit{Eq}\ a \Rightarrow a \to a$. In fact, all predicates containing a type variable which is generalized must be made part of the type scheme. To achieve this, we present a new definition for generalizing a type with respect to a set of monomorphic types, and in the context of a set of type class predicates.

Definition 10.2 (Generalization with qualifiers). *Given a set of monomorphic types $\mathcal{M}$, a set of type class predicates P, and a type τ. Then*

$$\mathit{gen}_\pi(\mathcal{M}, P, \tau) = \forall \overline{a}.Q \Rightarrow \tau$$

where $\overline{a} = \mathit{ftv}(\tau) - \mathit{ftv}(\mathcal{M})$ and $Q = \{\pi \mid \pi \leftarrow P, \mathit{ftv}(\pi) \cap \overline{a} \neq \emptyset\}$.

With this notion of generalization, we refine constraint satisfaction for generalization, instantiation, and skolemization constraints.

$$\begin{array}{lllll} \Theta \vdash_s & \sigma_v := \textsc{Gen}(\mathcal{M}, \tau) & =_{\mathit{def}} & \Theta(\sigma_v) = \mathit{gen}_\pi(\Theta(\mathcal{M}), \Pi_\Theta, \Theta(\tau)) \\ \Theta \vdash_s & \tau := \textsc{Inst}(\rho) & =_{\mathit{def}} & (\Pi_\Theta \Rightarrow \Theta(\tau)) < \Theta(\rho) \\ \Theta \vdash_s & \tau := \textsc{Skol}(\mathcal{M}, \rho) & =_{\mathit{def}} & \Theta(\rho) < \mathit{gen}_\pi(\Theta(\mathcal{M}), \Pi_\Theta, \Theta(\tau)) \end{array}$$

A generalization constraint is satisfied only if all predicates that contain a quantified type variable are included in the type scheme. Instantiating a type scheme with qualifiers implies that the instantiated predicates are entailed by the predicates in Θ. For a skolemization constraint, we require that the restrictions on τ posed by Π_Θ are entailed by the predicates in ρ. The following examples illustrate these definitions.

Example 10.7. Again, our solutions take the form of (S, Σ, Π).

$$\begin{aligned}
(id, [\sigma_0 := \forall a.a \rightarrow a], \{Eq\ v_1\}) &\not\vdash_s \sigma_0 := \textsc{Gen}(\emptyset, v_1 \rightarrow v_1) \\
(id, [\sigma_0 := \forall a.Eq\ a \Rightarrow a \rightarrow a], \{Eq\ v_1\}) &\vdash_s \sigma_0 := \textsc{Gen}(\emptyset, v_1 \rightarrow v_1) \\
([v_1 := v_2 \rightarrow v_2], \emptyset, \emptyset) &\not\vdash_s v_1 := \textsc{Inst}(\forall a.Eq\ a \Rightarrow a \rightarrow a) \\
([v_1 := v_2 \rightarrow v_2], \emptyset, \{Ord\ v_2\}) &\vdash_s v_1 := \textsc{Inst}(\forall a.Eq\ a \Rightarrow a \rightarrow a) \\
([v_1 := v_2 \rightarrow v_2], \emptyset, \{Eq\ v_2\}) &\not\vdash_s v_1 := \textsc{Skol}(\emptyset, \forall a.a \rightarrow a) \\
([v_1 := v_2 \rightarrow v_2], \emptyset, \{Eq\ v_2\}) &\vdash_s v_1 := \textsc{Skol}(\emptyset, \forall a.Ord\ a \Rightarrow a \rightarrow a)
\end{aligned}$$

10.4 Extending the framework

The type inference framework of Section 5.2 is extended to handle the new type constraints. While solving the constraints, we maintain two lists of predicates. The first list contains predicates that are assumed to hold, which corresponds to the Π component in the solutions. The second list contains predicates that should be validated. Predicates that are entailed by the assumed predicates can be removed at any time. When we are finished solving the constraints, this list should be empty.

We introduce a type class for monads that support solving type class constraints.

```
class Monad m ⇒ HasPreds m ⟨i⟩ | m → ⟨i⟩ where
   assumePredicate      :: Predicate → ⟨i⟩ → m ()
   provePredicate       :: Predicate → ⟨i⟩ → m ()
   contextReduction     :: m ()
   generalizeWithPreds :: Monos → Type → m TypeScheme
   reportAmbiguous      :: m ()
```

Instances of this type class should maintain two lists of predicates: we can add a predicate to the list of assumed predicates (with *assumePredicate*), or to the list of predicates we have to prove (with *provePredicate*). The function *contextReduction* performs context reduction on the predicates that are to be proven, and removes the predicates that are entailed by the assumed predicates. We will take a closer look at this function later. With *generalizeWithPreds*, a type is generalized, and the resulting type scheme contains predicates from the prove list that contain a type variable over which we quantify. The included predicates are, as a side-effect, added to the list of assumed predicates. The function *reportAmbiguous* takes all the remaining predicates that are not proven yet, and reports these as ambiguities. We apply this function once when all other constraints have been handled.

We now discuss how to solve the type constraints using these operations. We start with the two type class constraints.

```
solve (Prove (π) ⟨i⟩)    = provePredicate π ⟨i⟩
solve (Assume (π) ⟨i⟩) = assumePredicate π ⟨i⟩
```

Solving these constraints is straightforward, since we can use *provePredicate* and *assumePredicate* to add the type class constraint to one of the two lists. We do not have to change how equality constraints are solved, but solving the polymorphism constraints has to be adapted since we have to deal with the type class predicates in the type schemes. An instantiation constraint is solved as follows.

```
$solve\ ((\tau := \textsc{Inst}(\rho))\ \ \langle i\rangle) =$
  $\mathbf{do}\ \sigma \qquad\quad \leftarrow findScheme\ \rho$
      $P :\Rightarrow: \tau_1 \leftarrow instantiateM\ \sigma$
      $\mathbf{let}\ \langle i\rangle' = instScheme\ \sigma\ \langle i\rangle$
      $addConstraints\ [Prove\ (\pi)\ \ \langle i\rangle' \mid \pi \leftarrow P]$
      $addConstraint\ \ ((\tau \equiv \tau_1)\ \ \langle i\rangle')$
```

Two changes are made with respect to the old definition. First of all, instantiating the type scheme returns a qualified type, i.e., a type and a set of type class predicates. The pattern $P :\Rightarrow: \tau_1$ is used to split the predicates and the type. Secondly, all predicates in P have to be proven, and are added as constraints.

We now discuss the modified function for solving a skolemization constraint.

```
$solve\ ((\tau := \textsc{Skol}(\mathcal{M}, \rho))\ \ \langle i\rangle) =$
  $\mathbf{do}\ \sigma \qquad\qquad\quad \leftarrow findScheme\ \rho$
      $(P :\Rightarrow: \tau_1, skc) \leftarrow skolemizeM\ \sigma$
      $\mathbf{let}\ \langle i\rangle' = skolScheme\ \sigma\ \langle i\rangle$
      $addSkolems\ \langle i\rangle'\ skc\ \mathcal{M}$
      $addConstraints\ [Assume\ (\pi)\ \ \langle i\rangle' \mid \pi \leftarrow P]$
      $addConstraint\ \ ((\tau \equiv \tau_1)\ \ \langle i\rangle')$
```

The required changes are similar to the modifications for instantiation constraints. Skolemizing the type scheme yields a qualified type ($P :\Rightarrow: \tau_1$) and a list of the introduced skolem constants. However, for skolemization constraints, the predicates in P are added as *Assume* constraints.

The last type constraint we consider is the generalization constraint.

```
$solve\ ((\sigma_v := \textsc{Gen}(\mathcal{M}, \tau))\ \ \langle i\rangle) =$
  $\mathbf{do}\ makeConsistent$
      $contextReduction$
      $\mathcal{M}' \leftarrow applySubst\ \mathcal{M}$
      $\tau' \quad \leftarrow applySubst\ \tau$
      $\sigma \quad \leftarrow generalizeWithPreds\ \mathcal{M}'\ \tau'$
      $storeScheme\ \langle i\rangle\ \sigma_v\ \sigma$
```

After making the substitution state consistent, we perform context reduction. This simplifies the set of predicates to be proven, and while simplifying these constraints, we may encounter irreducible predicates, which we add as an error to the state. We use *generalizeWithPreds* to obtain the type scheme that we assign to σ_v. In fact, this function is nothing but gen_π.

We take a second look at context reduction, which consists of simplifying the qualifiers using the instance declarations, and removing superclasses and duplicates from the set. Of course, we want to keep track of information about the qualifiers, and how this set changes by context reduction. To achieve this, we introduce a type class to record this information. Hence, an extra class constraint appears in the type of *contextReduction*, in addition to the class constraint *HasPreds* m $\langle i \rangle$ from the class declaration.

```
contextReduction :: (HasPreds m ⟨i⟩, PredicateInfo ⟨i⟩) ⇒ m ()

class PredicateInfo ⟨i⟩ where
  byInstance   :: (Predicate, Predicate) → ⟨i⟩ → ⟨i⟩
  bySuperclass :: ⟨i⟩ → ⟨i⟩ → ⟨i⟩
  byDuplicate  :: ⟨i⟩ → ⟨i⟩ → ⟨i⟩
    -- default definitions
  byInstance _   = id
  bySuperclass _ = id
  byDuplicate _  = id
```

The three member functions of the type class correspond to the steps of context reduction. With *byInstance*, a pair of predicates is stored in the constraint information. The first component of this pair is the original predicate, the second is one of the predicates to which this original predicate is simplified by an instance declaration. Both *bySuperclass* and *byDuplicate* combine information carried by two predicates. The following example illustrates our approach.

Example 10.8. Consider the following set on which we perform context reduction. Each predicate is paired with its own constraint information.

$$\{\ \ (\mathit{Ord}\ v_0, \langle i_0 \rangle)\ \ ,\ \ (\mathit{Eq}\ (\mathit{Int}, v_0), \langle i_1 \rangle)\ \ ,\ \ (\mathit{Num}\ \mathit{Bool}, \langle i_2 \rangle)\ \ ,\ \ (\mathit{Ord}\ v_0, \langle i_3 \rangle)\ \ \}$$

Because *Num Bool* cannot be simplified, $\langle i_2 \rangle$ is added to the list of errors. Next, we simplify *Eq* (Int, v_0), which gives us *Eq Int* and *Eq* v_0. The former can be removed since $\Vdash_e$ *Eq Int*, the latter can be removed since we have *Ord* v_0. One of the two *Ord* v_0 predicates is removed, and, as a result, the simplified set contains only one predicate. Observe how the constraint information of this predicate contains a trace to its original predicates.

$$\{\ \ (\mathit{Ord}\ v_0, \mathit{byDuplicate}\ \langle i_3 \rangle\ (\mathit{bySuperclass}\ \langle i_1 \rangle'\ \langle i_0 \rangle))\ \ \}$$

where $\langle i_1 \rangle'$ is *byInstance* $(\mathit{Eq}\ (\mathit{Int}, v_0), \mathit{Eq}\ v_0)\ \langle i_1 \rangle$.

10.5 Modifying the type rules

Chapter 6 presents an extensive set of type rules to construct a tree with type constraints for a program. We discuss modifications to these type rules that are required for overloading a number of language constructs. Fortunately, most type rules are left untouched. First, we overload the numeric literals in our language. Next, we discuss overloading negations and enumerations, and, finally, we take a look at monadic do expressions.

$\boxed{\mathcal{T}_\mathcal{C} \vdash_l^\pi l : \tau}$ *Literal*

$$\frac{c = \mathit{Prove}\ (\mathit{Num}\ \beta)}{[c]^\bullet \vdash_l^\pi \text{Integer} : \beta}\ (\text{L-Num})^\pi \qquad \frac{c = \mathit{Prove}\ (\mathit{Fractional}\ \beta)}{[c]^\bullet \vdash_l^\pi \text{Float} : \beta}\ (\text{L-Frac})^\pi$$

$$\frac{}{\bullet \vdash_l^\pi \text{Char} : \mathit{Char}}\ (\text{L-Char})^\pi \qquad \frac{}{\bullet \vdash_l^\pi \text{String} : \mathit{String}}\ (\text{L-String})^\pi$$

$\boxed{\mathcal{E}, \mathcal{T}_\mathcal{C} \vdash_p p : \tau}$ *Pattern*

$$\frac{c = (\beta \equiv \tau) \qquad \mathcal{T}_\mathcal{C} \vdash_l^\pi l : \tau}{\emptyset, c \rhd \{\!\bullet\ \mathcal{T}_\mathcal{C}\ \bullet\!\} \vdash_p l : \beta}\ (\text{P-Lit})^\pi$$

$\boxed{\langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T}_\mathcal{C} \vdash_e e : \tau}$ *Expression*

$$\frac{c = (\beta \equiv \tau) \qquad \mathcal{T}_\mathcal{C} \vdash_l^\pi l : \tau}{\langle\mathcal{M}\rangle, \emptyset, c \rhd \{\!\bullet\ \mathcal{T}_\mathcal{C}\ \bullet\!\} \vdash_e l : \beta}\ (\text{E-Lit})^\pi$$

Figure 10.5. Type rules for overloaded literals

Overloaded literals

We use the type classes *Num* and *Fractional* to overload the numeric literals in our language. We aim at the following type schemes.

$$5 :: \forall a.\mathit{Num}\ a \Rightarrow a \qquad 3.14159 :: \forall a.\mathit{Fractional}\ a \Rightarrow a$$

In the original type rules, we use judgements of the form $\vdash_l l : \tau$ to type a literal. We extend these judgements to $\mathcal{T}_\mathcal{C} \vdash_l^\pi l : \tau$ such that we can create type class constraints for the overloaded literals. The new type rules for literals are shown in Figure 10.5. The type rules $(\text{L-Num})^\pi$ and $(\text{L-Frac})^\pi$ introduce a fresh type variable β, and create a type class constraint to restrict this type variable. The constraint trees for character and string literals are empty.

Because we changed the judgements for literals, we also have to adapt the type rules (P-Lit) and (E-Lit), which make use of these judgements. Figure 10.5 presents the modified rules.

Overloaded negation and enumeration

Figure 10.6 displays the type rules for overloaded negation (in a pattern or expression) and overloaded enumeration. The type rule $(\text{P-Neg})^\pi$ for a negated pattern

$\boxed{\mathcal{E}, \mathcal{T_C} \vdash_p p : \tau}$ *Pattern*

$$\frac{c = (\beta \equiv \tau) \qquad \mathcal{T_C} \vdash_l^{\pi} l : \tau}{\emptyset, c \rhd \{\!\{ \mathcal{T_C} \}\!\} \vdash_p \; -l : \beta} \; (\textsc{P-Neg})^{\pi}$$

$\boxed{\langle \mathcal{M} \rangle, \mathcal{A}, \mathcal{T_C} \vdash_e e : \tau}$ *Expression*

$$\frac{\begin{array}{c} \mathcal{C} = [\beta \equiv \tau, \mathit{Prove}\ (\mathit{Num}\ \beta)] \\ \langle \mathcal{M} \rangle, \mathcal{A}, \mathcal{T_C} \vdash_e e : \tau \end{array}}{\langle \mathcal{M} \rangle, \mathcal{A}, \mathcal{C} \trianglerighteq \{\!\{ \mathcal{T_C} \}\!\} \vdash_e \; -e : \beta} \; (\textsc{E-Neg})^{\pi}$$

$$\frac{\begin{array}{c} \mathcal{T_C}_{new} = (\mathcal{C} \trianglerighteq \{\!\{ c_1 \lhd \mathcal{T_C}_1, c_2 \lhd \mathcal{T_C}_2, c_3 \lhd \mathcal{T_C}_3 \}\!\}) \\ \mathcal{C} = [\beta_2 \equiv [\beta_1], \mathit{Prove}\ (\mathit{Enum}\ \beta_1)] \\ c_1 = (\tau_1 \equiv \beta_1) \qquad c_2 = (\tau_2 \equiv \beta_1) \qquad c_3 = (\tau_3 \equiv \beta_1) \\ \langle \mathcal{M} \rangle, \mathcal{A}_1, \mathcal{T_C}_1 \vdash_e e : \tau_1 \\ \langle \mathcal{M} \rangle, \mathcal{A}_2, \mathcal{T_C}_2 \vdash_{me} me_1 : \tau_2 \qquad \langle \mathcal{M} \rangle, \mathcal{A}_3, \mathcal{T_C}_3 \vdash_{me} me_2 : \tau_3 \end{array}}{\langle \mathcal{M} \rangle, \mathcal{A}_1 +\!\!+ \mathcal{A}_2 +\!\!+ \mathcal{A}_3, \mathcal{T_C}_{new} \vdash_e [e, me_1 \mathinner{..} me_2] : \beta_2} \; (\textsc{E-Enum})^{\pi}$$

Figure 10.6. Type rules for overloaded negation and enumeration

literal introduces a fresh type variable β, which must be equal to the type of the literal. We do not restrict β or τ to the *Num* type class, since syntactically only numeric literals are allowed – the type rules $(\textsc{L-Num})^{\pi}$ and $(\textsc{L-Frac})^{\pi}$ take care of this restriction.

However, for negating an expression, we do have to introduce a type class constraint. In addition to the equality constraint that equates the type of the negated expression (τ) with the type of the whole expression (β), we create the constraint *Prove* (*Num* β), since negation can only be performed on numeric values.

The type class *Enum* contains all types that can be enumerated, including integers and characters. The new type rule $(\textsc{E-Enum})^{\pi}$ for an enumeration is shown in Figure 10.6. Two new type variables are introduced: β_1 represents the type of the elements in the list, and β_2 is the type of the enumeration. The elements of the list must have a type in the *Enum* type class, and therefore we generate the constraint *Prove* (*Enum* β_1). Note that in the earlier type rule (that is, (E-Enum) on page 110) we assumed β_1 to be *Int*.

Overloaded do expressions

Overloading the monadic do notation poses a greater challenge. Instead of assuming all monadic computations to be in the *IO* monad, we use the *Monad* type class. In fact, this is a constructor class, as its members are not types, but type constructors

$\boxed{\langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{T\!C} \vdash_e e : \tau}$ *Expression*

$$\frac{\begin{array}{c}\mathcal{C} = [\beta \equiv \mu\ \tau, \mathit{Prove}\ (\mathit{Monad}\ \mu)] \\ \langle\mathcal{M}, \cdot\rangle, \mu, \mathcal{A}, \mathcal{T\!C} \vdash^{\pi}_{ms} ms : \tau\end{array}}{\langle\mathcal{M}\rangle, \mathcal{A}, \mathcal{C} \trianglerighteq \{\!\!\{ \mathcal{T\!C} \}\!\!\} \vdash_e \mathbf{do}\ ms : \beta}\ (\text{E-Do})^{\pi}$$

$\boxed{\langle\mathcal{M}, \tau^?\rangle, \mu, \mathcal{A}, \mathcal{T\!C} \vdash^{\pi}_{ms} ms : \tau^?}$ *Sequence of statements*

$$\frac{}{\langle\mathcal{M}, \tau^?\rangle, \mu, \emptyset, \bullet \vdash^{\pi}_{ms} \cdot : \tau^?}\ (\text{M-Empty})^{\pi}$$

$$\frac{\begin{array}{c}c = (\tau \equiv \mu\ \beta) \\ \langle\mathcal{M}\rangle, \mathcal{A}_1, \mathcal{T\!C}_1 \vdash_e e : \tau \qquad \langle\mathcal{M}, \beta\rangle, \mu, \mathcal{A}_2, \mathcal{T\!C}_2 \vdash^{\pi}_{ms} ms : \tau^?_2\end{array}}{\langle\mathcal{M}, \tau^?_1\rangle, \mu, \mathcal{A}_1 +\!\!+ \mathcal{A}_2, \{\!\!\{ c \triangleleft \mathcal{T\!C}_1, \mathcal{T\!C}_2 \}\!\!\} \vdash^{\pi}_{ms} e; ms : \tau^?_2}\ (\text{M-Expr})^{\pi}$$

$$\frac{\begin{array}{c}(\mathcal{M}', \mathcal{A}', \mathcal{T\!C}') = \mathit{bga}(\mathcal{M}, [\mathcal{B}_{new}, \mathcal{B}_1, \ldots, \mathcal{B}_n], \biguplus_{i=1}^{n} \Sigma_i) \\ \langle\mathcal{M}'\rangle, \mathcal{B}_i, \Sigma_i \vdash_d d_i \qquad 1 \leqslant i \leqslant n,\ n \geqslant 0 \\ \langle\mathcal{M}, \cdot\rangle, \mu, \mathcal{A}, \mathcal{T\!C} \vdash^{\pi}_{ms} ms : \tau^?_2 \qquad \mathcal{B}_{new} = (\emptyset, \mathcal{A}, \mathcal{T\!C})\end{array}}{\langle\mathcal{M}, \tau^?_1\rangle, \mu, \mathcal{A}', \mathcal{T\!C}' \vdash^{\pi}_{ms} \mathbf{let}\ d_1; \ldots; d_n; ms : \tau^?_2}\ (\text{M-Let})^{\pi}$$

$$\frac{\begin{array}{c}\mathcal{T\!C}_{new} = (c \triangleright \mathcal{C}_\ell \underline{\triangleright\circ} \{\!\!\{ \mathcal{T\!C}_1, \mathcal{T\!C}_2, \mathcal{T\!C}_3 \}\!\!\}) \\ c = (\mu\ \tau_1 \equiv \tau_2) \qquad \mathcal{C}_\ell = (\mathcal{E} \equiv \mathcal{A}_3) \\ \mathcal{E}, \mathcal{T\!C}_1 \vdash_p p : \tau_1 \qquad \langle\mathcal{M}\rangle, \mathcal{A}_2, \mathcal{T\!C}_2 \vdash_e e : \tau_2 \\ \langle\mathcal{M} +\!\!+ \mathit{ftv}(\mathcal{C}_\ell), \cdot\rangle, \mu, \mathcal{A}_3, \mathcal{T\!C}_3 \vdash^{\pi}_{ms} ms : \tau^?_2\end{array}}{\langle\mathcal{M}, \tau^?_1\rangle, \mu, \mathcal{A}_2 +\!\!+ \mathcal{A}_3 \backslash \mathit{dom}(\mathcal{E}), \mathcal{T\!C}_{new} \vdash^{\pi}_{ms} p \leftarrow e; ms : \tau^?_2}\ (\text{M-Gen})^{\pi}$$

Figure 10.7. Type rules for overloaded do expressions

of kind $* \rightarrow *$ (such as *Maybe*, [ ], and *IO*). We use a type variable as a place-holder for the monad of a do expression, and this type variable must also be of kind $* \rightarrow *$. Once we start overloading do expressions, we inevitably need to perform a kind analysis prior to type inference. Of course, this analysis can again be formulated as a constraint-problem, as we discussed in Section 5.4. Having said this, we continue our discussion on the type rules for overloaded do expressions without addressing kinding issues.

Not only do we have to change the type rule (E-Do), but also the four type rules for a sequence of statements need to be modified. The new type rules are listed in Figure 10.7. We introduce a fresh type variable to represent the type of the monad in the type rule $(\text{M-Empty})^{\pi}$ for an empty sequence. As a reminder, this type variable is written as μ, and it is included in the judgement for sequences

$$\boxed{S \textit{ improves } P} \quad \textit{Improvement relation}$$

$$\frac{}{id \textit{ improves } P}\ (\textsc{Id}) \qquad \frac{S_1 \textit{ improves } P \qquad S_2 \textit{ improves } (S_1 P)}{(S_2 \circ S_1) \textit{ improves } P}\ (\textsc{Compose})$$

Figure 10.8. Basic rules for qualifier improvement

of statements as a synthesized attribute. The three new type rules for non-empty sequences pass this type variable on without changing it. In the constraints created by $(\text{M-}\textsc{Expr})^{\pi}$ and $(\text{M-}\textsc{Gen})^{\pi}$, we no longer assume the computation to be in the *IO* monad, but use the special type variable instead. This is also the case for the equality constraint created by the type rule $(\text{E-}\textsc{Do})^{\pi}$. Additionally, the type class constraint *Prove* (*Monad* μ) is generated by this type rule.

10.6 Improving substitutions

In addition to simplification of qualifiers, we present improvement of qualifiers by means of an improving substitution [31]. Certain qualifiers allow us to make deductions about type variables, and we capture this knowledge in an improvement relation. Let S be a substitution, and let P be a set of qualifiers. We say that S *improves* P if the qualifiers in P imply the deductions on the type variables in S. Figure 10.8 lists two properties of improving substitutions. First of all, the identity substitution is a valid improvement for all qualifier sets (Id). We use this as the default improvement for qualifiers. Secondly, improving substitutions can be composed with the rule (Compose).

Improving qualifiers has some immediate consequences for the type inference framework presented thus far. Because of an improving substitution, more improvements can be triggered, and, after applying the substitution, more simplifications may be possible. Hence, simplification and improvement of qualifiers take the form of a fixpoint computation. A second issue is that improving substitutions can be involved in a type inconsistency. To explain the reasoning steps behind the type variable deductions of an improving substitution is of particular interest in a framework which focuses on providing high quality feedback.

As a case study of improving substitutions, we discuss dependency qualifiers. These qualifiers let us express that one type is uniquely determined by another type, and they are the basis of functional dependencies for multi-parameter type classes, as described by Jones [34]. Interestingly enough, dependencies in a more general form also capture the essence of System O [47]. This system, presented by Odersky et al., is an alternative (and less known) approach to deal with overloading. In the remaining part of this chapter, we introduce dependency qualifiers. We give examples how these qualifiers can be used to obtain overloading in style of System O, and how to encode functional dependencies for multi-parameter type classes.

$\boxed{P \Vdash_e P}$ *Entailment relation*

$$\frac{(\mathit{Dependency}\ \pi) \in \Gamma}{\Vdash_e \pi} \quad (\textsc{Dep-Entail})$$

$\boxed{S\ \mathit{improves}\ P}$ *Improvement relation*

$$\frac{\begin{array}{c} S = \mathit{mgu}(\tau_2, \tau_3) \\ P \Vdash_e X : \tau_1 \leadsto \tau_2 \qquad P \Vdash_e X : \tau_1 \leadsto \tau_3 \end{array}}{S\ \mathit{improves}\ P} \quad (\textsc{Dep-Impr})$$

Figure 10.9. Entailment and improvement rules for dependency qualifiers

Dependency qualifiers

A dependency qualifier is of the following form.

$$\pi_{dep} ::= X : \tau_1 \leadsto \tau_2 \qquad \textit{(dependency qualifier)}$$

Here, X comes from a set of labels. Such a dependency qualifier is pronounced as "under label X, type τ_1 determines type τ_2". Several dependency labels can be around, each corresponding to a different dependency relation.

Just as there is a design space for type classes, dependency qualifiers can be exposed to certain restrictions and extensions. For instance, it is quite useful to allow type variables in dependency qualifiers, such as $X : [a] \leadsto a$, or $Y : (a, b) \leadsto a$. Another issue is in what sense the determining types of a dependency relation should be different. Clearly, the dependencies $X : \mathit{Int} \leadsto \mathit{Int}$ and $X : \mathit{Int} \leadsto \mathit{Bool}$ are contradictory, and should be rejected. A fairly conservative approach is to require that each determining type of a dependency relation has a unique type constant in its left spine. For instance, $X : (\mathit{Int} \to \mathit{Int}) \leadsto \mathit{Int}$ and $X : (\mathit{Bool} \to \mathit{Bool}) \leadsto \mathit{Bool}$ violate this restriction, because the type constant $(\to)$ appears in the left spine of both dependencies. This approach is adopted by System O. The rule (Dep-Entail), shown in Figure 10.9, refines entailment for dependency qualifiers. We assume that the initial type environment Γ contains declarations of the form $(\mathit{Dependency}\ \pi_{dep})$ which define the dependency relations. This rule expresses that dependency qualifiers that are in Γ hold trivially.

We continue with a discussion on improving substitutions for dependencies. The rule (Dep-Impr) in Figure 10.9 specifies how dependency qualifiers can be improved. If we have $X : \tau_1 \leadsto \tau_2$ and $X : \tau_1 \leadsto \tau_3$, then the types τ_2 and τ_3 must be the same. Hence, we use a most general unifier of these types as an improving substitution. In the Top framework, this takes the form of introducing an equality constraint $\tau_2 \equiv \tau_3$. The rule for dependency improvement can be applied in two ways.

- *Direct dependency.* This is an improving substitution for one dependency from the set of qualifiers, and a dependency from the initial environment Γ. Assume that $\mathit{Dependency}\ (X : \mathit{Int} \leadsto \mathit{Bool}) \in \Gamma$, and the set of qualifiers P contains $X : \mathit{Int} \leadsto v_0$. Then we have $P \Vdash_e X : \mathit{Int} \leadsto \mathit{Bool}$ by (DEP-ENTAIL), and $P \Vdash_e X : \mathit{Int} \leadsto v_0$ by (MONO). Hence, the substitution $\mathit{mgu}(v_0, \mathit{Bool}) = [v_0 := \mathit{Bool}]$ improves P. Observe that the improvement makes the dependency superfluous. Once it is used, it can be discarded.
- *Indirect dependency.* This is an improving substitution for two dependency qualifiers, say $X : \tau_1 \leadsto \tau_2$ and $X : \tau_1 \leadsto \tau_3$, from P. Although we may not know τ_1 sufficiently to determine τ_2 or τ_3, these two types do have to be equal. After the improvement, the two dependency qualifiers are identical, and one can be removed.

Dependency qualifiers can also be in error. If we have a dependency $X : \mathit{Int} \leadsto v_0$, but the relation X is not defined for *Int*, then this qualifier produces an error. If we know that the determining types of a dependency relation all have a unique type constant in the left spine, then we know that after improving a set of dependencies, the remaining qualifiers are in head-normal form.

At a generalization point, dependency qualifiers may end up in a type scheme. These qualifiers can influence whether other qualifiers are ambiguous or not, in particular when we mix dependency qualifiers and type class predicates. Take a look at the following two type schemes.

$$\sigma_1 = \forall ab.(X : a \leadsto b, \mathit{Eq}\ b) \Rightarrow a \rightarrow \mathit{String}$$
$$\sigma_2 = \forall ab.(X : a \leadsto b, \mathit{Eq}\ a) \Rightarrow b \rightarrow \mathit{String}$$

In σ_1, $\mathit{Eq}\ b$ is no longer ambiguous since the dependency can be used to determine a type for b once we have a type for a. On the other hand, σ_2 is ambiguous because we cannot recover a type for a.

We conclude our discussion on dependency qualifiers with two examples.

Example 10.9. In this example, taken from *"A Second Look at Overloading"* [47], we illustrate how overloading in the style of System O can be expressed using dependency qualifiers. Consider the overloaded functions *first*, *second*, and *third* to obtain the corresponding component from a tuple or triple. First, we declare which functions are overloaded.

over *first*
over *second*
over *third*

This brings the three overloaded functions in scope. The function *first* is assigned the type scheme $\forall ab.(\mathit{first} : a \leadsto b) \Rightarrow a \rightarrow b$. The name of the overloaded function is also the name of the dependency label. We continue and define instances for tuples and triples. We omit the code that normally accompanies these declarations.

inst *first* $:: (a, b) \rightarrow a$
inst *second* $:: (a, b) \rightarrow b$

$$\begin{array}{ll} \textbf{inst}\ \mathit{first} & :: (a, b, c) \to a \\ \textbf{inst}\ \mathit{second} & :: (a, b, c) \to b \\ \textbf{inst}\ \mathit{third} & :: (a, b, c) \to c \end{array}$$

Each **inst** declaration is translated into a dependency that we put in Γ. For example, the first line is translated into *Dependency* $(\mathit{first} : (a, b) \leadsto a)$. We define a function f which uses the overloaded functions, and we collect its type constraints.

$$f\ x = (\mathit{second}\ x, \mathit{first}\ x)$$

Applying the constraint collecting rules results in the following constraint set.

$$\begin{array}{lllllllll} v_2 & \preceq & \forall ab.(\mathit{second} : a \leadsto b) \Rightarrow a \to b & v_3 & \equiv & v_1 & v_2 & \equiv & v_3 \to v_4 \\ v_5 & \preceq & \forall ab.(\mathit{first} : a \leadsto b) \Rightarrow a \to b & v_6 & \equiv & v_1 & v_5 & \equiv & v_6 \to v_7 \\ v_8 & \equiv & (v_4, v_7) & v_0 & \equiv & v_1 \to v_8 & \sigma_0 & := & \textsc{Gen}(\emptyset, v_0) \end{array}$$

The type scheme variable σ_0 represents the polymorphic type of f. Solving these constraints returns $\forall abc.(\mathit{second} : a \leadsto b, \mathit{first} : a \leadsto c) \Rightarrow a \to (b, c)$ for σ_0. We define *demo*, which uses the function f.

$$\mathit{demo} = f\ (1, \mathit{True}, \texttt{'a'})$$

For this function, the following (simplified) type constraints are generated.

$$\begin{array}{llllll} v_9 & \preceq & \sigma_0 & v_{10} & \equiv & (\mathit{Int}, \mathit{Bool}, \mathit{Char}) \\ v_9 & \equiv & v_{10} \to v_{11} & \sigma_1 & := & \textsc{Gen}(\emptyset, v_{11}) \end{array}$$

The type scheme variable σ_1 is assigned to the function *demo*. After the constraints are solved, we find $(\mathit{Bool}, \mathit{Int})$ for σ_1.

Example 10.10. Our second example shows that dependency qualifiers can model functional dependencies between arguments of type classes. Of course, this only makes sense for multi-parameter type classes (a widely accepted extension to Haskell 98 type classes). Consider the type class *Collects e ce*, where e is the type of the elements in the collection, and *ce* is the type of the collection itself.[2]

$$\begin{array}{l} \textbf{class}\ \mathit{Collects}\ e\ \mathit{ce} \mid \mathit{ce} \to e\ \textbf{where} \\ \quad \mathit{empty} \quad :: \mathit{ce} \\ \quad \mathit{insert} \quad :: e \to \mathit{ce} \to \mathit{ce} \\ \quad \mathit{member} :: e \to \mathit{ce} \to \mathit{Bool} \end{array}$$

The functional dependency in the first line states that the type of the collection determines the type of the elements. Without the functional dependency, the function *empty* would have an ambiguous type, since e does not occur in the type. Normally, we assign *empty* the type scheme $\forall ab.\mathit{Collects}\ a\ b \Rightarrow b$, and the functional dependency between the type variables is implicitly present in the type class predicate.

[2] In this example, *ce* is really a type, and not a type constructor such as *List*.

We present a different solution, which makes this dependency explicit and visible in the type scheme. The type schemes we assign to *empty* and *insert* are the following.

$$\begin{aligned}\sigma_{empty} &= \forall ab.(\mathit{Collects}\ a\ b, \mathit{Collects} : b \rightsquigarrow a) \Rightarrow b \\ \sigma_{insert} &= \forall ab.(\mathit{Collects}\ a\ b, \mathit{Collects} : b \rightsquigarrow a) \Rightarrow a \to b \to b\end{aligned}$$

Again, we use the name of the type class as a dependency label. Take a look at the function *test*, which inserts two elements in an empty collection.

$$\mathit{test}\ x\ y = \mathit{insert}\ x\ (\mathit{insert}\ y\ \mathit{empty})$$

The following constraint set is collected for the definition of *test*.

$$\begin{array}{lll} v_3 \preceq \sigma_{insert} & v_4 \equiv v_1 & v_5 \preceq \sigma_{insert} \\ v_6 \equiv v_2 & v_7 \preceq \sigma_{empty} & v_5 \equiv v_6 \to v_7 \to v_8 \\ v_3 \equiv v_4 \to v_8 \to v_9 & v_0 \equiv v_1 \to v_2 \to v_9 & \sigma_0 := \textsc{Gen}(\emptyset, v_0) \end{array}$$

In this constraint set, σ_0 corresponds to the polymorphic type of *test*. The types of the arguments x and y are unknown. However, they are inserted in the same collection, and since the type of the collection determines the type of the elements, we know that x and y should be of the same type. Hence, the type we find for test is

$$\mathit{test} :: \forall ab.(\mathit{Collects}\ a\ b, \mathit{Collects} : b \rightsquigarrow a) \Rightarrow a \to a \to b.$$

If we omit the dependency qualifiers in σ_{empty} and σ_{insert}, then we cannot determine anymore that the types of x and y are equal.

11

Type class directives

Overview. *We propose a number of directives to improve the type error messages in the presence of Haskell 98 type classes, in particular for the non-expert user. As a language feature, type classes are very pervasive, and strongly influence what is reported and when, even in relatively simple programs. Four type class directives are explained, and we discuss how our approach can be generalized to a small language for describing invariants on type classes. This chapter is based on "Type Class Directives" [21].*

Type classes have been studied thoroughly, both in theory and practice. In spite of this, very little attention has been devoted to compensate the effects type classes have on the quality of type error messages. We present a practical and original solution to improve the quality of type error messages by scripting the type inference process. To illustrate the problem type classes introduce, consider the following attempt to decrement the elements of a list.

$$f\ xs = map\ \ {-1}\ xs$$

The parse tree for this expression does not correspond to what the spacing suggests, but corresponds to $map - (1\ xs)$: the literal 1 is applied to xs, the result of which is subtracted from map. Notwithstanding, GHC will infer the following type for f (Hugs will reject f because of an illegal Haskell 98 class constraint in the inferred type):

$$\begin{aligned} f :: (&Num\ (t \to (a \to b) \to [a] \to [b]) \\ ,&Num\ ((a \to b) \to [a] \to [b]) \\ )&\Rightarrow t \to (a \to b) \to [a] \to [b] \end{aligned}$$

Both subtraction and the literal 1 are overloaded in the definition of f[1]. Although the polymorphic type of 1 is constrained to the type class *Num*, this restriction does not lead to a type error. Instead, the constraint is propagated into the type of f. Moreover, unifications change the constrained type into a type which is unlikely to be a member of *Num*. A compiler cannot reject f since the instances required could be given later on. This *open-world* approach for type classes is likely to cause problems at the site where f is used. One of our directives allows us to specify that function types will never be part of the *Num* type class. With this knowledge we can reject f at the site of definition.

[1] In Haskell, we have $1 :: Num\ a \Rightarrow a$ and $(-) :: Num\ a \Rightarrow a \to a \to a$.

We improve the type error messages for Haskell 98 type classes [49], in particular for the non-expert user. We continue with the design of a language for type inference directives, which we started in Chapter 9, with a focus on overloading. In this chapter, we make the following additions.

- We present four type class directives that help to improve the resolution of overloading (Section 11.1). With these directives we can report special purpose error messages, reject suspicious class contexts, improve inferred types, and disambiguate programs in a precise way.
- We discuss how the proposed directives can be incorporated into the process of context reduction (Section 11.2).
- We give a general language for describing invariants over type classes (Section 11.3). This language generalizes some of our proposed directives.
- In Section 11.4, we extend the specialized type rules to handle type classes. As a consequence, we can report precise and tailor-made error messages for incorrect uses of an overloaded function.

11.1 Type class directives

In Haskell, new type classes are introduced with a class declaration. If a list of superclasses is given at this point, then the instances of the type class must also be member of each superclass; this is enforced by the compiler. To make a type a member of a type class, we provide (or derive) an instance declaration. Other means for describing type classes do not exist in Haskell.

Therefore, some properties of a type class cannot be described: for example, we cannot exclude a type from a type class. To gain more flexibility, we propose type class directives to enrich the specification of type classes. Each of these have been implemented in our type inference framework.

The first directive we introduce is the *never* directive (Section 11.1.1), which excludes a single type from a type class. This is the exact opposite of an instance declaration, and limits the *open-world* character of that type class. Similar to this case-by-case directive, we introduce a second directive to disallow new instances for a type class altogether (Section 11.1.2). A closed type class has the advantage that we know its limited set of instances.

Knowing the set of instances of a type class opens the door for two optimizations. In the exceptional case that a type class is empty, we can reject every function that requires some instance of that class. If the type class X has only one member, say the type t, then a predicate of the form $X\ a$ can improve a to t. This is, in fact, an improving substitution in Jones' theory of qualified types [32]. If we have $(X\ a,\ Y\ a)$, and the set of shared instances is empty or a singleton, then the same reasoning applies. For example, if the instances of X are *Int* and *Bool*, and those of Y are *Bool* and *Char*, then a must be *Bool*. This is easily discovered for Haskell 98 type classes by taking intersections of sets of instances.

Our next directive, the *disjoint* directive, specifies that the intersection of two type classes should be empty (Section 11.1.3). This is another instance of an invariant over sets of types, which is formulated by the programmer, and maintained by the compiler. In Section 11.3, we present a small language to capture this invariant, and many others besides.

Finally, Section 11.1.4 discusses a *default* directive for type classes, which helps to disambiguate in case overloading cannot be resolved. This directive refines the ad hoc default declarations supported by Haskell.

In the remainder of this section, we explore the new directives in more detail, and conclude with a short section on error message attributes.

11.1.1 The never directive

Our first directive enables us to express explicitly that a type should never become a member of a certain type class. This statement can be accompanied with a special purpose error message, reported in case the forbidden instance is needed to resolve overloading. The main advantage of the *never* directive is the tailor-made error message for a particular case in which overloading cannot be resolved. In addition, the directive guarantees that the outlawed instance will not be given in future. We illustrate the *never* directive with an example. For the sake of brevity, we keep the error messages in our examples rather terse. Error message attributes can be used to create a more verbose message that depends on the actual program.

never $\mathit{Eq}\ (a \to b)$: `functions cannot be tested for equality`
never *Num Bool* : `arithmetic on booleans is not supported`

These two directives should be placed in a `.type` file, which is considered prior to type inference, but after collecting all the type classes and instances in scope. Before type inference, we should check the validity of the directives. Each inconsistency between the directives and the instance declarations results in an error message or warning. For example, the following could be reported at this point.

```
The instance declaration for
  Num Bool at (3,1) in A.hs
is in contradiction with the directive
  never Num Bool defined at (1,1) in A.type
```

We proceed with type inference if no inconsistency is found. If arithmetic on booleans results in a *Num Bool* predicate, we report our special purpose error message. For the definition

$$f\ x = \textbf{if}\ x\ \textbf{then}\ x + 1\ \textbf{else}\ x$$

we simply report that arithmetic on booleans is not supported, and highlight the operator +. An extreme of concision results in the following type error message.

```
(1,19): arithmetic on booleans is not supported
```

The *never* directive is subject to the same restrictions as any instance declaration in Haskell 98: a class name followed by a type constructor and a list of unique type variables (we took the liberty of writing an infix function arrow in the example presented earlier). Haskell 98 does not allow overlapping instances, and similarly we prohibit overlapping *never*s. This ensures that there is always a unique directive for determining the error message.

Example 11.1. Consider the following overlapping *never* directives.

never $Eq\ (Int \rightarrow a)$ `: message #1`
never $Eq\ (b \rightarrow Bool)$ `: message #2`

It is unclear what will be reported for the type class predicate $Eq\ (Int \rightarrow Bool)$. One way to cope with this situation is to require a third directive for the overlapping case, namely **never** $Eq\ (Int \rightarrow Bool)$. This implies that we can always find and report a most specific directive. Note that in the context of overlapping *never* directives, we have to postpone reporting a violating class predicate, since more information about a type variable in this assertion may make a more specific directive a better candidate.

11.1.2 The close directive

With the *never* directive we can exclude one type from a type class. Similar to this case-by-case directive, we introduce a second type class directive which closes a type class in the sense that no new instances can be defined. As a result of this directive, we can report special error messages for unresolved overloading for a particular type class. A second advantage is that the compiler can assume to know all instances of the given type class since new instances are prohibited, which can be exploited when generating the error message.

One subtle issue is to establish at which point the type class should be closed. This can be either *before* or *after* having considered the instance declarations defined in the module. In this section we only discuss the former. A possible use for the latter is to close the *Num* type class in `Prelude.type` so that everybody who imports it may not extend the type class, but the `Prelude` module itself may specify new instances for *Num*.

Before we start to infer types, we check for each closed type class that no new instance declarations are provided. A special purpose error message is attached to each *close* directive, which is reported if we require a non-instance type to resolve overloading for the closed type class. Such a directive can live side by side with a *never* directive. Since the latter is strictly more informative, we give it precedence over a *close* directive if we have to create a message.

Example 11.2. As an example, we close the type class for *Integral* types, defined in the standard Prelude. Hence, this type class will only have *Int* and *Integer* as its members.

close *Integral* : `the only instances of Integral are Int and Integer`

The main advantage of a closed type class is that we know the fixed set of instances. Using this knowledge, we can influence the type inference process. As discussed in the introduction of Section 11.1, we can reject definitions early on (in case the set of instances for a certain type class is empty) or improve a type variable to a certain type (in case the set of instances is a singleton).

Example 11.3. Consider a function $f :: (Bounded\ a, Num\ a) \Rightarrow a \rightarrow a$. The type class *Bounded* contains all types that have a minimum and maximum value, including *Int* and *Char*. However, *Int* is the only numeric type among these. Hence, if both *Bounded* and *Num* are closed, then we may safely improve *f*'s type to $Int \rightarrow Int$.

The advantages of the *close* directive would be even higher if we drop the restrictions of Haskell 98 on type classes, because this directive allows us to reject incorrect usage of a type class early on. We illustrate this with an example.

Example 11.4. The type class *Similar* has one member function.

class *Similar a* **where**
  $(\approx) :: a \rightarrow a \rightarrow Bool$

instance *Similar Int* **where**
  $(\approx) = (==)$

We import these declarations, and close the type class *Similar*.

close *Similar* : `the only instance of Similar is Int.`

We define a function f, which requires that lists are in *Similar*.

$$f\ x\ xs = [x] \approx xs$$

GHC version 6.2 (without extensions) accepts the program above, although an instance for *Similar* $[a]$ must still be provided to resolve overloading. The type inferred for f is

$$f :: forall\ t.(Similar\ [t]) \Rightarrow t \rightarrow t \rightarrow Bool$$

although this type cannot be declared in a type signature for f.[2] This type makes sense: the function f can be used in a different module, provided that the missing instance declaration is supplied. However, if we intentionally close the type class, then we can generate an error for f at this point.

[2] In our opinion, it should be possible to include each type inferred by the compiler in the program. In this particular case, GHC suggests to use the Glasgow extensions, although these extensions are not required to infer the type.

In this light, the *close* directive may become a way to moderate the power of some of the language extensions by specifying cases where such generality is not desired. An alternative would be to take Haskell 98 as the starting point, and devise type class directives to selectively overrule some of the language restrictions. For instance, a directive such as **general** X could tell the compiler not to complain about predicates concerning the type class X that cannot be reduced to head-normal form. Such a directive would allow more programs. In conclusion, type class directives give an easy and flexible way to specify these local extensions and restrictions.

11.1.3 The disjoint directive

Our next directive deliberately reduces the set of accepted programs. In other words: the programs will be subjected to a stricter type discipline. The *disjoint* directive specifies that the instances of two type classes are disjoint, i.e., no type is shared by the two classes. A typical example of two type classes that are intentionally disjoint are *Integral* and *Fractional* (see the Haskell 98 Report [49]). If we end up with a type *(Fractional a, Integral a)* $\Rightarrow$ *....* after reduction, then we can immediately generate an error message, which can also explain that "fractions" are necessarily distinct from "integers". Note that without this directive, a context containing these two class assertions is happily accepted by the compiler, although it undoubtedly results in problems when we try to use this function. Acknowledging the senselessness of such a type prevents misunderstanding in the future. A *disjoint* directive can be defined as follows.

disjoint *Integral Fractional* :
```
  something which is fractional can never be integral
```

The numeric operations in Haskell's standard library are all overloaded, and accommodated in an intricate hierarchy of type classes (see Figure 11.1). Because *Floating* is a subclass of *Fractional* (each type in the former must also be present in the latter), the directive above implies that the type classes *Integral* and *Floating* are also disjoint.

In determining whether two type classes are disjoint, we base our judgements on the set of instance declarations for these classes, and not on the types implied by the instances. Therefore, we reject instance declarations $C\ a \Rightarrow C\ [a]$ and $D\ b \Rightarrow D\ [b]$ if C and D must be disjoint. A more liberal approach is to consider the set of instance types for C and D, so that their disjointness depends on other instances given for these type classes.

Example 11.5. Take a look at the following definition, which mixes fractions and integrals.

$$wrong\ x = (div\ x\ 2, x\ /\ 2)$$

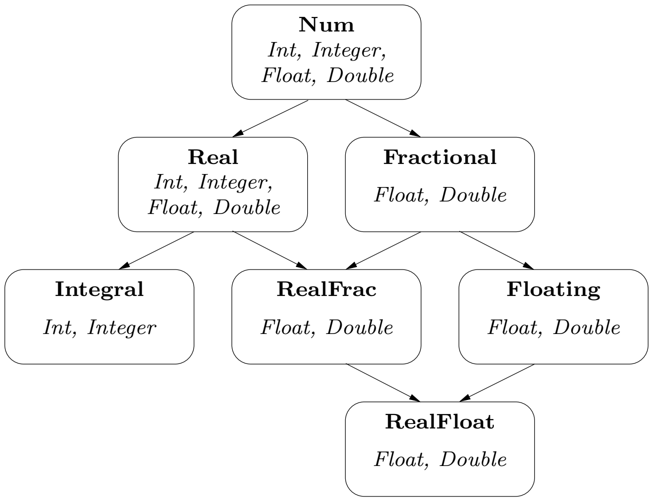


Figure 11.1. Hierarchy of Haskell's standard numeric type classes

Without directives, the following type is inferred for *wrong*.

$$wrong :: (Integral\ a, Fractional\ a) \Rightarrow a \rightarrow (a, a)$$

The directive **disjoint** *Integral Fractional* identifies the contradiction in the class context, and assists in reporting an appropriate error message for *wrong*.

11.1.4 The default directive

One annoying aspect of overloading is that seemingly innocent programs are in fact ambiguous. For example, *show* [] is not well-defined, since the type of the elements must be known (and showable) in order to display the empty list. This problem can only be circumvented by an explicit type annotation. A default declaration is included as special syntax in Haskell to help disambiguate overloaded numeric operations. This approach is fairly ad hoc, since it only covers the (standard) numeric type classes. Our example suggests that a programmer could also benefit from a more liberal defaulting strategy, which extends to other type classes. Secondly, the exact rules when defaulting should be applied are unnecessarily complicated (see the Haskell Report [49] for the exact specification). We think that a *default* declaration should be nothing but a type class directive, and that it should be placed amongst the other directives instead of being considered part of the programming language. Taking this viewpoint paves the way for other, more complex defaulting strategies as well.

One might wonder at this point why the original design is so conservative. Actually, the caution in applying a general defaulting strategy is justified since it changes the semantics of a program. Inappropriate defaulting, unnoticed by a programmer, is unquestionably harmful. By specifying *default* directives, the user has full control over the defaulting mechanism. A warning should be raised to inform the programmer that a class predicate has been defaulted. Although we do not advocate defaulting in large programming projects, it is unquestionably useful from time to time: for instance, for showing the result of an evaluated expression in an interpreter. Note that GHCi (the interpreter that comes with GHC) departs from the standard, and applies a more liberal defaulting strategy in combination with the emission of warnings, which works fine in practice.

Take a look at the following datatype definition for a binary tree with values of type a.

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Bin}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a) \mid \mathit{Leaf}\ \textbf{deriving}\ \mathit{Show}$$

A function to show such a tree can be derived automatically, but it requires a show function for the values stored in the tree. This brings us to the problem: *show Leaf* is of type *String*, but it is ambiguous since the tree that we want to display is polymorphic in the values it contains. We define default directives to remedy this problem.

default *Num* (*Int*, *Integer*, *Float*, *Double*)
default *Show* (*String*, *Bool*, *Int*)

The first directive is similar to the original default declaration, the second defaults predicates concerning the *Show* type class. Obviously, the types which we use as default for a type class must be a member of the class.

We apply the following procedure to default type variables. For type variable a, the set $P = \{X_1\ a, X_2\ a, \ldots, X_n\ a\}$ consists of all predicates in the context that contain a. This set fully determines the default type of a, if it exists. A default type exists only if at least one of the X_i has a default directive. We consider the default directives for each of the predicates in P in turn: for each of these directives, we determine the first type which satisfies all of P. If this type is the same for all default directives of P, then we choose this type for a. If the default directives cannot agree on their first choice, then defaulting for a does not take place.

Example 11.6. Let P be $\{\mathit{Num}\ a, \mathit{Show}\ a, \mathit{Eq}\ a, \mathit{Show}\ b\}$, and consider our defaulting procedure for this set of predicates. First, we try to default the type variable a. We select the three predicates in P containing a: for two of these predicates we have a default directive (declared above). For both directives, *Int* is the first type that is an instance of all three type classes: *Num*, *Show*, and *Eq*. (We assume that *String* and *Bool* are not in *Num*.) Hence, a is defaulted to *Int*. Likewise, we default b to *String*.

If we have an instance for *Num String*, then we cannot default a. With this new instance, the default directive for *Show* selects *String* as the first type which is in all three type classes. Because *Num* selects a different type (*Int*), we refrain from choosing a default for a.

If default directives are given for a type class and for its subclass, we should check that the two directives are coherent. For instance, *Integral* is a subclass of

Num, and hence we expect that defaulting *Integral a* and *Num a* has the same result as defaulting only *Integral a*.

Considering defaulting as a directive allows us to design more precise defaulting strategies. For instance, we could have a different default strategy for showing values of type $[a]$: this requires maintaining information about the instantiated type of the overloaded function. We illustrate this with an example.

Example 11.7. Consider the expression *show* $[\,]$. Overloading cannot be resolved for this expression: *show* has the polymorphic type $\forall a.\mathit{Show}\ a \Rightarrow a \to \mathit{String}$, which is instantiated to (for instance) $v_5 \to \mathit{String}$ and the predicate $\mathit{Show}\ v_5$. Because *show* is applied to the empty list, v_5 becomes $[v_8]$ because of unification. The predicate, which is now $\mathit{Show}\ [v_8]$, is simplified to $\mathit{Show}\ v_8$. This is the predicate that cannot be resolved: the information that it originates from the function *show*, and that it was $\mathit{Show}\ [v_8]$ before context reduction enables us to choose a different default strategy.

11.1.5 Error message attributes

The error messages given so far are context-insensitive, which is not sufficient for a real implementation. Again, error message attributes (see Section 9.2.1, page 161) are used to display context dependent information. We restrict ourselves to an example for the *close* directive.

Example 11.8. Consider the following directive to close the *Show* type class.

```
close Show :
  The expression @expr.pp@ at @expr.range@ has the type
  @expr.gentype@. This type is responsible for the introduction
  of the class predicate @errorpredicate@, which is not an instance
  of @typeclass@ due to the close directive defined at
  @directive.range@.
```

The attributes in the error message are replaced by information from the actual program. For instance, `@directive.range@` is changed into the location where the *close* directive is defined, and `@expr.pp@` is unfolded to a pretty printed version of the expression responsible for the introduction of the erroneous predicate. We can devise a list of attributes for each directive. These lists differ: in case of the *disjoint* directive, for instance, we want to refer to the origin of both class predicates that contradict.

A complicating factor is that the predicate at fault may not be the predicate which was introduced. Reducing the predicate $\mathit{Eq}\ [(\mathit{String}, \mathit{Int} \to \mathit{Int})]$ will eventually lead to $\mathit{Eq}\ (\mathit{Int} \to \mathit{Int})$. We would like to communicate this reasoning to the programmer as well, perhaps by showing some of the reduction steps.

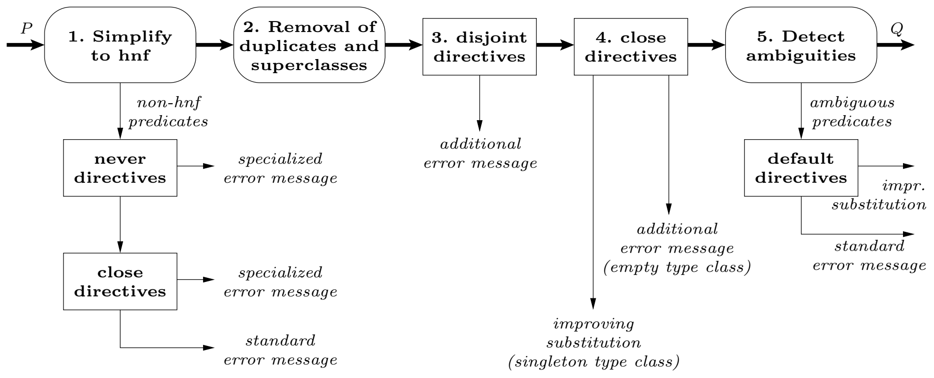


Figure 11.2. Context reduction with type class directives for Haskell 98

11.2 Implementation

For each binding group, we perform context reduction, which serves to simplify sets of type class predicates, and to report predicates that cannot be resolved. We continue with a discussion on how the four type class directives can be incorporated into this process. Figure 11.2 gives an overview. The first, second, and fifth step correspond to phases of the traditional approach: simplification using instances, removal of superfluous predicates, and detection of ambiguous predicates, respectively. The bold horizontal line reflects the main process in which the set of predicates P is transformed into a set of predicates Q.

The first modification concerns the predicates that cannot be simplified to head-normal form. If a *never* or *close* directive is specified for such a predicate, then we report the specialized error message that was declared with the directive (precedence is given to the *never* directives). Otherwise, we proceed as usual and report a standard error message.

The *disjoint* directives and closed type classes are handled after removal of duplicates and superclasses. At this point, the predicates to consider are in head-normal form. A *disjoint* directive creates an error message for a pair of predicates that is in conflict. Similarly, if we can conclude from the closed type classes that no type meets all the requirements imposed by the predicates for a given type variable, then an error message is constructed. If we, on the other hand, discover that there is a single type which meets the restrictions, then we assume this type variable to have that particular type. This is an improving substitution [32]. Because we consider all predicates involving a certain type variable at once, the restrictions of Haskell 98 guarantee that improving substitutions cannot lead to more reduction steps.

To recognize empty or singleton sets for combinations of type classes, we need to merge information from instance declarations, the superclass hierarchy, and the declared type class directives. The following example illustrates the interaction between directives and instance declarations.

Example 11.9. Assume we have type classes X, Y, and Z. Furthermore, we know that *Int* and *Bool* are both instances of X, and *Int* and *Char* are the only two instances of Z. In this context, we write the following type class directives.

disjoint X Y : `the type classes X and Y are disjoint`
close Z : `only Int and Char are members of Z`

Because instance declarations and *disjoint* directives interact, we (implicitly) have two more *never* directives.

never Y *Int* : `(because Int in X, and disjoint X Y)`
never Y *Bool* : `(because Bool in X, and disjoint X Y)`

These *never* directives may trigger more improving substitutions. Suppose we have the type class predicates $(Y\ a, Z\ a)$. Because Z is closed, a can only be *Int* or *Char*. The implicit *never* directives eliminate the first option. Hence, we get $[a := \mathit{Char}]$.

The use of improving substitutions leads to more programs being accepted, while others are now rejected. The sum of their effects can be hard to predict, and not something to rely on in large programming projects. If desired, the improving substitutions can be switched off altogether: even without improving substitutions, the *never*, *close*, and *disjoint* directives can be quite useful.

Finally, we modify the fifth step (in Figure 11.2), in which ambiguous predicates are detected and reported. We try to avoid reporting ambiguous predicates by inspecting the given *default* directives. Defaulting a type variable (as described in Section 11.1.4) results again in an improving substitution. A standard error message is produced for ambiguous predicates that cannot be defaulted.

11.3 Generalization of directives

In this section, we sketch a generalization of the first three type class directives. This part has not been implemented, but gives an idea how far we expect type class directives can go, and what benefits accrue.

Essentially, a type class describes a (possibly infinite) set of types, and most of the proposed directives can be understood as constraints over such sets. In fact, they describe invariants on these sets of types, enriching the means of specification in Haskell, which is limited to membership of a type class (instance declaration), and a subset relation between type classes (class declaration).

We present a small language to specify invariants on the class system. The language is very expressive, and it may be necessary to restrict its power for reasons of efficiency and decidability, depending on the type (class) system to which it is added.

Constraint	::=	*Type EltOp Set* | *Set SetOp Set*
Set	::=	*Set BinOp Set* | *SetLiteral* | *Class*
SetLiteral	::=	$\{\}$ | $\{$ *Type* (, *Type*)* $\}$
EltOp	::=	$\in$ | $\notin$
SetOp	::=	$\subseteq$ | $=$ | $\supseteq$
BinOp	::=	$\cap$ | $\cup$ | $-$

Each constraint can be followed by an error message. If necessary, syntactic sugar can be introduced for special directives such as *never* and *disjoint*.

Example 11.10. never and *disjoint* directives can be translated into this new language straightforwardly. This translation also helps to identify interactions between directives using principles in set theory.

never *Eq* $(a \to b)$	*becomes*	$(a \to b) \notin Eq$
disjoint *Integral Fractional*	*becomes*	$Integral \cap Fractional = \{\}$

A *close* directive cannot be translated directly since we do not have the instances of the closed type class. If we know all members, we can close a type class as follows.

close *Integral*	*becomes*	$Integral = \{Int, Integer\}$

Instead of writing $\{Int, Integer\}$, we could also introduce notation for the members of *Integral* at the moment we close the type class. This set of types (when we close the type class) is a subset of, but not necessarily the same as *Integral*, which consists of all declared instances.

Example 11.11. Take a look at the following invariants on type classes.

$Monad \ = \{Maybe, [\,], IO\}$: `only Maybe, [], and IO are monads today.`
$Read \quad = Show$
$Egglayer \cap Mammal \subseteq \{Platypus\}$

The first example directive prevents new instances for the *Monad* type class, while $Read = Show$ demands that in this module (and all modules that import it) the instances for *Show* and *Read* are the same. A nice example of an invariant is the third directive, which states that only the duckbilled platypus can be both in the type class for egg layers and in *Mammal*. This directive might be used to obtain an improving substitution (as discussed in Section 11.1): if we have the predicates *Mammal a* and *Egglayer a*, then *a* must be *Platypus*. This example shows that the directives can be used to describe domain specific invariants over class hierarchies.

11.4 Specialized type rules (revisited)

In Chapter 9, we introduced specialized type rules to improve type error messages. This facility is especially useful for domain specific extensions to a base language (such as Haskell), because the developer of such a language can now specify error messages which refer to concepts in the domain to replace the error messages phrased in terms of the underlying language. We present an extension of these type rules which allows class assertions (to deal with overloading) among the equality constraints. This extension has been implemented in the Helium compiler [26].

Example 11.12. Consider the function *spread*, which returns the difference between the smallest and largest value of a list, and a specialized type rule for this function, given that the function is applied to one argument.

$$\begin{aligned}&\mathit{spread} :: (\mathit{Ord}\ a, \mathit{Num}\ a) \Rightarrow [\,a\,] \rightarrow a\\&\mathit{spread}\ \mathit{xs} = \mathit{maximum}\ \mathit{xs} - \mathit{minimum}\ \mathit{xs}\end{aligned}$$

For this overloaded function, we write the following specialized type rule.

$$\frac{\mathit{xs} :: t_1;}{\mathit{spread}\ \mathit{xs} :: t_2;}$$

$t_1 \equiv [\,t_3\,]$: `@xs.pp@ must be a list`
$t_3 \equiv t_2$: `@expr.pp@ should return a value of type @t3@`
$\mathit{Eq}\ t_2$: `@t2@ is not an instance of Eq, let alone Ord or Num`
$\mathit{Ord}\ t_2$: `@t2@ should have a linear ordering imposed on it`
$\mathit{Num}\ t_2$: `@t2@ should allow numerical operations`

The first equality constraint states that the type of *xs* (t_1) is a list type, with elements of type t_3 (t_3 is still unconstrained at this point). The next equality constraint constrains the type t_3 to be the same as the type of *spread xs*. Note that the listed constraints are verified from top to bottom, and this fact can be exploited to yield very precise error messages.

Class assertions are listed after the equality constraints, and again we exploit the order of specification. Although membership of *Ord* or *Num* implies membership of *Eq*, we can check the latter first, and give a more precise error message in case it fails. Only when *Eq* t_2 holds, do we consider the class assertions *Ord* t_2 and *Num* t_2. Note that the assertion *Eq* t_2 does not change the validity of the rule.

Context reduction for a binding group takes place after having solved the equality constraints of that binding group. This implies that listing class assertions before the equality constraints makes little sense, and only serves to confuse people. Therefore, we disallow this.

Equality constraints can be moved into the deduction rule, in which case it is associated with a standard error message. This facility is essential for practical reasons: it should be possible to only list those constraints for which we expect special treatment. Similarly, we may move a class assertion into the deduction rule. Notwithstanding, this assertion is checked after all the equality constraints.

All specialized type rules are automatically examined so that they leave the underlying type system unchanged. This is an essential feature, since a mistake is easily made when writing these rules. In Section 9.3.2, we presented a procedure to compare the set of constraints implied by the specialized type rule (say S) with the set that would have been generated by the standard inference rules (say T). Broadly speaking, a type rule is only accepted if S equals T under the entailment relation. Soundness of a specialized type rule with class assertions is checked by combining entailment for type class predicates and entailment for equality constraints.

11.5 Summary

Elements of type class directives can be found in earlier papers: closed type classes were mentioned by Shields and Peyton Jones [57], while the concepts of disjoint type classes and type class complements were considered by Glynn et al. [19]. Type class directives lead to improving substitutions which are part of the framework as laid down by Jones [32]. All these efforts are focused on the type system, while we concentrate on giving good feedback by adding high-level support to compilers via compiler directives. Moreover, we generalize these directives to invariants over type classes.

The techniques described in this chapter offer a solution to compensate the effect that the introduction of overloading (type classes) has on the quality of reported error messages. In general, the types of overloaded functions are less restrictive, and therefore some errors may remain undetected. At the same time, a different kind of error message is produced for unresolved overloading, and these errors are often hard to interpret.

A number of type class directives have been proposed to remedy the loss of clarity in error messages, and we have indicated how context reduction can be extended to incorporate these directives. The directives have the following advantages.

- Tailor-made, domain-specific error messages can be reported for special cases.
- Functions for which we infer a type scheme with a suspicious class context can be detected (and rejected) at an early stage.
- An effective defaulting mechanism assists to disambiguate overloading.
- Type classes with a limited set of instances help to improve and simplify types.

Furthermore, we have added type class predicates to the specialized type rules, and the soundness check has been generalized accordingly.

12

Conclusion and future work

We have presented the constraint-based type inference framework Top, which has been designed primarily to report informative type error messages. The constraints of this framework cover the entire Haskell 98 standard, including overloading with type classes. We summarize the contributions of our work.

12.1 Conclusion

To a great extent, the "quality of a type error message" depends on personal preferences, and is influenced by factors such as level of expertise and style of programming. Because there is no single best type inference algorithm which suits everyone's needs, we developed an infrastructure which is highly customizable.

In this thesis, we pursued a constraint-based approach to type inference. This approach offers a firm basis to experiment with various existing type inference strategies, which resulted in the design of new techniques to improve the error reporting facility.

In our framework, each type constraint carries around information: for instance, where the constraint was created, and why. This has proven to be a powerful method to manage the flow of information during constraint-based type inference: constraint information is used by heuristics during constraint solving, and, ultimately, it is used to create the error message.

Given a constraint solver that considers constraints sequentially, the relative order of the constraints strongly influences where an inconsistency is detected (Chapter 5). By using specific constraint orderings, we can emulate well-known inference algorithms, such as $\mathcal{W}$ and $\mathcal{M}$. In fact, our approach generalizes various Hindley-Milner type inference algorithms. A correctness proof of the constraint-based approach was presented in Section 4.5.

The proposed type inference framework has been implemented in the Helium compiler [26], which has been used for several introductory courses to functional programming at Utrecht University. The implementation confirms that the described techniques are feasible in practice, and it proves that our approach scales well to a full-blown programming language. Constraint-collecting type rules for nearly the entire Haskell 98 standard have been discussed in Chapter 6.

In Chapter 7, we have presented type graphs: a data structure to represent substitutions, which can be in an inconsistent state, and which keeps track of the reasons for a unification. With type graphs, we can analyze a program in a global way, and circumvent the notorious left-to-right bias present in most type inference algorithms. Extra overhead caused by the type graph increases compilation time. However, by partitioning constraint sets into subproblems, and by combining constraint solvers, we can get the best of two worlds: an efficient constraint solver for well-typed parts of a program, while keeping highly accurate type error messages for type inconsistencies. A number of heuristics that work on type graphs help to pinpoint the most likely source of a mistake, including heuristics that search for possible ways to correct the type error.

Type inference directives (Chapter 9 and Chapter 11) let users influence the type inference process, and adapt the reported type error messages to their own liking. For a domain-specific library, for instance, directives provide error messages in the domain of the library, which greatly enhances their usefulness. Furthermore, directives help to anticipate known pitfalls in a library. A special collection of directives has been presented to improve the quality of type error messages for languages that supports overloading.

In Chapter 10, we have shown how the Top framework can be extended to support type classes. The extended framework covers the entire Haskell 98 standard. Our solution for dealing with qualifiers departs from the standard approach taken in most compilers: type class predicates are collected and resolved at a global level.

12.2 Future work

We conclude with a list of possible directions for further research.

- *Repair heuristics.* Throughout this thesis, we have suggested techniques to automatically repair ill-typed programs, for instance by rearranging the abstract syntax tree. Repair directives, presented in Section 9.2.5, let programmers define their own transformations to correct programs. By composing these small program corrections, we can obtain more complex transformations. A special-purpose language is needed in which all kinds of corrections can be declared. An implementation of such a facility allows further experimentation with automatically repairing programs.
- *Generalization of type class directives.* A language to specify invariants on the class system (see Section 11.3) seems to be a promising direction, and this requires further investigation. The more expressive such a language becomes, the more need there is for some form of analysis of these invariants. In fact, these invariants can be seen as set constraints (see Aiken [1] for an overview), which have a long history. However, this generality is not necessary to cover our type class directives, and a simpler approach would suffice here.

- *Extensions to specialized type rules.* Extensions which we are taking into consideration are adding flexibility in specifying the priority of specialized type rules, and extending the facilities for phasing: at this point, phasing is a purely global operation, which might be too coarse for some applications. By their nature, the specialized type rules follow the structure of the abstract syntax tree. However, syntactic matching is not always satisfactory, and we could benefit from a more liberal matching strategy.

- *Type inference and programming environments.* Integrating type inference in a programming environment adds new dimensions to reporting understandable type error messages. Not only do we have different ways to present type information at our disposal, it also becomes easier for a user to extract insightful information from the type inference process. At its simplest, this takes the form of inspecting inferred types of subexpressions.
  Instead of only suggesting program fixes to a programmer, a programming environment would enable us to truly apply the corrections, thereby correcting type errors automatically. To fully integrate type inference in a dedicated program editor, we need incremental type inference. Essentially, incremental inference in a constraint-based setting is nothing but removing and inserting constraints: a technique already used by some of our heuristics. Note that having previous versions of a program available can also be helpful for heuristics to determine the most likely source of a type error.

- *Formalizing type class directives.* The type class directives presented in Chapter 11 change the underlying type system: if these directives are used, then a different set of programs is accepted. We see the need for a formal approach, so that the effects of our directives can be fully understood. Constraint handling rules (introduced by Frühwirth [16], and used by Glynn et al. [19]) are a good starting point for such a formalization.

- *Extending the class system.* Another direction is to explore directives for a number of the proposed extensions to the type class system [50], and to come up with new directives to alleviate the problems introduced by these extensions. For Haskell in particular, most of the proposals to extend the language with new features result in increasingly complicated type systems. The more complicated a type system becomes, the more the need arises for high quality feedback about the inference process, and explanations of encountered inconsistencies in particular – also for experienced programmers.

- *Logging facility.* During introductory courses on functional programming at Utrecht University, we have set up a logging facility to record compiled programs. Thousands of compiled programs produced by participating students were stored on disk, and this data collection can give insight in the learning process over time. At the moment, the data has been analyzed only superficially: a more thorough study is required to draw profound conclusions. We plan to use the data to justify our heuristics, and, if necessary, to adjust these.

- *Other analyses.* Although we have focused solely on type inference for functional languages, the constraint-based approach we followed lends itself to other program analyses as well (such as strictness analysis and soft typing). Constraints separate specification and implementation of analyses, and by storing constraint information with each constraint, we can manage to hold on to the information of our interest. Especially our compiler directive approach could be applied to guide other analyses.

References

1. A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2–3):79–111, 1999.
2. A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *FPCA'93: Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 31–41. ACM Press, June 1993.
3. L. Augustsson. Implementing Haskell overloading. In *FPCA'93: Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 65–73. ACM Press, June 1993.
4. M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2(4):17–30, March 1993.
5. K. L. Bernstein and E. W. Stark. Debugging type errors (full version). Technical report, State University of New York at Stony Brook, November 1995.
6. R. Bird. *Introduction to Functional Programming using Haskell*. Series in Computer Science. Prentice Hall, second edition, April 1998.
7. O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ICFP'01: Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming*, pages 193–204. ACM Press, September 2001.
8. V. Choppella. *Unification Source-tracking with Application to Diagnosis of Type Inference*. PhD thesis, Department of Computer Science, Indiana University, August 2002.
9. V. Choppella and C. T. Haynes. Diagnosis of ill-typed programs. Technical Report 426, Indiana University, 1994.
10. P. T. Cox. Finding backtrack points for intelligent backtracking. In *Implementations of Prolog*, pages 216–233. Ellis Horwood/Halsted Press/Wiley, 1984.
11. L. Damas and R. Milner. Principal type schemes for functional programs. In *POPL'82: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.
12. T. B. Dinesh and F. Tip. A slicing-based approach for locating type errors. In *DSL'97: Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 77–88, October 1997.
13. G. J. Duck, S. Peyton Jones, P. J. Stuckey, and M. Sulzmann. Sound and decidable type inference for functional dependencies. In D. Schmidt, editor, *ESOP'04: Proceedings of the 13th European Symposium on Programming*, volume 2986 of *LNCS*, pages 49–63. Springer Verlag, 2004.

14. D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.
15. R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.
16. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
17. M. Gandhe, G. Venkatesh, and A. Sanyal. Correcting errors in the curry system. In V. Chandru and V. Vinay, editors, *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *LNCS*, pages 347–358. Springer Verlag, 1996.
18. GHC Team. *The Glasgow Haskell Compiler*. `http://www.haskell.org/ghc`.
19. K. Glynn, P. J. Stuckey, and M. Sulzmann. Type classes and constraint handling rules. In *First Workshop on Rule-Based Constraint Reasoning and Programming*, July 2000.
20. C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In P. Degano, editor, *ESOP'03: Proceedings of the 12th European Symposium on Programming*, LNCS, pages 284–301, April 2003.
21. B. Heeren and J. Hage. Type class directives. In M. Hermenegildo and D. Cabeza, editors, *PADL'05: Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, volume 3350 of *LNCS*, pages 253–267. Springer Verlag, January 2005.
22. B. Heeren, J. Hage, and S. D. Swierstra. Constraint based type inferencing in Helium. In M.-C. Silaghi and M. Zanker, editors, *Workshop Proceedings of Immediate Applications of Constraint Programming*, pages 59–80, Cork, September 2003.
23. B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. In *ICFP'03: Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, pages 3–13. ACM Press, 2003.
24. B. Heeren and D. Leijen. Functioneel programmeren met Helium. In *NIOC 2004 Proceedings*, pages 73–82. Passage, November 2004.
25. B. Heeren and D. Leijen. Gebruiksvriendelijke compiler voor het onderwijs. *Informatie*, 46(8):46–50, October 2004.
26. B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for learning Haskell. In *Haskell'03: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 62–71. ACM Press, 2003.
27. J. R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.
28. F. Huch, O. Chitil, and A. Simon. Typeview: A tool for understanding type errors. In M. Mohnen and P. Koopman, editors, *IFL'00: Proceedings of the 12th International Workshop on Implementation of Functional Languages*, volume 2011 of *LNCS*, pages 63–69. RWTH Aachen, Springer Verlag, September 2000.
29. P. Hudak. *The Haskell School of Expression: Learning Functional Programming Through Multimedia*. Cambridge University Press, New York, 2000.
30. G. F. Johnson and J. A. Walz. A maximum flow approach to anomaly isolation in unification-based incremental type inference. In *POPL'86: Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 44–57, St. Petersburg, FL, January 1986. ACM Press.
31. M. P. Jones. *Qualified Types: Theory and Practice*. PhD thesis, University of Nottingham, November 1994.
32. M. P. Jones. Simplifying and improving qualified types. In *FPCA'95: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 160–169, June 1995.

33. M. P. Jones. Typing Haskell in Haskell. In H. J. M. Meijer, editor, *Proceedings of the 1999 Haskell Workshop*, pages 7–22, October 1999. Published in Technical Report UU-CS-1999-28, Utrecht University, The Netherlands.
34. M. P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *ESOP'00: Proceedings of the 9th European Symposium on Programming*, volume 1782 of *LNCS*, pages 230–244. Springer Verlag, March 2000.
35. M. P. Jones et al. *The Hugs 98 system.* OGI and Yale, `http://www.haskell.org/hugs`.
36. O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, July 1998.
37. O. Lee and K. Yi. A generalized let-polymorphic type inference algorithm. Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology, March 2000.
38. B. J. McAdam. On the unification of substitutions in type inference. In K. Hammond, T. Davie, and C. Clack, editors, *IFL'98: Proceedings of the 10th International Workshop on Implementation of Functional Languages*, volume 1595 of *LNCS*, pages 139–154. Springer Verlag, September 1998.
39. B. J. McAdam. Generalising techniques for type debugging. In P. Trinder, G. Michaelson, and H.-W. Loidl, editors, *Trends in Functional Programming*, volume 1, pages 50–59, Bristol, UK, 2000. Intellect.
40. B. J. McAdam. How to repair type errors automatically. In K. Hammond and S. Curtis, editors, *Trends in Functional Programming*, volume 3, pages 87–98, Bristol, UK, 2002. Intellect.
41. B. J. McAdam. *Repairing Type Errors in Functional Programs.* PhD thesis, Laboratory for Foundations of Computer Science, The University of Edinburgh, 2002.
42. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, August 1978.
43. M. Neubauer and P. Thiemann. Discriminative sum types locate the source of type errors. In *ICFP'03: Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, pages 15–26, New York, 2003. ACM Press.
44. M. Neubauer and P. Thiemann. Haskell type browser. In *Haskell'04: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 92–93. ACM Press, 2004. Demonstration Abstract.
45. M. Odersky and K. Läufer. Putting type annotations to work. In *POPL'96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–67, New York, 1996.
46. M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.
47. M. Odersky, P. Wadler, and M. Wehr. A second look at overloading. In *FPCA'95: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 135–146. ACM Press, June 1995.
48. J. Peterson and M. P. Jones. Implementing type classes. In *PLDI'93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 227–236. ACM Press, 1993.
49. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.
50. S. Peyton Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Proceedings of the 1997 Haskell Workshop*, June 1997.
51. S. Peyton Jones and M. Shields. Practical type inference for arbitrary-rank types. 74 pages, Submitted to The Journal of Functional Programming, April 2004.

52. B. C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.
53. G. S. Port. A simple approach to finding the cause of non-unifiability. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 651–665. The MIT Press, 1988.
54. M. Rittri. Finding the source of type errors interactively, 1993. Draft, Department of Computer Science, Chalmers University of Technology, Sweden.
55. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
56. M. M. Schrage. *Proxima – a presentation-oriented editor for structured documents.* PhD thesis, Utrecht University, The Netherlands, October 2004.
57. M. Shields and S. Peyton Jones. Object-oriented style overloading for Haskell. In *BABEL'01: Workshop on Multi-Language Infrastructure and Interoperability*, September 2001.
58. H. Soosaipillai. An explanation based polymorphic type checker for Standard ML. Master's thesis, Department of Computer Science, Heriot-Watt University, Edinburgh, Scotland, September 1990.
59. P. J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in Haskell. In *Haskell'03: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 72–83, New York, 2003. ACM Press.
60. P. J. Stuckey, M. Sulzmann, and J. Wazny. Improving type error diagnosis. In *Haskell'04: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 80–91. ACM Press, 2004.
61. S. D. Swierstra. Combinator parsers: From toys to tools. In G. Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41. Elsevier Science Publishers, August 2001.
62. S. D. Swierstra, A. I. Baars, and A. Löh. The UU-AG attribute grammar system. `http://www.cs.uu.nl/groups/ST`.
63. S. Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley Longman, second edition, 1999. `http://www.cs.ukc.ac.uk/people/staff/sjt/craft2e`.
64. M. Wand. Finding the source of type errors. In *POPL'86: Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 38–43. ACM Press, January 1986.
65. J. Yang. Explaining type errors by finding the source of a type conflict. In P. Trinder, G. Michaelson, and H.-W. Loidl, editors, *Trends in Functional Programming*, volume 1, pages 59–68, Bristol, UK, 2000. Intellect.
66. J. Yang. *Improving polymorphic type explanations.* PhD thesis, Heriot-Watt University, May 2001.
67. J. Yang and G. Michaelson. A visualisation of polymorphic type checking. *Journal of Functional Programming*, 10(1):57–75, January 2000.
68. J. Yang, G. Michaelson, and P. Trinder. Explaining polymorphic types. *The Computer Journal*, 45(4):436–452, 2002.
69. J. Yang, G. Michaelson, P. Trinder, and J. B. Wells. Improved type error reporting. In M. Mohnen and P. Koopman, editors, *IFL'00: Proceedings of the 12th International Workshop on Implementation of Functional Languages*, volume 2011 of *LNCS*, pages 71–86. RWTH Aachen, Springer Verlag, September 2000.

Samenvatting

Computerprogramma's nemen een steeds belangrijkere plaats in binnen de hedendaagse maatschappij. Dat programma's fouten bevatten is eerder regel dan uitzondering; dit geldt zowel voor kleine applicaties als voor grootschalige softwareprojecten. De consequenties van een foutief programma hangen voornamelijk af van het toepassingsgebied. In de meeste gevallen zal een fout slechts irritatie opwekken bij de gebruiker, bijvoorbeeld omdat een computerprogramma onverwachts stopt, of omdat een bepaalde internetpagina niet getoond kan worden. In andere gevallen kan een foutief programma een grotere impact hebben. Te denken valt aan beveiligingsproblemen van een bedrijfsnetwerk, veroorzaakt door falende software, of aan de gevolgen van een fout in de computerprogrammatuur voor het besturen van een vliegtuig. Het ontwerpen van betrouwbare en foutloze computerprogramma's is één van de grootste uitdagingen voor de software-industrie.

Om de kwaliteit van een programma te kunnen garanderen is het belangrijk om in een vroeg stadium programmeerfouten te detecteren. Dit kan door een programma gedurende het ontwikkelingsproces uitvoerig te testen. Echter, zelfs de meest zorgvuldige testprocedure kan onmogelijk alle fouten aan het licht brengen. Een aanvullende methode om fouten op te sporen is de programmacode automatisch te laten analyseren. Deze controle vindt plaats zonder dat het geanalyseerde programma wordt uitgevoerd, en staat bekend als *statische analyse*. Dit soort analyses hebben twee belangrijke voordelen ten opzichte van het testen van software. Ten eerste vindt er een snelle terugkoppeling plaats naar de ontwerper van het programma. De vroegtijdige rapportage van fouten zorgt ervoor dat deze direct en efficiënt hersteld kunnen worden. Een bijkomend voordeel is dat de programmeur geconfronteerd wordt met zijn fouten, en niet de gebruiker. Ten tweede kan een analyse het optreden van bepaalde foutsituaties categorisch uitsluiten, wat de kwaliteit van de programmatuur ten goede komt.

Een moderne compiler (vertaler) biedt vele geavanceerde programma-analyses aan om vroegtijdig fouten in programma's te rapporteren. Het genereren van efficiënt executeerbare code is traditioneel een belangrijk aandachtsveld voor vertalerbouwers. Vaak wordt echter minder aandacht geschonken aan de presentatie van foutmeldingen. Het toevoegen van duidelijke en behulpzame foutmeldingen aan een

compiler is niet alleen veel werk, maar is mede afhankelijk van factoren zoals de programmeerstijl en de deskundigheid van de programmeur. Met name voor de meer geavanceerde analyses geldt dat de foutmeldingen vaak cryptisch van aard zijn. Een gevolg hiervan is dat het voor een programmeur dikwijls onduidelijk is waarom een programma als incorrect wordt beschouwd, en hoe de gemaakte fout hersteld kan worden.

Ter illustratie van het probleem bekijken we de werking van een geautomatiseerde spelling- en grammaticacontrole voor een stuk Nederlandse tekst. Hoe praktisch zou zo'n toepassing zijn als enkel wordt vermeld of de tekst in zijn geheel goed dan wel fout is? Het is wenselijk dat de locatie van een gevonden fout zo nauwkeurig mogelijk wordt vermeld, al kan deze niet altijd eenduidig worden aangewezen. De toepassing wordt pas echt gebruiksvriendelijk als ook wordt aangegeven hoe een gemaakte fout verbeterd zou kunnen worden, eventueel met verwijzingen naar het *Groene Boekje*. Een soortgelijke functionaliteit is ook wenselijk voor de fouten die een vertaler rapporteert.

Functionele programmeertalen

Welke programma-analyses kunnen worden toegepast hangt voornamelijk af van de gebruikte programmeertaal. De bekendste familie van programmeertalen heeft een imperatief karakter. Een programma dat geschreven is in zo'n taal bestaat uit een serie van instructies die na elkaar moeten worden uitgevoerd. Functionele programmeertalen bieden een alternatief waarbij een programma wordt beschreven door een verzameling van (wiskundige) functies. Deze talen zijn gebaseerd op de principes van de lambda-calculus. Recente functionele programmeertalen zijn onder andere ML en Haskell.

Karakteristiek voor deze laatste twee talen is dat ze impliciet getypeerd zijn. Een type beschrijft de waarden die een expressie aan kan nemen, zoals "een getal", "een lijst met getallen", "een paar van twee karakters" of "een functie". De programmeur hoeft niet langer voor iedere expressie het type op te schrijven, maar kan vertrouwen op een krachtig mechanisme om automatisch de types af te leiden (te infereren). Tijdens het infereren van de ontbrekende types kunnen inconsistenties in een programma worden ontdekt, en deze worden vervolgens gerapporteerd aan de programmeur. Als er geen fout gevonden wordt, dan zijn alle functies in een programma gegarandeerd op een juiste manier toegepast. Door het controleren van de types kan een significant deel van de fouten worden geëlimineerd.

Het interpreteren van typeringsfoutmeldingen wordt in het algemeen als lastig ervaren, en vergt enige training. Voor de ervaren programmeur is dit geen probleem: in sommige gevallen zal er zelfs nauwelijks naar de foutmelding gekeken worden. Voor beginnende programmeurs is de situatie anders: zij zullen voornamelijk het gevoel krijgen te worden tegengewerkt door het type-inferentieproces aangezien de foutmeldingen niet voldoende duidelijk maken wat er fout is, en hoe dit verbeterd kan worden. Deze onduidelijkheid belemmert het aanleren en waarderen van een functionele programmeertaal. Dit proefschrift beschrijft een aantal technieken om betere typeringsfoutmeldingen te genereren voor een programmeertaal zoals Haskell.

Waarom is het lastig om duidelijke typeringsfoutmeldingen te rapporteren? Omdat het niet verplicht is alle types op te schrijven kan er gemakkelijk een mismatch ontstaan tussen de types verwacht door de programmeur en de types afgeleid door de compiler. Dit soort verschillen kan elders in het programma problemen veroorzaken. Daarnaast ondersteunt Haskell hogere-orde functies (functies kunnen als argument meegegeven worden aan functies) en polymorfie (een functie kan worden gebruikt met verschillende types). Beide concepten zijn een extra uitdaging voor het type-inferentiemechanisme.

Type-inferentie met constraints

Het rapporteren van duidelijke foutmeldingen komt feitelijk neer op het bijhouden van voldoende informatie en het op een handige manier organiseren van deze informatie. Foutmeldingen van traditionele inferentie-algoritmen hebben te lijden onder de mechanische wijze waarop de algoritmen te werk gaan. Het is beter om van deze benadering af te stappen en een programma in z'n geheel te bekijken zoals een expert dat zou doen. Een dergelijke globale analyse kan gebruik maken van een aantal heuristieken die de kennis van een deskundige vastleggen. Deze heuristieken kunnen onder meer bereiken dat veelgemaakte fouten worden herkend, en dat zij op een inzichtelijke manier worden gemeld aan de programmeur.

Het bovenstaande doel hebben we bereikt door gebruik te maken van *constraints*. Deze aanpak is ook voor andere programma-analyses uiterst succesvol gebleken. Een constraintverzameling beschrijft nauwkeurig de relaties tussen de types in een programma, en levert een natuurlijke splitsing op van de specificatie van de analyse (het verzamelen van de constraints) en de implementatie (het oplossen van de constraints). Deze splitsing heeft als pluspunt dat diverse oplossingsmethoden voor een constraintverzameling naast elkaar kunnen bestaan. Meerdere oplossingsmethoden zijn ondergebracht in het Top framework voor type-inferentie met goede foutmeldingen.

Top foutmeldingen

Het Top framework stelt vertalerbouwers in staat om een compiler op eenvoudige wijze uit te breiden met type-inferentie, daarbij gebruikmakend van allerlei technieken voor het genereren van goede foutmeldingen. De naam van dit framework staat enerzijds voor *"Typing Our Programs"* en anderzijds voor het symbool $\top$ uit de wiskunde, dat een foutsituatie representeert. Een uitgangspunt bij het ontwerp van Top is dat het inferentieproces kan worden afgestemd op de wensen en de deskundigheid van een programmeur. Deze flexibiliteit is noodzakelijk aangezien de gewenste informatie bij een foutmelding nogal persoonlijk is. Eén algoritme voor type-inferentie dat optimaal functioneert voor iedereen bestaat om deze reden niet. Top is gebaseerd op constraints, en maakt de aanname dat iedere constraint zijn eigen informatie met zich meedraagt. Deze informatie beschrijft bijvoorbeeld waarom de constraint is gegenereerd, en waar hij vandaan komt uit het oorspronkelijke programma. Verder kan de informatie van een constraint heuristieken aansturen, of een foutmelding aandragen.

Naast het verzamelen en het oplossen van constraints onderscheidt het framework nog een extra fase: het ordenen van de verzamelde constraints alvorens deze worden opgelost. De volgorde waarin constraints worden afgehandeld is niet relevant voor de oplossing die voor een verzameling gevonden wordt, maar beïnvloedt wel in sterke mate het moment waarop een inconsistentie wordt ontdekt, en welke fout er wordt gerapporteerd. Dit geldt in het bijzonder als de constraints één voor één bekeken worden. Een globale oplossingsstrategie heeft hier minder last van. Een ordening komt tot stand door eerst een constraintboom op te bouwen en deze vervolgens af te breken met een boomwandeling naar keuze. Hiermee kunnen klassieke algoritmen uit de literatuur (zoals $\mathcal{W}$ [11] en $\mathcal{M}$ [36]) worden nagebootst, en kan er geëxperimenteerd worden met variaties op deze algoritmen.

Het Top framework biedt naast een standaard oplossingsmethode ook een gespecialiseerde methode die het mogelijk maakt om zeer gedetailleerd een inconsistentie in een constraintverzameling uit te leggen. Hierbij wordt gebruik gemaakt van een typeringsgraaf. Dit is een geavanceerde datastructuur voor het representeren van substituties, met als kenmerkende eigenschap dat een substitutie zich ook in een inconsistente toestand kan bevinden. Met deze datastructuur kan een programma globaal worden geïnspecteerd, en kunnen heuristieken worden ingezet ter verbetering van de foutmeldingen. Hoewel deze oplossingsmethode meer rekentijd vergt dan de standaardmethode, kan de analyse nog steeds binnen redelijke tijd worden voltooid.

Directieven voor foutmeldingen

Tenslotte biedt Top de mogelijkheid om door middel van een eenvoudig te gebruiken scriptingtaal de foutmeldingen van buitenaf te beïnvloeden, en deze in te richten naar eigen smaak. Met behulp van directieven kan het type-inferentieproces worden gestuurd. Deze technologie is met name waardevol gebleken in de context van domeinspecifieke programmeertalen. Zonder een compiler aan te hoeven passen kunnen foutmeldingen worden uitgedrukt in termen van concepten uit het probleemdomein in plaats van die uit de onderliggende programmeertaal. Een collectie van directieven is speciaal ontworpen om het negatieve effect dat overloading en typeklassen hebben op foutmeldingen te beperken. Ook voor andere programma-analyses zou een dergelijke scriptingtaal van waarde kunnen zijn.

Gebruiksvriendelijke compiler

De Helium compiler [26] is ontwikkeld aan de Universiteit Utrecht met als doel om precieze, gebruikersvriendelijke foutmeldingen te genereren. De compiler maakt gebruik van de technologieën uit het Top framework, en is inmiddels al enkele malen ingezet tijdens de introductiecursus *Functioneel Programmeren* aan de Universiteit Utrecht. Helium ondersteunt bijna de volledige Haskell 98 standaard [49], wat aantoont dat de concepten uit Top daadwerkelijk toegepast kunnen worden op een volwassen programmeertaal. Tevens ondersteunt de compiler de faciliteit om zelf directieven te definiëren. Er zijn twee artikelen in het Nederlands verschenen die de ondersteunende rol van Helium binnen het programmeeronderwijs toelichten [25, 24].

Dankwoord

Functioneel programmeren en het informatica-onderwijs in Utrecht zijn een unieke combinatie. Na vele boeiende colleges over dit onderwerp te hebben gevolgd vanuit de collegebanken, voelde ik me als assistent in opleiding helemaal op mijn plaats binnen de Software Technology groep. De zeer uiteenlopende discussies met collega's hebben mij een heel eigen kijk gegeven op de elegante programmeerconcepten die een functionele taal te bieden heeft. Door het uitwisselen van ideeën en problemen hebben zij gezorgd voor een prettige werkomgeving. Ook vele collega's buiten de vakgroep hebben bijgedragen aan de goede sfeer. Bedankt!

Om te beginnen wil ik mijn promotor Doaitse Swierstra bedanken voor de begeleiding in de afgelopen periode. Wat vijfenhalf jaar geleden begon met een gezamenlijke reis naar Cochabamba, heeft uiteindelijk geleid tot het schrijven van dit proefschrift. Je enthousiasme voor functionele talen is een bron van inspiratie. Ik ben zeer dankbaar voor de feedback die ik heb gekregen en voor het doorsturen van al die weerzinwekkende typeringsfoutmeldingen.

Mijn dank gaat ook uit naar Jurriaan Hage. De rol van dagelijks begeleider heb je meer dan uitstekend ingevuld. Je hebt altijd de moeite genomen om geduldig te luisteren naar mijn vaak maar half uitgewerkte ideeën en je wist altijd weer een uurtje voor me vrij te maken. Tot het laatst toe was je zeer zorgvuldig in het doorlezen van mijn teksten, hoe vaak ik deze ook bij je inleverde.

Ook wil ik Johan Jeuring en Pablo Azero Alcocer nogmaals bedanken voor de supervisie van mijn afstudeerscriptie. De vele ideeën die in deze periode zijn ontstaan zijn een goede basis geweest voor het verdere verloop van mijn onderzoek.

De Helium compiler heeft een belangrijke bijdrage geleverd aan de voortgang van mijn onderzoek, en daar wil ik Arjan van IJzendoorn in het bijzonder voor bedanken. De combinatie van theorie en praktijk is van onschatbare waarde gebleken. Door een toevallige samenloop van omstandigheden zijn we drie jaar geleden begonnen met het ontwerpen van een gebruiksvriendelijke compiler voor het onderwijs. Een enorme klus, maar met een prachtig resultaat. Het door jou geïntroduceerde *pair programming* zorgde op zijn tijd voor een leuke en leerzame onderbreking. Ook ben ik dank verschuldigd aan alle anderen die in de loop der jaren betrokken zijn geweest bij dit project.

Verder wil ik Daan Leijen bedanken voor de getoonde interesse in mijn werk en voor de prettige samenwerking bij het schrijven van een aantal artikelen. Ook al hebben we een tegenovergestelde werkwijze, onze samenwerking blijkt wel te werken. Bedankt voor al je constructieve opmerkingen over mijn proefschrift. Ik ben Andres Löh zeer dankbaar voor het met mij delen van zijn inzichten in type-inferentie met constraints en voor alle hulp die ik heb gekregen om mijn Haskell-code te typesetten met *lhs2TeX*.

Ook ben ik dank verschuldigd aan al diegenen die steeds weer bereid waren om mijn teksten door te lezen. Martijn Schrage, Dave Clarke, Arthur van Leeuwen, Alexey Rodriguez Yakushev, Arthur Baars en Jeroen Fokker: allemaal hartstikke bedankt. A special thanks goes to Phil Trinder and Greg Michaelson, both connected to the Heriot-Watt University, for providing me with a copy of Helen Soosaipillai's thesis.

De prachtige tekening op het omslag van mijn proefschrift is gemaakt door Lydia van den Ham. Na een langdurige brainstorm-sessie, waarin we enkel de grote lijnen van het ontwerp hebben doorgesproken, was het eindresultaat voor mij een aangename verrassing. Het sluit perfect aan bij de oorspronkelijke titel van mijn onderzoek *"Why did we reach Top?"*, en het geeft dit boekje een heel eigen uitstraling. Ook Janny Beumer wil ik hartelijk bedanken voor het onder de loep nemen van de vormgeving en alle aanbevelingen die hieruit volgden. Ik dank Cor Kruft voor zijn hulp bij het ontwerpen van het omslag en Piet van Oostrum voor het installeren van de LaTeX fonts.

Als laatste wil ik mijn familie en vrienden bedanken voor de ondersteuning en de interesse die jullie hadden in mijn onderzoek. Misschien was het niet altijd even duidelijk met welke vraagstukken ik mij bezighield; hopelijk maakt dit proefschrift het voor jullie iets meer concreet. De laatste woorden zijn bestemd voor Anja, die me tijdens het schrijven telkens weer heeft aangemoedigd en altijd voor me klaar stond. Ook de voltooiing heb je van zeer dichtbij meegemaakt. Ik kan je hier niet genoeg voor bedanken.

Index

Curriculum Vitae

Bastiaan Johannes Heeren

28 mei 1978
Geboren te Alphen aan den Rijn.

augustus 1990 - juni 1996
Atheneum aan de Openbare Scholengemeenschap Schoonoord te Zeist.
Diploma behaald op 12 juni 1996.

september 1996 - oktober 2000
Studie Informatica aan de Universiteit Utrecht.
Doctoraal diploma behaald op 30 oktober 2000.

november 2000 - oktober 2004
Assistent in Opleiding aan het Informatica Instituut van de Universiteit Utrecht.

maart 2005 - heden
Junior universitair docent aan het Departement Informatica van de Universiteit Utrecht.

Titles in the IPA Dissertation Series

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09