THE UNIVERSITY OF
MELBOURNE

Type inference and type error diagnosis for Hindley/Milner with extensions

Jeremy Richard Wazny

January 2006

Submitted in total fulfilment of the requirements
of the degree of Doctor of Philosophy

Computer Science and Software Engineering
The University of Melbourne
Parkville, Melbourne

Abstract

Types and formal type systems have proven invaluable in the design and use of modern declarative languages. The Hindley/Milner type system forms the basis of many type systems in practical use since it is reasonably expressive (with support for parametric polymorphism), and type inference is well-understood and can easily yield principal types. Unfortunately this combination of polymorphism and inference can make reporting type errors difficult. Inferred types may not correspond to intended types, and polymorphism means that type information can travel some distance from its source before a conflict is discovered.

Typically, type inference is performed while traversing a source program's abstract syntax tree. A type is generated for each node either by looking it up in an environment, or by reasoning about the types of sub-nodes. This reasoning usually involves unifying types, updating them, in order to derive new types. Normally, when types are unified, their original form and the reason for the unification are forgotten. This makes providing accurate and helpful reasons for any eventual errors difficult.

We avoid these problems by reducing type inference to a constraint solving problem. An unsatisfiable constraint corresponds to a type error. Other type-related conditions, like subsumption of declared type annotations, can also be phrased in terms of constraints. By using constraints we are able to maintain a connection between program locations and the effect they have, which allows us to reason about their influence on typing results. Essentially, type error diagnosis becomes constraint reasoning. We develop algorithms for finding minimal sets of unsatisfiable and implicant constraints, which allow us to identify

locations in the source program which are responsible for a specific result. We also describe a series of optimisations which allow us to find all minimal unsatisfiable subsets of a constraint relatively efficiently.

Previous work in type error diagnosis has almost exclusively focused on standard Hindley/Milner. Most versions of the type system in actual use, however, extend it with support for features like explicit type annotations, algebraic data types and patterns, as well as some form of ad hoc overloading. We use a programmable constraint solver, based on Constraint Handling Rules, to provide support for type classes and functional dependencies, and later for extended algebraic data types.

Declaration

This is to certify that:

(i) the thesis comprises only my original work towards the PhD except where indicated in the Preface,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Signed,

Jeremy Richard Wazny
13th January 2006

Preface

Parts of this thesis have been derived from earlier work, all of which the author of this thesis has contributed to. Chapters 3,4,5 and Appendix A are derived from the as-yet unpublished work in [81]. Chapter 6 is based on the paper [79]. In Chapter 7, the algorithm for finding all minimal unsatisfiable subsets of a constraint is taken from [25]. The definition of the implication solver in Chapter 8 is the same as in [85]. The description of the interactive Chameleon type debugger of Chapter 9 is an updated version of the work in [77].

Please see Section 1.3 for an overview of the contents of this thesis.

Acknowledgements

Throughout the course of my enrolment I have benefited greatly from the support and direction of many people at the university. I am especially grateful to my supervisors Dr. Sulzmann and Prof. Stuckey, whose steadfast guidance and consistent enthusiasm for this work have propelled our efforts along at an exciting and rewarding pace. My time spent working in the office has been made lighter by the combined presence of those around me. There are too many people to mention, but special thanks go to Greg for his interesting conversations on all things unrelated to types, and to Bernie for his keen insights into functional programming.

During my candidature I've had the good fortune of spending a great deal of time overseas at the National University of Singapore. Thanks to everybody who made my stays in Singapore so rewarding and enjoyable, especially Kenny, Meng, Edmund, and all the others in the lab.

To my parents and family goes my gratitude for not only tolerating me for all of these years, but for their unwavering encouragement and support in all things in life. Special thanks go to my sister Yvette whose tireless proof-reading has been invaluable and much appreciated.

Contents

List of Figures

Chapter 1

Introduction

Types have long been an important aspect of programming languages and programming language design. Most programmers will already have an intuitive understanding of what a *type* is, gained from whichever programming languages they have used. They will probably also already have some pre-conceived ideas about the sorts of properties a type can express. For now, though, we will introduce types and type systems in fairly general, high-level terms.

In general, we can think of a type as an abstraction of the values a variable may eventually take at run time. Cardelli [7] defines a type as an upper-bound on this set of values. An alternative view is to think of a type as defining an 'interface'. A variable's type encapsulates all of the ways in which it can be safely used with meaningful result. Both points of view can be useful when thinking about types in different contexts. Note that when we mention 'variables' we often mean both values that may vary over the execution of a program, as well as constants, like fixed numbers or function definitions, which remain unchanged. The names we give to unchanging values can be thought of as 'single-assignment' variables, which are set once and for all whenever such a binding is encountered.

There are a number of reasons why we may be interested in knowing the types of elements of a program. Optimising compilers which know the exact types of the values being manipulated can often generate specialised code, which can execute more efficiently. Types are also a good form of program documentation, and allow programmers to think about their code at a higher, more abstract level. Most importantly though, and most directly relevant to this thesis, when incorporated into a type system, types can be used to detect and prevent the execution of ill-defined programs.

Types are simply labels we assign to program fragments. Obviously, if we could give any part of a program any type, those types would be meaningless. We would essentially be saying that all types are equivalent, since they could all be freely interchanged.

A *type system* formally describes the types that can be assigned to different parts of a program.Type systems differ from other forms of program analysis and verification in that they are usually fairly lightweight, and are described in a syntax-directed way [71]. Since they are defined inductively on the structure of a language, type systems can usually be applied in a single pass over a source program. Other forms of program analysis, such as abstract interpretation [12], may require multiple iterations over a program. Type systems are lightweight in the sense that their implementations can be expected to run in reasonable time, and process programs automatically, without the intervention of a programmer. As a result, type systems are most commonly implemented directly as part of a compiler.

Often when we attempt to classify all fragments of some program according to type, we find that it is not possible to do so, while respecting the type system. This may happen because some part of a program implies one type, while some other part implies a different, incompatible type. Obviously no program fragment can have multiple incompatible types; certainly, when that part of the program executes, we want it to work correctly in whatever context it is used. When this happens we say that the program has a *type error*.

As a simple example, imagine we wrote an expression like `(1 + 'a')`, in a language whose type system allows arithmetic only on numbers. The value `'a'`, an alphabetic character, is clearly not a number and yet we try to add `1` to it. The type system would forbid this program. Some languages, like C [48], may allow this expression, since they contain rules for uniformly converting characters to numbers. Without such rules, we would have no good way of predicting just what would happen at run-time. Indeed, we may have no basis to even conclude that this expression would yield the same result every time it is executed (although, this would probably be the case in any practical language implementation.)

A type system is a form of program verification since we can use it to ensure that all elements of a program are appropriately typed and that these elements always work and are used in a manner consistent with their type. When we can write a *type reconstruction* or *inference* (as we shall call it) algorithm for our type system, then it also becomes an analysis tool, enabling us to discover information

about programs we have written. Having an inference algorithm is also extremely desirable from a practical point of view; it allows us to avoid writing excessively many type annotations, since we can trust the system to just infer them for us anyway.

Some strongly typed languages offer a feature, or a number of related features, called *polymorphism*[1] which enables the use of a single identifier with different types in the same program. There are two main forms of polymorphism used in functional languages today: *parametric polymorphism*, where types can contain variables which are later instantiated to specific ground types; and *ad hoc polymorphism*, which allows us to use an identifier at a number of different, unrelated types.

Parametric polymorphism is often employed when writing functions that process data structures independently of the contents of the structure. For example, the `length` function in Haskell can calculate the length of any list, regardless of the type of the list elements. In a language without parametric polymorphism, it may be necessary to declare a separate length function for each possible list element type. This would be unnecessarily troublesome, since the length of a list should never depend on the type of its elements anyway.

In Haskell, ad hoc polymorphism is available in the form of type classes. An application of the Haskell function `show`, for example, can turn a value of any "showable" type into a string. A "showable" type is one for which a `show` declaration has been provided. In Standard ML, which does not have support for ad hoc polymorphism, we would have to use a different name for the specific `show` function on each of those types. e.g. `showInt`, `showBool`, etc.

Both forms of polymorphism are extremely useful for writing generic functions that can safely operate on values of many different types. Parametric polymorphism is particularly prominent, since it is more common in strongly typed functional languages, and is used extensively.

Tractable type inference and let-polymorphism (a restricted form of parametric polymorphism) are both important features of the Hindley/Milner type system [63]. This combination of expressiveness and computational efficiency has lead to its adoption by many modern declarative programming languages. The type systems of languages such as Standard ML [64], Objective Caml [6],

[1]The term 'polymorphism', literally 'many shapes', can mean different things in different programming language communities – which is fitting, given the name. In this thesis we are not concerned with polymorphism as it is used in an object-oriented context.

Haskell [30], Clean [11] and Mercury [62] are all based on the Hindley/Milner system. In all cases, Hindley/Milner has been extended to support additional features in these languages.

In this thesis we will focus on the Haskell language. Most significantly, Haskell adds type classes, a powerful type-directed mechanism for grouping overloading identifiers. We will see later how type classes can be incorporated into the Hindley/Milner type system.

Whenever we mention 'Haskell' in this thesis, we are actually referring to the language described by the most recent iteration of the Haskell language standard, Haskell 98 [30]. On occasion, when we need to move beyond the language defined in the Haskell 98 language report, we will make this clear.

In most of the examples which contain Haskell programs, we use the Glasgow Haskell Compiler (GHC) [26]. In our experience, GHC tends to produce more useful and expressive type error reports than the other Haskell implementations we have tried. We have used version 6.2.2 of GHC almost exclusively, since that was the most recent release of the compiler available at the time most of this thesis was written. Since some of the advanced language extensions of Chapter 8 are only available in GHC as of version 6.4, it was necessary to use that version of the compiler for some examples in that chapter.

1.1 A type error example

A type system can provide a great deal of assurance to programmers that their programs are safe to run. When a program that violates the type system is recognised, it is rejected and it is the programmer's responsibility to find and fix the error. Unfortunately most current type system implementations provide very little information about rejected programs. Indeed, due to the mechanical nature of the type checking process, it is possible that the diagnosis given by the compiler may not only be unhelpful, but arguably even misleading.

Let us take a look at a simple ill-typed program, and see how three different Haskell implementations report the type error.

Example 1 Consider the following simple function declaration.

```
split xs = case xs of
               []  -> ([],[])
               [x] -> ([], x)
```

```
(x:y:zs) -> let (xs, ys) = split zs
            in  (x:xs, y:ys)
```

□

The purpose of this function is to take a single list as input, and split its contents into two equal-sized[2] lists, which are returned as a pair. It contains three cases (and clauses), each of which depends on the length of the input list. If the input is empty or singleton, the result is easy to determine. In the case of a list of at least two elements, it works by recursively splitting the remainder of the list, and then appending those two elements to the front of each.

This program has a type error. In the case of a single-element list, containing an arbitrary value `x`, `split` returns a pair of an empty list, and `x` itself. Instead of `x`, it should clearly return `[x]`, a list containing `x`. Note that the standard, built-in Haskell lists we use here are homogeneous; they can only contain elements of the same type.

The problem, in simple terms, boils down to two issues. Firstly, the third clause indicates that the lists returned by `split` must be of the same type as the input list. We can clearly see this, because the output list is constructed using `x`, which is from the input. But, the second clause contradicts this by returning, as its second list `x` itself. Together, these two assumptions imply that `x` must have the same type as a list of `x`; an impossibility. Let us see how three different Haskell implementations report this error.

Version 1.6 of Nhc98, a "small, easy to install, standards-compliant compiler for Haskell 98" [69], reports the following when we try to compile `split`.

```
====== Errors after type inference/checking:
Type error (type-variable occurrence check fails)
when trying to unify expression part of alternatives at 2:13.
```

This error message is not very useful at all since it only tells us that 1) there is an "occurrence check" failure[3], and 2) it is caused by something within the "expression part" (the right-hand side) of the case alternatives. The unfortunate thing is that the programmer has nowhere specific to start the search for the bug. Checking a large program in the same way would prove tedious at best.

[2] Obviously the resulting lists can only be of equal size if the length of the input list is even. If it is odd, one of the output lists will contain an additional element.

[3] This relates to the fact that `split` tries to return both an `x` and a list of `x` in different cases.

Hugs is a functional language interpreter, which provides an "almost complete" implementation of Haskell 98, plus various extensions [37]. When we load `split` into the November 2003 version of Hugs, it reports the following.

```
ERROR "split.hs":1 - Type error in case expression
*** Term           :  let {xs (ys $0) ((xs,ys) (4 (split zs))) $0 $0}
                      in (x :  xs,y :  ys)
*** Type           :  ([[b]],[[b]])
*** Does not match :  ([a],[b])
*** Because        :  unification would give infinite type
```

This error report is confusing for a number of reasons. Firstly, even though it indicates where in the program Hugs found the error, the term that it displays doesn't actually appear in the original program; it is obviously a transformed version of the third case alternative. Hugs reports a conflict between types `([[b]], [[b]])` and `([a],[b])`, which represent `split`'s return value, however, the connection between these and the type the programmer may have expected, `([a],[a])` is not at all clear. Moreover, even though the second case alternative, which actually contains the bug, has had an effect on the form of this error report (giving rise to the strange types), it is not represented directly at all.

Finally, we try to load our program using GHC, a "state-of-the-art" compiler and interactive implementation of Haskell, which also supports many additional language extensions [26].

```
split.hs:4:  Occurs check: cannot construct the infinite type: a=[a]
             Expected type: [[a]]
             Inferred type: [a]
    In the second argument of `(:)', namely `ys'
    In a case alternative:
        (x : y : zs) -> let (xs, ys) = split zs in (x : xs, y : ys)
```

This error message is clearly better than the one produced by Hugs in that it presents the part of the program where it found the error in its original form. It also removes some of the unnecessary detail that Hugs presented about the return type of `split`. GHC mentions only the type of the second returned list, which is where the problem lies. Unfortunately again, the preceding case clauses have been processed and accepted as correct, but their contribution to the type error is not reported.

The one thing common to all of these error reports is that none of them indicate, or even hint at the true source of the error. Moreover, they all contain undisclosed assumptions about various types within the program. To truly decipher them and gain the big-picture view that will allow us to spot the mistake, we essentially need to be able to reconstruct the inference process up until the point where failure was detected.

The Nhc98 error message leaves us no recourse but to manually re-infer all of the types in the case alternatives and check them ourselves. With Hugs and GHC, we at least have a program location and some types to start reasoning from. In either case, we are forced to re-discover information that the type system implementation has already deduced, but has lost track of by the time the error is detected. This is obviously a frustrating, yet avoidable, burden.

Our goal is to assist programmers as much as possible in diagnosing type errors. This means reporting errors in a clear and concise way, while exposing as much information as is necessary to gain a full understanding of the problem. The type inference and error reporting system presented in this thesis, and implemented as part of the Chameleon system [84], can give us the following error report.

```
split.ch:1: ERROR: Type error
Problem : Case alternatives incompatible
Types   : [a] -> (b,a)
          [a] -> (c,[a])
Conflict: split xs = case xs of
                       [] -> ([],[])
                       [x] -> ([], x)
                       (x:y:zs) -> let (xs, ys) = split zs
                                   in  (x:xs, y:ys)
```

This error message tells us that two case alternatives are incompatible. It presents the types of both of those alternatives, and furthermore tells us which alternative each type corresponds to. Each type is highlighted in a different style, and the locations in the program which give rise to that type, in this case the two alternatives, are similarly highlighted. Our implementation highlights types in different colours. Since we are unable to reproduce those here, we use grey tones and box outlines instead.

The great advantage of this error report is that it not only reports the conflicting types, as Hugs and GHC do, but it also identifies the set of program locations which contribute to those types. As we saw earlier, none of the other Haskell implementations made any mention of the second case alternative, even though it is a necessary part, indeed the cause, of the error. Our error report does not hide the second clause's contribution.

Unlike the earlier error messages, we are not limited to identifying an entire alternative, or in Nhc98's case, the entire case expression, as the source of the problem. The locations highlighted are precisely those which cause the error; no more and no less. Seeing which locations are not highlighted can itself be useful. For our example program we know, for instance, that the first case alternative is fine, as is the first component of all of the returned pairs.

The `split` program above can be typed using the standard Hindley/Milner type system extended with support for lists and pairs (or algebraic data types), patterns and monomorphic recursion. These are trivial, well-understood extensions that are available in any practical functional language. Even so, we have demonstrated that it already poses a problem for most implementations when it comes to presenting accurate and meaningful type error information.

Many modern declarative languages go beyond the features typeable by standard Hindley/Milner. Haskell, Clean and Mercury, for example, contain type classes. Mercury and some Haskell implementations even allow multiple parameter type classes and functional dependencies. To accommodate these features, type systems expand and become more complicated, and consequently more and new opportunities for type errors occur.

It is telling that almost all of the work which relates to assisting programmers with type errors is limited to the standard Hindley/Milner type system (as we will see in Chapter 10.) In this thesis we will go beyond Hindley/Milner to look at the inference and error reporting issues once we add language features like: type classes, lexically scoped type annotations, and extended algebraic data types.

1.2 Contributions

This thesis covers, and is an extension of, the work presented in [25, 77, 78, 79, 81, 85]. Here we provide for a more in-depth discussion, provide proofs of soundness and completeness for our inference scheme, based on Constraint Handling Rules

(CHR) and introduce new methods for generating text messages, which explain type errors, from minimal unsatisfiable subsets of constraints.

Beyond the published work, we also present a new, unified inference framework which allows us to extend the language, and error reporting facilities even further, with support for: existential data types, extended algebraic data types, and lexically-scoped type variables.

Our contributions are:

- A translation of the typing problem for Hindley/Milner which includes type annotations and Haskell-style overloading (and indeed more complex type extensions [82]) into CHR.

- A refinement of CHR solving which keeps track of justifications, i.e. program locations, attached to constraints.

- Soundness and completeness results for our CHR-based inference scheme.

- The identification of constraint reasoning procedures which are sufficient to locate the minimal set of contributing locations in case of an error or unexpected result.

- Methods for selectively generating type error messages which can focus on any program location involved in an error.

- Advanced error reporting for all of our extensions beyond standard Hindley/Milner. This includes subsumption failure, ambiguous type schemes and missing instances.

- A unified inference framework, based on implication constraints, which subsumes our simpler scheme, and allows us to cleanly support further advanced language extensions such as extended algebraic data types (EADTs.)

- Type error reporting for the extended language, and a demonstration that our implication-based framework encompasses and extends our previous work, showing that the error-reporting principles discussed earlier can be neatly incorporated.

1.3 Thesis outline

The core of the thesis can be viewed as two separate, yet interrelated parts. In chapters 4,5 and 6 we develop a typing framework for Hindley/Milner extended with type classes and type annotations, and discuss the handling of type errors in this setting. This system is a good match for the type system found in modern declarative programming languages such as Haskell and Mercury.

In Chapter 8, we present a new typing framework which is strictly more expressive in that it captures all of the previously discussed features, and adds more. In this new system we are able to straightforwardly express lexically scoped type annotations, as well as extended algebraic data types. As before, we present the inference system and discuss the handling of type errors.

Apart from the distinction between these core components, the rest of this thesis is structured as follows.

We give an overview of the Hindley/Milner type system in Chapter 2, where we begin by describing the core type system, and gradually expand the scope of our discussion to cover various useful extensions. We also look at the issue of type inference for the Hindley/Milner type system, and current practices.

We begin laying the foundations of our own work in Chapter 3, where we discuss constraints and constraint solving. In that chapter, we also define and discuss a number of important constraint operations, like an algorithm for finding a minimal unsatisfiable subset of a constraint, which we will come to rely upon later, for error diagnosis.

In Chapter 4 we return to type systems, and give a thorough description of type inference for Hindley/Milner in terms of constraint solving and Constraint Handling Rules. We then go on to extend our source language, and type system, with support for type annotations and type classes, and describe the necessary changes to the inference system.

Having described inference, we are able in Chapter 5 to finally discuss possibilities for type error reporting in our system. We also discuss the reporting of errors related to the various type extensions we had earlier introduced. In Chapter 6 we go beyond this and look at generating more useful and informative type error messages.

We look at the problem of efficiently finding all minimal unsatisfiable subsets of a constraint in Chapter 7. We consider how the error reporting of earlier chapters could be enhanced when multiple minimal unsatisfiable subsets are available.

In Chapter 8 we present a new type system capable of encoding lexically scoped type annotations, and extended algebraic data types; two important new language features not possible in the earlier system. We show that the error reporting techniques and strategies of earlier chapters can be applied in this new system to useful effect.

Chapter 9 demonstrates how the constraint reasoning steps introduced earlier can be applied in an interactive setting. It is here that we introduce the Chameleon type debugger, an interactive tool for examining type errors, which embodies the type debugging methods we describe earlier.

In Chapter 10 we take a look at the existing type error reporting and debugging literature. We examine various proposals for alternative inference algorithms, type error reporting procedures, and interactive type debugging tools and compare their designs and capabilities to those that we present.

Finally, in Chapter 11 we conclude by summarising the main ideas and key contributions of this thesis. We also look ahead and briefly consider some possible future directions for our work.

Chapter 2

Background

In this chapter we will take a look at the Hindley/Milner[1] type system, which forms the core of the type systems we will study and develop. We begin by introducing the type system itself, independent of any specific implementation details (Section 2.1.1). Following that, we then go on to describe a number of useful language extensions, which can be found in Haskell, and its implementations (Sections 2.1.2-2.1.4). Our coverage of these extensions will remain necessarily high-level for now since there are no generally-accepted, universal formalisation of them all. We will eventually formalise all of these extensions ourselves in terms of a single, uniform type system, in Chapter 8.

After that, we will take a look at the practical details of implementing Hindley/Milner type inference, where we discuss algorithm $\mathcal{W}$ (Section 2.2.1) and its implications for type error reporting (Section 2.2.2).

2.1 Standard Hindley/Milner and extensions

2.1.1 The Hindley/Milner type system

We give a description of the Hindley/Milner system, in the style of Milner [63]. The Hindley/Milner type system is defined for a simple expression language based on the λ-calculus, extended with let bindings.

$$\textbf{Expressions} \quad e \quad ::= \quad x \mid e\ e \mid \lambda x.e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e$$

[1] The same type system is sometimes also referred to as 'Damas/Milner'.

$$\text{(Var)} \qquad \Gamma \vdash x : \sigma \quad (x : \sigma \in \Gamma)$$

$$\text{(Abs)} \qquad \frac{\Gamma_x \cup \{x : t\} \vdash e : t'}{\Gamma \vdash \lambda x.e : t \to t'}$$

$$\text{(App)} \qquad \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}$$

$$\text{(Let)} \qquad \frac{\Gamma \vdash e : \sigma \qquad \Gamma_x \cup \{x : \sigma\} \vdash e' : t'}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : t'}$$

$$(\forall\ \text{Intro}) \qquad \frac{\Gamma \vdash e : t \qquad \bar{\alpha} \notin fv(\Gamma)}{\Gamma \vdash e : \forall \bar{\alpha}.t}$$

$$(\forall\ \text{Elim}) \qquad \frac{\Gamma \vdash e : \forall \bar{\alpha}.t'}{\Gamma \vdash e : [\bar{t}/\bar{\alpha}]t'}$$

Figure 2.1: The Hindley/Milner type system

Note that these let bindings are non-recursive; the expression $\mathsf{let}\ x = x\ \mathsf{in}\ x$ is not typeable if x is not already bound somewhere outside of this.

The language of types and type schemes is defined as follows:

$$\begin{array}{lrcl} \textbf{Type} & t & ::= & \alpha \mid t \to t \\ \textbf{Type scheme} & \sigma & ::= & t \mid \forall \bar{\alpha}.t \end{array}$$

Type variables α represent monotypes, whereas for type schemes $\forall \bar{\alpha}.t$ the variables $\bar{\alpha}$ may be instantiated differently at every use. This means quantified types may be used polymorphically. We write $t \to t'$ as the type of functions from arguments to t to results t'. The $\to$ constructor associates to the right, so we will employ parentheses to overcome this in places. Sometimes we will refer to 'type schemes' simply as 'types' when we do not especially care about any quantification. We may also call unquantified types *simple types*.

As is common, we present the type system as a natural deduction system. Well-typedness of an expression e is represented by a *typing judgement* $\Gamma \vdash e : \sigma$, where the *environment* Γ is a mapping from expression variables to type schemes, and σ is a valid type scheme for e. We will often write Γ using set notation. We

write Γ_x for the environment $\Gamma - \{x : \sigma\}$, for any σ.

A *substitution* $[\bar{t}/\bar{\alpha}]$ is a function which replaces type variables $\bar{\alpha}$ with types $\bar{t}$. Applying a substitution to a type yields a new type corresponding to the old one with its variables updated according to the substitution,
e.g. $[(\beta \to \beta)/\alpha](\alpha \to \alpha) = (\beta \to \beta) \to (\beta \to \beta)$. When substitution ϕ is applied to an environment $\Gamma = \{x_1 : t_1, ..., x_n : t_n\}$, the result is a new environment $\{x_1 : \phi(t_1), ..., x_n : \phi(t_n)\}$. The result of applying a substitution ϕ_1 to another substitution ϕ_2 (also known as the *composition* of ϕ_1 and ϕ_2), written $\phi_1\ \phi_2$ or $\phi_1(\phi_2)$, is a new substitution that has the effect of first applying ϕ_1 to its argument, then ϕ_2. i.e. if $\phi_3 = \phi_1\ \phi_2$, then $\phi_3(t) = \phi_2(\phi_1(t))$.

We also make use of a function *fv* which returns all of the free (unbound) variables in its argument. The application $fv(t)$, where t is a simple type, returns all of the variables in t. Applied to a type scheme, $fv(\forall\bar{\alpha}.t)$, yields $fv(t) - \bar{\alpha}$. For a type environment, as in $fv(\{x_1 : t_1, ..., x_n : t_n\})$, it returns $\bigcup_{i=1}^{n} fv(t_i)$.

The *typing rules* for this language are shown in Figure 2.1. As usual, the typing (proof) rules allow us to conclude the judgement below the line, if the judgements above (premises) hold. The (Var) rule, with an implicit, empty premise, simply allows us to conclude that x has type σ if that is part of Γ. The (Abs) rule gives us a type $t \to t'$ for a λ-abstraction, if we assume the variable x has some type t given which, the body e, can be found to have type t'. The type of the result of an application e_1e_2 is t_2 if e_1 is $t_1 \to t_2$ and e_2 is t_1. In the (Let) rule, a type σ is found for the bound variable x, and the body of the expression, e', is typed under an environment extended with $x : \sigma$. The last two rules deal with polymorphic type schemes. Rule ($\forall$ Intro) allows us to *generalise* an existing type, by quantifying over any variables not present in the environment. Finally, ($\forall$ Elim) allows universal type variables to be instantiated to specific types, thereby eliminating the quantifier.

The Hindley/Milner type system forms the basis of type systems for many modern declarative languages. The expression language we showed at the start of this section, however, is not very practical for productive programming. Functional languages like ML, OCaml, Haskell and Clean, provide additional features which make it easier to structure and write programs in practice.

Commonly, these languages include support for: recursive let bindings; some form of *ad hoc* polymorphism, whether under programmer control or not, e.g. for convenient use of arithmetic operators; explicit programmer-declared type annotations; and programmer-defined data structures, in the form of algebraic

data types, or records/objects. Recursive definitions are so common that we will not explicitly mention them again as a distinctive 'feature' until we revisit the issue of type inference again in Chapter 4. As for the others, we will now go on to look at them in the context of Haskell 98, and common extensions to it. At this stage we will cover these features from a high-level, informal perspective, simply to motivate our more thorough definitions and discussions later on.

2.1.2 Type classes

For notational convenience, programmers sometimes like to reuse the same variable names in different ways. We often take it for granted that the `+` (addition) operator in a language can be applied consistently to arguments whether they are integers or floating point numbers. Similarly, it is useful to be able to test whether two values are equal using the same operator, `==`, regardless of the types of those values. Many programmers would baulk at the idea of having to use separate names like `eqInt` for *Int*s or `eqBool` for *Bool*s. Furthermore, the use of such type-specific functions makes it harder to write polymorphic programs, as we will see.

The Hindley/Milner type system presented earlier, however, does not allow us to assign disjoint types like $Int \to Int \to Bool$ and $Bool \to Bool \to Bool$ to a single function like `==`, even though sometimes we would like it to accept *Int* arguments, and sometimes *Bool*s. To keep things as polymorphic as possible, we would like `==` to have a type like $\forall a.a \to a \to Bool$, so that we could apply it to arguments of any type. The problem with this type, however, is that not all values we could possibly pass to `==` can be tested for equality, and as such, we may compromise the soundness of the type system.

Example 2 Consider the following program. The `replace` function takes two values and a list of elements of the same type. It returns the same list, except that all occurrences of the first argument are replaced by the second.

```
replace x y []     = []
replace x y (z:zs) = if z == x then y : replace x y zs
                               else x : replace x y zs
```

What type should `replace` have? If we assume that the `==` function has type $\forall a.a \to a \to Bool$, i.e. that its arguments must be of the same type, and that it

returns a Boolean value, then `replace` would be $\forall a.a \to a \to [a] \to [a]$. But, as explained above, giving `==` this type does not make sense, since operationally, `==` may not be defined (in the way we expect) for all possible values.

Nevertheless, we would like `replace` to be as polymorphic as possible. We do not want to have to write different versions depending on the specific types of its arguments.

One solution would be to require the equality function to be explicitly passed to `replace`, as in the following.

```
replaceEq eq x y []     = []
replaceEq eq x y (z:zs) = if eq z x then y : replaceEq eq x y zs
                                    else x : replaceEq eq x y zs
```

In this version `replaceEq` does not rely on any externally declared equality function. If we infer `replaceEq`'s type, we would find it to be $\forall a.(a \to a \to \mathit{Bool}) \to a \to a \to [a] \to [a]$.

Wherever we want to apply `replace` to a list of *Int*s we would write `replaceEq eqInt`, where `eqInt` defines equality on *Int*s. We would need to do the same thing for *Bool*s, *Char*s and other types.

Dealing with this extra parameter would be cumbersome, however, if we want to use `replace` within another polymorphic function. Consider the following.

```
replaceMany []         zs = zs
replaceMany ((x,y):xs) zs = replaceMany xs (replace x y zs)
```

The function `replaceMany` takes a list of pairs of substitutions, and invokes `replace` on each pair, and the list `zs`.

We still face the same problem. If we want `replaceMany` to be polymorphic in its arguments, we will need it to use `replaceEq`, and take the explicit equality function to be used as an extra parameter. We might rewrite it as follows:

```
replaceManyEq eq []         zs = zs
replaceManyEq eq ((x,y):xs) zs = replaceManyEq eq xs
                                                (replaceEq eq x y zs)
```

The function `replaceManyEq` has the type $\forall a.(a \to a \to \mathit{Bool}) \to [(a,a)] \to [a] \to [a]$. As we can see, threading an explicit equality function through every function that will use it (even if only indirectly) would be quite a burden. Furthermore, it clutters up those function definitions and makes their types much harder to read.

□

Many languages have tackled this problem in different ways. In Standard ML [64], for example, type variables must be explicitly marked to indicate that they only permit types which can be compared for equality. In Miranda [65], equality for values of any type is built directly into the implementation. Consequently, the version of equality provided by the system for some type may not correspond to the notion the programmer has about equality of those values. This is especially so in the case of equality on functions and other abstract types.

In Haskell, this problem is solved using type classes [86, 28]. Type schemes are extended with type class constraints which effectively limit instantiations of those variables to only types which are members of a particular class. With the addition of type classes, the form of acceptable type schemes becomes:

$$\begin{array}{lrcl} \textbf{Type class} & c & ::= & U\ \alpha \\ \textbf{Type scheme} & \sigma & ::= & t \mid \forall \bar{\alpha}.\bar{c} \Rightarrow t \end{array}$$

In the form $U\ \alpha$, U represents a type class name, and α is a type variable. A type scheme $\forall \bar{\beta}.C\ \alpha \Rightarrow t$, where $\alpha \in fv(t)$ and $\alpha \in \bar{\beta}$, limits the instantiation of the otherwise universal variable α to only types which are members of the type class C. To make a type t a *member* of a type class, we must provide an implementation of the type class for t.

Type classes provide a uniform, type-directed way to safely overload groups of related functions. A type class is identified by a unique name, and consists of a set of *methods*, as well as an optional, user-specified dependency on a (possibly empty) set of other type classes. We call those classes *super classes*. The methods within a type class can be used as freely as any other top-level variables.

In Haskell, a new type class is introduced with a `class` declaration, like in the following:

```
class Eq a where
      (==) ::  a -> a -> Bool
      (/=) ::  a -> a -> Bool

class Eq a => Ord a where
      (<=) ::  a -> a -> Bool
```

Here two classes *Eq* and *Ord* are declared. Similar definitions can be found in the Haskell Prelude (standard library.) The *Eq* class contains two methods: `==`,

which represents an equality test – it should return `True` if its two arguments are equal; and `/=` which, correspondingly represents the not-equal-to relation. The *Ord* class has a method `<=`, which is the less-than-or-equal-to relation. By writing `Eq a => Ord a` we are able to express the requirement that any member of the *Ord* class must also already be a member of *Eq*; *Eq* is a super class of *Ord*.

These class declarations bring their methods into scope. Although we have no implementations for them yet, the types of those methods can be read directly from their declarations. The function `==` has type $Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$.

A type class declaration is just a skeleton, though. Without any instances, we cannot make use of these overloaded methods. We can write an `instance` declaration in Haskell to supply these methods. The following is an example.

```
instance Eq a => Eq [a] where
    (==) []     []     = True
    (==) (x:xs) (y:ys) = x == y && xs == ys
    (==) _      _      = False

    (/=) x y = not (x == y)
```

Here we declare an instance of *Eq* for lists. As in the class declaration for *Ord*, we have constrained the usage of this instance. By writing `Eq a => Eq [a]` we ensure that our instance only applies to lists whose elements are already themselves of the *Eq* class.

The definition of `==` works by simple induction on lists. If both arguments are empty, then they are trivially equal. If neither is empty then the two lists are equal if their heads (leading elements) are equal and the remainder of the lists are equal. The third case applies when one list is empty and the other non-empty, and so they cannot be equal.

The second case is most interesting since there the same `==` identifier is used both to test equality of two elements of the argument lists, and recursively to test the remainder of the lists. Since the instance declaration promises that *Eq* is already defined on elements of the argument list, and since we know *Eq* is defined on such lists (we are defining it right now!), we know these two uses of `==` can be satisfied.

Now that we have provided an instance of *Eq* on lists, we would be able to also declare a similar instance of *Ord*. We will omit the specific declaration since the definition of the `<=` method is essentially the same as that of `==`.

In the same way, Haskell provides overloaded versions of: arithmetic operations, like addition and multiplication, provided by *Num* and other classes; methods for enumerating values of a type, as provided by the *Enum* class; and methods for converting values to and from strings, as part of the *Show* and *Read* classes. Instances of these (and other) classes on Haskell's basic types are available in the Haskell Prelude.

Type classes have been an important feature of Haskell from the beginning [28], and they appear prominently in Haskell's standard library. Type classes have since proven popular enough to be adopted in directly a number of other statically typed, declarative languages, like Clean and Mercury.

Stuckey and Sulzmann [82] provide a semantics for type classes in terms of constraints and Constraint Handling Rules. We will make use of their design when we describe type inference for Hindley/Milner with type classes in Chapter 4.

Type class ambiguity

In Haskell, when a type scheme contains any type class constraints it is required that all of the constrained variables also appear within the type component of the type scheme [30]. This restriction ensures that at use sites, when eventually all type variables are ground, the type class variables are also all ground. A type scheme which does not meet this condition is said to be *ambiguous*. For example, the type scheme $\forall a.Eq\ b \Rightarrow a \rightarrow a$, is considered ambiguous since *Eq* constrains b, which does not appear in the type component $a \rightarrow a$.

This restriction is necessary since all occurrences of overloaded identifiers need to be replaced with specific instance methods before they can be executed. A type class constraining an unbound variable does not correspond to any instance in particular, and so it is not obvious which should be used.

Example 3 The following program is a standard example of an ambiguous Haskell program. The functions `show` and `read` are standard Prelude functions and have the types $\forall a.Show\ a \Rightarrow a \rightarrow String$, and $\forall a.Read\ a \Rightarrow String \rightarrow a$, respectively. We use `read` to convert a *String* into a *Read*able value, and `show` to convert a *Show*able value back into a *String*.

```
ident str = show (read str)
```

Inferring `ident`'s type, we find it to be $\forall a.(Show\ a,\ Read\ a) \Rightarrow String \rightarrow String$. This is clearly ambiguous, since the variable a appears only in the type class context and not in the type component.

The reason for this ambiguity is simply that the type a, produced by `read` is immediately consumed as an argument to `show`. Since `show`'s result type doesn't mention its input type, the a never appears in the type of `ident`.

□

We will return to the ambiguity problem for type classes again in Chapter 4, and address the issue of suggesting to the programmer how it can be resolved.

Multi-parameter type classes and functional dependencies

In Haskell, as in our previous examples, type classes are restricted to a single argument. Consequently, we can only use them to represent properties which depend on a single type. One obvious advancement is to extend type classes to multiple parameters. This would allow us to capture an even broader range of programming abstractions in a convenient way. Indeed, they allow us to think of type classes as generic type relations. In recent years, many have exploited this idea, using instance declarations to essentially define type class programs [50].

Multi-parameter type classes have proven to be a popular extension to Haskell, and both Hugs and GHC provide support for them. A number of the libraries supplied with these systems use multi-parameter type classes to good effect. For example, a ternary type class *MArray a e m*, declared in module `Data.Array.MArray`, provides a generic interface to mutable arrays. It is parameterised in terms of an array constructor type a, an element type e, and the type of the monad, m, from within which the array can be accessed. The result is that a great many specific instances can all be accessed uniformly, allowing specific implementations to be swapped in and out quite easily.

The following is a typical example of a multi-parameter type class [43].

```
class Collects c e
      insert ::  e -> c -> c
      lookup ::  c -> Int -> Maybe e
      empty  ::  c
```

Here we define a class *Collects* over two variables, c and e. The class represents a collection parameterised in the type of the collection itself, c, and the type of the individual elements, e. The method `insert` adds a new element to an existing collection, returning a new collection. The `lookup` method provides a way to retrieve an element, specified by an index argument, from the collection. This

method has a *Maybe* return type since the supplied index may not correspond to any actual element in the given collection. Finally, the `empty` method represents a new, empty collection.

One problem with multi-parameter type classes is that, in practical use, we often have to deal with ambiguity. In the `Collects` example above, we can see immediately that the type of `empty` only makes reference to one of the type class's arguments, c, but not the other, e. Indeed, `empty`'s type is $\forall c, e.\mathit{Collects}\ c\ e \Rightarrow c$; we can read it directly out of the class declaration. That means that every occurrence of `empty` is going to be unavoidably ambiguous. This is clearly a problem, since as we stated earlier, we cannot generally resolve ambiguous overloading, and we simply will not be able to run any program making use of `empty`.

Example 4 In the following program we try to use the *Collects* class by writing a function that will generate a new singleton collection, initialised with the value of its argument.

```
init e = insert e empty
```

If we infer `init`'s type, we find $\forall c, e, e'.(\mathit{Collects}\ c\ e,\ \ \mathit{Collects}\ c\ e') \Rightarrow e \rightarrow c$. Both `insert` and `empty` give rise to separate type class constraints, which both share the same collection type c. The argument `e` lends its type to the type of elements `insert` expects in the collection. In the case of the constraint arising from `empty`, however, there is no connection at all to the element type of the collection it represents. In the inferred type we have written this as e'.

Despite the fact that we may have expected `insert` and `empty` to represent collections of the same type, we have no way yet to directly enforce this. The result is that `init`'s type is ambiguous. □

Jones provides a solution to this dilemma, in the form of *functional dependencies* [43]. A functional dependency allows us to express a simple relation amongst the arguments of a type class. Specifically, a functional dependency $a_1, ..., a_n \rightarrow b_1, ..., b_m$ states that the arguments $a_1, ..., a_n$ uniquely determine the arguments $b_1, ..., b_m$. That is, a single set of as always implies the same bs. In Hugs and GHC a functional dependency may be expressed as part of a `class` declaration. We are not limited to a single functional dependency; we can state as many as are necessary for any particular class.

The following is the `Collects` type class of before, but now amended with a functional dependency `c -> e`. This states that in all *Collects* type class constraints, the collection type c determines the element type e.

```
class Collects c e | c -> e
      insert ::  e -> c -> c
      lookup ::  c -> Int -> Maybe e
      empty  ::  c
```

This solves the problem we faced with the `empty` method, since now whenever we have the type of a collection c, the functional dependency ensures that we will be able to infer a specific element type e.

Example 5 For the `init` function of Example 4, we had previously inferred the type $\forall c, e, e'.(\mathit{Collects}\ c\ e,\ \mathit{Collects}\ c\ e') \Rightarrow e \rightarrow c$. Taking the functional dependency into account, we know that every collection type determines a specific element type. Since the two *Collects* type class constraints contain the same collection type c, it must therefore be the case that their element types must also be the same. That means that $e' = e$. We can now be satisfied that both `insert` and `empty` refer to collections of the exact same type — as we would naturally expect.

Having inferred this additional information, we can simplify `init`'s inferred type to $\forall c, e.\mathit{Collects}\ c\ e \Rightarrow e \rightarrow c$. □

Duck, Peyton-Jones, Stuckey and Sulzmann [16] have studied functional dependencies in a more general CHR-based type class setting. We will make use of their encoding of functional dependencies in terms of Constraint Handling Rules later (in Chapter 4) when we revisit the idea as part of the type system we develop.

2.1.3 Type annotations

One obvious extension to the standard Hindley/Milner type system presented, is the addition of programmer-declared type annotations. Even though these can (usually[2]) be inferred anyway, providing explicit annotations is often a good idea. A type annotation serves as a specification of a variable's behaviour, and is an important form of automatically-verified program documentation.

[2]As we will explain, type annotations must always be provided in the case of polymorphic recursive definitions.

In Haskell, we can declare types of let-bound variables by providing a type signature declaration anywhere in the same binding group. Expressions can also be explicitly annotated.

Obviously the types we declare cannot be completely arbitrary. A declared type must agree with the inferred type of the annotated entity. We call this the *subsumption* condition, and say that for an annotated type to be correct it must be subsumed by the inferred type. Put informally, this condition requires that the declared type is no more general than the inferred type.

Example 6 Let us now revisit the `replace` function of Example 2. We have added an explicit type annotation.

```
replace ::  Eq b => b -> b -> [a] -> [a]
replace x y []     = []
replace x y (z:zs) = if z == x then y : replace x y zs
                               else x : replace x y zs
```

We know from earlier that `replace`'s (most general) inferred type is $\forall\, a.Eq\, a \Rightarrow a \rightarrow a \rightarrow [a] \rightarrow [a]$. The type we have declared though is more general than this since it allows the type of the first two arguments to differ from the element type of the lists. This annotated version of `replace` should be rejected. □

We will return to the subsumption problem again, in Chapter 5 when we consider ways in which we can best report subsumption errors.

Lexically scoped type variables

In Haskell, all type annotations are implicitly *closed*. Every type annotation $e : t$, where e and t are expressions and types respectively, in fact stands for $e : \forall\, fv(t).t$. This makes type annotations easy to interpret, but means we cannot explicitly annotate an expression or let-binding whose type depends on a variable bound outside of its definition.

Example 7 Here again is the `replace` function of Example 2, which we have reformulated slightly. To avoid unnecessarily passing around the first two values, we have factored the list traversal out of `replace`, and placed it in the nested function `rep`. We have also explicitly annotated `replace` with its most-general type.

```
replace ::  Eq a => a -> a -> [a] -> [a]
replace x y xs = let rep [] = []
                     rep (z:xs) = if z == x then y : rep xs
                                            else x : rep xs
                 in  rep xs
```

Unfortunately we are not able to correctly annotate `rep` in Haskell. If we were to declare `rep :: [a] -> [a]`, this would be interpreted as $\forall\ a.[a] \to [a]$ — these as are not the same as as in `replace`'s type. That cannot be `rep`'s type, however, since the elements of its list argument must share the same type as `replace`'s first two arguments (which is what we intended by reusing the a variable.) It certainly cannot accept lists of elements of any arbitrary type, as the universal quantifier implies.

We will see an example later where type checking will only succeed if just such an annotation were allowed. □

In the context of Haskell 98, this inability to express certain nested annotations is only significant if polymorphic recursion[3] is necessary. The (Let) rule of Figure 2.1 is for a non-recursive binding. From the rule, we see that the type of a variable x is always fully determined before x is added to Γ, and can be subsequently used. If we allow for recursive bindings, which are a fundamental part of any practical functional language, we would (somehow) need to know the type of x before we had completely inferred it. The common restriction, as found in ML and Haskell, is to limit x to have a simple/monomorphic type within its own definition. That is, every (recursive) occurrence of the variable within its own definition must have the same type. It turns out that this restriction is necessary since type inference for polymorphic recursive bindings has been found to be undecidable [35, 49].

Haskell allows us to circumvent this problem, by allowing variables whose types have been explicitly declared to be used polymorphically. In this way, the problem of inferring the variable's type is completely avoided, since we can just use the type provided by the programmer. The subsumption check we mentioned earlier suffices to ensure that the declared type is indeed correct. By forcing all annotations to be closed, however, Haskell prevents us from nesting certain

[3] 'Polymorphic recursion' refers to the use of a let-bound variable with different, non-unifiable types within the body of its own definition.

polymorphic recursive functions: we cannot write an unannotated polymorphic recursive function, and we often cannot annotate nested functions at all.

In our system, we will allow for scoped type annotations. A type annotation binds all new variables within it, implicitly universally quantifying them. If any of those variables appear in a nested annotation, they are treated as the same; once a variable is bound, it cannot be re-bound. We could annotate the nested function of Example 2 as suggested, without encountering the problem we would face from Haskell.

As useful as they are, lexically scoped type annotations are largely folklore and have only been formalised to some extent [64, 45]. We will develop this idea further in Chapter 8.

2.1.4 Algebraic data types

Algebraic data types provide a direct way for programmers to encode (possibly recursive) data structures. Common data structures such as lists and trees can be readily represented. Given their structural definition they naturally allow for pattern matching. The immediacy of algebraic data type declarations and their obvious resemblance to grammatical production rules, makes them ideal for encoding the syntax of languages. Algebraic data types are a common feature of modern declarative languages and can be found in ML, Haskell, Mercury and others.

Example 8 The following program contains an encoding of an extremely simple expression language as an algebraic data type, as well as an evaluator for that language.

```
data Exp = N Int
         | B Bool
         | If Exp Exp Exp

eval :: Exp -> Exp
eval (N n) = N n
eval (B b) = B b
eval (If e1 e2 e3) = case eval e1 of
                          B True  -> eval e2
                          B False -> eval e3
                          _       -> error "not a B!"
```

The `data` declaration defines three data constructors: `N` with type $Int \rightarrow Exp$, `B` of type $Bool \rightarrow Exp$, and `If` which has type $Exp \rightarrow Exp \rightarrow Exp \rightarrow Exp$. These can be applied just like functions in order to build values of type Exp.

The language we have represented here supports three kinds of expressions: numbers, encoded as *Int*s; Boolean values (`True` or `False`); and conditional, if-then-else expressions.

The evaluation function `eval` simply returns any constant, whether numeric or Boolean, or recursively reduces an if-then-else. To reduce an if-then-else expression we first must evaluate the conditional part – the first argument. If it is `True`, the result is the evaluated second expression, if `False`, then the third. If the conditional is not a Boolean value, then we cannot proceed; the evaluator reports an error and halts.[4]

Note that in the error case, the result of evaluating the conditional must be a number. We know that because: 1) if it were a Boolean one of the preceding cases would have applied, and 2) by examining the return values, we can see that `eval` only ever returns a number or a Boolean, never an if-then-else expression.

Furthermore, since we also know from the type given to `eval` that its result is always an `Exp`, this rules out the possibility that the conditional could evaluate to some partially-applied constructor. Indeed the type system of the host language, Haskell, affords our embedded language a useful degree of type safety. We can rest assured that `eval` will never be invoked on a syntactically ill-formed expression, such as `If (B True) (If) (N 2)`.

What Haskell does not guarantee, however, is that any expression in the conditional position will always be a Boolean. In Haskell, for instance, the expression `eval (If (N 1) (N 2) (N 3))`, is well-typed, even though we know that the evaluator will fail on the 'conditional' `(N 1)`. That is why we need to insert the third case which, when reached, essentially corresponds to a run-time type error. □

We will now briefly examine a number of extensions to algebraic data types which, although not part of Haskell 98, have been adopted by a number of Haskell implementations.

[4]We make no effort to ensure that the result of evaluating either branch of the if-then-else yield values of the same type. That would be quite inefficient, as it would require evaluating both branches, in all cases, even though only the result of one of them is ever required

Existential data types

Läufer and Odersky describe an extension to algebraic data types which allow for local, existentially typed components [53]. This effectively allows us to encode abstract data types [66], i.e. data types with components whose types are completely hidden.

Although not part of the Haskell 98 standard, existential data types have proven to be a useful and popular extension. Both Hugs and GHC can handle Haskell programs which make use of existential data types.

Example 9 The following program contains an encoding of a stack data structure using an existential data type.

```
data Stack a = forall s.  Stack s              -- self
                                (a -> s -> s) -- push
                                (s -> s)      -- pop

push ::  a -> Stack a -> Stack a
push x (Stack s push' pop) = Stack (push' x s) push' pop

pop ::  Stack a -> Stack a
pop (Stack s push pop') = Stack (pop' s) push pop'

mkListStack ::  [a] -> Stack a
mkListStack xs = Stack xs (:) tail
```

We declare an algebraic data type *Stack*, with one visible type argument a, and a hidden, existential argument s. The variable a is to represent the type of the values on the stack, while s stands for the type of the actual, concrete stack. We keep s hidden because we want to be able to treat *Stack*s uniformly; independent of their exact representation. We only care about: 1) a stack's contents, and 2) its interface.

We call s an existential type, but in the syntax used above (supported by both Hugs and GHC), the keyword `forall` is used. This is because, in the type of the `Stack` constructor s is indeed universal. `Stack` has type $\forall s, a.s \rightarrow (a \rightarrow s \rightarrow s) \rightarrow (s \rightarrow s) \rightarrow \mathit{Stack}\ a$, and we certainly do not want to restrict the type of the concrete stacks we can build.

A `Stack` data structure contains three components: the actual stack, of type s; a function for pushing a new value onto an existing s stack; and a function for

removing a value from an s stack. Since we would like to be able to manipulate a stack without knowing the make up of the specific data structure, we provide global push and pop functions which can operate on any `Stack`. These work by simply extracting the corresponding method, and by applying that method, creating a new `Stack` with an updated initial field.

The function `mkListStack` actually creates a stack data structure, which is encoded as a standard Haskell list. The stack is initialised with the list passed to `mkListStack`, and the standard functions `:` and `tail` are to be used for pushing and popping respectively.

(A more practically useful version of this data structure might contain additional methods for actually accessing values in the stack. We omit these to keep the example a more managable length.)

□

Existential data types with type classes

Läufer describes an extension to algebraic data types which combines both existential components and type classes [52]. We will demonstrate the possibilities that this allows for by example.

Example 10 We recast the encoding of stacks from Example 9. Instead of packing all stack operations within the data structure, we instead constrain the type of the stack itself so that only types which are members of the `StackM` type class are permitted. The type class defines methods which can be used to manipulate its member types as stacks. If you look back at Example 9, you will notice that these are precisely the same methods that we had previously bundled into the `Stack` constructor ourselves.

```
class StackM s a | s -> a where
    push :: a -> s -> s
    pop  :: s -> s

data Stack a = forall s.  StackM s a => Stack s

instance StackM (Stack a) a where
    push x (Stack s) = Stack (push x s)
    pop (Stack s)    = Stack (pop s)
```

```
instance StackM [a] a where
    push = (:)
    pop  = tail

mkListStack ::  [a] -> Stack a
mkListStack xs = Stack xs
```

The first instance we provide allows us to treat `Stack` data structures as stacks, via the methods of `StackM`. These methods simply lift the `push` and `pop` methods from the level of concrete stack implementations (as contained within a `Stack`) to abstract `Stack`s.

The next instance contains declarations of stack methods, `push` and `pop` for a stack represented as a list. Again, these are the same functions as we used in the earlier example.

Finally, we again declare a function `mkListStack` which constructs a stack represented as a list, initialised with the value of its argument. In this case, though, there is no need to explicitly place the stack-manipulation functions within the data structure itself. The `StackM` type class in the definition of `Stack` ensures that whatever the concrete stack representation, it will always have `push` and `pop` methods defined for it.[5] □

Guarded recursive data types

Guarded recursive data types (GRDTs) are an extension to algebraic data types, developed by Xi, Chen and Chen, which allows the types of constructors to be refined independently of each other [90]. We demonstrate the idea behind GRDTs, as well as their usefulness with a pair of examples.

Example 11 We yet again reformulate the `replace` program from Example 7.

```
data Zero
data Succ n
```

[5]The reader may wonder how an implementation would know which method to invoke for a given abstract/existential type. In practice, whenever one of these type-class constrained existential data types is constructed, the compiler silently inserts the methods corresponding to the existential variable's concrete type. Essentially we are again exploiting the compiler's ability to resolve type class overloading in order to avoid manually passing arguments around. For more details see [52]

```
data List a b = (b = Zero) => Nil
              | forall b'. (b = Succ b') => Cons a (List a b')

replace :: Eq a => a -> a -> List a b -> List a b
replace x y xs = let rep :: List a c -> List a c
                     rep Nil = Nil
                     rep (Cons z xs) = if z == x then Cons y (rep xs)
                                                 else Cons x (rep xs)
                 in  rep xs
```

The type constructors `Zero` and `Succ` are introduced to encode the natural numbers; `Zero` is the first number, and `Succ` represents the successor of its argument. We will only be using these at the type-level, so no term-level representation is necessary, hence no term constructors are defined.

We define a new list data structure `List` which is parameterised in two types: a, the type of elements; and b, a type representing the length of that list. The type we have given to `replace` states that the length b of the input list must be the same as the length of the output list, whatever b is. The `Nil` constructor, with type $\forall a.\mathit{List}\ a\ \mathit{Zero}$ represents an empty list, and so its length is given by `Zero`. `Cons`, with type $\forall a, b'.a \rightarrow \mathit{List}\ a\ b' \rightarrow \mathit{List}\ a\ (\mathit{Succ}\ b')$, constructs a new list from a given element and another list. The length of its tail list is represented by b', and so since `Cons` simply adds one more element, the resulting list is of length $\mathit{Succ}\ b'$.

The reader may wonder how we could possibly annotate rep with the type $\forall c.\mathit{List}\ a\ c \rightarrow \mathit{List}\ a\ c$, when we know that the `Nil` case requires that c be Zero, while `Cons` makes it $\mathit{Succ}\ b'$, for some b'. The answer is that additional constraints, associated with those constructors only apply to the body of the function clause they are part of. This means that the constraints on the length of the list are never combined, and therefore can never conflict in this program.

Nevertheless, it still looks like there must be a subsumption error. In both cases the declared type of the list length c is more general than what is inferred. Usually when we check a type annotated function, we begin with the assumption that the declared type is correct. In the case of GRDTs, that type is temporarily refined. This refinement takes effect when checking the right-hand side of a pattern match involving a GRDT constructor.

Let us take a look at the `Nil` case. We begin with the fact that `rep` has type $\forall c.\mathit{List}\ a\ c \rightarrow \mathit{List}\ a\ c$, and then refine that with the constraint, from the

definition of `Nil`, that $c = Zero$. Our starting assumption is therefore that `rep` has type $List\ a\ Zero \rightarrow List\ a\ Zero$ on the right-hand side of this pattern match (only). A cursory inspection of the definition shows that this is a valid type for the first clause of `rep`.

The same reasoning applies in the `Cons` case. A system capable of supporting GRDTs, and lexically scoped annotations, will find this version of `replace` to be type correct. □

Example 12 We return to the simple expression language of Example 8, which we have reformulated below using guarded recursive data types.

```
data Exp a = (a = Int)  => N a
           | (a = Bool) => B a
           | If (Exp Bool) (Exp a) (Exp a)

eval :: Exp a -> a
eval (N n) = n
eval (B b) = b
eval (If e1 e2 e3) = case eval e1 of
                          True  -> eval e2
                          False -> eval e3
```

With these additional constraints in place, the type system prevents us from even constructing a value like `eval (If (N 1) (N 2) (N 3))`. This would be immediately rejected because `(N 1)` must have type `Exp Int`, while `If`'s first argument can only be of type `Exp Bool`. In this way, we have guaranteed that our evaluator need never perform any run-time checks.

We have eliminated the possibility of a run-time type error, but at the cost of a new restriction. In addition to enforcing that the conditional part of an if-then-else is truly a Boolean, we also require that the two possible results must be of the same type. This eliminates some perfectly good expressions such as `If (B True) (N 1) (B True)`. Despite the fact that this expression is ruled out, we could safely execute it without encountering any run-time error. □

A number of experimental languages, like Omega [73] and ATS [89] feature GRDTs as a key element. They focus on the idea of combining programming with lightweight theorem proving, as supported by GRDTs. We can treat constructor specific types as proof terms, and relate them to enforce additional restrictions

on the form of acceptable programs. The GRDT version of the `replace` program is an example of this. There we encoded the property that the length of the output list must be the same as the length of the input list by assigning each list an additional type representing its length, and ensuring that those types are the same for both lists.

Peyton-Jones, Washburn and Weirich [46] also study a practical type inference system for GRDTs which they term 'GADTs' (generalised algebraic data types.) Cheney and Hinze's 'first-class phantom types' [9] also provide essentially the same feature. As for the other implementations mentioned above, these works deal with inference for (their version of) GRDTs separated from inference for type classes, whereas our work unifies all of these features. As we will see in Example 14, GRDTs are superficially redundant once we allow for functional dependencies.

GRDTs and type classes

When we program with additional type parameters, which could represent various kinds of program properties, we may want to relate those types in various ways. In Example 11, we saw that by repeating a type we could enforce that the lengths of two lists were the same. Often, though, we require more expressiveness than this in order to capture more interesting properties. As we saw earlier, multi-parameter type classes, when coupled with functional dependencies can be used to relate multiple types in useful ways, since they admit a limited (and indirect) form of type-level programming.

Example 13 The following program contains a definition of the `append` function (for concatenating two lists). We again make use of length indexed lists, and an encoding of natural numbers as types.

```
data Zero
data Succ n

class Add a b c | a b -> c
instance Add Zero b b
instance Add a b c => Add (Succ a) b (Succ c)

data List a b = (b = Zero) => Nil
              | forall b'. (b = Succ b') => Cons a (List a b')
```

```
append :: Add l m n => List a l -> List a m -> List a n
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

The above is a standard encoding of append, except that we now use length indexed lists, and by the type annotation, enforce the additional restriction that the length of the output list must be the sum of the lengths of the input lists.

The *Add* constraint is simply a type class. We will forgo discussion of the details of its definition for now, since that would involve a more thorough understanding of type class semantics than what we have presented so far. We will formalise this in sufficient detail to describe the encoding of the type-level *Add* relation (and any type class relation in general), in Chapter 4. □

With guarded recursive data types, we could add constructor-specific equations which affected the types of those constructors. One obvious step beyond GRDTs is to widen the domain of constraints allowed in constructors to include type classes. We call these *extended algebraic data types*, or EADTs.

EADTs could be viewed as a generalisation of existential types with type classes, since we now add equations to algebraic data type contexts. We can show, though, that allowing equations is not strictly necessary if we already have functional dependencies.

Example 14 In the following program, we use an abstract type class *MkInt a* which does nothing but force its argument *a* to be *Int*.

```
class MkInt a | -> a
instance MkInt Int

data Exp a = (a = Int) => N a
           | (MkInt a) => N' a
```

With no variables on its left, the functional dependency simply states that *MkInt*'s argument *a* must always have the same type. The two constructors `N` and `N'` both have type $Int \rightarrow Exp\ Int$. □

The system we will present in this thesis is the first to comprehensively and consistently unify all of these forms of algebraic data types, along with a fully programmable constraint domain. Previously, these ideas have only been studied separately.

2.2 Type inference for standard Hindley/Milner

So far we have described the Hindley/Milner type system, and discussed some useful extensions of it. In this section we will finally look at the practical matter of type inference. As before, we will focus on a core which involves standard Hindley/Milner. Specific details of type inference or checking for the various extensions are too numerous to go into here. Details of various proposals can be found in the papers we have already cited. We will ourselves describe a single type system which unifies these ideas, and provides great leverage for detailed error reporting, in Chapter 8.

Algorithm $\mathcal{W}$ is still the basis for many[6] type inference implementations in use today [26, 37, 6, 65]. Because of this, the type error reporting and diagnosis capabilities of these systems remains broadly similar. Of course, improvements to specific implementations are always possible, but there is a fundamental limit to the expressiveness of diagnosis based on $\mathcal{W}$. We will discuss this further in Chapters 5, 6 and 10.

2.2.1 Algorithm W

Milner presented a tractable inference procedure for the Hindley/Milner type system, which he called algorithm $\mathcal{W}$ [63]. In the same work he proved the soundness of this algorithm. Later, Damas and Milner showed that it is also complete, and also computes principal types [13]. This algorithm, presented in a slightly different style, can be found in Figure 2.2.

Algorithm $\mathcal{W}$ is a function from a type environment Γ (mapping variables to type schemes) and an expression, to a pair of a substitution S and a result type scheme σ, *or* a failure. The function returns *fail* when it is unable proceed and infer a type.

The type of a variable can be instantiated as the algorithm progresses. A type that originally appeared quite general can become more specific once we have seen it in a different context. Algorithm $\mathcal{W}$ keeps track of the current type of each bound variable via the current environment Γ, and a substitution S. If a type changes, the types in Γ are never updated directly, rather a new substitution which reflects this modification is generated. At any point, the last substitution

[6] Although we cannot be absolutely sure, it is likely that almost all practical implementations of type inference for Hindley/Milner are based on algorithm $\mathcal{W}$. Certainly, no alternative, fundamentally different algorithms are as well known.

$$
\begin{array}{lll}
\mathcal{W}(\Gamma, x) & = \textsf{if } x : \forall \bar{a}.t \in \Gamma \textsf{ then } (\emptyset, [\bar{\beta}/\bar{\alpha}]t) \qquad \bar{\beta} \textit{ fresh} \\
 & \qquad\qquad \textsf{else } \textit{fail} \\
\mathcal{W}(\Gamma, e_1\ e_2) & = \textsf{let } (S_1, t_1) = \mathcal{W}(\Gamma, e_1) \\
 & \qquad (S_2, t_2) = \mathcal{W}(S_1\Gamma, e_2) \\
 & \qquad S_3 = \textit{unify}(S_2 t_1, t_2 \to \beta) \qquad \beta \textit{ fresh} \\
 & \textsf{in if } S_3 = \textit{fail} \textsf{ then } \textit{fail} \\
 & \qquad\qquad \textsf{else } (S_3 S_2 S_1, S_3\beta) \\
\mathcal{W}(\Gamma, \lambda x.e_1) & = \textsf{let } (S_1, t_1) = \mathcal{W}(\Gamma_x \cup \{x : \beta\}, e_1) \qquad \beta \textit{ fresh} \\
 & \textsf{in } (S_1, S_1(\beta \to t_1)) \\
\mathcal{W}(\Gamma, \textsf{let } x = e_1 \textsf{ in } e_2) & = \textsf{let } (S_1, t_1) = \mathcal{W}(\Gamma, e_1) \\
 & \qquad (S_2, t_2) = \mathcal{W}(\Gamma_x \cup \{x : \textit{gen}(S_1\Gamma, t_1)\}, e_2) \\
 & \textsf{in } (S_2 S_1, t_2)
\end{array}
$$

Figure 2.2: Algorithm $\mathcal{W}$

generated is the most current. Thus, the type as inferred *so far*, of any variable x in scope can be determined by applying the current substitution S to the type t, given that $x : t \in \Gamma$. In practical implementations of algorithm $\mathcal{W}$, for efficiency, no explicit substitutions are maintained. Instead, type variables are updated directly.

In the definition of $\mathcal{W}$, we have made use of a number of auxiliary functions. We call $\textit{unify}(t_1, t_2)$ to calculate a *most general unifier* (mgu) of two types (not type schemes) t_1 and t_2, if one exists or fail if not [72]. We use *gen* to generalise a type. Given an environment Γ and a type t, $\textit{gen}(\Gamma, t)$ returns the type scheme $\forall \bar{\alpha}.t$, where $\bar{a} = \textit{fv}(t) - \textit{fv}(\Gamma)$. This corresponds to the ($\forall$ Intro) rule of Figure 2.1. The algorithm is defined inductively on the structure of the language, with one case for each kind of expression.

The type of a variable x is obtained by looking it up directly in the environment. We write $\emptyset$ for the empty/identity substitution that is also returned. The $\forall$ Elim rule is built directly into this first case. Whenever it retrieves the type of a variable from Γ, any universal variables are immediately renamed. Obviously if the variable is not present in the environment, the algorithm must fail. This is not a very interesting kind of failure, however, since we can easily detect and eliminate such programs with a simple syntactic check before $\mathcal{W}$ is even invoked. Although it causes algorithm $\mathcal{W}$ to fail, we do not consider this a type error.

In the case of an application, the type of the function t_1 is inferred, and unified with a type skeleton of form $t_2 \to \beta$, where t_2 is the inferred type of the argument,

and β is a fresh variable. This unification step represents another opportunity for algorithm $\mathcal{W}$ to fail. If $S_2 t_1$ cannot be unified with $t_2 \rightarrow \beta$ then obviously no unifying substitution exists, and *unify* will fail. This is a more interesting case of failure than that mentioned earlier. We consider that this failure of $\mathcal{W}$ is the result of a *type error*. This could happen in practice, for example, if $S_2 t_1$ were a non-function type, like *Int* or *Bool*.[7] Alternatively, it could be that $S_2 t_1$ is a function type, but that the type of the argument does not match t_2, or that unifying them could lead to an occurs check error.[8]

In the case of a λ abstraction, $\mathcal{W}$ guesses a type for the argument, and proceeds to type the body of the function under the environment extended with that type. Finally, when processing a let expression, algorithm $\mathcal{W}$ first types the right-hand side of the binding, then generalises that type, making it as polymorphic as possible, and uses it to type the body of the let.

2.2.2 Type errors and algorithm $\mathcal{W}$

Let us now take a closer look at the way in which type errors are detected by algorithm $\mathcal{W}$, and the sorts of measures it can support in reporting those errors. In fairness, the algorithm was (probably) not designed with informative type error reporting as a goal. Nevertheless, since many implementations of the Hindley/Milner type system are based on it, it is important for us to explore its capabilities and limitations.

Many researchers have identified shortcomings of algorithm $\mathcal{W}$ when it comes to reporting errors [54, 59, 92, 88]. Broadly, the problems that have been identified relate to:

- The distinction between the error site reported and the actual site of the program mistake.
- The small amount of information $\mathcal{W}$ can report at the point of failure.
- The fact that $\mathcal{W}$, being a bottom-up algorithm, finds errors later than it otherwise could.

[7] Obviously this could not happen for the language of types we outlined in Section 2.1.1, since there we only have variable and function types. It is a common cause of failure for any type system in practical use.

[8] The unification algorithm mentioned earlier detects and rejects terms whose unification would lead to an infinite term. For instance, it would fail for an invocation like $\mathit{unify}(\alpha, \alpha \rightarrow \alpha)$.

One effect that algorithm $\mathcal{W}$ has on the quality of type errors is the location in the program at which it fails. It is important to distinguish between a program location which contains a mistake, which leads to a type error, and the location at which the type inference algorithm actually discovers and reports an inconsistency. These are often not the same location.

The following expression illustrates the idea that the location at which algorithm $\mathcal{W}$ fails is completely arbitrary. The point of failure is a side-effect of the precise implementations of the algorithm.

$$\lambda f.\lambda x.(f\ x)\ (x\ f)$$

Given $\Gamma = \{f : t_f, x : t_x\}$, if the left-hand side of the application is processed first then the 'current' substitution will be $[t_x \rightarrow \beta/t_f]$. If the right-hand side were processed first, then the substitution would be $[t_f \rightarrow \beta'/t_x]$. As soon as these are combined through the unification step, regardless of which of the two sub-expressions it happens in, the algorithm will fail.

Although this is a simplistic program, it contains fundamentally the same sort of type error as the `split` function of Chapter 1. In both programs, the order in which the program is traversed ends up affecting the site which is found to be untypeable. If, as in our definition of $\mathcal{W}$, the function part of the expression is visited first, then the argument will be found to be ill-typed and vice versa.

Because the algorithm is fixed once written, this ordering manifests itself as a kind of bias, which means that some perfectly plausible error sites will never be reported [59]. For this program it is quite possible, as was the case for `split`, that the site reported is not the source of the mistake. Hence, algorithm $\mathcal{W}$ will never report the actual site of the mistake for this program.

As mentioned above, type errors are detected by algorithm $\mathcal{W}$ when there is a failure to unify the type of a variable in a function position with a function type skeleton, containing the inferred type of the argument. Whenever a compiler, or any static analysis tool, discovers a failure in a program it ought to report it in such a way that the programmer is able to understand and address the cause of the error with as little effort as possible.

At that point in the calculation, algorithm $\mathcal{W}$ has access to a few nuggets of useful information. It has available the two types which could not be unified, as well as the two most current substitutions. Even though we have access to these substitutions, it may not be obvious what should be reported about them. It is

possible that these are quite large, and it is certainly not clear which variables within them can be usefully reported.

The real problem is that at the point of the error, the algorithm has the conflicting types, but is missing is any *reason* for those types. We see this clearly again in the `split` program. All that an algorithm $\mathcal{W}$-based system can usefully report is: the two conflicting types, and the location at which the conflict was discovered. There is no mention of any other locations, whose well-typedness was taken for granted, which may also have influenced the types and lead to failure.

The inescapable conclusion is that algorithm $\mathcal{W}$ often completely fails to mention the actual source of a program error. This can sometimes be confusing for programmers to deal with. Although this is admittedly a simple and contrived example, it only underscores the fact that in general, there is no way to automatically know the intended meaning of a program; no algorithm can be expected to always guess where the actual mistake is in an ill-typed program.

Another common criticism of algorithm $\mathcal{W}$ is that the locations at which it reports errors are unnecessarily large. The reason for this is that $\mathcal{W}$ is a bottom-up algorithm; it always infers the types of all sub-expressions before starting on the type of its current expression.

Consider the following expression. We use the constant, *True*, with type *Bool* to simplify the example a little. The symbol E represents an extremely large expression, of type $\alpha \to \alpha \to \alpha$.

$$\lambda f.\lambda x.f\ (E\ f\ True\ ))$$

This expression is obviously ill-typed, since the outer occurrence of f must be a function, whereas the inner f is of type *Bool* (E enforces this.) Algorithm $\mathcal{W}$, will discover this inconsistency at the top-most application site. This means that a programmer trying to understand the conflict would have to examine all of f's argument, which is notable for the very large expression (E) it contains. In some cases, it would be nicer if the inference algorithm could point to a smaller slice of the program. In the expression above, that would be the nested occurrence of f.

There have been some proposals of alternatives to $\mathcal{W}$, like Lee and Yi's algorithm $\mathcal{M}$, which they formalised after realising it was being used in an implementation of type inference for the Caml-Light [6] language at the time [54]. These variations of $\mathcal{W}$ differ in the place at which unification is performed during inference. For example, if we already unify the type of the function component of an application with a skeleton like $\beta \to \beta'$, where the βs are fresh, then we could

discover type errors if this expression is found not to be a function, while exploring either sub-expression. Unlike $\mathcal{W}$, this variation would not need to return to the outer-most application site to discover an error of this sort.

In general, however, we cannot convincingly argue that any particular traversal or ordering of unification steps leads to the discovery of 'better' error locations than any other. There are certainly examples of programs for which it can be argued that the location algorithm $\mathcal{W}$ finds is closer to the actual source of the error than other alternatives.[9] We will explore this topic more thoroughly in Chapter 10.

For these reasons, we consider that the error reporting capabilities supported by algorithm $\mathcal{W}$ are fundamentally limited. Any implementation based on it will suffer from the same basic limitations. We will revisit some of these ideas again in Chapters 5, 6 and 10, in light of the inference and error reporting systems we will present in this thesis.

2.3 Summary

In this chapter we began by giving a description of the Hindley/Milner type system. We followed that with a high-level overview of the various language and type system extensions that we will focus on in this thesis. Finally, we took a look at the practical matter of type inference for Hindley/Milner. We presented the classic algorithm $\mathcal{W}$ of Damas and Milner, still the basis of most type system implementations, and considered the effect of changing the order of unification.

[9] If we place the previous example in a practical context, it is possible that either use of f could be the actual source of the mistake. If the outer f is the problem, then $\mathcal{W}$ finds the superior solution, otherwise if it is the inner f that is wrong, then $\mathcal{M}$ will arguably give a better result.

Chapter 3

Constraints and Constraint Handling Rules

In the previous chapters we introduced the basic Hindley/Milner type system, and the prototypical type inference algorithm $\mathcal{W}$, and discussed some of the problems associated with generating type error messages for ill-typed programs. The main contribution of this thesis is the development of our own, novel inference and type error diagnosis systems which address the limitations of $\mathcal{W}$, and other inference algorithms. These systems are based firmly on an interpretation of typing in terms of constraints and constraint solving. Before we can explain our inference and error diagnosis methods, we must formally introduce our constraint domain and the language of Constraint Handling Rules, which we will do in this chapter.

We will begin by specifying our language of justified constraints (Section 3.1). Following that, we will introduce Constraint Handling Rules (CHR), describing their logical and operational semantics (Section 3.2). We also address the issue of extending the standard CHR semantics to preserve the constraint justifications necessary for error diagnosis. Having established the meaning of constraints in our framework, we then take a look at a number of essential constraint operations which we depend upon for reasoning about type errors. We present a procedure for discovering a single minimal unsatisfiable subset of a constraint set (Section 3.3), and a related algorithm for finding a single minimal implicant (Section 3.4). Finally, we look at the issue of finding *all* minimal unsatisfiable subsets of a constraint set (Section 7.2).

$$\begin{array}{lrcl} \text{Primitive} & p & ::= & t_1 = t_2 \mid U\ \bar{t} \mid \mathit{True} \\ \text{Constraint} & c & ::= & p_J \mid c \wedge c \end{array}$$

Figure 3.1: Justified constraints

3.1 Constraints

The language of constraints we employ is shown in Figure 3.1. The two most significant kinds of primitive constraints we make use of are: *equations* and *user-defined constraints*. An equation, or *Herbrand constraint* is of the form $t_1 = t_2$, where t_1 and t_2 are types that share the same structure as types in the programming language we will use.[1] We write user-defined constraints as either $U\ \bar{t}$ or $f(\bar{t})$ where U and f are both *predicate symbols*, and $\bar{t}$ denotes a sequence of types $t_1, ..., t_n$, for some n. Commonly, we refer to user constraints written in the form $U\ \bar{t}$ as *type-class constraints* and $f(\bar{t})$ as *function predicates*. Note that in Haskell, type class constraints are limited to a single type argument, whereas here we allow for any number of arguments, i.e. we are able to represent multi-parameter type classes. When we write concrete type-class constraints, we separate each type argument by a space, e.g. $U\ t_1\ t_2$, whereas the arguments of function predicates will be comma-separated, e.g. $f(t_1, t_2, t_3)$. The meaning of user constraints is defined by a CHR program. We will discuss CHR in Section 3.2. In addition to these two kinds of primitives, we also have a special always-satisfiable constraint *True*, which is equivalent to an empty conjunction of constraints. We sometimes write *False* when we mean some arbitrary, unsatisfiable (Herbrand) constraint.

We sometimes write conjunctions of constraints using a comma separator instead of the Boolean connective $\wedge$, and we often treat conjunctions as sets of constraints. If C is a conjunction then C_e represents the equations in C and C_u is the user-defined constraints in C.

The primitive constraints we use will often be explicitly justified. A *justification* is simply a list of numbers, which we write enclosed in brackets, e.g $[1, 2, 3]$. As shown in Figure 3.1, we write these justifications as trailing subscripts. To keep things more legible, we typically write singleton justifications without brackets, e.g. the justified primitive constraint written p_1 is equivalent to $p_{[1]}$. A constraint written without an explicit justification can be assumed to

[1] We leave the formal description of types until Chapter 4.

have an empty justification, e.g. p can be read as $p_{[]}$. Similarly, an 'unjustified' constraint can be interpreted simply as an equivalent constraint with an empty justification. Justifications are simply tags, and unless otherwise specified, they have no bearing on the logical meaning of any constraints.

We will use justifications as references to source program locations. In our system, each program location will be assigned a unique number, and a constraint whose justification contains that number can be said to have 'arisen from', or 'depend on', that particular source location. Often, we will write that a justification consists of not just numbers, but *locations*. We will make this connection clearer in the next chapter.

Regularly, we will need to generate new justifications from existing justifications. We write $J_1 +\!\!+ J_2$ to represent the result of appending justification J_2 to the end of J_1. A justification can be *normalised* at any time by removing any number which appears to the right of another occurrence of that number. e.g. the justification $[1, 2, 1, 3, 2]$ becomes $[1, 2, 3]$. We will often do this automatically to keep justifications a manageable length. We also make use of a function *just*, which returns the justification attached to a primitive, i.e. $\mathit{just}(p_J) = J$, or a constraint, $\mathit{just}(p_J \wedge c) = J +\!\!+ \mathit{just}(c)$.

We employ a number of standard operations and relations on constraints. We carry over all of the definitions introduced in Chapter 2, e.g. substitutions, most general unifier (mgu), etc., and extend them to apply to constraints in the usual way. Given a substitution ϕ, then $\phi((t_1 = t_2)_J) = (\phi t_1 = \phi t_2)_J$, and $\phi((U\ t_1 ... t_n)_J) = (U\ \phi t_1 ... \phi t_n)_J$. We will often write ϕ_C to represent the mgu of a constraint C. We use $fv(p)$ to find the free variables of a primitive: $fv(t_1 = t_2)_J = fv(t_1) \cup fv(t_2)$, and $fv(U\ t_1 ... t_n)_J = \bigcup_{i=1}^{n} fv(t_i)$. The free variables in a constraint are obtained as follows $fv(p \wedge C) = fv(p) \cup fv(C)$. We write $|C|$ to denote the number of constraints (or 'size'/'length') of a constraint C, i.e. $|p| = 1$, and $|p \wedge C| = 1 + |C|$. We can *project* a set of equations C onto a variable α, by finding ϕ_C, the mgu of C, and evaluating $\phi_C \alpha$. When we write $C_1 \equiv C_2$ we are stating that constraints C_1 and C_2 are syntactically equivalent.

In addition to the Boolean operator $\wedge$ (conjunction), we make use of $\neg$ (negation), $\supset$ (implication), $\leftrightarrow$ (equivalence) and quantifiers $\exists$ (existential) and $\forall$ (universal) to express conditions in formal statements, typing rules etc. We assume that $fv(o)$ computes the free variables not bound by any quantifier in an object o. We write $\bar{\exists}_V.F$ as a short-hand for $\exists fv(F) - V.F$ where F is a first-order formula and V is a set of variables. Unless otherwise stated, we assume that

formulae are implicitly universally quantified. Sometimes, we also make use of the model-theoretic entailment relation. The statement $G_1 \models G_2$ says that any model of $\forall \bar{a}.G_1$ where $\bar{a} = fv(G_1)$ is also a model of $\forall \bar{b}.G_2$ where $\bar{b} = fv(G_2)$. We use the predicate *sat* to test whether a set of constraints is satisfiable; $sat(C)$ is *True* when $\exists fv(C).C_e$. For more details on first-order logic, please consult one of the many fine textbooks available [74, 38].

3.2 Constraint Handling Rules with justifications

Constraint Handling Rules (CHR) is a committed-choice, rule-based language for writing constraint solvers [23]. A CHR program consists of a set of rewrite rules which can be applied to a global set of primitive constraints, transforming them into some final solved form. We use CHR to provide a meaning for our user constraints. Note that the term 'CHR' is commonly used as the name of the language of constraint rewriting rules, as well as to refer to a specific (constraint-handling) rule. Hence we could say that a simplification 'CHR', or 'CHR rule', is written in CHR.

We make use of the following two forms of CHR rules, written as follows[2]

$$\begin{array}{rrrcl} \text{simplification} & (r1) & c_1, \ldots, c_n & \iff & d_1, \ldots, d_m \\ \text{propagation} & (r2) & c_1, \ldots, c_n & \implies & d_1, \ldots, d_m \end{array}$$

In the above, $c_1, \ldots, c_n$ are unjustified user-defined constraints, $d_1, \ldots, d_m$ are (arbitrary) justified constraints, and *r1* and *r2* are labels by which we can refer to these rules. The constraints appearing to the left of the arrow are called the rule *head*, those to the right, the *body*. All rules must contain at least a single head constraint, i.e. $n \geq 1$, but are not required to have any constraints in their bodies. (An empty body is equivalent to *True*.) When $n = 1$, we call the rule *single-headed*. If $n > 1$, then the rule is *multi-headed*. We will usually omit explicit rule labels when they are not necessary, and will often simply refer to rules by their head constraint when they are single-headed and such a reference would be unambiguous.

The logical interpretation of the rules is as follows. Let $\bar{x}$ be the variables occurring in $\{c_1, \ldots, c_n\}$, and $\bar{y}$ be the other variables occurring in the rule. The

[2]Some implementations of CHR [24] provide for a third kind of rule, known as a 'simpagation' rule. A simpagation rule written $C/D \iff E$ is equivalent to the simplification rule $C, D \iff C, E$, where C, D, E are constraints.

logical reading is

$$\begin{aligned} \text{simplification} \quad & \forall \bar{x}((c_1 \wedge \cdots \wedge c_n) \leftrightarrow \exists \bar{y}\ (d_1 \wedge \cdots \wedge d_m)) \\ \text{propagation} \quad & \forall \bar{x}((c_1 \wedge \cdots \wedge c_n) \supset \exists \bar{y}\ (d_1 \wedge \cdots \wedge d_m)) \end{aligned}$$

Since we want to actually compute with CHRs, rather than just express logical relations, we also need to describe the operational behaviour of rules. We extend the usual derivation steps of CHR [24] to maintain and extend the justifications of constraints appearing in rule bodies.

A *simplification derivation step* applying a renamed rule instance

$$(r) \quad c_1, \ldots, c_n \iff d_1, \ldots, d_m$$

to a set of constraints C is defined as follows. Let $E \subseteq C_e$ be such that the most general unifier of E is θ. Let $D = \{c'_1, \ldots, c'_n\} \subseteq C_u$, and suppose there exists substitution σ on variables in r such that $\{\theta(c'_1), \ldots, \theta(c'_n)\} = \{\sigma(c_1), \ldots, \sigma(c_n)\}$, i.e. a subset of C_u *matches* the left hand side of r under the substitution given by E. The *justification* J of the matching is the union of the justifications of $E \cup D$. Note that there may be multiple subsets of C_e which satisfy the above condition and allow matching to occur. For our purposes, however, we require the subset E to be *minimal*. i.e. no strict subset of E can allow for a match.[3] An algorithm for finding such an E is detailed in Section 3.4.

Then we create a new set of constraints $C' = C - \{c'_1, \ldots, c'_n\} \cup \{\theta(c'_1) = c_1, \ldots, \theta(c'_n) = c_n, (d_1)_{J+}, \ldots, (d_n)_{J+}\}$. Note that the equation $\theta(c'_i) = c_i$ is shorthand for $\theta(s_1) = t_1, \ldots, \theta(s_m) = t_m$ where $c'_i \equiv p(s_1, \ldots, s_m)_{J'}$ and $c_i \equiv p(t_1, \ldots, t_m)$.

The annotation $J+$ indicates that we add the justification set J to the *beginning* of the original justification of each d_i. The other constraints (the equality constraints arising from the match) are given empty justifications. This is sufficient; the connection to the original location in the program text is retained by propagating justifications to constraints on the right hand side.

A *propagation derivation step* applying a renamed rule instance

$$(r) \quad c_1, \ldots, c_n \implies d_1, \ldots, d_m$$

[3]It may also be that there are multiple minimal subsets of constraints which imply the rule match. See Example 18 for a demonstration of this. In practice we simply pick the first such subset that we find, ignoring the others.

is defined similarly except the resulting set of constraints is $C' = C \cup \{\theta(c'_1) = c_1, \ldots, \theta(c'_n) = c_n, (d_1)_{J+}, \ldots, (d_n)_{J+}\}$.

A derivation step from global set of constraints C to C' using an instance of rule r is denoted $C \longrightarrow_r C'$. We often say that r *fires* on C. A *derivation*, denoted $C \longrightarrow_P^* C'$ is a sequence of derivation steps using rules in P where no derivation step is applicable to C'. Unless otherwise specified, rules are applied non-deterministically. The operational semantics of CHR exhaustively applies rules to the global set of constraints, being careful not to apply any propagation rule to the same constraints more than once (to avoid infinite propagation). For details on avoiding re-propagation see Abdennadher [1].

For efficiency, many practical implementations of CHR do not make non-deterministic choices about rule application [17]. For the rules we will employ, however, these sorts of low-level, implementation-specific details are not important.

3.2.1 Termination and confluence

It is possible to write CHR programs which will not terminate. Consider the simple rule:

$$F \iff F$$

Given an initial goal F this rule will replace F with another copy, which can then immediately be replaced again, and so on. In general, since CHR is Turing complete, the termination of general CHR programs is an undecidable problem.

In the CHR operational semantics we described above, rules are selected and applied non-deterministically. If we interpret the CHR rules logically, any final result that is obtained can be viewed as a logical consequence of those rules. Usually, though, we will want the result of applying a set of rules to an initial goal to be in some normal form. That way, we can test the equivalence of two final constraints syntactically [82].

Consider the two rules:

$$\begin{array}{lll} (r1) & F & \iff G \\ (r2) & F & \iff \mathit{False} \end{array}$$

Given an initial goal of F, either of the following two derivations is possible.

$$F \longrightarrow_{r1} G \qquad F \longrightarrow_{r2} \mathit{False}$$

Logically we know that both results are equivalent, since $F \leftrightarrow G, F \leftrightarrow \mathit{False}$, then $G \leftrightarrow \mathit{False}$, but we have no good way to check this algorithmically [38].

A CHR program P is confluent if given any two initial goals C_1 and C_2, there exist derivations $C_1 \longrightarrow_P^* C_3$ and $C_2 \longrightarrow_P^* C_3'$ such that C_3 and C_3' are syntactically equivalent, up to variable renaming [24]. A sound and complete confluence checking algorithm for terminating CHR programs is known [1]. The two rules above can be made confluent with the addition of the rule $G \Longleftrightarrow \mathit{False}$. In some cases, it is possible to automatically 'complete' a set of CHR rules in order to make them confluent [2]. In general, however, coming up with a 'complete' completion algorithm does not seem feasible.

Unless otherwise stated, all of the CHR rules we use will be both confluent and terminating. This also goes for CHR rules which we automatically generate from source programs. As we will see, either a specific CHR rule generation procedure will always guarantee this, or there will be simple syntactic conditions we can enforce to ensure it.[4]

3.2.2 CHR examples

We will present a number of example CHR derivations in order to illustrate the concepts introduced above. In each CHR derivation, we will underline the constraints which are matched in each step. As usual, in examples where all of the variables used in the CHR rules are distinct, and no rule fires more than once, we will not bother to rename rules before adding their constraints.

Our first example contains CHR rules whose head constraints contain only variables. Rule matching, and propagation of justifications is therefore straightforward.

Example 15 Let P be the following set of rules.

$$\begin{array}{lrcl} (f) & F\ a & \Longleftrightarrow & (G\ a)_1 \\ (g) & G\ b & \Longrightarrow & (b = \mathit{Int})_2 \\ (h) & h(c) & \Longleftrightarrow & (F\ d)_3, (c = d)_4, (d = \mathit{Float})_5 \end{array}$$

[4] Any restrictions we place on the shape of source programs are the same, if not more liberal than, those required by Haskell in any case.

We begin with the goal $h(t)_6$, and perform the following derivation.

$$\begin{array}{lll} \underline{h(t)_6} & \longrightarrow_h & t = c, \underline{(F\ d)_{[6,3]}}, (c = d)_{[6,4]}, (d = \mathit{Float})_{[6,5]} \\ & \longrightarrow_f & t = c, d = a, \underline{(G\ a)_{[6,3,1]}}, (c = d)_{[6,4]}, (d = \mathit{Float})_{[6,5]} \\ & \longrightarrow_g & t = c, d = a, a = b, (b = \mathit{Int})_{[6,3,1,2]}, (c = d)_{[6,4]}, (d = \mathit{Float})_{[6,5]} \end{array}$$

Note that whenever new constraints are added, their justifications are extended, at the front, by those of the constraint which caused the rule to fire. □

In the next example, we make use of rules with slightly more complicated heads. Matching becomes non-trivial, as multiple constraints may now be required for a rule to fire.

Example 16 Let P be the following set of rules.

$$\begin{array}{lll} (f) & f(a) & \Longleftrightarrow\ (G\ b)_1, h(c)_2, (c = b \rightarrow a)_3 \\ (g) & G\ \mathit{Int} & \Longleftrightarrow\ \mathit{True}_4 \\ (h) & h(d) & \Longleftrightarrow\ (d = \mathit{Int} \rightarrow \mathit{Bool})_5 \end{array}$$

We start our derivation with the goal $f(t)_7$.

$$\begin{array}{lll} \underline{f(t)_7} & \longrightarrow_f & t = a, (G\ b)_{[7,1]}, \underline{h(c)_{[7,2]}}, (c = b \rightarrow a)_{[7,3]}, \\ & \longrightarrow_h & t = a, c = d, \underline{(G\ b)_{[7,1]}}, \underline{(d = \mathit{Int} \rightarrow \mathit{Bool})_{[7,2,5]}}, (c = b \rightarrow a)_{[7,3]} \\ & \longrightarrow_g & t = a, c = d, b = \mathit{Int}, \mathit{True}_{[7,2,5,1,4]}, (d = \mathit{Int} \rightarrow \mathit{Bool})_{[7,2,5]}, \\ & & (c = b \rightarrow a)_{[7,3]} \end{array}$$

Note that in this derivation, rule g can only fire after h. Until h adds $(c = \mathit{Int} \rightarrow \mathit{Bool})$, the $G\ b$ is not sufficiently instantiated to cause g to fire.

We have underlined multiple constraints to show that g depends on both the equation from h as well as the user constraint from f. □

In the following example, we make use of a multi-headed CHR rule.

Example 17 Let P be the following set of rules.

$$\begin{array}{lll} (f) & f(a) & \Longleftrightarrow\ g(a)_1, (a = b \rightarrow c)_2, (H\ b\ c)_3 \\ (g) & g(d) & \Longleftrightarrow\ (d = e \rightarrow f)_4, (H\ e\ f)_5 \\ (h) & H\ g\ h,\ H\ g\ i & \Longrightarrow\ (h = i)_6 \end{array}$$

Starting with the goal $f(t)_7$, we perform the following derivation.

$$\begin{array}{rll}
\underline{f(t)_7} & \longrightarrow_f & t = a, \underline{g(a)_{[7,1]}}, (a = b \rightarrow c)_{[7,2]}, (H\ b\ c)_{[7,3]} \\
& \longrightarrow_g & t = a, \underline{a = d}, \underline{(d = e \rightarrow f)_{[7,1,4]}}, \underline{(H\ e\ f)_{[7,1,5]}}, \underline{(a = b \rightarrow c)_{[7,2]}}, \\
& & \underline{(H\ b\ c)_{[7,3]}} \\
& \longrightarrow_h & t = a, a = d, e = g, f = h, a = g, c = i, (h = i)_{[7,1,4,5,2,3,6]}, \\
& & (d = e \rightarrow f)_{[7,1,4]}, (H\ e\ f)_{[7,1,5]}, (a = b \rightarrow c)_{[7,2]}, (H\ b\ c)_{[7,3]}
\end{array}$$

The justification added to $h = i$ represents all of the locations responsible for the constraint. That includes not only those associated with the two H constraints, but also those that imply $e = b$, and thus allow the h rule to fire. □

In the previous examples, there was only ever a single subset of constraints which caused a match. As the next example illustrates, however, it's possible for there to be multiple such subsets. In such a case, we simply select one arbitrarily. Indeed, in practice, we simply commit to the first subset we find (without even checking to see whether there are any other alternatives.)

Example 18 Consider the following set of CHR rules.

$$\begin{array}{lrcl}
(f) & f(a) & \Longleftrightarrow & g(a)_1, (a = Int)_2, (H\ a\ b)_3 \\
(g) & g(c) & \Longleftrightarrow & (c = Int)_4 \\
(h) & H\ Int\ d & \Longrightarrow & (d = Bool)_6
\end{array}$$

We perform a CHR derivation, beginning with the goal $f(t)_7$.

$$\begin{array}{rll}
\underline{f(t)_7} & \longrightarrow_f & g(a)_{[7,1]}, \underline{(a = Int)_{[7,2]}}, \underline{(H\ a\ b)_{[7,3]}} \\
& \longrightarrow_h & a = Int, b = d, \underline{g(a)_{[7,1]}}, (a = Int)_{[7,2]}, (H\ a\ b)_{[7,3]}, (d = Bool)_{[7,2,3,6]} \\
& \longrightarrow_g & a = c, a = Int, b = d, (c = Int)_{[7,1,4]}, (a = Int)_{[7,2]}, (H\ a\ b)_{[7,3]}, \\
& & (d = Bool)_{[7,2,3,6]}
\end{array}$$

The following is another equally valid CHR derivation.

$$\begin{array}{rll}
\underline{f(t)_7} & \longrightarrow_f & \underline{g(a)_{[7,1]}}, (a = Int)_{[7,2]}, (H\ a\ b)_{[7,3]} \\
& \longrightarrow_g & \underline{a = c}, \underline{(c = Int)_{[7,1,4]}}, (a = Int)_{[7,2]}, \underline{(H\ a\ b)_{[7,3]}} \\
& \longrightarrow_h & a = Int, b = d, a = c, (c = Int)_{[7,1,4]}, (a = Int)_{[7,2]}, (H\ a\ b)_{[7,3]}, \\
& & (d = Bool)_{[7,1,4,2,3,6]}
\end{array}$$

Both CHR derivations are possible. Since our CHR rules are confluent, the results are logically equivalent, modulo justifications.[5]

The only difference between the two results is the justification attached to the $d = \mathit{Bool}$ constraint. In the second derivation, we (arbitrarily) chose the constraint added by the g rule as part of the explanation for the h rule firing. We may have just as well chosen the initial $a = \mathit{Int}$ constraint from f, and ended up with the same justifications.

□

Now that we have seen how constraints are solved, we will look at a number of fundamental operations we can use to reason about the results.

3.3 Minimal unsatisfiable subsets

Assume, for CHR program P we have $C \longrightarrow_P^* D$ for some constraint C and D where D is unsatisfiable. For D to be unsatisfiable it must be that D_e is unsatisfiable, since user-defined constraints only contribute new equations.

We are interested in finding a minimal subset E of D_e such that E is unsatisfiable. An unsatisfiable set is *minimal* if the removal of any constraint from that set leaves it satisfiable. i.e. assume $E \subseteq D_e$, and E is unsatisfiable, if $E - e$ is satisfiable, for any $e \in E$, then E is a minimal unsatisfiable subset of D_e. We will make use of minimal unsatisfiable subsets to diagnose type errors in later chapters.

Note that there may be multiple minimal unsatisfiable subsets within an unsatisfiable set, but in practice we will usually only be interested in finding a single one. We can also determine which constraints are present in *all* minimal unsatisfiable subsets. This can be useful if there are multiple minimal unsatisfiable subsets with a non-empty intersection.

The most naive approach to finding minimal unsatisfiable subsets involves simply enumerating and testing all subsets of some initial set. This procedure would eventually uncover all minimal unsatisfiable subsets of a constraint D after $2^{|D|}$ steps, or a single minimal unsatisfiable subset in $O(2^{|D|})$ steps; this is clearly impractical.[6]

[5] In fact, they are syntactically equivalent if we ignore justifications, since we did not rename any variables.

[6] Intuitively, for every constraint added to the original set, the number of subsets we need

```
min_unsat(D)
   M := ∅
   while satisfiable(M) {
      C := M
      while satisfiable(C) {
         let e ∈ D − C
         C := C ∪ {e}
      }
      D := C
      M := M ∪ {e}
   }
   return M
```

Figure 3.2: Finding a minimal unsatisfiable subset

Using an incremental equation solver (as almost all unification algorithms are) we can quickly determine a minimal unsatisfiable subset of D by adding the equations one at a time and detecting the first time the set is unsatisfiable. The last added equation must be involved in the minimal unsatisfiable subset. Applying this principle repeatedly results in the algorithm presented in Figure 3.2. The complexity for this algorithm is $O(|D|^2)$ calls to the *satisfiable* procedure.

The intuition behind this algorithm (and that of Section 3.4) is as follows. Assume constraint set D has a minimal unsatisfiable subset A. Our procedure, via the nexted loop, transfers constraints one-by-one from D into a temporary set C, until C becomes unsatisfiable. It is clear that C will not become unsatisfiable until all of A has been transferred into it. We know that the last constraint transferred from D into C must be in A, because the inner-loop of the procedure stops as soon as C becomes unsatisfiable. Note that we do not know which (if any) of the other constraints moved into C from D may also be in A. The algorithm accumulates, in M, all of the constraints definitely known to be from A, via the outer-loop, and stops as soon as M becomes unsatisfiable, ensuring that it is minimal.

Example 19 Consider the following set of constraints.

$$(t_1 = Char), t_2 = t_7, (t_5 = Bool), (t_6 = Bool),$$
$$(t_7 = t_5 \rightarrow t_6), (t_2 = t_1 \rightarrow t_3), (t_4 = t_3)$$

to test doubles. i.e. because we need to test those subsets with the new constraint, and those without.

The system of constraints is detected as unsatisfiable as the second last constraint $(t_2 = t_1 \rightarrow t_3)$ is added. Hence $(t_4 = t_3)$ can be excluded from consideration. Solving from the beginning, starting with $(t_2 = t_1 \rightarrow t_3)$, unsatisfiability is detected at $(t_7 = t_5 \rightarrow t_6)$. In the next iteration, starting with $(t_7 = t_5 \rightarrow t_6)$ and $(t_2 = t_1 \rightarrow t_3)$, unsatisfiability is detected at $(t_5 = Bool)$. Therefore, $(t_6 = Bool)$ can be excluded. The final result M is

$$(t_1 = Char), t_2 = t_7, (t_5 = Bool),$$
$$(t_7 = t_5 \rightarrow t_6), (t_2 = t_1 \rightarrow t_3)$$

Note that M is the only minimal unsatisfiable constraint for this example. □

Example 20 Let C be the following set of unsatisfiable constraints:

$$(a = [b]), (b = c), (d = b), (c = Char), (d = Char), (b = e), (Int = e)$$

As described earlier, a constraint $c \in C$ must appear within *all* minimal unsatisfiable subsets of C if $C - c$ is satisfiable.

Considering all the constraints above, only $C - (Int = e)$ is satisfiable; therefore $(Int = e)$ must be common to all minimal unsatisfiable subsets of C. Furthermore, since $(Int = e)$ is itself satisfiable (and is the only constraint to appear in all minimal unsatisfiable subset), there must be more than one minimal unsatisfiable subset within C.[7]

Employing the algorithm described above, We can begin our search for a minimal unsatisfiable subset knowing that $(Int = e)$ must be part of it. Progressing through C in order, our system of constraints becomes unsatisfiable as soon as we add $(b = e)$. Starting with $(Int = e), (b = e)$, we next discover unsatisfiability when we get to $(c = Char)$. We can discard the remainder of the list, $(d = Char)$ from further consideration. We next begin with $(Int = e), (b = e), (c = Char)$, and detect that our constraints are unsatisfiable upon the addition of $(b = c)$. At this point, the constraints we have accumulated are themselves unsatisfiable, and so we can stop. The constraints in the minimal unsatisfiable subset are

$$(Int = e), (b = e), (c = Char), (b = c)$$

[7] If the constraints found to be common to all minimal unsatisfiable subsets are unsatisfiable, then there can only be one minimal unsatisfiable subset.

```
in_all_min_unsat(D)
   M := ∅
   while ∃e ∈ D {
      if satisfiable(D − e) {
         M := M ∪ {e}
      }
   }
   return M
```

Figure 3.3: Finding the intersection of all minimal unsatisfiable subsets

As an aside, the other minimal unsatisfiable subset present is

$$(Int = e), (b = e), (d = Char), (d = b)$$

though our algorithm will not discover it. If we had reversed the order of the constraints in C, we would have found this subset instead.

Whenever we say that we make use of an 'arbitrary' minimal unsatisfiable subset, we generally mean the one that our algorithm happens to discover.

□

3.3.1 Intersection of all minimal unsatisfiable subsets

Given an unsatisfiable constraint D, we can straightforwardly determine which constraints $e \in D$ must occur in *all* minimal unsatisfiable subsets, since this is exactly those where $D - \{e\}$ is satisfiable. The algorithm can be found in Figure 3.3.

The complexity of this procedure is $O(|D|^2)$, using an incremental unification algorithm.

Example 21 Let us consider the constraints of Example 20 again, which we call C, and have repeated below for reference. You will recall that this set of constraints contains two minimal unsatisfiable subsets. By applying the above-mentioned algorithm, we will find the intersection of both of those subsets.

$$(a = [b]), (b = c), (d = b), (c = Char), (d = Char), (b = e), (Int = e)$$

We remove the first constraint from C and find that $C - (a = [b])$ is still

unsatisfiable. This is understandable, since $(a = [b])$ does not appear in either of the minimal unsatisfiable subsets we found in C in the previous example, therefore both subsets still remain unbroken. We then remove $(b = c)$. By doing so, we break the first minimal unsatisfiable subset we found, but not the second, and so $C - (b = c)$ is still unsatisfiable. We continue in the same vein, until we try $C - (b = e)$, which we find to be satisfiable. Since $(b = e)$ is part of both minimal unsatisfiable subsets, its removal makes them and C satisfiable. The same applies to the next constraint, $(Int = e)$.

The constraints in C found to be part of all minimal unsatisfiable subsets are $(b = e)$ and $(Int = e)$. □

3.4 Minimal implicants

In addition to finding minimal unsatisfiable constraints, we are also interested in finding minimal systems of constraints that ensure that a type has a certain shape. We've already seen the need for this in Section 3.2; whenever a CHR rule is applied, we must find a minimal subset of store constraints which allowed the rule to fire. In later sections, we will also make use of this to explain why certain unexpected, or problematic types arise, allowing us to diagnose fairly complex errors.

Assume that $C \longrightarrow_P^* D$ where $\models D \supset \exists\bar{\alpha}.D'$. We want to identify a minimal subset E of D such that $\models E \supset \exists\bar{\alpha}.D'$. The algorithm for finding minimal implicants is closely related to that for minimal unsatisfiable subsets; the code for min_impl is identical to min_unsat except the test $satisfiable(S)$ is replaced by $\neg implies(S, \exists\bar{\alpha}.D')$. The algorithm can be found in Figure 3.4.

The test $implies(M, \exists\bar{\alpha}.D')$ can be performed as follows. If D' is a system of equations only, we first add M_e to an incremental equation solver, and then add D to them, and check that no variable apart from those in $\bar{\alpha}$ is further bound from the state with M.

If D' includes user-defined constraints, then for each user-defined constraint $c_i \in D'_u$ we nondeterministically choose a user-defined constraint $c'_i \in M$. We then check that $implies(M, \exists\bar{\alpha}.(D'_e \cup \{c_i = c'_i\})$ holds as above. We need to check all possible choices for c'_i (although we can omit those which obviously lead to failure, e.g. $c_i = C\ a$ and $c'_i = D\ b$).

Example 22 Let C be the following set of constraints. We wish to find a minimal

```
min_impl(D, D')
   M := ∅
   while ¬implies(M, ∃ᾱ.D') {
      C := M
      while ¬implies(C, ∃ᾱ.D') {
         let e ∈ D − C
         C := C ∪ {e}
      }
      D := C
      M := M ∪ {e}
   }
   return M
```

Figure 3.4: Finding a minimal implicant

subset of C which implies $D = \{c = Int\}$.

$$a = (b, c), b = Int, e = a \to f, a = (d, d), f = d$$

We begin by stepping through C, accumulating constraints. Once we encounter $a = (d, d)$, we have found a subset which implies D. We can drop the remainder of the list, $f = d$ from further consideration. Returning to the start, and continuing with $a = (d, d)$, we find we have accumulated sufficient constraints to imply D upon the addition of $b = Int$, allowing us to remove $e = a \to f$ from consideration. Since $a = (d, d), b = Int$ does not imply D, we restart, and immediately add $a = (b, c)$. The accumulated constraints do imply D. Our result is:

$$a = (b, c), b = Int, a = (d, d)$$

□

In the following example, we use the minimal implicant algorithm to identify a minimal subset of store constraints which cause a CHR rule to fire. We need to find this subset in order to accurately justify the constraints added by the rule to the store.

Example 23 Assume we have the following constraint store C

$$(R\ c)_1, (a = b \to c)_2, (c = b)_3, (d = e \to b)_4, (d = f \to [f])_5, (b = [g])_6$$

and a CHR rule

$$(r) \quad R\ [t] \iff (R\ t)_7$$

and we want to find a minimal subset of C which causes rule r to fire. There's only one user constraint in C which matches the head of r, namely $R\ c$. We need to find a minimal subset of C_e which implies $\exists t.R\ c = R\ [t]$, or equivalently $\exists t.c = [t]$, which we will call D.

As usual, we begin by stepping through C_e, accumulating constraints until the condition is met. In this case, we are successful as soon as we add $(d = f \to [f])_5$. We discard $(b = [g])_6$, and restart with $(d = f \to [f])_5$. On the next pass, we get to $(d = e \to b)_4$ before we can stop. Finally, continuing with $(d = f \to [f])_5, (d = e \to b)_4$, we find that adding $(c = b)_3$ gives us an accumulated constraint that implies D. Since the constraints we have collected are themselves sufficient, we stop. The final result is:

$$(d = f \to [f])_5, (d = e \to b)_4, (c = b)_3$$

When rule r fires, we'll end up replacing $(R\ c)_1$ by $c = [t], (R\ t)_{[5,4,3,7]}$.

□

3.5 Summary

In this chapter we have introduced justified constraints and the language of Constraint Handling Rules, which will form the basis of our type system implementation. We also presented a number of important constraint reasoning and manipulation algorithms which we will use later as the basis of our type error diagnosis systems.

Chapter 4

Hindley/Milner with type classes and annotations

Having laid the foundations for our constraint and CHR-based framework in the previous chapter, we now turn to encoding the Hindley/Milner type system in this setting. Our approach to type inference follows Demoen, García de la Banda and Stuckey [14] by translating the typing problem into a constraint problem (Section 4.2.1) and inferring types by constraint solving (Section 4.2.2). However, in contrast to [14] where translation results in a set of Horn clauses, we map the typing problem to a set of Constraint Handling Rules (CHR) [23]. The advantage of CHR is that our type inference scheme can easily accommodate more advanced type extensions, like type classes and functional dependencies (see Section 4.3). Another difference between our approach and that of [14] is that we attach justifications to constraints in order to keep track of their source program locations. This allows us to identify the program locations which actually contribute to a result by simply observing the final constraints in the store, and applying the constraint reasoning operations developed earlier.

4.1 Language of expressions and types

We start off by defining the set of well-formed declarations, expressions, patterns and types we are interested in. Note that the '|' characters in the first grammar

rule are part of the syntax of data declarations.

$$
\begin{array}{lllll}
\textbf{Declarations} & d & ::= & \mathsf{data}\ T\ \bar{a} = K_1\ \bar{t} \mid ... \mid K_n\ \bar{t} \\
\textbf{Expressions} & e & ::= & f_l \mid x_l \mid (\lambda x_l.e)_l \mid (e\ e)_l \mid \mathsf{let}\ f = e\ \mathsf{in}\ e \mid \\
 & & & (\mathsf{case}\ e\ \mathsf{of}\ [p_i \rightarrow e_i]_{i \in I})_l \\
\textbf{Patterns} & p & ::= & x_l \mid (K_l\ p_1\ ...\ p_n)_l \\
\textbf{Types} & t & ::= & a \mid T \mid t\ t \\
\textbf{Type Schemes} & \sigma & ::= & t \mid \forall \bar{\alpha}.C \Rightarrow t
\end{array}
$$

Our expression language is the standard Hindley/Milner language, as presented in Chapter 2, extended with case expressions to pattern match over user-defined data types. Data declarations can be used to introduce user-defined data types, represented by K. We assume patterns are linear, i.e. no variable appears more than once in the same pattern. We often write a top-level, outermost let-binding without an explicit let keyword, and with a trivial expression body. e.g. when we write definitions of the form $f = e$, as we often will in our examples, they can be interpreted as $\mathsf{let}\ f = e\ \mathsf{in}\ f$.

Our expressions, and some other program locations, are fully *labelled*, i.e. we decorate program locations with unique numbers. We indicate these labels by a subscript following the expression, as can be seen in the language description above. Labels will become important when generating constraints from a source program. For convenience, in some examples where they are not important, we will omit explicit location numbers.

We assume that λ-bound and let-bound variables have been renamed to avoid name clashes. We commonly use x, y, z to refer to λ-bound variables and $f, g, h, \ldots$ to refer to (let-bound) user-defined functions. In any case, we assume that there is no overlap in the naming of let and λ variables.

Using case expressions we can easily encode multiple-clause definitions, with their own formal parameter:

$$
\left(\begin{array}{lll} f\ p_1 & = & e_1 \\ ... & & \\ f\ p_n & = & e_n \end{array}\right)_l \equiv \begin{array}{l} f\ =\ (\backslash x_{l'_1}\ \rightarrow\ (\mathsf{case}\ x\ \mathsf{of}\ \ p_1 \rightarrow e_1 \\ \qquad\qquad\qquad\qquad\qquad\quad\ \ ... \\ \qquad\qquad\qquad\qquad\qquad\quad\ \ p_n \rightarrow e_n)_l)_{l'_2} \end{array}
$$

and if-then-else expressions:

$$(\mathsf{if}_{l_1}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)_{l_2} \equiv \begin{array}{l}(\mathsf{case}\ e_1\ \mathsf{of}\ \ \mathrm{True}_{l_1} \rightarrow e_2 \\ \qquad\qquad\qquad\quad \mathrm{x}_{l'_1} \rightarrow e_3)_{l_2}\end{array}$$

which we will make use of in our examples.

In the above, we use a single location to represent the combined type of all clauses of a definition, which we indicate by labelled parentheses around all of the clauses. In the case of if-then-else expressions we make use of two separate locations which correspond to the conditions that: 1) the expression in the conditional part is of type *Bool*, and 2) the then and else branches have the same type. In the above, these are locations l_1 and l_2 respectively. The variable $\mathrm{x}_{l'_1}$ is a new, fresh variable.

For our purposes it is only necessary to label one pattern (*True* in the above) with location l_1, the other can be assigned some arbitrary, fresh location number. Subscripts of form l'_i represent such locations in the above description.

Our type language consists of variables a, type constructors T and type application, e.g. $T\ a$. We use common Haskell notation for writing function, pair and list types, e.g. $a \rightarrow b$, (a, b), $[a]$. Note that we impose no *kind*[1] restrictions on types. This means that our system is capable of handling what, in Haskell, are higher-kinded types and type classes.

The terminology that we use when discussing types is the same as in Chapter 2, but is worth revisiting briefly here. A type scheme is of the form $\forall\bar{\alpha}.C \Rightarrow t$ where $\bar{\alpha}$ refers to the set of bound variables, C is a set of constraints and t is a type. When C is omitted it is considered to be *True*. Note that for standard Hindley/Milner the constraint C will only consist of equations among types. We often use the term *types* to refer to both type schemes and types. Occasionally we use the term *polymorphic type* to refer to type schemes and *simple type* to refer to types not containing universal quantifiers.

4.2 Hindley/Milner with constraints and algebraic data types

The rules describing well-typedness of expressions, in Figure 4.1, are essentially the ones from HM(X) [70, 83] extended with the standard rules to deal with

[1] Just as a type is an abstraction of a value, a kind is an abstraction of a type. A kind system dictates whether a type is well-formed.

$$\text{(Var)}\quad C, \Gamma \vdash v_l : \sigma \quad (v : \sigma \in \Gamma) \qquad \text{(Abs)}\quad \frac{C, \Gamma.x : t_1 \vdash e : t_2}{C, \Gamma \vdash (\lambda x_{l_1}.e)_{l_2} : t_1 \to t_2}$$

$$\text{(Let)}\quad \frac{C, \Gamma \vdash e : \sigma \quad C, \Gamma.f : \sigma \vdash e' : t'}{C, \Gamma \vdash \mathsf{let}\ f = e\ \mathsf{in}\ e' : t'} \qquad \text{(App)}\quad \frac{C, \Gamma \vdash e_1 : t_1 \to t_2 \quad C, \Gamma \vdash e_2 : t_1}{C, \Gamma \vdash (e_1\ e_2)_l : t_2}$$

$$\text{(}\forall\text{ Intro)}\quad \frac{C \wedge D, \Gamma \vdash e : t \quad \bar{a} \notin fv(\Gamma, C)}{C, \Gamma \vdash e : \forall \bar{a}.D \Rightarrow t} \qquad \text{(}\forall\text{ Elim)}\quad \frac{C, \Gamma \vdash e : \forall \bar{a}.D \Rightarrow t' \quad F \models C \supset [\bar{t}/\bar{a}]D}{C, \Gamma \vdash e : [\bar{t}/\bar{a}]t'}$$

$$\text{(Case)}\quad \frac{C, \Gamma \vdash e : t_1 \quad C, \Gamma \vdash (p_i \to e_i)_i : t_1 \to t_2 \quad \text{for } i \in I}{C, \Gamma \vdash (\mathsf{case}\ e\ \mathsf{of}\ [(p_i \to e_i)_i]_{i \in I})_l : t_2}$$

$$\text{(Alt)}\quad \frac{p : t_1 \vdash \Gamma_p \quad C, \Gamma \cup \Gamma_p \vdash e : t_2}{C, \Gamma \vdash p \to e : t_1 \to t_2}$$

$$\text{(Pat-Var)}\quad x_l : t \vdash \{x : t\} \qquad \text{(Pat-K)}\quad \frac{K : \forall \bar{a}.t_1 \to ... \to t_n \to T\ \bar{a} \quad p_i : [\bar{t}_i/\bar{a}_i]t_i \vdash \Gamma_{p_i} \quad \text{for } i = 1, ..., n}{(K_{l_1}\ p_1\ ...\ p_n)_{l_2} : T\ \bar{t} \vdash \bigcup_{i \in \{1,...,n\}} \Gamma_{p_i}}$$

Figure 4.1: Hindley/Milner with constraints and algebraic data types

algebraic data types. Typing judgements are of the form $C, \Gamma \vdash e : \sigma$, where C is a constraint scoping over the variables in variable environment Γ, the expression e, and its type/type scheme σ. Instead of $\mathit{True}, \Gamma \vdash e : \sigma$ we may write $\Gamma \vdash e : \sigma$ for short. Both λ and let-bound variables are recorded in a variable environment Γ, which we treat as an (ordered) list of elements, though we sometimes use set notation. The statement $\Gamma.x : \sigma$ represents a new, extended type environment whose prefix is Γ, and last element is $x : \sigma$. For example, if $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$, then $\Gamma.x : \sigma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n, x : \sigma\}$.

In rule (Var), we assume that v either refers to a λ- or let-bound variable,

whose type we simply look up in Γ. This and rules (Abs), (App), and (Let) are the standard Hindley/Milner rules, with the added constraint component. In rule ($\forall$ Intro), we build type schemes by pushing in the "affected" constraints D from the constraint context $C \wedge D$. Note that we slightly deviate from the standard HM(X) ($\forall$ Intro) where instead of C we find $C \wedge \exists \bar{a}.D$ in the judgement in the conclusion. However, the current rule is good enough for a lazy language[2], and our main target is Haskell. In rule ($\forall$ Elim), we assume that F refers to a first-order formula specifying relations among user-defined constraints.

The remainder of the rules are there to handle pattern matching of algebraic data types, and were not present in Chapter 2. The rule (Case) works by typing each case alternative, and unifying the pattern types with the type of the scrutinised expression, as well as combining the types of all the possible results. The (Alt) rule types a single case alternative. We make use of an auxiliary judgement of the form $p : t \vdash \Gamma$ to compute the variable binding of pattern variables. Rule (Pat-Var) simply assigns some type to a pattern variable, while (Pat-K) looks up the type of a pattern constructor in Γ, renames it, and unifies it with the types of its arguments.

We assume that the types of all pattern constructors, K, are known in advance, as in rule (Pat-K). For instances, a declaration like

```
data List a = Nil
            | Cons a (List a)
```

is recorded in the initial environment $\{Nil : \forall a.List\ a, Cons : \forall a.a \rightarrow List\ a \rightarrow List\ a\}$. Our assumption is that types of constructors K are of the form $\forall \bar{a}.t_1 \rightarrow ... \rightarrow t_n \rightarrow T\ \bar{a}$ where $fv(t_1, ..., t_n) \subseteq \bar{a}$. That is, all variables in the types of of the constructor's arguments must appear in its result type. We do not (yet) consider existential types [53] or other recent extensions such as guarded recursive data types [90].[3]

4.2.1 Translation to CHR

The basic idea of our translation is that for each definition (let-bound variable) `f = e`, we introduce a CHR of the form $f(t, x) \iff C$. The type parameter t refers to the type of `f` whereas x refers to the set of types of λ-bound variables in

[2]See Odersky, Sulzmann and Wehr [70] for a detailed discussion.

[3]In Chapter 8 we re-formulate our type inference scheme to accommodate these and other extensions.

scope (i.e. the types of free variables which come from the enclosing definition). The reason for x is that we must ensure that λ-bound variables remain monomorphic. The set C contains the constraints generated out of expression `e` plus some additional constraints restricting x. We use list notation (on the level of types) to refer to the "set" of types of λ-bound variables. In order to avoid confusion with lists of values, we write $\langle x_1, \ldots, x_n \rangle$ to denote the list of types $x_1, \ldots, x_n$. We write $\langle x | r \rangle$ to denote the list of types with head x and tail r. Since CHR rules operate at the same level (regardless of the nesting of the original binding), what we are performing here is a form of λ-lifting [41] on the level of types.

The key idea is that each function predicate $f(t, x)$ that arises during solving represents an instantiation of `f`'s type scheme (due to a call to `f` in some expression), which in turn is defined by a CHR simplification rule of form $f(t', x') \iff C$. Application of this rule yields essentially the same constraints as the let-expanded version of any such function call. The following example provides some intuition about our translation scheme.

Example 24 Consider the following program. We have left out explicit location numbers, since they are not important here.

```
k z = let h w = (w,z)
      in  let f x = let g y = (x,y)
                    in (g 1, g True, h 3)
          in f z
```

A (partial) description of the resulting CHR rules *might* look as follows. For simplicity, we leave out the constraints generated out of expressions. We write t_x to denote the type of λ-bound variable `x`.

$$\begin{array}{lrcl} \text{(k)} & k(t,x) & \iff & x = \langle\rangle, \ldots \\ \text{(h)} & h(t,x) & \iff & x = \langle t_z \rangle, \ldots \\ \text{(f)} & f(t,x) & \iff & x = \langle t_z, t_w \rangle, \ldots \\ \text{(g)} & g(t,x) & \iff & x = \langle t_z, t_x \rangle, \ldots \end{array}$$

Note that the λ parameter x refers exactly to the set of types of all free (λ-bound) variables in scope.

Consider the sub-expression `(g 1, g True, h 3)`. At each instantiation site, we need to correctly specify the set of types of λ-bound variables which were in scope at the function definition site. In the above program, λ variables `z` and `x`

are in scope at `g`'s definition site, whereas only `z` is in scope at the point of `h`'s binding. Among others, we might generate the following constraints out of this expression:

$$g(t_1, x_1), x_1 = \langle t_z, t_x \rangle, t_1 = Int \to t'_1,$$
$$g(t_2, x_2), x_2 = \langle t_z, t_x \rangle, t_2 = Bool \to t'_2,$$
$$h(t_3, x_3), x_3 = \langle t_z \rangle, t_3 = Int \to t'_3, \ldots$$

At function instantiation sites our constraint generation algorithm needs to remember the types of the λ-variables which were in scope at the function definition site. We use a simple trick to avoid such calculations.

We leave "open" the list of types of λ-bound variables within definition CHRs. In other words, we terminate these lists with a fresh variable, rather than a '*Nil*' constructor. This means that at instantiation sites, we can simply pass in the complete list of variables in scope at that point. The prefix of this list will match the open list in the definition, and any additional type variables can be absorbed by the list's tail variable. The assumption, of course, is that these lists are extended only at their tails.[4]

Our actual translation yields a (more complete) instance of the following.

$$\begin{array}{lllll}
\text{(k')} & k(t,x) & \Longleftrightarrow & x = r, f(t, x_1), x_1 = \langle t_z \rangle, \ldots \\
\text{(h')} & h(t,x) & \Longleftrightarrow & \underline{x = \langle t_z | r \rangle}, \ldots \\
\text{(f')} & f(t,x) & \Longleftrightarrow & x = \langle t_z | r \rangle, g(t_1, x_1), x_1 = \langle t_z, t_x \rangle, g(t_2, x_2), x_2 = \langle t_z, t_x \rangle, \\
 & & & h(t_3, x_3), \underline{\underline{x_3 = \langle t_z, t_x \rangle}}, \ldots \\
\text{(g')} & g(t,x) & \Longleftrightarrow & x = \langle t_z, t_x | r \rangle, \ldots
\end{array}$$

In rule (h') we require that variable `z` is in scope but allow for possibly more variables, which can be innocently bound to the fresh variable r (see underlined constraint). In rule (f'), we pass in the (somewhat redundant) variable t_x as part of the x parameter at the instantiation site of `h` (see double-underlined constraint). There is no harm in doing so, because there can be no reference to variable t_x on the right hand side of rule (h'). □

The formal translation of the typing problem consists of a mutually recursive process of generating constraints out of expressions and generating CHRs for function definitions.

[4]The description of extending a type environment Γ, in Section 4.2, conforms to this requirement.

$$\text{(Var-x)} \quad \frac{(x:t) \in \Gamma \quad t_l \text{ fresh}}{E, \Gamma, x_l \vdash_{Cons} ((t_l = t)_l \mid t_l)}$$

$$\text{(Var-f)} \quad \frac{\begin{array}{c} f \in E \quad \Gamma = \{\overline{y : t_x}\} \quad t_l, t \text{ fresh} \\ C = \{f(t,x)_l \wedge x = \langle \bar{t_x} \rangle \wedge t_l = t\} \end{array}}{E, \Gamma, f_l \vdash_{Cons} (C \mid t_l)}$$

$$\text{(Abs)} \quad \frac{E, \Gamma.x : t_{l_1}, e \vdash_{Cons} (C \mid t) \quad t_{l_1}, t_{l_2}, t' \text{ fresh}}{E, \Gamma, (\lambda x_{l_1}.e)_{l_2} \vdash_{Cons} (C, (t_{l_2} = t' \to t)_{l_2}, (t_{l_1} = t')_{l_1} \mid t_{l_2})}$$

$$\text{(App)} \quad \frac{E, \Gamma, e_1 \vdash_{Cons} (C_1 \mid t_1) \quad \Gamma, e_2 \vdash_{Cons} (C_2 \mid t_2) \quad t, t_l \text{ fresh}}{E, \Gamma, (e_1\ e_2)_l \vdash_{Cons} (C_1, C_2, (t_1 = t_2 \to t)_l, (t_l = t)_l) \mid t_l)}$$

$$\text{(Let)} \quad \frac{E \cup \{f\}, \Gamma, e_2 \vdash_{Cons} (C \mid t)}{E, \Gamma, \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \vdash_{Cons} (C \mid t)}$$

Figure 4.2: Justified constraint generation

Constraint generation is formulated as a logical deduction system with clauses of the form $E, \Gamma, e \vdash_{Cons} (C \mid t)$ where the environment E, containing all let-defined and pre-defined functions, environment Γ consisting of lambda-bound variables, and expression e are input parameters and constraint C and type t are output parameters. Each individual sub-expression gives rise to a constraint which is justified by the location attached to this sub-expression. We ensure that for each location l we generate a constraint of the form $(t_l = t)_l$ for some t. We will make use of this canonical form in later chapters.

Figure 4.2 contains a full specification of this process, which we will describe. In rule (Var-x) we simply look up the type of a λ-bound variable in Γ. In rule (Var-f) we generate an "instantiation" constraint, to represent the type of a let-defined or pre-defined function. The constraint $f(t,x), x = \langle t_{x_1}, \ldots, t_{x_n} \rangle$ demands an instance of `f` on type t where $(t_{x_1}, \ldots, t_{x_n})$ refers to the set of types of λ-bound variables in scope. We will usually combine these constraints, writing $f(t, \langle t_{x_1}, \ldots, t_{x_n} \rangle)$ instead of the above. The actual type of `f` will be described by a CHR where the set of types of λ-bound variables is left open. Note that, as mentioned earlier, we impose an ordering on elements in the environment Γ; we assume that type assignments in the environment Γ are ordered according to the

$$\text{(Case)}\quad \frac{\begin{array}{c} E,\Gamma,e \vdash_{Cons} (C_e \mid t_e) \\ E,\Gamma,(p_i \to e_i)_i \vdash_{Cons} (C_i \mid t_i) \quad \text{for } i \in I \\ C = \left\{\bigcup_{i\in I}(t_i = t_e \to t' \wedge C_i)\right\} \wedge (t_l = t')_l \quad t', t_l \text{ fresh} \end{array}}{E,\Gamma,(\mathsf{case}\ e\ \mathsf{of}\ [p_i \to e_i]_{i\in I})_l \vdash_{Cons} (C \mid t_l)}$$

$$\text{(Alt)}\quad \frac{\begin{array}{c} p \vdash_{Cons} (C_p \mid t_p \mid \Gamma') \quad E,\Gamma \cup \Gamma', e \vdash_{Cons} (C_e \mid t_e) \\ t = t_p \to t_e \quad t \text{ fresh} \end{array}}{E,\Gamma,p \to e \vdash_{Cons} (C \mid t)}$$

$$\text{(Pat-Var)}\quad \frac{t_l, t_x \text{ fresh}}{x_l \vdash_{Cons} ((t_l = t_x)_l \mid t_l \mid \{x : t_x\})}$$

$$\text{(Pat-K)}\quad \frac{\begin{array}{c} p_i \vdash_{Cons} (t_{p_i} \mid C_{p_i} \mid \Gamma_{p_i}) \quad \text{for } i = 1,...,n \\ K : \forall \bar{a}.t_K \quad t'_K = [\bar{b}/\bar{a}]t_K \\ \Gamma_p = \bigcup_{i\in\{1,...,n\}} \Gamma_{p_i} \quad t_{l_1}, t_{l_2}, \bar{b} \text{ fresh} \\ C' = \left\{ \begin{array}{c} t'_K = t_{p_1} \to ... \to t_{p_n} \to t'_{l_2}, \\ (t_{l_1} = t'_K)_{l_1}, (t_{l_2} = t'_{l_2})_{l_2} \end{array} \right\} \cup \bigcup_{i\in\{1,..,n\}} C_{p_i} \end{array}}{(K_{l_1}\ p_1\ ...\ p_n)_{l_2} \vdash_{Cons} (C' \mid t_{l_2} \mid \Gamma_p)}$$

Figure 4.2: *(continued)*

scope of variables. The rules (Abs), (App) and (Let) mirror their corresponding typing rules. Notice that in (App), instead of generating a single constraint $(t_1 = t_2 \to t_l)_l$ for location l, we break this into $(t_1 = t_2 \to t)_l, (t_l = t)_l$ to maintain the invariant that for each location l, there is a constraint of form $t_l = t$. In the (Case) rule, we generate constraints from all alternatives, and produce constraints to unify all of those alternative types and the type of the scrutinised expression.

The (Alt) rule generates constraints for a single case alternative. We make use of auxiliary judgements of the form $p \vdash_{Cons} (C \mid t \mid \Gamma_p)$, where p is a pattern, C and t are the constraint and type generated from that pattern and Γ_p is the environment mapping all of the pattern variables nested within p to their types. We use (Pat-Var) to simply assign a new, fresh type variable t_x to a pattern variable, and return an environment reflecting this mapping. In the (Pat-K) rule we (again) assume that the type of constructors is known in advance. We rename that type, from t_K to t'_K, and unify it with the types of all of the sub-

$$\text{(Var)} \qquad E, \Gamma, v_l \vdash_{Def} \emptyset$$

$$\text{(Abs)} \qquad \frac{t_l \text{ fresh} \quad E, \Gamma.x : t_{l_1}, e \vdash_{Def} P}{E, \Gamma, (\lambda x_{l_1}.e)_{l_2} \vdash_{Def} P}$$

$$\text{(App)} \qquad \frac{E, \Gamma, e_1 \vdash_{Def} P_1 \quad \Gamma, e_2 \vdash_{Def} P_2}{E, \Gamma, (e_1\ e_2)_l \vdash_{Def} P_1 \cup P_2}$$

$$\text{(Let)} \qquad \frac{\begin{array}{c} E, \Gamma, e_1 \vdash_{Cons} (C \mid t) \quad \Gamma = \{x_1 : t_1, \ldots, x_n : t_n\} \quad l, r \text{ fresh} \\ E, \Gamma, e_1 \vdash_{Def} P_1 \quad E \cup \{f\}, \Gamma, e_2 \vdash_{Def} P_2 \\ P = P_1 \cup P_2 \cup \{f(t, x) \Longleftrightarrow C, x = \langle t_1, \ldots . t_n | r \rangle\} \end{array}}{E, \Gamma, \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \vdash_{Def} P}$$

$$\text{(Case)} \qquad \frac{\begin{array}{c} E, \Gamma, e \vdash_{Def} P \\ p_i \vdash_{Cons} (\_ \mid \_ \mid \Gamma_{p_i}) \quad E, \Gamma \cup \Gamma_{p_i}, e_i \vdash_{Def} P_i \quad \text{for } i \in I \end{array}}{E, \Gamma, (\mathsf{case}\ e\ \mathsf{of}\ [p_i \to e_i]_{i \in I})_l \vdash_{Def} P \cup \bigcup_{i \in I} P_i}$$

Figure 4.3: CHR rule generation for Hindley/Milner

patterns that the constructor has been applied to. The return value includes the combined constraints of the sub-patterns plus the constructor, the result type of the constructor application, and the combined type environments of sub-patterns.

Often, when we are proceeding informally, we will not bother to desugar a function binding to a simple let binding and a number of nested abstractions. We will typically generate the 'obvious' constraints, in place. For example, a definition $(f\ p_1 \ldots p_n = e)_0$ will yield $t_0 = t_1 \to \ldots \to t_n \to t_e$, where the t_is are the types of the corresponding patterns, and t_e is the type of the expression.

Generation of CHR rules is also formulated as a logical deduction system with clauses of the form $E, \Gamma, e \vdash_{Def} P$ where input parameters E, Γ and e are as before and the set P of CHRs is the output parameter. Figure 4.3 contains the full specification of this procedure.

No rule can be generated from a single variable, as in (Var). The (Abs) and (App) rules are straightforward, in that they only serve to combine the rules generated from sub-expressions.

The (Let) rule is most interesting. As mentioned earlier, for each let-bound variable we generate a new CHR rule. This involves invoking the constraint

generation procedure just described on the body of binding e_1, yielding constraints C. We also need to generate a type list for the x component, which contains all of the λ-bound variables in scope. As discussed, we leave the list of types of lambda-bound variables open at definition sites.

A subtlety of rule (Case), is that when generating rules P_i arising out of the body e_i of alternative clauses we need to take into account the pattern binding Γ_{p_i}. For instance, for the expression

$$\lambda x.\mathsf{case}\ x\ \mathsf{of}\ (x', y') \to \mathsf{let}\ f = \lambda z.(z, x')\mathsf{in} f\ x$$

we need to extend Γ with entries for x' and y' before translating the let binding. We can, however, neglect the types and constraints arising out of p_i. In the (Case) rule we indicate this by employing "don't-care" variables _.

It is straightforward to see that both constraint and CHR generation procedures are terminating, since each rule invokes the same procedure on an ever smaller sub-expression. Importantly, we can also state that the rules generated are terminating and confluent.

Any set of rules P, generated from $\vdash_{Def}$ must terminate because our source language does not contain recursive bindings; therefore, if $f(t, x) \longrightarrow_P^* D \longrightarrow_P \ldots$, then $f(t', x') \notin D$, for any t, x, t', x'. Furthermore, P must be confluent because $\vdash_{Def}$ never generates overlapping CHRs rules; a let-bound variable f gets a single rule $f(t, x) \iff C$ (see the (Let) rule of Figure 4.3).

4.2.2 Type inference via CHR solving

We consider a Hindley/Milner *typing problem* (Γ, e) consisting of an environment Γ and expression e. For convenience, we assume that Γ can be split into a component Γ_{init} and Γ_λ such that $fv(\Gamma_{init}) \subseteq fv(\Gamma_\lambda)$ and types in Γ_λ are simple, i.e. not universally quantified. In essence, we demand that if a type scheme in Γ contains an unbound variable, it must be mentioned in some simple type. For each function (let-bound variable) f in Γ_{init} we introduce a binary predicate symbol f which we record in E_{init}. We build a set $P_{E_{init}}$ of CHR rules by generating for each $f : \forall \bar{a}.C \Rightarrow t \in \Gamma_{init}$ the rule $f(t', x) \iff C, t' = t$ where t' and x are fresh. To represent this, we write $P_{E_{init}}, E_{init} \sim \Gamma_{init}, \Gamma_\lambda$.

Type inference proceeds as follows: We first compute $E_{init}, \Gamma_\lambda, e \vdash_{Cons} (C \mid t)$ and $E_{init}, \Gamma_\lambda, e \vdash_{Def} P$. To infer the type of e, we perform $C \longrightarrow^*_{P_{E_{init}}, P} D$.

Note that by construction these CHRs are terminating and confluent. If D_e is satisfiable, then e has type $\forall \bar{a}.\phi(D_u) \Rightarrow \phi(t)$ where $\phi = mgu(D_e)$ and $\bar{a} = fv(\phi D_u, \phi t) - fv(\phi \Gamma_\lambda)$. Note that in the case of standard Hindley/Milner we have that D_u, the set of all user-defined constraints in D, always equals *True*.

We can state that our formulation of type inference is sound and complete w.r.t. the standard Hindley/Milner typing rules. Proofs can be found in Appendix A.1. For soundness we demand that let-defined functions are "realisable", i.e. used in the program text. We say expression e_1 is *let-realisable* iff for each sub-expression of form $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ it is the case that $x \in fv(e_2)$. We demand let-realisability because our scheme is a bit 'lazier' than other formulations.

Example 25 Consider

```
e = let f = True True
    in  False
```

Our (simplified) translation to CHRs yields

$$\begin{array}{lcl} f(t) & \Longleftrightarrow & t_1 = Bool, t_1 = t_2 \rightarrow t_3, t_2 = Bool, t_3 = t \\ e(t) & \Longleftrightarrow & t = Bool \end{array}$$

For simplicity, we omit justifications and the x parameter. Type inference for expression `e` succeeds, although function `f` is ill-typed. There is no occurrence of `f` in the let body, hence we never execute the CHR belonging to `f`. In a traditional approach, like algorithm $\mathcal{W}$, type inference for `e` proceeds by first inferring the type of `f`, which would immediately detect that `f` is not well-typed. □

To detect such cases, we could additionally check that all defined functions must be type correct, by simply executing the corresponding CHRs. The alternative is to demand that all let-defined functions are realisable. In fact, we could enforce realisability directly in our translation to CHRs. We refer to [80] for details.

The advantage of our inference scheme is that, by employing the justification-propagating CHR semantics of Chapter 3, we keep track of precisely those program locations which contribute to the final result.

Example 26 Consider the following location-annotated program.

$(\mathtt{f}\ \mathtt{x}_2\ \mathtt{=}\ \mathtt{let}\ \mathtt{g}_3\ \mathtt{y}_4\ \mathtt{=}\ (\mathtt{x}_6\mathtt{,}\mathtt{y}_7)_5\ \mathtt{in}\ (\mathtt{g}_8\ \mathtt{x}_9)_{10})_1$

From this source we generate the following rules.

$$\begin{array}{lcl} f(t,x) & \Longleftrightarrow & t = t_1, x = r, t_1 = t_2 \rightarrow t_{10}, g(t_8, \langle t_x \rangle)_8, (t_9 = t_x)_{[2,9]}, \\ & & (t_8 = t_9 \rightarrow t_{10})_{10} \\ g(t,x) & \Longleftrightarrow & t = t_3, x = \langle t_x | r \rangle, (t_3 = t_4 \rightarrow t_5)_3, (t_5 = (t_6, t_7))_5, (t_6 = t_x)_{[2,6]}, \\ & & (t_7 = t_y)_{[4,7]} \end{array}$$

We now begin a CHR derivation in order to infer `f`'s type.

$$\begin{array}{ll} & f(t,x) \\ \rightarrow_f & t = t_1, x = r, t_1 = t_2 \rightarrow t_{10}, g(t_8, \langle t_x \rangle)_8, (t_9 = t_x)_{[2,9]}, (t_8 = t_9 \rightarrow t_{10})_{10} \\ \rightarrow_g & t = t_1, x = r, t_1 = t_2 \rightarrow t_{10}, t' = t_8, (x' = \langle t_x \rangle)_8, (t' = t_3)_8, (x' = \langle t'_x | r' \rangle)_8, \\ & (t_3 = t_4 \rightarrow t_5)_{[8,3]}, (t_5 = (t_6, t_7))_{[8,5]}, (t_6 = t_x)_{[8,2,6]}, (t_7 = t_y)_{[8,4,7]}, \\ & (t_9 = t_x)_{[8,2,9]}, (t_8 = t_9 \rightarrow t_{10})_{[8,10]} \end{array}$$

Solving the result (i.e. unifying equations in the final constraint store), we find `f` has type $\forall t_x.t_x \rightarrow (t_x, t_x)$. Note that we are able to universally quantify over t_x since t_x is not bound outside `f`'s scope. We know this because $\phi t_x \notin fv(\phi x)$, where ϕ is the mgu of the result. The x here plays the same role as the Γ_λ in the description of type inference at the start of this section.

Note that when the second rule fires, we re-justify all of `g`'s constraints to indicate that they originated from the specific call site at location 8. Also important to note is the use of the x parameter in the CHRs above. By lifting the type of the λ-bound variable `x` into x we are able to keep `x`'s type consistent across rule firings.

□

The following example shows how one of the simple constraint reasoning steps of Chapter 3, allows us to identify all those program locations which actually contribute to the final inference result, or even just some chosen part of that result.

Example 27 Consider again the program introduced in Example 26, and the constraints inferred on `f`'s type, both repeated below.

```
f1 x2 = let g3 y4 = (x6,y7)5 in (g8 x9)10
```

$$D = \left\{ \begin{array}{c} t = t_1, t_1 = t_2 \to t_{10}, t' = t_8, (l' = \langle t_x \rangle)_8, (t' = t_3)_8, (l' = \langle t_x | r \rangle)_8, \\ (t_3 = t_4 \to t_5)_{[8,3]}, (t_5 = (t_6, t_7))_{[8,5]}, (t_6 = t_x)_{[8,2,6]}, (t_7 = t_y)_{[8,4,7]}, \\ (t_9 = t_x)_{[8,2,9]}, (t_8 = t_9 \to t_{10})_{[8,10]} \end{array} \right\}$$

In the above, t represents `f`'s type. Solving, we infer `f` has type $\forall t_x.t_x \to (t_x, t_x)$.

Assume now that we are interested in discovering why `f`'s result is a tuple. That is, which program locations enforce that `f`'s type is of form $a \to (b, c)$, for some a, b, c which we're not interested in. Let $D' = \{t = a \to (b, c)\}$.

Using the minimal implicant algorithm of Section 3.4, i.e. $\mathsf{min\_impl}(D, D')$, we find:

$$\left(\begin{array}{l} t = t_1, t_1 = t_2 \to t_{10}, (t_8 = t_9 \to t_{10})_{[8,10]}, (t' = t_8)_8, \\ (t' = t_3)_8, (t_3 = t_4 \to t_5)_{[8,3]}, (t_5 = (t_6, t_7))_{[8,5]} \end{array} \right) \supset \bar{\exists}_t.t = a \to (b, c)$$

Its clear that not all of the constraints from the original derivation result are necessary to establish that `f`'s return type is a tuple; only the ones above are required. By reading the locations of the justifications in the above implicant constraints, we see that only locations 3,5,8 and 10 of the original program are important for this result. Indeed, we could completely replace the other program locations with *any* other expressions, even ill-typed, and the above result would still hold. For example, replacing $\mathtt{x}_9$ with `True` would not affect the result.

□

In Chapter 5, we build on these ideas to show that in the event of a type error the justifications attached to minimal unsatisfiable constraints support basic type error reporting. Chapter 6 discusses more sophisticated methods. In the next section, however, we consider how to extend our inference framework to support a number of other type system extensions.

4.2.3 Inferring types of sub-expressions

In the previous section we described the process of inferring the type of a top-level let-bound identifier. For a let-bound variable `f` this involves generating CHR rules P out of our program, performing a CHR derivation $f(t, x) \longrightarrow_P^* C$, and projecting C onto t. It is obvious that we could also project C onto other variables, which represent the types of various program locations, in order to

determine their types. If we want to infer the principal types of expressions within nested let-bindings, however, we face a slight difficulty.

Example 28 Consider the following program. The function `not`, has the same type as its counterpart in Haskell, i.e. $Bool \to Bool$.

```
f = \x -> let g = \y -> x
          in  g (not x)
```

The CHR rules we generate will look like the following. We include the rule for `not`, which simply asserts the type we know it should have. We have simplified the rules below slightly to clarify the issue.

$$\begin{array}{rrcl}
(f) & f(t,x) & \iff & t = t_x \to t_r, not(t_x \to t_n), g(t_g, \langle t_x \rangle), t_g = t_n \to t_r \\
(g) & g(t,x) & \iff & t = t_y \to t_x, x = \langle t_x | r \rangle \\
(not) & not(t) & \iff & t = Bool \to Bool
\end{array}$$

Inferring `f`'s type, we perform the following CHR derivation (also slightly simplified.)

$$\begin{array}{rl}
 & \underline{f(t,x)} \\
\longrightarrow_f & t = t_x \to t_r, \underline{not(t_x \to t_n)}, g(t_g, \langle t_x \rangle), t_g = t_n \to t_r \\
\longrightarrow_{not} & t = t_x \to t_r, \underline{g(t_g, \langle t_x \rangle)}, t_g = t_n \to t_r, t_x \to t_n = Bool \to Bool \\
\longrightarrow_g & t = t_x \to t_r, t_g = t_n \to t_r, t_x \to t_n = Bool \to Bool, t_g = t'_y \to t'_x, \\
 & \langle t_x \rangle = \langle t'_x | r \rangle
\end{array}$$

We can determine `f`'s most general type by simply projecting the above constraints on t, which yields $Bool \to Bool$. If we want to find `g`'s principal type, however, we face a problem. We cannot simply solve for t_g, as t_g corresponds to a specific instantiation of `g`'s type, i.e. at the call site in `f`. Certainly, the type we would get, $Bool \to Bool$, is a valid type for `g`, but we are interested in finding the most general type, $\forall t_y.t_y \to Bool$ in this case.

Furthermore, unlike for `f` we cannot simply run a goal $g(t,x) \longrightarrow^* C$ and project C on t. The reason for this is that there may be expressions outside of `g` which constrain λ variables used in `g`, thereby affecting its type. Indeed, that is the case here with the variable `x`, which `f` constrains via the `not` function. Without taking this effect into account we would infer $\forall t_y, t_x.t_y \to t_x$, which is clearly too polymorphic. □

In general, assume we have a program of the following form.

```
f = ...  let g = ...  e1 ...
         in  ...  g2 ...
```

The variable `f` is bound at the top-level, `g` is bound at some arbitrary sub-expression within `f`'s definition (it could be nested within other lets, to any depth), and e_1 is an immediate sub-expression of `g`, i.e. it does not occur in the definition of any other let-binding that may be in `g`. We also assume that `g` is let-realisable, i.e. `f` calls it, whether directly or indirectly via a call to some other let variable.

We can infer the type of e_1 as follows. Assume that t_1 is the variable assigned to represent the type of e_1. We generate CHRs P out of our program, and perform the derivation $f(t, x) \longrightarrow_P ... \longrightarrow_P C \cup \{g(t', x')\}$, where t, x are fresh variables, and C_u contains no function predicates, i.e. we perform a normal CHR derivation except that we apply rules to all other user constraints in preference to g, and we stop when a lone g constraint is the only function predicate in the store.[5] We then run the CHR derivation $\{g(t'', x')\} \cup C \longrightarrow^*_P C'$, where t'' is fresh. The type of expression e_1 can then be obtained from C' as usual, i.e. it is $\forall \bar{a}.D \Rightarrow t'_1$, where $D = \phi_{C'}(C'_u)$, $t'_1 = \phi_{C'}(t_1)$, $\phi_{C'}$ is the mgu of C' and $\bar{a} = fv(D, t'_1) - fv(\phi_{C'}(x'))$.

By breaking up the derivation, and throwing away the t component of the g constraint, we get the types within `g` independent of that particular instantiation. At the same time, we retain the effect that `f`, or any other let-binder outside of `g`, has on the λ-bound variables that `g` sees.

4.3 Hindley/Milner extensions

We will now extend our inference scheme from the previous section, in stages, to deal with monomorphic recursive functions (Section 4.3.1), type annotations[6] (Section 4.3.2) and type-class style overloading (Section 4.3.3). These extensions require additional inference tests besides checking for satisfiability.

In previous work, Stuckey, Sulzmann, et al. [82, 16] showed that CHRs serve as a low-level notation for type classes. Before we can use them, we must check that

[5] If there were multiple calls to g then all but one will have been reduced. Note that there may also be unresolved type class user constraints in the store.

[6] With the addition of explicit type annotations we will be able to type polymorphic recursive let-bound variables.

CHRs generated from type classes are "consistent" and "terminating". These conditions can be guaranteed by imposing syntactic restrictions on type classes [16].

Furthermore, as described briefly in Chapter 2 we need to check that types are "unambiguous", and the extension to type annotations implies that we also need to test for "subsumption" among type schemes. Both of these checks can be phrased in terms of CHRs.

4.3.1 Monomorphic recursive functions

It is well-known that type inference for recursive functions is undecidable unless we impose some restrictions [49]. The restriction we impose is that recursive functions must be *monomorphic*, i.e. they must have a simple type. This is a standard limitation, and is present in both ML and Haskell. The following typing rule added to Figure 4.1 enforces this restriction. As usual we assume that program locations are labelled with unique numbers.

$$\text{(Mono-Rec)} \quad \frac{C, \Gamma.f : t \vdash e : t}{C, \Gamma \vdash (\mathsf{rec} f \ \mathsf{in}\ e)_l : t}$$

We introduce a new language construct, $\mathsf{rec} f \ \mathsf{in}\ e$, for defining recursive functions. As is common, we will use some syntactic sugar when defining such functions. For example, the definition of the factorial function

$$fac = \lambda n.\mathsf{if}\ 0 == n\ \mathsf{then}\ 1\ \mathsf{else}\ fac\ (n-1) * n$$

is considered syntactic sugar for

$$fac = \mathsf{rec} f\ \mathsf{in} \lambda n.\mathsf{if}\ 0 == n\ \mathsf{then}\ 1\ \mathsf{else}\ f\ (n-1) * n$$

Immediately, we can extend our inference scheme, and the respective soundness and complete results, for desugared programs by adding the following rule to Figure 4.2.

$$\text{(Mono-Rec)} \quad \frac{E, \Gamma.f : t_1, e \vdash_{Cons} (C \mid t_2) \quad t_1 \text{ fresh}}{E, \Gamma, (\mathsf{rec} f \ \mathsf{in}\ e)_l \vdash_{Cons} (C, (t_1 = t_2)_l \mid t_1)}$$

The original purpose of Γ is to keep track of lambda-bound variables, but there

is no harm if we also allow monomorphic recursive functions to be recorded in Γ.

In our actual implementation, we do something slightly different. We perform inference directly on the original recursive definition rather than the 'rec-desugared' program. For instance, inference for

$$\mathsf{let}\ fac = (\lambda n.\mathsf{if}\ 0 == n\ \mathsf{then}\ 1\ \mathsf{else}\ fac\ (n-1) * n)\ \mathsf{in}\ fac\ 4$$

yields the (simplified) constraint $fac(Int \to t', \langle\rangle)$ and type t' under the (simplified) CHR

$$fac(t, x) \iff t = Int \to t_1, t_1 = Int, t_2 = Int, fac(Int \to t_2, \langle Int \rangle)$$

The advantage is that no new rule needs to be added. The disadvantage is that we may create cycles among CHRs which leads to non-termination. For the above example, we find that

$$\begin{array}{ll} & fac(Int \to t', \langle\rangle) \\ \longrightarrow & Int \to t' = t, t = Int \to t_1, t_1 = Int, t_2 = Int, fac(Int \to t_2, \langle Int \rangle) \\ \longrightarrow & Int \to t' = t, t = Int \to t_1, t_1 = Int, t_2 = Int, \\ & Int \to t_2 = t'', t'' = Int \to t_1'', t_1'' = Int, t_2'' = Int, fac(Int \to t_2'', \langle Int \rangle) \\ \ldots & \end{array}$$

Obviously, when solving constraints we still need to enforce the condition that recursive functions must be of monomorphic type. The trick is to break cycles by equating types when encountering calls to recursive predicates. For our running example, we find

$$\begin{array}{ll} & fac(Int \to t', \langle\rangle) \\ \longrightarrow & Int \to t' = t, t = Int \to t_1, t_1 = Int, t_2 = Int, \underline{fac(Int \to t_2, \langle Int \rangle)} \\ \longrightarrow_{Mono-Rec} & Int \to t' = t, t = Int \to t_1, t_1 = Int, t_2 = Int, \underline{Int \to t_2 = Int \to t'} \end{array}$$

We have underlined the recursive constraint in the second store, as well as the equation we replaced it with, in the third store.

We need to be careful not to be overly aggressive when breaking cycles.

Example 29 Consider the following expression

```
let f = f1 in (f2 'a', f3 True)
```

We have annotated the different use sites of `f` with explicit location numbers. Inference for the let-body requires solving the constraint

$$f(Char \to t_1, \langle\rangle)_2, f(Bool \to t_2, \langle\rangle)_3, t = (t_1, t_2)$$

under the (slightly simplified) CHR

$$f(t, x) \iff f(t, x)_1$$

We find that

$$\begin{array}{ll} & f(Char \to t_1, \langle\rangle)_2, f(Bool \to t_2, \langle\rangle)_3, t = (t_1, t_2) \\ \longrightarrow & \underline{f(Char \to t_1, \langle\rangle)_{[2,1]}}, f(Bool \to t_2, \langle\rangle)_3, t = (t_1, t_2) \\ \longrightarrow_{Mono-Rec} & \underline{\underline{Char \to t_1 = Bool \to t_2}}, f(Bool \to t_2, \langle\rangle)_3, t = (t_1, t_2) \end{array}$$

In the first step, CHR application only results in propagation of justifications. In the second step, we break the cycle between $f_3(Bool \to t_2, \langle\rangle\rangle$ and $f_{2,1}(Char \to t_1, \langle\rangle\rangle$, by equating their types, which results in a type error. We would expect the expression to be type correct, since outside of its definition, `f` is polymorphic.

The obvious conclusion is that we should not equate the types of arbitrary instances of the same function. Indeed, if we had performed the following derivation instead, the resulting constraints would be satisfiable.

$$\begin{array}{ll} & f(Char \to t_1, \langle\rangle)_2, f(Bool \to t_2, \langle\rangle)_3, t = (t_1, t_2) \\ \longrightarrow & \underline{f(Char \to t_1, \langle\rangle)_{[2,1]}}, f(Bool \to t_2, \langle\rangle)_3, t = (t_1, t_2) \\ \longrightarrow_{Mono-Rec} & \underline{\underline{Char \to t_1 = Char \to t_1}}, f(Bool \to t_2, \langle\rangle)_3, t = (t_1, t_2) \\ \longrightarrow & Char \to t_1 = Char \to t_1, \underline{f(Bool \to t_2, \langle\rangle)_3}, t = (t_1, t_2) \\ \longrightarrow_{Mono-Rec} & Char \to t_1 = Char \to t_1, \underline{\underline{Bool \to t_2 = Bool \to t_2}}, t = (t_1, t_2) \end{array}$$

□

In detail, we assume that set R records all predicate symbols belonging to recursive functions. This set can be obtained by a simple dependency analysis. A *monomorphic-cycle removal derivation step* is defined as follows: Let $f(t, x)_J \in C$ and $f(t', x')_{J'} \in C'$ and given a derivation $... \longrightarrow C \longrightarrow ... \longrightarrow C'$ such that $f \in R$ and J is a prefix in J', then $... \longrightarrow C \longrightarrow ... \longrightarrow C' \longrightarrow_{Mono-Rec} (C' - f(t', x')_{J'}) \cup \{(t = t')_{J'}\}$.

We assume that any monomorphic-cycle removal steps are applied before sim-

plification and propagation steps (see Section 3.2). Note that it is not necessary to equate the x and x' component.[7] It is, however, necessary to justify the cycle-breaking equation.

Example 30 Consider the following simple variation of the program above. Here, `f` is used monomorphically at all call sites.

$\texttt{f}_1\ \texttt{x} = ((\texttt{f}_2\ \texttt{'a'}_3)_4, (\texttt{f}_5\ \texttt{True}_6)_7)$

To infer `f`'s type we must essentially solve the constraint $f(t)_8$ under the CHR

$$f(t_1, x) \iff (t_1 = t_x \to Bool)_1, f(t_2, \langle t_x \rangle)_2, (t_2 = t_3 \to t_4)_4, (t_3 = Char)_3, \\ f(t_5, \langle t_x \rangle)_5, (t_5 = t_6 \to t_7)_7, (t_6 = Bool)_6, x = r$$

We proceed as follows

$$\begin{array}{ll} & f(t, \langle\rangle)_8 \\ \longrightarrow & t = t_1, x = \langle\rangle, (t_1 = t_x \to Bool)_{[8,1]}, \underline{f(t_2, \langle t_x \rangle)_{[8,2]}}, (t_2 = t_3 \to t_4)_{[8,4]}, \\ & (t_3 = Char)_{[8,3]}, f(t_5, \langle t_x \rangle)_{[8,5]}, (t_5 = t_6 \to t_7)_{[8,7]}, (t_6 = Bool)_{[8,6]}, x = r \\ \longrightarrow_{Mono-Rec} & t = t_1, x = \langle\rangle, (t_1 = t_x \to Bool)_{[8,1]}, \underline{\underline{(t = t_2)_{[8,2]}}}, (t_2 = t_3 \to t_4)_{[8,4]}, \\ & (t_3 = Char)_{[8,3]}, \underline{f(t_5, \langle t_x \rangle)_{[8,5]}}, (t_5 = t_6 \to t_7)_{[8,7]}, (t_6 = Bool)_{[8,6]}, x = r \\ \longrightarrow_{Mono-Rec} & t = t_1, x = \langle\rangle, (t_1 = t_x \to Bool)_{[8,1]}, (t = t_2)_{[8,2]}, (t_2 = t_3 \to t_4)_{[8,4]}, \\ & (t_3 = Char)_{[8,3]}, \underline{\underline{(t = t_5)_{[8,5]}}}, (t_5 = t_6 \to t_7)_{[8,7]}, (t_6 = Bool)_{[8,6]}, x = r \end{array}$$

The resulting constraints are unsatisfiable, with the following single minimal unsatisfiable subset.

$$(t = t_2)_{[8,2]}, (t_2 = t_3 \to t_4)_{[8,4]}, (t_3 = Char)_{[8,3]}, \\ (t = t_5)_{[8,5]}, (t_5 = t_6 \to t_7)_{[8,7]}, (t_6 = Bool)_{[8,6]}$$

If we had not justified the 'cycle-breaking' equations, we would have no reference to locations 2 and 5, which are necessary for any explanation of this result. □

4.3.2 Type annotations

We will now add explicit type annotations to our language. We couple type annotations with let-definitions. As usual, we assume that type annotations are

[7] Equating the x and x' components would result in an unsatisfiable constraint if they happen to be of different lengths. e.g. for expression `\x -> let f = f1 in \z -> f2`, we get $f(t_1, \langle t_x \rangle)_1$, and $f(t_2, \langle t_x, t_z \rangle)_2$

labelled with a unique number. The additional typing rule is standard.

$$\text{(LetA)}\quad \frac{C, \Gamma.f : \sigma \vdash e : \sigma \quad C, \Gamma.f : \sigma \vdash e' : t' \quad fv(\sigma) = \emptyset}{C, \Gamma \vdash \mathsf{let}\ \begin{array}{l}(f :: \sigma)_l \\ f = e\end{array}\ \mathsf{in}\ e' : t'}$$

In our formulation we allow for polymorphic recursion [35]. A let-bound function may be defined recursively and have a polymorphic type as long as the type is explicitly provided. As in Haskell, our type annotations are *closed*,[8] i.e. all variables in a declared type are implicitly universally quantified. In examples, when we write $f :: C \Rightarrow t$, this is a short-hand for $f :: \forall \bar{a}.C \Rightarrow t$ where $\bar{a} = fv(C, t)$.

We can straightforwardly extend our translation scheme to CHRs. For constraint generation we add the following two rules to Figure 4.2.

$$\text{(VarA-f)}\quad \frac{\begin{array}{c}f_a \in E \quad \Gamma = \{\overline{x : t_x}\} \quad t, t_l \text{ fresh} \\ C = \{f_a(t, x)_l \wedge x = \langle \overline{t_x} \rangle \wedge (t_l = t)_l\}\end{array}}{E, \Gamma, f_l \vdash_{Cons} (C, \mathbb{I}\, t)}$$

$$\text{(LetA)}\quad \frac{E \cup \{f_a\}, e_2 \vdash_{Cons} (C \,\mathbb{I}\, t)}{E, \Gamma, \mathsf{let}\ \begin{array}{l}(f :: C' \Rightarrow t')_l \\ f = e_1\end{array}\ \mathsf{in}\ e_2 \vdash_{Cons} (C \,\mathbb{I}\, t)}$$

Our assumption is that the annotated type of f will be represented by its own separate CHR. We adopt the convention that f_a refers to the predicate symbol capturing the valid relations for the annotated type of f whereas predicate symbol f represents the inferred type of f. For rule generation we add the following rule

[8] In Chapter 8 we will look at open type annotations, where type variables scope over nested annotations.

to Figure 4.3.

$$\text{(LetA)}\quad \frac{\begin{array}{c} E \cup \{f_a\}, \Gamma, e_1 \vdash_{Cons} (C'' \mid t'') \quad \Gamma = \{x_1 : t_1, \ldots, x_n : t_n\} \quad t, x, r \text{ fresh} \\ E \cup \{f_a\}, \Gamma, e_1 \vdash_{Def} P_1 \quad E \cup \{f_a\}, \Gamma, e_2 \vdash_{Def} P_2 \\ P = P_1 \cup P_2 \cup \left\{ \begin{array}{l} f_a(t,x) \Longleftrightarrow t = t_l, (t_l = t')_l, (C')_l, \\ f(t,x) \Longleftrightarrow f_a(t,x), C'', t = t'', x = \langle t_1, \ldots . t_n | r \rangle \end{array} \right\} \end{array}}{E, \Gamma, \mathsf{let} \begin{array}{l} (f :: C' \Rightarrow t')_l \\ f = e_1 \end{array} \mathsf{\ in\ } e_2 \vdash_{Def} P}$$

Note that we generate two CHR rules instead of one. We refer to the first rule as the *annotation* CHR of f and to the second rule as f's *inference* CHR. Note that on the right hand side of the inference CHR we include a call to the annotation CHR. This is common practice. When typing an expression we wish to make use of any additional type information in the annotation. Also note that we may make use of f_a (representing the annotated type) in the let-definition itself and the body of the let-statement. Hence, we will not create any cyclic CHRs for recursive functions which are type annotated.

Example 31 Consider the following program. Here `const` is the standard Haskell function of the same name, with type $a \to b \to a$.

```
f x = const True (f True, f 'a')
```

From this we generate the (simplified) rule:

$$\begin{array}{rcl} f(t,x) & \Longleftrightarrow & const(t_k), f(t_1,x), f(t_2,x), t_k = Bool \to (t_{r1}, t_{r2}) \to t_r, \\ & & t = t_x \to t_r, t_1 = Bool \to t_{r1}, t_2 = Char \to t_{r2} \end{array}$$

It is quite clear that no normal CHR derivation involving constraint $f(t,x)$ would terminate. If we treat `f` as monomorphic and employ the *monomorphic-cycle removal step*, introduced in Section 4.3.1, then the derivation would terminate, but the resulting constraints would be unsatisfiable. In enforcing monomorphism, we would have to equate t_1 and t_2, which are non-unifiable types.

As in Haskell, polymorphic recursion is allowed in the event that a type annotation is provided. We declare `f`'s type as follows.

```
f :: a -> Bool
f x = const True (f True, f 'a')
```

Employing the newly described constraint and CHR generation rules, we would generate these simplification rules.

$$\begin{array}{rcl} f_a(t,x) & \Longleftrightarrow & t = a \to Bool \\ f(t,x) & \Longleftrightarrow & const(t_k), f_a(t,x), f_a(t_1,x), f_a(t_2,x), \\ & & t_k = Bool \to (t_{r1}, t_{r2}) \to t_r, t = t_x \to t_r, x = r, \\ & & t_1 = Bool \to t_{r1}, t_2 = Char \to t_{r2} \end{array}$$

The important thing to notice now is that the recursive calls to `f` are represented by the constraints $f_a(t_1,x), f_a(t_2,x)$, and that the f_a rule, standing for `f`'s type annotation, is not cyclic. We would now be able to successfully infer `f`'s type. □

The presence of type annotations demands an additional test. We need to check that the inferred type subsumes the annotated type. In our formulation, we will need to check that the annotation CHR entails the inference CHR w.r.t. the given set of CHRs. More precisely, consider a Hindley/Milner typing problem (Γ, e) where $P_{E_{init}}, E_{init} \sim \Gamma_{init}, \Gamma_\lambda$ and $\Gamma = \Gamma_{init} \cup \Gamma_\lambda$. As before, we first compute $E_{init}, \Gamma_\lambda, e \vdash_{Cons} (C \mid t)$ and $E_{init}, \Gamma_\lambda, e \vdash_{Def} P$. For each pair (f, f_a) of inference and annotation CHR in P we check the *subsumption condition* that $P_{init}, P \models f_a(t,x) \leftrightarrow f(t,x)$. Since the inference CHR includes the annotation CHR, $f_a(t,x) \supset f(t,x)$ is equivalent to $f_a(t,x) \leftrightarrow f(t,x)$. In the subsumption condition we assume that P_{init} and P refer to the logical reading of CHRs. The subsumption condition holds iff (1) we execute $f_a(t,x) \longrightarrow^*_{P_{init},P} C_1$ and $f(t,x) \longrightarrow^*_{P_{init},P} C_2$, and (2) we have that $\models (\bar{\exists}_{t,l}.C_1) \leftrightarrow (\bar{\exists}_{t,l}.C_2)$. In Chapter 5, we will discuss how to assist the user in case the subsumption check fails.

Example 32 We return to the annotated program, and CHRs, of Example 31. Running $f_a(t,x) \longrightarrow^* t = a \to Bool$, and $f(t,x) \longrightarrow^* t = t_x \to Bool, C$ (where C contains only Herbrand constraints,) we find that indeed $\models (\bar{\exists}_{t,l}.t = a \to Bool) \leftrightarrow (\bar{\exists}_{t,l}.t = t_x \to Bool, C)$. Hence, the type we have declared for `f` is correct. □

We can state that type inference in the presence of type annotations is sound and complete for standard Hindley/Milner. By 'standard Hindley/Milner' we refer to the system where equations are the only primitive constraints. Note that the completeness result does not extend to more complicated type extensions such

as type-class overloading [21, 80]. Please refer to Appendix A.2 for more details, formal results and proofs.

4.3.3 Type class overloading

We will now add class and instance declarations to our language. The programs we admit allow for any number of such declarations appearing at the top-level. We leave out instance bodies for simplicity. Sometimes we will also omit class methods as well, when they are not important. Also note that the | symbol in the first line is part of the syntax of class declarations.

$$
\begin{array}{llll}
\text{Declaration} & d & ::= & \mathsf{class}\ (Ctx \Rightarrow (U\ t_1\ \ldots\ t_n))_{l_1} \mid fd_1, ..., fd_n \\
 & & & \quad \mathsf{where}\ f :: (C \Rightarrow t)_{l_2} \\
 & & & \mid \mathsf{instance}\ Ctx \Rightarrow (U\ t_1\ \ldots\ t_n)_{l_0} \\
\text{Context} & Ctx & ::= & U_1\ \overline{t_1}, ..., U_n\ \overline{t_n} \\
\text{FD} & fd & ::= & \bar{a} \text{->}_l\ \bar{a}
\end{array}
$$

We require that only type class constraints appear in the context of a class or instance declaration. Our description supports multi-parameter type classes [44] and functional dependencies (FDs) [43]. Their meaning will become clear in the translation to CHRs. For simplicity, we omit the description of typing rules. We refer the interested reader to Stuckey, Sulzmann, et al. [82, 16] for a detailed treatment.

Class and instance declarations are converted into CHRs as follows The class declaration

$$\mathsf{class}\ (Ctx \Rightarrow (U\ t_1\ \ldots\ t_n))_{l_1} \mid fd_1, ..., fd_n\ \mathsf{where}\ f :: (C \Rightarrow t)_{l_2}$$

creates the following *super class* and *method* CHR rules, respectively.

$$
\begin{array}{l}
U\ t_1\ \ldots\ t_n \Longrightarrow Ctx_{l_1} \\
f(t_{l_2}) \Longleftrightarrow t = t_{l_2}, C_{l_2}, (U\ t_1\ \ldots\ t_n)_{l_2}
\end{array}
$$

And for each functional dependency $fd_i \equiv a_{i_1} \ldots a_{i_k} \text{->}_l\ a_{i_0}$ the *FD* rule

$$U\ t_1\ \ldots\ t_n, U\ s_1\ \ldots\ s_n \Longrightarrow (t_{i_0} = s_{i_0})_l$$

where $s_j = t_j$ if $j = a_{i_o}$ for some $1 \leq o \leq k$, otherwise s_j is a new variable.

From each instance declaration `instance` $Ctx \Rightarrow (U\ t_1\ \ldots\ t_n)_{l_0}$, we create the *instance* CHR rule

$$U\ t_1\ \ldots\ t_n \iff Ctx_{l_0}$$

and for each functional dependency for the class definition $fd_i \equiv a_{i_1} \ldots a_{i_k}$`->`$_l\ a_{i_0}$ an *instance improvement* CHR rule

$$U\ s_1\ \ldots\ s_n \Longrightarrow (t_{i_0} = s_{i_0})_{[l_0,l]}$$

where (again) $s_j = t_j$ if $j = a_{i_o}$ for some $1 \leq o \leq k$ and otherwise s_j is a new variable. We often refer to the FD and instance improvement CHRs as *improvement* rules.

Here are some example translations.

Example 33 Given the class and instance declarations below,

```
class (Eq a)1 where
    (==) ::  (a -> a -> Bool)2
class (Eq a => Ord a)3 where
    (>) ::  (a -> a -> Bool)4
instance (Ord a => Ord [a])5
instance (Ord Bool)6
```

we generate the following CHRs

$$\begin{array}{rcl}
Eq\ a & \Longrightarrow & True_1 \\
(==)(t_2) & \iff & (t_2 = a \to a \to Bool)_2, (Eq\ a)_2 \\
Ord\ a & \Longrightarrow & (Eq\ a)_3 \\
(>)(t_4) & \iff & (t_4 = a \to a \to Bool)_4, (Ord\ a)_4 \\
Ord\ [a] & \iff & (Ord\ a)_5 \\
Ord\ Bool & \iff & True_6
\end{array}$$

The super-class relation encoded by the third rule states that each occurrence of $Ord\ a$ implies $(Eq\ a)_3$. The right-hand sides of all CHR rules generated are justified, so we can keep track of which rules were involved, and their original source locations, when inspecting justifications attached to constraints. □

Example 34 For the following declarations,

```
class (Collects ce e)1 | ce ->2 e where
    empty :: ce3
    insert :: (e -> ce -> ce)4

instance (Collects [a] a)5
```

we produce the rules below.

$$\begin{array}{rcl}
\mathit{Collects}\ ce\ e & \Longrightarrow & \mathit{True}_1 \\
\mathit{Collects}\ ce\ e,\ \mathit{Collects}\ ce\ e' & \Longrightarrow & (e = e')_2 \\
\mathit{empty}(t) & \Longleftrightarrow & (t = ce)_3, (\mathit{Collects}\ ce\ e)_3 \\
\mathit{insert}(t) & \Longleftrightarrow & (t = e \rightarrow ce \rightarrow ce)_4, (\mathit{Collects}\ ce\ e)_4 \\
\mathit{Collects}\ [a]\ a & \Longleftrightarrow & \mathit{True}_5 \\
\mathit{Collects}\ [a]\ b & \Longrightarrow & b = a_{[5,2]}
\end{array}$$

The interesting bit here is our use of CHRs to encode the improvement rules implied by functional dependencies. □

Our assumptions are that CHRs generated from class and instance declarations are "consistent" and terminating. Consistency is expressed as confluence [82]. Fortunately, the conditions imposed on Haskell type classes guarantee these two important properties. An in-depth discussion can be found in [16].

Type inference for type classes additionally demands that all type schemes (whether annotated or inferred) are "unambiguous". The need for unambiguity is related to the way in which type classes are translated during compilation.[9] We refer to Jones [42, 82] for details. Generally speaking, a type scheme is ambiguous if grounding all variables in the type component does not ground all variables in the constraint component.

More formally, we consider a function f and set of CHRs P such that $f(t, x) \longrightarrow^*_P C$, and therefore $f :: \forall\bar{\alpha}.C \Rightarrow t$, where $\phi = mgu(C_e)$, $\bar{\alpha} = fv(C) - fv(\phi(x))$. Let ρ be a renaming on $\bar{\alpha}$, and $C \wedge \rho C \wedge t = \rho t \longrightarrow^*_P D$. If $\models D \supset \alpha = \rho\alpha$ for all $\alpha \in \bar{\alpha}$ then f is *unambiguous*. We say that f is ambiguous in any α which fails this test.

This is a more complicated ambiguity test than is necessary in Haskell 98, which simply requires that for $\forall\bar{\alpha}.C \Rightarrow t$, it is the case that $(fv(C) \cap \bar{\alpha}) - fv(t) = \emptyset$, i.e. all universal variables in the constraint context appear in the type component.

[9] Intuitively, if a constraint is ambiguous, we have no way to consistently decide which instance will be required.

The reason is that with the addition of functional dependencies, it is possible that the instantiation of one variable can determine another, and so that other variable need not appear in the type component. For example, given a class declaration class $C\ a\ b \mid a\texttt{->}b$, the type scheme $\forall a, b.C\ a\ b \Rightarrow a \rightarrow Bool$, is not ambiguous at all since if we know a, then by the dependency, we must also know b.

We note that in reality each method in an instance declaration would require an ambiguity test (and subsumption test). Since we have omitted instance bodies from our source language, to simplify matters, we will overlook these checks here.

We now take a look at a number of examples to get a feel for CHR-based type class inference.

Example 35 Consider the following (slightly desugared) program, together with the class and instance declarations of Example 33.

```
(lteq x1 y2 = (not7 (((>3) x4)5 y6)8)9)10
not ::  Bool -> Bool16
```

where `>` is a member of the `Ord` class, as in Haskell. The translation process yields the following (again for simplicity we ignore the λ-bound variable argument):

$$\begin{array}{rcl} lteq(t_{10}) & \Longleftrightarrow & (t_1 = t_x)_1, (t_2 = t_y)_2, gt(t_3,)_3, (t_4 = t_x)_4, \\ & & (t_3 = t_4 \rightarrow t_5)_5, (t_6 = t_y)_6, not(t_7)_7, \\ & & (t_5 = t_6 \rightarrow t_8)_8, (t_7 = t_8 \rightarrow t_9)_9, \\ & & (t_{10} = t_1 \rightarrow t_2 \rightarrow t_9)_{10} \\ not(t_{16}) & \Longleftrightarrow & (t_{16} = Bool \rightarrow Bool)_{16} \end{array}$$

These rules are generated directly from the program text.

Type inference for $\texttt{(lteq}_{22}\ \texttt{[w}_{17}\texttt{]}_{18}\texttt{)}_{23}$ generates the constraints

$$(t_{17} = t_w)_{17}, (t_{18} = [t_{17}])_{18}, lteq(t_{22})_{22}, (t_{22} = t_{18} \rightarrow t_{23})_{23}$$

This is the initial constraint which we run the CHR program on. For brevity, we show the whole derivation in a simplified form, just showing $\theta(C_u) \wedge t_{23} = \theta(t_{23})$ where θ is the mgu of C_e for C at each step, and omit justifications. That is, we only show the user-defined constraints and the top type variable t_{23}, under the

effect of the equations in C ignoring justifications.

$$\begin{array}{rl} & lteq([t_w] \to t_{23}), t_{23} = t_{23} \\ \longrightarrow_{lteq} & not(t_8 \to t_9), gt([t_w] \to t_2 \to t_8), (t_{23} = t_2 \to t_9) \\ \longrightarrow_{not} & gt([t_w] \to t_2 \to Bool), t_{23} = t_2 \to Bool \\ \longrightarrow_{gt} & Ord\ [t_w], t_{23} = [t_w] \to Bool \\ \longrightarrow_{Ord\ [a]} & Ord\ t_w, t_{23} = [t_w] \to Bool \\ \longrightarrow_{Ord\ a} & Ord\ t_w, Eq\ t_w, t_{23} = [t_w] \to Bool \end{array}$$

In other words the type inferred for the original expression is `(Ord a, Eq a) => [a] -> Bool`. Note that we are more "verbose" than Hugs or GHC which would report `Ord a => [a] -> Bool`. Clearly, the constraint `Eq a` is "redundant", since every instance of `Ord` must be an instance of `Eq` as specified by the class declaration for `Ord`. In [82], we show how to remove such redundant constraints. However, for type error reporting purposes it is desirable to keep all constraints for more thorough type explanations. □

4.4 Summary

In this chapter we returned to the Hindley/Milner type system, and extended it first with constraints and algebraic data types. We described a type inference procedure for this language, which involves generating constraints and CHR rules out of a source program, and performing constraint solving. We also briefly demonstrated the application of one of the constraint reasoning algorithms of Chapter 3 in the explanation of an inference result. We then extended our language, and inference system to include monomorphic recursion, explicit type annotations and type classes.

Chapter 5

Handling type errors

In this chapter we address the problem of reporting type errors, in the event that any of the checks introduced in the previous chapters has failed. We first look at a simple scheme for reporting ill-typed programs in terms of the source locations which together give rise to the type error (Section 5.1). Then we address the problem of reporting failure of other type-related checks, specifically subsumption (Section 5.2) and ambiguity (Section 5.3). In Haskell there is an additional requirement that types appearing within inferred type class constraints must be of some limited form. Constraints which do not meet this requirement cause a 'missing instance' error. We formulate this condition in terms of our inference system and address the error reporting issues in Section 5.4.

We are not aware of any other system that can report subsumption, ambiguity and missing instance errors with the same degree of accuracy as ours. A general advantage of all of our error reporting procedures is their completeness. Whenever we report an error, we always indicate all of the program locations which contributed. This is in contrast to the more conventional style of error reporting supported by algorithm $\mathcal{W}$-style systems, where errors are always reported as a problem at a single program location.

5.1 Satisfiability errors

A program is ill-typed in our system if the constraints on its type are unsatisfiable. Before such a program can be run it must be modified, but obviously any such modification needs to actually fix the problem at hand. Our task then is to report the type error in such a way that the programmer is directed towards the locations

in the source code which are potentially the source of the error. Modifying those locations appropriately should fix the program.

Although this seems obvious, conventional type inference algorithms, like algorithm $\mathcal{W}$, do not report errors in this way. Instead, they only indicate a single location in the source program at which the problem was first detected.

Example 36 Consider the following ill-typed Haskell program. As in Haskell, the `length` function has type $\forall a.[a] \rightarrow Int$. The mistake here is that `sumLengths` ought to return `0` rather than `[]`.

```
sumLengths [] = []
sumLengths (xs:xss) = length xs + sumLengths xss
```

GHC reports the following error for this program.

```
sumLengths.hs:2:
    Couldn't match 'Int' against '[a]'
        Expected type:  Int
        Inferred type:  [a]
    In the application 'sumLengths xs'
    In the second argument of '(+)', namely 'sumLengths xs'
```

This tells us that there is a problem in the application of `sumLengths` to `xs`, and that it was expected to have type *Int*, but was actually found to have type $[a]$. What this error message does not mention is why it expected an *Int*, or why it inferred $[a]$. Moreover, the location of the reported error is far removed from the actual mistake in the program; changing that application would not fix the program at all (assuming we have the obvious, intended meaning of the program in mind.)

□

We now look at an extremely simple scheme which takes advantage of justified constraints in order to find all of the locations in a source text that are potentially the source of a type error mistake. Note that this method alone is inadequate for producing meaningful type error messages for programmer consideration, but it does form the basis of the more refined scheme we will introduce in Chapter 6.

Assume we are interested in finding the type of a function `f`, with respect to CHRs P. We run a CHR derivation $f(t, x) \longrightarrow_P^* C$, and find that C is

unsatisfiable, i.e. `f` is ill-typed. We then look for M, a single minimal subset of C, employing the algorithm of Section 3.3, i.e. $M = \mathsf{min\_unsat}(C)$. Such a set represents the smallest collection of program locations which caused that type error. Taking the justifications of the constraints in M, we immediately have the program locations which led to the type error, i.e. the locations of interest are simply $\mathit{just}(M)$.

Example 37 We return to the program of Example 36, which we repeat below in a slightly desugared form (`+` is applied in a prefix position), and annotated with location numbers.

$$\left(\begin{array}{l}\texttt{sumLengths []}_2\texttt{ = []}_3 \\ \texttt{sumLengths (xs}_5\texttt{:}_4\texttt{xss}_6\texttt{)}_7\texttt{ = (((+)}_8\texttt{ (length}_9\texttt{ xs}_{10}\texttt{)}_{11}\texttt{)}_{12} \\ \qquad\qquad\qquad\qquad\texttt{(sumLengths}_{13}\texttt{ xss}_{14}\texttt{)}_{15}\texttt{)}_{16}\end{array}\right)_1$$

For brevity, we will omit the CHRs we would generate from the above program, as well as the (trivial) derivation, and instead present the final justified typing constraints. We will often do this when the CHR derivation would only consist of a single step or two, and the result is just the union of the constraints in the rule bodies.

$(t_1 = t_r)_1, (t_a = t_e \to t_r), (t_b = t_e \to t_r), (t_a = t_2 \to t_3), (t_2 = [a])_2, (t_3 = [b])_3,$
$(t_b = t_7 \to t_{16}), (t_4 = c \to [c] \to [c])_4, (t_4 = t_5 \to t_6 \to t_7)_4, (t_8 = d \to d \to d)_8,$
$(Num\ d)_8, (t_8 = t_{11} \to t'_{12})_8, (t_{12} = t'_{12})_{12}, (t_9 = [e] \to Int)_9, (t_9 = t_{10} \to t'_{11})_9,$
$(t_{11} = t'_{11})_{11}, (t_{10} = t_5)_{10}, (t_{13} = t_b)_{13}, (t_{13} = t_{14} \to t'_{15})_{15}, (t_{15} = t'_{15})_{15},$
$(t_{14} = t_6)_{14}, (t_{12} = t_{15} \to t'_{16})_{16}, (t_{16} = t'_{16})_{16}$

This set of constraints is unsatisfiable. A minimal unsatisfiable subset of these constraints is below.

$(t_a = t_e \to t_r), (t_b = t_e \to t_r), (t_a = t_2 \to t_3), (t_3 = [b])_3, (t_b = t_7 \to t_{16}),$
$(t_8 = t_{11} \to t'_{12})_8, (t_8 = d \to d \to d)_8, (t_{12} = t'_{12})_{12}, (t_{12} = t_{15} \to t'_{16})_{16},$
$(t_9 = [e] \to Int)_9, (t_9 = t_{10} \to t'_{11})_9, (t_{11} = t'_{11})_{11},$

Removing any one of these constraints would make the set satisfiable. Correspondingly, fixing any of the locations in the justifications above, such that the associated constraint is not generated, would also solve the problem. The locations in question are 3,8,9,11,12,16.

We could report the type error by highlighting those locations in the source program like so. Since there are no explicit tokens in the source which represent application sites, we cannot highlight locations 11,12 or 16. We leave it up to the user to understand that whenever we highlight a token in a function position we may also be referring to its application. We highlight the original, non-desugared program as follows.

```
sumLengths [] = []
sumLengths (xs:xss) = length xs + sumLengths xss
```

Such an error report does not attempt to explain *why* there is a type error, or even to try to guess which of the candidate locations is the actual source of the problem. Its great advantage is that it clearly indicates to the programmer the *precise* set of sites potentially in error. Most significantly, the real site of the mistake, here at location 3, will always be highlighted. By comparison, GHC's error singled out location 15 alone. □

Example 38 The following program contains a type error that involves a functional dependency. The functional dependency, combined with the instance enforce the condition that both arguments of the C type class constraint must always be the same.

```
class C a b | a -> b where c ::  a -> b -> a
instance C a a
e = c (\x -> x) 'a'
```

Again, we desugar it slightly, making application sites more explicit, and labelling all relevant locations. The result is the following.

```
class (C a b)1 | a ->2 b where (c ::  a -> b -> a)3
instance (C a a)4
(e = ((c9 (\x11 -> x12)10)7 'a'8)6)5
```

From this program, we would generate the following CHR rules. We have omitted the x parameter (which normally keeps track of λ-bound variables in

scope), to simplify the presentation, since there are no nested definitions anyway.

$$
\begin{array}{lrcl}
(r1) & C\ a\ b & \Longrightarrow & \mathit{True} \\
(r2) & C\ a\ b, C\ a\ c & \Longrightarrow & (b = c)_2 \\
(r3) & c(t) & \Longleftrightarrow & (C\ a\ b)_3, (t = a \to b \to a)_3 \\
(r4) & C\ a\ b & \Longrightarrow & (b = a)_{[4,2]} \\
(r5) & C\ a\ a & \Longleftrightarrow & \mathit{True}_4 \\
(r6) & e(t) & \Longleftrightarrow & (t_5 = t_6)_5, (t_7 = t_8 \to t'_6)_6, (t_6 = t'_6)_6, \\
 & & & (t_8 = \mathit{Char})_8, (t_9 = t_{10} \to t'_7)_7, (t'_7 = t_7)_7, c(t_9)_9, \\
 & & & (t_{10} = t'_{11} \to t_{12})_{10}, (t_{11} = t'_{11}), (t_{12} = t_{11})_{12}
\end{array}
$$

To infer `e`'s type, we perform the following CHR derivation. The initial constraint is unjustified (or equivalently, has an empty justification []). We underline each constraint that causes the following derivation step to occur, and only rename variables whenever it is required to avoid a name clash.

$$
\begin{array}{ll}
 & \underline{e(t)} \\
\longrightarrow_{(r6)} & (t_5 = t_6)_5, (t_7 = t_8 \to t'_6)_6, (t_6 = t'_6)_6, (t_8 = \mathit{Char})_8, (t_9 = t_{10} \to t'_7)_7, \\
 & (t'_7 = t_7)_7, \underline{c(t_9)_9}, (t_{10} = t'_{11} \to t_{12})_{10}, (t_{11} = t'_{11}), (t_{12} = t_{11})_{12} \\
\longrightarrow_{(r3)} & (t_5 = t_6)_5, (t_7 = t_8 \to t'_6)_6, (t_6 = t'_6)_6, (t_8 = \mathit{Char})_8, (t_9 = t_{10} \to t'_7)_7, \\
 & (t'_7 = t_7)_7, (t_{10} = t'_{11} \to t_{12})_{10}, (t_{11} = t'_{11}), (t_{12} = t_{11})_{12}, \\
 & t_9 = t', \underline{(C\ a\ b)_{[9,3]}}, (t' = a \to b \to a)_{[9,3]} \\
\longrightarrow_{(r4)} & (t_5 = t_6)_5, (t_7 = t_8 \to t'_6)_6, (t_6 = t'_6)_6, (t_8 = \mathit{Char})_8, (t_9 = t_{10} \to t'_7)_7, \\
 & (t'_7 = t_7)_7, (t_{10} = t'_{11} \to t_{12})_{10}, (t_{11} = t'_{11}), (t_{12} = t_{11})_{12}, t_9 = t', \\
 & (C\ a\ b)_{[9,3]}, (t' = a \to b \to a)_{[9,3]}, (b = a)_{[9,3,4,2]}
\end{array}
$$

We stop here, as the final store above is unsatisfiable. There is only one minimal unsatisfiable subset, which is as follows:

$$
\begin{array}{l}
(t_7 = t_8 \to t'_6)_6, (t_8 = \mathit{Char})_8, (t_9 = t_{10} \to t'_7)_7, (t'_7 = t_7)_7, \\
(t_{10} = \mathit{Bool})_{10}, t_9 = t', (t' = a \to b \to a)_{[9,3]}, (b = a)_{[9,3,4,2]}
\end{array}
$$

By collecting the justifications from those constraints, we see that locations 2,3,4,6,7,8,9 and 10 were involved. We can report this error by highlighting those locations in the source program as follows.

```
class C a b | a -> b where c ::  a -> b -> a
instance C a a
```

```
e = c (\x -> x) 'a'
```

This indicates that the call to `c`, along with its arguments are involved. Furthermore, part of the problem lies with the combination of `C`'s functional dependency and the highlighted instance. The problem overall is that the dependency and the instance, together require that `c`'s arguments must be of the same type, which is not the case here. Importantly, nothing specific to the function passed as `c`'s first argument is highlighted, i.e. it is enough that the argument is a function at all, regardless of what it computes.

□

5.1.1 Reporting locations common to all errors

To remain efficient, we only consider a single minimal unsatisfiable subset of constraints at a time. Given the number of constraints generated during inference, calculating all minimal unsatisfiable subsets is generally not feasible. This is particularly so for the purposes of generating static text error messages, since the user may need to wait some time for the process to finish before regaining control of the terminal. In Chapter 7, we will look at the problem of finding all minimal unsatisfiable subsets relatively efficiently, and suggest some possible uses, assuming we have them all. Even with the optimisations we will discuss, the process is still too time consuming in almost all cases. As mentioned in Section 3.3.1, however, it is inexpensive to find any constraints which appear in all minimal unsatisfiable subsets.

For type error reporting purposes, finding a non-empty intersection of all minimal unsatisfiable subsets is significant, since those constraints correspond to source locations which are part of every type conflict present. These common locations are much more likely to be the actual source of the mistake.

Example 39 The following simple program is ill-typed. The functions `toUpper` and `toLower`, are the standard Haskell functions of the same names, both with type $Char \rightarrow Char$.

$$(\texttt{f}\ x_2 = (\texttt{if}_3\ x_4\ \texttt{then}_5\ (\texttt{toUpper}_6\ x_7)_8\ \texttt{else}_9\ (\texttt{toLower}_{10}\ x_{11})_{12})_1$$

Again, for brevity, we will omit the trivial CHR generation and derivation steps and go directly to the justified constraints generated from this program.[1]

$$(t_1 = t_2' \to t_3)_1, (t_2 = t_2')_2, (t_4 = \mathit{Bool})_4, (t_4 \to t' = t_5 \to t_6),$$
$$(t_F \to t' = t_5 \to t_7), (t_3 = t')_3, (t_5 = t_2)_5, (t_8 = \mathit{Char} \to \mathit{Char})_8, (t_9 = t_2)_9,$$
$$(t_8 = t_9 \to t_6')_6, (t_6 = t_6')_6, (t_{10} = \mathit{Char} \to \mathit{Char})_{10}, (t_{11} = t_2)_{11},$$
$$(t_{10} = t_{11} \to t_7')_7, (t_7 = t_7')_7$$

It is plain to see that there is a conflict between the use of `x` at type *Bool* in the conditional, and at type *Char* in both branches of the if-then-else. Hence, there are two minimal unsatisfiable subsets of the above constraints. Common to both of these are the constraints listed below.

$$(t_4 = \mathit{Bool})_4, (t_5 = t_2)_5, t_4 \to t' = t_5 \to t_7$$

This strongly suggests that the real source of the mistake in this program lies at location 4 or 5. We might report this by highlighting the source text as follows. Highlighting `toLower` and its argument instead, would also have been a possibility.

```
conflicting locations:
(f x = (if x then (toUpper x) else (toLower x))
locations which appear in all conflicts:
(f x = (if x then (toUpper x) else (toLower x))
```

Indeed, changing the expression at location 4 to something like `x > 'm'` would resolve both type conflicts, whereas changing either of the two branches would only fix one.

□

So far we have only briefly touched upon the methods we can employ to make sense of type errors represented as sets of unsatisfiable, justified constraints. Obviously, we would like to provide far more meaningful type error messages, which not only indicate the program locations in conflict, but also provide some explanation for conflicts. We address this issue in Chapter 6.

[1] Note that our translation of the if-then-else is based on desugarring it to a case expression, as described in Section 4.1

5.2 Subsumption errors

As discussed in Section 4.3.2, for each type-annotated function, we need to check that the declared type is subsumed by the function's inferred type. In other words, for the declared type to be acceptable it can be no more general than the inferred type. Whenever this check fails, we attempt to report it in such a way that the programmer is able to go directly to the source locations which are responsible for the problem.

Before we begin, we present two examples of subsumption errors.

Example 40 In the following program, the declared type is not subsumed by the inferred type.

```
f :: a -> (a, b)
f x = (x, x)
```

The (most general) inferred type of `f` is $a \rightarrow (a, a)$. In the declared type, the `b` in the tuple is too general, since it would allow `x`, which is monomorphic, to have two distinct types. □

Example 41 In this next program, the subsumption error involves a type class constraint:

```
notNull :: Eq a => [a] -> Bool
notNull xs = xs > []
```

Given the usual Haskell definition of `(>)` with type $\forall a.Ord\ a \Rightarrow a \rightarrow a \rightarrow a$, the inferred type of `notNull` is $\forall a.Ord\ a \Rightarrow [a] \rightarrow Bool$. The inferred Ord however is not subsumed by the Eq of the declared type. Indeed, `notNull` can only work for types $[a]$ where a is a member of Ord, a subclass of Eq. □

As illustrated by these examples, a subsumption error may arise in one of two ways: either the declared type component is more general than the inferred type, or the inferred user constraints cannot be matched against declared user constraints. First we consider the case where an inferred constraint is not represented by a corresponding declared constraint.

Given a set of CHRs P and function predicates $f(t, x)$ and $f_a(t, x)$ representing the inferred and declared types of a function f, we perform derivations $f(t, x) \longrightarrow^*_P C$, and $f_a(t, x) \longrightarrow^*_P C'$. The subsumption test will fail for any user

constraint $u \in C_u$ if $u \notin C'_u$.[2] Let U be the set of all such us. We restrict ourselves to reporting one unmatched constraint in U. We select the $u \in U$ with the fewest number of locations in its justification. If there are multiple candidates, we pick one arbitrarily. For constraint $u_{[...,l]}$ we can then report that constraint u arising from location l in the program is unaccounted for by the annotation.[3]

Returning to the `notNull` program of Example 41, we report the following.

```
notNull.hs:2: ERROR: Inferred type does not subsume
                     declared type
Declared: forall a. Eq a => [a] -> Bool
Inferred: forall a. Ord a => [a] -> Bool
Problem : Constraint Ord a, from following location,
          is unmatched.
          notNull :: Eq a => [a] -> Bool
          notNull xs = xs > []
```

It should be noted that GHC also seems to do well at reporting this sort of error; it appears to record the source location of each user constraint, so it can then report where any unaccounted for constraints come from.

GHC raises the following error:

```
notNull.hs:2:
    Could not deduce (Ord a) from the context (Eq a)
    arising from use of ‘>’ at notNull.hs:2
    Probable fix:
        Add (Ord a) to the type signature(s) for ‘notNull’
    In the definition of ‘notNull’:  notNull xs = xs > []
```

Other Haskell systems such as Hugs and Nhc98 report the error without identifying the program locations responsible.

We now consider the case where the subsumption test fails because the declared type is too polymorphic. The aim is to find any type variables in the declaration, which correspond to more-instantiated types in the inferred type.

Given a set of CHRs P and function predicates $f(t, x)$ and $f_a(t', x')$ representing the inferred and declared types of a function f, we perform derivations

[2]Remember that the CHR for $f(t, x)$ will contain a call to $f_a(t, x)$ (see Section 4.3.2), so we are looking for any user constraints implied by the combined inferred and declared types, which are not already present in the declared type.

[3]Recall that, as described in Section 3.2, justifications are always extended at the front of the list. Therefore, the last location number will always refer to the constraint's original source location.

$f(t,x) \longrightarrow_P^* C$, and $f_a(t',x') \longrightarrow_P^* C'$. Let $C'' \equiv C' \wedge C \wedge t = \alpha \wedge t' = \alpha$ so that $C'' \longrightarrow^* D$ and $\phi_1 = mgu(C'_e)$ and $\phi_2 = mgu(D_e)$. The subsumption test will fail if $\phi_1(t') \neq \phi_2(t')$ which can only occur when $\phi_2(t')$ is more specialised than $\phi_1(t')$ (since D includes the constraints C'_e). We select an $\alpha \in fv(t')$ such that $\phi_1(\alpha) \neq \phi_2(\alpha)$. Such an α is an annotation variable which is instantiated by the type of the program body. To explain this unexpected instantiation, we find a minimal subset D_2 of C'' such that $\models D_2 \rightarrow \alpha = \phi_2(\alpha)$. We make use of the `min_impl` algorithm described in Section 3.4 to find D_2, i.e. $D_2 = \mathsf{min\_impl}(C'', (\alpha = \phi_2\alpha))$.

Example 42 The following program contains a subsumption error.

```
notEq :: Eq a => a -> b -> Bool
notEq x y = if x == y then False else True
```

Attempting to compile this with GHC, the following error message is issued:

```
notEq.hs:1:
  Inferred type is less polymorphic than expected
    Quantified type variable ‘b’ is unified with
    another quantified type variable ‘a’
  When trying to generalise the type inferred for ‘notEq’
    Signature type: forall a b. (Eq a) => a -> b -> Bool
    Type to generalise: a -> a -> Bool
  When checking the type signature for ‘notEq’
  When generalizing the type(s) for ‘notEq’
```

The Chameleon system reports:

```
notEq.hs:2: ERROR: Inferred type does not subsume
                   declared type
Declared: forall a,b. Eq a => a -> b -> Bool
Inferred: forall b. Eq b => b -> b -> Bool
Problem : The variable ‘a’ makes the declared type
          too polymorphic
          notEq :: Eq a => a -> b -> Bool
          notEq x y = if x == y then False else True
```

The advantage of this error message is that it clearly indicates the program locations which violate the declared type; it is `(==)` applied to `x` and `y` which force them to have the same type.

□

5.3 Ambiguity errors

In the presence of type class overloading, we require that all type schemes, whether inferred or declared, are unambiguous. Generally speaking, a type scheme is ambiguous if grounding all variables in the type component does not ground all variables in the constraint component. The ambiguity check was described in detail in Section 4.3.3.

Before re-visiting the details, we give an example of a program with an ambiguous type scheme.

Example 43 The definition of `f` in the following program is ambiguous.

```
class Mul a b c | a b -> c where mul :: a -> b -> c
instance Mul Int Int Int where mul = (*)


f xs = length xs `mul` 10
zero = f []
```

In the above, `f`'s type inferred type is $\forall a, b, c.\ (Num\ a,\ Mul\ Int\ a\ b) \Rightarrow [c] \rightarrow b$ The problem here is that, since the number `10` is overloaded, the type of `mul`'s second argument is unknown.[4]

Loading this program with GHC results in the following error message.

```
mul.hs:5:
    No instance for (Mul Int b c)
      arising from use of `f' at mul.hs:5
    Possible cause: the monomorphism restriction applied to
                    the following:
      zero :: c (bound at mul.hs:5)
    Probable fix: give these definition(s) an explicit
                  type signature
    In the definition of `zero': zero = f []
```

Interestingly, GHC reports the problem at the call to `f` in the body of `zero`. This error message correctly reports that there is no instance of class `Mul` to satisfy the demand at the call site. By ignoring `f`'s ambiguity it has reported the error quite far from the location where the programmer would actually fix the problem. □

[4] In Haskell, *Num*'s argument type cannot be automatically defaulted in this case, since *Mul* is not a standard Haskell type class [30].

When reporting an ambiguity error, we assume that the inferred class constraints are correct, and that the error should be fixed by simply eliminating the ambiguous variables. This can be done through the introduction of additional type information to the program, which could be in the form of a type annotation. Therefore, the aim of our ambiguity reporting scheme is to uncover those locations in the program where additional type information can be inserted by the programmer to ground the problematic variables and resolve the ambiguity.

We will overlook the possibility of user-declared ambiguous types, since they are trivial to report. Any ambiguous user-declared type simply must be changed before it can be accepted.

In order to find the locations where ambiguous type variables appear, we proceed as follows. We start with a user constraint $f(t, x)$, corresponding to the type of ambiguous definition `f`, and CHR set P, and run $f(t, x) \longrightarrow^*_P C$. We then apply the ambiguity check defined in Section 4.3.3 to find the set of ambiguous variables $\bar{\alpha} \subseteq fv(C)$. We select one variable $\alpha \in \bar{\alpha}$, and build ρ, the m.g.u. of C_e, without substituting for α. Given variables $t_l, ..., t_{l+n} \in fv(C)$, representing the types of locations $l, ..., l + n$, if $\alpha \in fv(\rho t_i)$, then the type at location i contains the variable α. Therefore, we report it as a candidate for a type annotation.

Example 44 We revisit the program of Example 43. The definition of `f` is repeated below in a slightly desugared form, with source location numbers made explicit.

$$\texttt{(f xs}_2\texttt{ = ((mul}_6\texttt{ (length}_8\texttt{ xs}_9\texttt{)}_7\texttt{)}_4\texttt{ 10}_5\texttt{)}_3\texttt{)}_1$$

From this, and the preceding class declaration we can generate the following constraints. We will use these constraints directly, rather than generating a CHR rule and performing a single-step derivation.

$(t_1 = t_2 \rightarrow t_3)_1, (t_4 = t_5 \rightarrow t'_3)_3, (t_3 = t'_3)_3, (t_5 = d)_5, (Num\ d)_5, (t_6 = t_7 \rightarrow t'_4)_4,$
$(t_4 = t'_4)_4, (t_6 = a \rightarrow b \rightarrow c)_6, (Mul\ a\ b\ c)_6, (t_8 = t_9 \rightarrow t'_7)_7, (t_7 = t'_7)_7,$
$(t_8 = [e] \rightarrow Int), (t_9 = t_2)_{[2,9]}$

We can work out, by applying the ambiguity test of Section 4.3.3, that the ambiguous variable here is b. We now need to normalise these constraints, without substituting for b, and determine which locations have types that contain b. In this case we find the following normalised equations $t_4 = b \rightarrow t_3, t_5 = b, t_6 = a \rightarrow b \rightarrow c$. These represent the types of locations 4,5 and 6. Placing an appropriate type annotation at any of those locations would fix b and resolve the ambiguity.

```
mul.hs:4: ERROR: Inferred type scheme is ambiguous:
Type scheme: forall a,b,c. (Num a, Mul Int a b) => [c] -> b
Suggestion : Ambiguity can be resolved at these locations
             a: f xs = length xs `mul` 10
```

One possible fix is to make `10`'s type explicit with a type annotation, like so.

```
f xs = length xs `mul` (10::Int)
```

□

5.4 Missing instance errors

In Haskell, the arguments of type classes appearing within a type declaration or inferred function type, must be of one of two forms, either a single variable (α) or a variable applied to a number of types ($\alpha\ t_1 \ldots t_n$). For example, $Eq\ [a]$ is not allowed, whereas $Eq\ a$ and even $Eq\ (a\ [b])$ are.

A non-conforming constraint is one whose arguments have not been reduced to one of those forms, indicating that some instance of that class is missing. We aim to report the error so that the programmer can see and address the source of the problem directly, by either changing the program so that it no longer requires such an instance, or by adding that instance if possible.

Example 45 The following program violates the condition on type classes mentioned earlier. It is typical of the sort of mistake that beginners make. The complexity of the reported error is compounded by Haskell's overloading of numbers.

```
sumLists = sum .  map sum

sum [] = []
sum (x:xs) = x + sum xs
```

GHC does not report the error in `sum` until a monomorphic instance is required, at which point it discovers that no instance of $Num\ [a]$ exists. This means that unfortunately such errors may not be found through type checking alone – it may remain undiscovered until someone attempts to run the program. The function `sumLists` forces that here, and GHC reports:

```
sum.hs:3:
No instance for (Num [a])
  arising from use of ‘sum’ at sum.hs:3
Possible cause: the monomorphism restriction applied to
  the following:
  sumLists :: [[[a]]] -> [a] (bound at sum.hs:3)
Probable fix: give these definition(s) an explicit
  type signature
In the first argument of ‘(.)’, namely ‘sum’
In the definition of ‘sumLists’:
  sumLists = sum .  (map sum)
```

□

It would be useful to report not only the missing instance, but also the program locations which lead to the type for which we lack an instance. In other words, we need to find a minimal subset of constraints which imply the erroneous class constraint.

We restrict ourselves to single parameter type classes for this description, although the idea extends to any number of parameters. Assume we are interested in the type of function f, and $f(t) \longrightarrow^* C$, so that $f :: \forall fv(C).C \Rightarrow t$. Note that since C is unsimplified, all user constraints in C_u are of form $U\ \alpha$, where α is a type variable. Let $\phi = mgu(C_e)$ and find a $U\ t \in C_u$ such that $\phi(t)$ is not a variable or a variable applied to any number of types. For some type constructor T, if $\phi(t) = T$, then let $t' = T$, otherwise if $\phi(t) = T\ \bar{t}$, let $t' = T\ \bar{\alpha}$, where αs are fresh variables; t' is the type of the missing instance – we need to find where it comes from in the program text. Using the algorithm for finding minimal sets of implicants, we find a minimal subset $D_e \subseteq C_e$, such that $\models D_e \supset t = t'$, i.e. $D_e = \mathsf{min\_impl}(C_e, t = t')$. The justifications of constraints in D_e identify the problematic locations.

For the program in Example 45, we can report the following.

```
sum.hs:4: ERROR: Missing instance
Instance:Num [a]: sum [] = []
                  sum (x:xs) = x + sum xs
```

This indicates that the demand for this instance arises from the interaction between `[]` on the first line and `(+)` on the second.

5.5 Summary

In this chapter we have introduced a number of methods for reporting, and suggesting fixes to, the various kinds of type-related errors that can arise. We described methods which precisely locate the site of subsumption and missing-instance violations, as well a procedure for finding the locations at which ambiguity errors can be resolved.

We presented a simple method for finding the cause of type conflicts (satisfiability errors), which involves finding a minimal unsatisfiable subset of the conflicting constraints. In the next chapter we will discuss ways to make further use of the constraints available, to produce better, more informative error messages.

Chapter 6

Generating text error messages

So far we have presented a method for performing type inference via constraint solving using CHRs, and an algorithm which, in the event of a type error, will find a minimal subset of the constraints that correspond to the error. Since we use justified constraints, we can then trace those erroneous constraints back to specific program locations.

In this chapter we outline how our system processes this information to generate text error messages. The most obvious thing to do with a set of erroneous locations is to highlight them in the source program, as we discussed in Chapter 5. Though this does benefit the programmer, we have found that highlighting alone is often not sufficient to describe the error in an intuitive way.

First we address this problem by proposing a method for generating more conventional text error messages, which show the error as a conflict between some number of incompatible types at certain locations within the program (Section 6.1). We then describe an extension which allows programmers to customise advice that can be printed as part of the error message (Section 6.2).

6.1 Location-oriented text error messages

Generating insightful text error messages is difficult in general since there is no definite way to 1) decide which location is the actual source of the error, or 2) determine which are the correct, intended types.

As we have seen, one problem with traditional $\mathcal{W}$-style type inference algorithms is that they process programs in a fixed order. They proceed through the abstract syntax tree, generating and unifying types until they terminate success-

fully, or a unification failure occurs and the algorithm cannot continue. In the event of failure, these algorithms simply stop and report that the last location they visited could not be typed. Moreover, the error reported will always consist of exactly two incompatible types, an 'expected' type and an 'inferred' type, which could not be unified.[1]

In our constraint-based formulation of inference we are able to take a broader view, and consider all locations (not just one) involved in a type error, when deciding which conflicting types to report. In addition, we are not restricted to reporting only two problematic types. Indeed, we will see cases where it may be advantageous to report three or more types.

Briefly, our procedure for generating text error messages with type information, from a minimal unsatisfiable set of justified constraints, involves the following steps:

1. Select a location to report the type conflict about.

2. Find the types that conflict at that location.

   - Assign each a colour and determine which other locations contribute to it.

3. Diagnose the error in terms of the conflicting types at the chosen location. Highlight each location involved in the colours of the types it contributes to.

We address each of these steps in the following subsections.

6.1.1 Selecting an erroneous location

When a type error occurs, there are often many locations involved in the conflict, each of which is potentially the source of the programming mistake. The first problem is to determine which of these locations to select for more detailed reporting. Lacking any guidance, the system cannot determine whether one location is more likely to have been a mistake than another. Therefore, any scheme for selecting a location is at best a heuristic.

[1] Section 2.2.2 covered these issue in some detail. In algorithm $\mathcal{W}$ failure occurs at application sites; the 'expected' type comes from the function expression, whereas the 'inferred' type arises from the argument.

When choosing a location to report, one strategy that we have found to be reasonably successful is to select the location of the expression from amongst those in the minimal unsatisfiable subset which appears 'highest' in the program's abstract syntax tree (AST). By convention, when labelling programs from now on, we will assign lower numbers to locations which we consider 'higher' in the AST. The most compelling reason for this selection is that fresh variables and ground types remain in their respective scopes/sub-trees. Note that here we limit ourselves to selecting locations that correspond to expressions.

Example 46 The following program is ill-typed since the branches of the if-then-else have disjoint types.

```
f c = if c then \ g x -> g x
           else \ e y -> y e
```

GHC prints the following error message:

```
top.hs:2:
    Occurs check: cannot construct the infinite type:
        t = (t -> t1) -> t1
        Expected type: t
        Inferred type: (t -> t1) -> t1
    In the first argument of ‘y’, namely ‘e’
    In a lambda abstraction:  e y -> y e
```

It is our conjecture that most programmers would prefer to consider the types of the two branches in isolation, and that combining them leads to confusion – particularly when the order of processing is arbitrary.

By selecting the highest program location involved in the error, the point at which the two branches are forced to have the same type, we can treat the types of the two branches separately, and report the following:

```
top.hs:1: ERROR: Type error - one error found
Problem : Branches of if-then-else have incompatible types
Types   : (a -> b) -> a -> c (then)
          d -> (d -> e) -> f (else)
Conflict: f x = if x then \ g x -> g x
                     else \ e y -> y e
```

□

6.1.2 Finding the types to report

In this section we present an algorithm which, given a minimal unsatisfiable set of constraints and a program location to report, will generate an appropriate text message to present to the programmer. This algorithm is designed so that it can adopt a different error reporting strategy for each type of location that could lead to an error. For example, if the location we pick is an application node in the AST, we can complain specifically about the conflict between the function being applied and its argument; or if the nominated location is an if-then-else we could report the conflict between the 'then' and 'else' branches, and so on.

Furthermore, instead of just reporting a set of conflicting types, it can relate each of those types to locations in the program. As such, the algorithm is necessarily tied closely to both the language grammar of Section 4.1 as well as the specific constraint generation scheme.[2] It can obviously be adapted for use with other languages, like the complete language of expressions available in Haskell and Chameleon.

The algorithm, which finds the conflicting types to report, for a location l is presented in Figure 6.1 (which has been split into two parts.) We assume that the minimal unsatisfiable set of constraints which constitutes the type error is C. Variables t represent fresh types. The procedure for finding the reportable types depends on the sort of program location l is. As such, each type of location at which an error could arise must be accounted for. In our description, we include cases for reporting errors in terms of if-then-else expressions and multiple-clause definitions, even though they are not a part of the *core* language. The rules for dealing with these extensions assume the constraints were generated from semantically equivalent case expressions, as described in Section 4.1 (where we show how to desugar these constructs.) Note that we leave out a separate case for let-expressions, since they are subsumed by the 'multiple-clause definition' case, i.e. the opportunities for error reporting are the same in both cases.

As mentioned earlier, this algorithm is intimately related to the constraint generation scheme. The intuition behind it is as follows. Given a minimal unsatisfiable set of constraints C, we cannot extract any concrete type information to

[2]In particular, our algorithm depends on a constraint of form $a_l = t$, where a_l is a variable representing location l and t is some type, being present for each location l. If you refer back to the constraint generation scheme of Figure 4.2 you will notice that in the case of application expressions, we add an additional, otherwise-redundant constraint in order to fulfil this requirement.

location	x_l *(λ-bound id.)*
for some	$T_l = \{t_{bnd(x)} = t \mid t_{bnd(x)} = t \in C\}$
let	$R_l = \{t \mid t_{bnd(x)} = t \in T_l\}$
	$\phi = mgu(C - T_l)$
report	ϕR_l *(uses of x)*

location	f_l *(recursive let-bound id.)*
for some	$(t_l = t) \in C$
let	$\phi = mgu(C - \{(t_l = t)\})$
report	ϕt *(definition skeleton)*, ϕt_i *(recursive use)*

location	f_l *(non-recursive or annotated let-bound id.)*
for some	$(t_l = t) \in C$
let	$\phi = mgu(C - \{(t_l = t)\})$
report	ϕt *(definition of f)*, ϕt_l *(use at l)*

location	$(\backslash x_i \rightarrow e_j)_l$
for some	$(t_l = t_i \rightarrow t_j) \in C$
	$(t = t_l) \in C$
let	$\phi = mgu(C - (t = t_l))$
report	ϕt_l *(λ abstraction)*, ϕt *(elsewhere)*

location	$(e_i\ e_j)_l$
for some	$(t_i = t_j \rightarrow t) \in C$
let	$\phi = mgu(C - \{(t_i = t_j \rightarrow t)\})$
report	ϕt_i *(function)*, ϕt_j *(argument)*

location	$(\mathsf{case}\ e\ \mathsf{of}\ [p_i \rightarrow e_i]_{i \in I})_l$
for some	$D = \{b \mid b \equiv (t_i = t_{l_{p_i}} \rightarrow t'_{l_{e_i}}) \in C\}$
let	$\phi = mgu(C - D)$
report	ϕD *(alternatives in isolation)*

Figure 6.1: Finding the conflicting types at a specific location

report. Since C is minimal, though, we need only remove a single constraint to make it satisfiable. For each kind of location in the program, the algorithm finds any constraints which can tell us something useful about the type error from the point of view of that location. (This information is shown in parentheses following each reported type.) At least one constraint is then removed from C, the substitution ϕ (of the remaining constraints) is built, and the interesting types can be extracted as necessary. So in essence, when we remove a constraint and then build the unifier, we are doing inference for all locations involved except one, and

$$
\begin{array}{rl}
\text{location} & (\text{if}_l\ e_h \text{ then } e_i \text{ else } e_j)_k \quad \textit{(conditional)} \\
\text{for some} & (t_l = \mathit{Bool})_l \in C \\
\text{let} & \phi = mgu(C - (t_l = \mathit{Bool})_l) \\
\text{report} & \phi t_h \ (\textit{conditional expression})
\end{array}
$$

$$
\begin{array}{rl}
\text{location} & (\text{if}_h\ e_i \text{ then } e_j \text{ else } e_k)_l \quad \textit{(branches)} \\
\text{for some} & D = \left\{ \begin{array}{l} (t_l = t')_l, \\ (t'_e \to t_j = t_e \to t'), \\ (t'_e \to t_k = t_e \to t') \end{array} \right\} \in C \\
\text{let} & \phi = mgu(C - D) \\
\text{report} & \phi t_j \ (\textit{then}),\ \phi t_k \ (\textit{else})
\end{array}
$$

$$
\begin{array}{rl}
\text{location} & \left(\begin{array}{lcl} f\ p_1 & = & e_1 \\ \ldots & & \\ f\ p_n & = & e_n \end{array} \right)_l \\
\text{for some} & D = \{b \mid b \equiv (t_i = t_{l_{p_i}} \to t'_{l_{e_i}}) \in C\} \\
\text{let} & \phi = mgu(C - D) \\
\text{report} & \phi D \ \textit{(alternatives in isolation)}
\end{array}
$$

$$
\begin{array}{rl}
\text{location} & \mathsf{let} \begin{array}{l} (f :: \sigma)_l \\ f = e \end{array} \ \mathsf{in}\ e' \\
\text{for some} & (t_l = t)_l \in C \\
\text{let} & \phi = mgu(C - (t_l = t)_l) \\
\text{report} & \phi(t) \ \textit{(declared type)},\ \phi(t_l) \ \textit{(inferred)}
\end{array}
$$

Figure 6.1: *(continued)*

then reporting the reason why that last step cannot proceed. Note that all constraint manipulation performed reflects the generation scheme of Section 4.2.1, and is guaranteed to succeed by construction.

Since the types that are reported are built only from the minimal unsatisfiable set of constraints, they may not be as ground as may be expected. For instance, in Example 46, it is clear that the types of the two branches must be equal, and hence the variables `c` and `f` in the two conflicting types that Chameleon reports would have to be the same. Because the constraint enforcing the equality of the two branches was the one that was removed in order to produce that error message, however, we lose that additional information, and hence the two different variables are reported.

Fundamentally, our system will only report the minimal amount of type information to indicate an error. In general, of course, it may not be possible to deduce any additional type information, since there could be multiple minimal

unsatisfiable subsets in the inference result.

6.1.3 Finding the source of each type

Although we can now present a great deal of information about a type error, the connection between the types reported and the locations highlighted may not be obvious. We will further refine our approach to highlighting by connecting it to the conflicting types that were found.

For each type that is reported, we will find all those constraints amongst the minimal unsatisfiable subset which collectively cause that type to occur. We will then assign each reported type its own colour, and highlight the locations of the constraints which contributed to it in the same colour.

Assume that we find type t for the expression at location l, whose type is represented by variable t_l, in the minimal unsatisfiable constraint C. We then find a minimal $D \subseteq C$, such that $\models D \supset t_l = t$, where $\models$ is just the semantic entailment relation in first-order logic. To do this, we simply employ the minimal-implicant algorithm presented in Section 3.4, i.e. $D = \mathsf{min\_impl}(C, \{t_l = t\})$. Having picked a distinct colour for location l, we can then highlight all locations which justify the constraints in D by that colour.

Locations contributing to multiple conflicting types should be assigned a different, distinct colour. We have chosen to highlight them in a mix of the colours of the types which arose from them.

Example 47 In the following program there is an obvious conflict amongst the arguments to the `(==)` function (which we assume has the same type as in Haskell, $\forall a.Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$.)

```
idEq = \ x -> x == [x]
```

GHC reports the following:

```
idEq.hs:1:
    Occurs check:  cannot construct the infinite type:  t = [t]
        Expected type:  t
        Inferred type:  [t]
    In the list element:  x
    In the second argument of ‘(==)’, namely ‘[x]’
```

This message demonstrates how a particular traversal of the program by the inference algorithm can lead to a counterintuitive error message. Rather than report that the first `x` is forced to have types t_x and $[t_x]$ simultaneously, it complains that the second `x` ought to have type t_x, but has type $[t_x]$.

Chameleon produces the following type error message:

```
idEq.hs:1:  ERROR: Type error - one error found
Problem :  Function can't be applied to that argument
Types   :  a -> Bool (function)
           [a]       (argument)
Conflict:  idEq = \x-> x == [x]
```

Note that all of the locations highlighted are involved in the error, and replacing any with an appropriate expression could fix the program. Furthermore, by selecting the reporting location that we did, we were able to avoid the same confusion as the GHC error message where the type variable representing `x`'s type is prematurely unified with a list type.

□

Example 48 Consider the following ill-typed program:

```
f 'a' b    z = error "'a'"
f c   True z = error "True"
f x   y    z = if z then x else y
f x   y    z = error "last"
```

Here `error` is the standard Haskell function with type $\forall a.[Char] \rightarrow a$.

GHC reports:

```
mdef.hs:4:
    Couldn't match 'Char' against 'Bool'
        Expected type: Char
        Inferred type: Bool
    In the definition of 'f': f x y z = if z then x else y
```

What's confusing here is that GHC combines type information from a number of clauses in a non-obvious way. In particular, in a more complex program, it may not be clear at all where the *Char* and *Bool* types it complains about come from. Indeed, it is not even obvious where the conflict in the above program is. Is it complaining about the two branches of the if-then-else (if so, which is *Char*

and which *Bool*?), or about `z` which might be a *Char*, but as the conditional must be a *Bool*?

The Chameleon system reports:

```
multi.hs:1: ERROR: Type error - one error found
Problem : Definition clauses not unifiable
Types   : Char -> a -> b -> c
          d -> Bool -> e -> f
          g -> g -> h -> i
Conflict: f 'a' b z = error "'A'"
          f c True z = error "True"
          f x y z = if z then x else y
```

This message makes clear precisely where the conflicting types arose from. By reporting the error from the highest location, it has kept the type in the three clauses separate. The error message does not mention the last definition equation since it is irrelevant to the error.

□

In general, we might need to report any number of types. Consider the following (where we write $\mathtt{p_1@p_2}$ to give pattern $\mathtt{p_2}$ the alias $\mathtt{p_1}$):

$$\mathtt{f\ p_1@('a')\ p_2\ \ldots\ \ p_{n-1}\ p_n@(True)} = \overset{n-1}{_{i=1}}\ \mathtt{if\ undefined\ then\ p_i\ else\ p_{i+1}}$$

Here we would have $n-1$ clauses and would need to display a type for each. In practice, however, we'd expect that cases where a large number of types need to be printed are uncommon. Regardless, we feel that displaying all individual, clause-specific types is always preferable to reporting some arbitrary combination of those types.

6.1.4 Reporting errors involving functional dependencies

So far we have only considered reporting errors which involve the standard Hindley/Milner type system extended with Haskell-style type classes. Some extensions to the type system, such as functional dependencies can introduce type errors and their involvement must also be reported in a useful way.

The problem is that the error reporting scheme we just described is insufficient in the presence of functional dependencies. The scheme outlined earlier (in the beginning of Section 6.1.2) depends on constraints justified by some locations to be of a certain form. For example, if location l is an application site that is part of the set of erroneous locations, then we expect a constraint $(t = s \to t'_l)_l$ to be part of the minimal unsatisfiable subset. Unfortunately, this may not be the case once propagation rules are involved, and justifications can be freely copied between Herbrand constraints.

Example 49 Consider the following program.

```
class (G a b)1 | b ->2 a where (g :: a -> b)3
instance (G Bool Bool)4
(f = (not7 (g9 (g11 'a'12)10)8)6)5
```

□

From the program above we generate the following rules. We have also added a rule representing the definition of `not`, omitted from the source code.

$$
\begin{array}{lrcl}
(r1) & G\ a\ b & \Longrightarrow & True_1 \\
(r2) & G\ a\ b,\ G\ a\ c & \Longrightarrow & (b = c)_2 \\
(r3) & g(t) & \Longleftrightarrow & (G\ a\ b)_3,\ (t = a \to b)_3 \\
(r4) & G\ Bool\ Bool & \Longleftrightarrow & True_4 \\
(r5) & G\ a\ Bool & \Longrightarrow & (a = Bool)_{[4,2]} \\
(r6) & f(t_5, x) & \Longleftrightarrow & (t_5 = t_6)_5, (t_7 = t_8 \to t'_6)_6, (t_6 = t'_6)_6, not_a(t_7)_7, \\
 & & & (t_9 = t_{10} \to t'_8)_8, (t_8 = t'_8)_8, g(t_9)_9, (t_{10} = t'_{10})_{10}, \\
 & & & (t_{11} = t_{12} \to t'_{10})_{10}, g(t_{11})_{11}, (t_{12} = Char)_{12}, x = r \\
(r7) & not_a(t) & \Longleftrightarrow & t = Bool \to Bool
\end{array}
$$

Note that since there are no nested definitions, we omit the x parameter just to simplify the rules slightly.

To infer the type of `f`, we begin a derivation with $f(t)$ and proceed as follows.

$$
\begin{array}{rl}
 & \underline{f(t, x)} \\
\to_{(r6)} & t = t_5, (t_5 = t_6)_5, (t_7 = t_8 \to t'_6)_6, (t_6 = t'_6)_6, \underline{not_a(t_7)_7}, \\
 & (t_9 = t_{10} \to t'_8)_8, (t_8 = t'_8)_8, g(t_9)_9, (t_{11} = t_{12} \to t'_{10})_{10}, \\
 & (t_{10} = t'_{10})_{10}, g(t_{11})_{11}, (t_{12} = Char)_{12}, x = r
\end{array}
$$

$$
\begin{array}{ll}
\rightarrow_{(r7)} & t = t_5, (t_5 = t_6)_5, (t_7 = t_8 \rightarrow t'_6)_6, (t_6 = t'_6)_6, t_7 = t''', \\
& (t''' = \mathit{Bool} \rightarrow \mathit{Bool})_7, (t_9 = t_{10} \rightarrow t'_8)_8, (t_8 = t'_8)_8, \underline{g(t_9)_9}, \\
& (t_{11} = t_{12} \rightarrow t'_{10})_{10}, (t_{10} = t'_{10})_{10}, g(t_{11})_{11}, (t_{12} = \mathit{Char})_{12}, x = r \\
\rightarrow_{(r3)} & G\ a'\ b'_{[9,3]}, (t' = a' \rightarrow b')_{[9,3]}, t_9 = t', t = t_5, (t_5 = t_6)_5, (t_7 = t_8 \rightarrow t'_6)_6, \\
& (t_6 = t'_6)_6, t_7 = t''', (t''' = \mathit{Bool} \rightarrow \mathit{Bool})_7, (t_9 = t_{10} \rightarrow t'_8)_8, (t_8 = t'_8)_8, \\
& (t_{11} = t_{12} \rightarrow t'_{10})_{10}, (t_{10} = t'_{10})_{10}, \underline{g(t_{11})_{11}}, (t_{12} = \mathit{Char})_{12}, x = r \\
\rightarrow_{(r3)} & G\ a''\ b''_{[11,3]}, (t'' = a'' \rightarrow b'')_{[11,3]}, t_{11} = t'', \underline{G\ a'\ b'_{[9,3]}}, \underline{(t' = a' \rightarrow b')_{[9,3]}}, \\
& \underline{t_9 = t'}, t = t_5, (t_5 = t_6)_5, \underline{(t_7 = t_8 \rightarrow t'_6)_6}, (t_6 = t'_6)_6, \underline{t_7 = t'''}, \\
& \underline{(t''' = \mathit{Bool} \rightarrow \mathit{Bool})_7}, \underline{(t_9 = t_{10} \rightarrow t'_8)_8}, \underline{(t_8 = t'_8)_8}, (t_{11} = t_{12} \rightarrow t'_{10})_{10}, \\
& (t_{10} = t'_{10})_{10}, (t_{12} = \mathit{Char})_{12}, x = r \\
\rightarrow_{(r5)} & \underline{(a''' = \mathit{Bool})_{[9,3,6,7,8,2,4]}}, \underline{a' = a'''}, \underline{G\ a''\ b''_{[11,3]}}, \underline{(t'' = a'' \rightarrow b'')_{[11,3]}}, \\
& \underline{t_{11} = t''}, G\ a'\ b'_{[9,3]}, \underline{(t' = a' \rightarrow b')_{[9,3]}}, \underline{t_9 = t'}, t = t_5, (t_5 = t_6)_5, \\
& (t_7 = t_8 \rightarrow t'_6)_6, (t_6 = t'_6)_6, t_7 = t''', (t''' = \mathit{Bool} \rightarrow \mathit{Bool})_7, \\
& \underline{(t_9 = t_{10} \rightarrow t'_8)_8}, (t_8 = t'_8)_8, \underline{(t_{11} = t_{12} \rightarrow t'_{10})_{10}}, (t_{10} = t'_{10})_{10}, \\
& (t_{12} = \mathit{Char})_{12}, x = r \\
\rightarrow_{(r5)} & (a'''' = \mathit{Bool})_{[11,3,9,6,7,8,2,4,10]}, a'' = a'''', (a' = \mathit{Bool})_{[9,3,6,7,8,2,4]}, a' = a''', \\
& G\ a''\ b''_{[11,3]}, (t'' = a'' \rightarrow b'')_{[11,3]}, t_{11} = t'', G\ a'\ b'_{9,3]}, (t' = a' \rightarrow b')_{[9,3]}, \\
& t_9 = t', t = t_5, (t_5 = t_6)_5, (t_7 = t_8 \rightarrow t'_6)_6, (t_6 = t'_6)_6, t_7 = t''', \\
& (t''' = \mathit{Bool} \rightarrow \mathit{Bool})_7, (t_9 = t_{10} \rightarrow t'_8)_8, (t_8 = t'_8)_8, (t_{11} = t_{12} \rightarrow t'_{10})_{10}, \\
& (t_{10} = t'_{10})_{10}, (t_{12} = \mathit{Char})_{12}, x = r
\end{array}
$$

The final constraints in the derivation above are unsatisfiable. The only minimal unsatisfiable subset C is: $(a'''' = \mathit{Bool})_{[11,3,9,6,7,8,2,4,10]}$, $a'' = a''''$, $(t'' = a'' \rightarrow b'')_{[11,3]}$, $t'' = t_{11}$, $(t_{11} = t_{12} \rightarrow t_{10})_{10}$, $(t_{12} = \mathit{Char})_{12}$.

If we were to apply the approach outlined in Section 6.1.1 we would attempt to explain the error in terms of location 6. (We can ignore locations 2,3 and 4 which arise from class and instance declarations.)

Since 6 is an application site, we would require a constraint of form $(t_7 = t_8 \rightarrow t'_6)_6$ in order to report the error, according to Figure 6.1. Unfortunately, no such constraint is present. The only reason 6 is part of the justification set is because it was copied when an earlier rule fired. Indeed, location 6 is a critical part of the error, but all that we know from the minimal unsatisfiable subset is that 6, amongst others, caused some rule to fire (which led to the error.)

Clearly we need a different approach to handle errors arising from functional dependencies in a useful way. Our new method, specialised for errors involving improvement rules, is conceptually similar to the previous method. It is based on

extracting types from a minimal unsatisfiable result, by first removing a constraint and thus making the result satisfiable. In this case, however, the constraint we will remove is always one propagated by an improvement rule.

We report the error in terms of the types affected by the improvement rule. We first show the two types (and their justifications) that caused the error when they were unified by the improvement rule. We then show why the improvement rule fired (the locations responsible) and the effect of the rule, in causing the error.

We start from the minimal unsatisfiable subset C that arises when inferring the type for `f`. From this we need to determine the last constraint $s = t$ added by an improvement rule. Let $f(t) \longrightarrow^* C' \longrightarrow_R C''$ be the derivation for `f` where C'' is unsatisfiable, and the last derivation step involved an improvement rule R. We can always assume that the last step in a derivation leading to a functional-dependency error is a propagation rule step.[3] Now $s = t$ is the justified constraint occurring in $C'' - C'$. Let $\phi = mgu(C - \{s = t\})$. We report the types ϕs and ϕt (highlighting their minimal justifications) and then report the rule R and the justification for it firing, including the original location of the user constraints involved. The source position of the user constraints which caused the last rule to fire can be easily recovered from their justifications.

When reporting such an error we also explain how the improvement rule was derived, i.e. the combination of class and instance declarations it arose from.

We will demonstrate our approach with a couple of examples.

Example 50 We now revisit the program and constraints of Example 49.

GHC reports:

```
gs.hs:3:
  Couldn't match 'Bool' against 'Char'
      Expected type: Bool
      Inferred type: Char
  When using functional dependencies to combine
    G Bool Bool, arising from use of 'g' at gs.hs:3
    G Char Bool, arising from use of 'g' at gs.hs:3
  When generalizing the type(s) for 'f'
```

[3] We have assumed that all rules are confluent. Consider that the only new simplification rules a propagation rule could possibly cause to fire are those arising from instance declarations, as described in Section 4.3.3. None of those simplification rules can result in an error, since they never add new equations.

We determine the last constraint added to the minimal unsatisfiable subset by an improvement rule (r5) is $s = t \equiv (a'''' = Bool)_{[11,3,9,6,7,8,2,4,10]}$. We calculate $\phi = mgu(C - \{s = t\})$ and then report $\phi a'''' = Char$ justified by $[3, 10, 11, 12]$, $\phi Bool = Bool$ justified by nothing, and finally the rule firing on constraint $G\ a''\ b''_{[11,3]}$ justified altogether by $[11, 3, 9, 6, 7, 8, 2, 4, 10]$

The Chameleon system reports the following:

```
gs.hs:3: ERROR: Functional dependency causes type error
Types   : Char
          Bool
Problem : class G a b | b -> a ...
          instance G Bool Bool ...
          Enforce: G a Bool ==> a = Bool
          On constraint:
             G Char Bool (from line 3, col. 14)
Conflict: f = not (g (g 'a'))
```

□

Example 51 Consider the following program.

```
class Collects ce e | ce -> e where
    empty :: ce
    insert :: e -> ce -> ce

f c = insert 'a' (insert True c)
```

GHC declares:

```
collects.hs:5:
    Couldn't match `Bool' against `Char'
        Expected type: Bool
        Inferred type: Char
    When using functional dependencies to combine
      Collects ce Bool, arising from use of `insert'
                        at collects.hs:7
      Collects ce Char, arising from use of `insert'
                        at collects.hs:7
    When generalizing the type(s) for `f'
```

We report:

```
collects.hs:5: ERROR: Functional dependency causes
                      type error
Types   : Char
          Bool
Problem : class Collects ce e | ce -> e ...
          Enforces: Collects ce e1,
                    Collects ce e2 ==> e1 = e2
          On constraints:
             Collects ce Char (from line 5, col. 7)
             Collects ce Bool (from line 5, col. 19)
Conflict: f c = insert 'a' (insert True c)
```

The advantage of our error report is that we are not limited to identifying just the locations of the `Collects` constraints above, we can straightforwardly point out all of the other complicit locations, and identify which of the conflicting types they contribute to.

□

It is clear that once we add functional dependencies, in order to fully explain complex type errors we really need to be able to break down the derivation process and explain individual steps. The problem becomes more complicated when other type extensions (programmed using CHRs) are allowed as in Chameleon. Once we allow for arbitrary CHR programs, we open the type system up to all the kinds of bugs that plague running programs. In general, we would need to resort to a debugger for CHR programs, which is beyond the scope of our work.

6.2 User-specified type error messages

Up until now we have been looking at the problem of supporting manual type error reporting as well as reporting type errors without any specialised knowledge about the program. Naturally, our methods have been generic, the consequence being that sometimes the error message that is reported is not the best description of the actual problem.

In this section we look at ways to allow the user to provide very specific type error messages which can be reported in certain circumstances. These messages

can be structured in terms more aligned with the specific use of a function. This is particularly useful for library writers, since they can tailor the error messages to reflect any domain-specific knowledge inherent in their libraries.

The methods presented here are inspired by the functionality described by Heeren et al [34]. We show that these ideas can be realised naturally and straightforwardly in our setting and discuss some extensions not covered there.

6.2.1 Assigning messages to constraints

We begin by describing a simple mechanism for associating descriptive text messages with types which are used during the inference process. Whenever an error involves one of these text-annotated constraints, the associated message can be printed as part of the error report.

Example 52 Consider the following program.

```
map f []     = []
map f (x:xs) = f x :  map f xs

test = map True ['a','b','c']
```

The standard Chameleon type error report reads as follows:

```
map.ch:4: ERROR: Type error - some locations are also
                 part of other errors
Problem : Function can't be applied to that argument
Types   : (a -> b) -> c  (function)
          Bool           (argument)
Conflict: map f [] = []
          map f (x:xs) = f x :  map f xs
          test = map True "abc"
```

□

The error message above is accurate. It shows that in the body of `test`, `map` is being incorrectly applied to `True`, and the reason this is incorrect is because in `map`'s definition its first argument is used as a function. Hence, `map` has a type of form `(a -> b) -> c`, and it has been applied to a `Bool`.

This error is a bit convoluted, though. We do not necessarily want to look deeply into a function's definition whenever that function is used incorrectly somewhere. We can avoid this by giving `map` a type annotation, which will be used (provided it is correct) wherever `map` is invoked.

Example 53 The following is a variation of the program in Example 52, above.

```
map ::  (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x :  map f xs

test = map True ['a','b','c']
```

This time Chameleon could report the following:

```
map.ch:5: ERROR: Type error - one error found
Problem : Function can't be applied to that argument
Types   : (a -> b) -> c  (function)
          Bool           (argument)
Conflict: map :: (a -> b) -> [a] -> [b]
          map f [] = []
          map f (x:xs) = f x :  map f xs
          test = map True "abc"
```

Note that, as mentioned in Section 6.1.2, the conflicting types that Chameleon reports are derived from the minimal unsatisfiable subset, and therefore contain just enough information to represent the error. The type of `map` is $(a \to b) \to [a] \to [b]$, but the fact that its first argument is a function is sufficient to cause this error, hence Chameleon reports the conflict involving function type $(a \to b) \to c$. □

This error message is better since it no longer requires the reader to understand the definition of `map`. Furthermore, our subsumption check (described earlier in Section 5.2), ensures that the annotation is correct with respect to the annotated type.

We can do better than this by refining the type declaration. We will increase the expressiveness of type annotations by allowing for Herbrand constraints in type contexts, where previously, as in Haskell, only type class constraints were allowed. Furthermore, we will associate with each of these constraints an optional

text string, which can be reported in case that constraint is involved in a type error.

Example 54 We have again modified the original program of Example 52, giving `map` a much more descriptive type annotation this time.

```
map :: ( t = a -> b -> c ; map has two arguments,
         a = d -> e      ; map's first argument must be a function,
         b = [f]         ; map's second argument must be a list,
         c = [g]         ; map returns a list,
         d = f           ; the argument type of the function must
                           match the element type of the input list,
         e = g           ; the result type of the function must
                           match the element type of the returned
                           list
       ) => t
map f []     = []
map f (x:xs) = f x :  map f xs

test = map True ['a','b','c']
```

Chameleon can now report:

```
map.ch:13: ERROR: Type error - one error found
Problem  : Function can't be applied to that argument
Reason   : map's first argument must be a function
Types    : (a -> b) -> c  (function)
           Bool           (argument)
Conflict : map :: (a -> b) -> [a] -> [b]
           map f [] = []
           map f (x:xs) = f x :  map f xs
           test = map True "abc"
```

□

In this last error message, we have provided all of the same basic information as in the previous annotated program case, as well as a more-specific plain language explanation of the problem, as you can see in the new `Reason` field.

Note that our subsumption check works in exactly the same way for this extended form of type annotation with Herbrand constraints allowed in the context.

As usual, we can be sure that the type annotation is valid for that definition of `map`.

Indeed there is no restriction on how annotations can specified as some combination of a type and associated context constraints. The following annotation is just as correct, though it does not offer the broad range of error messages that are available above.

```
map :: (a = d -> e ; map's first argument must be a function)
       => a -> [b] -> [c]
```

The implementation of this feature is rather simple. As before, we begin with a single minimal unsatisfiable set of constraints, C. We consider each type signature of form $f :: D \Rightarrow t$, where there exists a $d_J \in D$ and a $c_l \in C$, such that $J = [\ldots, l]$, and select one such signature. We then pick a single constraint, which appears in C, from the selected signature. The text message associated with that constraint becomes the `Reason` we give for the type error.

By convention, as in Example 54, the constraints we specify in a function's signature become more specific as we travel down the list. If we represent the function's type by a single variable, like t, then the first constraint will bind that variable to some other skeleton type, like $a \to b \to c$ in our example. Subsequent constraints will then affect those new variables, adding more specific information to the overall type as we proceed.

The important point to notice is that whenever a constraint low in the list is involved in a type error, it must be (if we follow our convention) that one of the higher constraints will also be involved. Indeed in Example 54, it's plain to see that if `map` is ever involved in a type error, the constraint $t = a \to b \to c$ must be part of it. Because of this, whenever we need to select a constraint in order to report a reason for a type error, we will always pick the *lowest* constraint available.

6.2.2 Attaching messages to CHR rules

In the Chameleon language it is possible to write CHR rules directly in a program, by using rule declarations. These must be written at the top-level, just like classes and instances, and consist of the keyword `rule`, followed directly by the CHR rule itself. For the purpose of type inference we simply collect and use all such rules as written. We will temporarily make use of this feature here, and will extend the explicit error message mechanism to support these user-declared rules.

Example 55 Consider the following program.

```
rule Eq (a -> b) ==> False

test f g = f 'a' || f == g
```

Attempting to type `test` results in an error, since `f` is clearly a function, and we are testing it for equality against `g`. The propagation rule explicitly forbids instances of `Eq` on functions.

Chameleon reports the following:

```
eq.ch:7:  ERROR: Type error - rule(s) involved
Problem : Rule:  Eq (a -> b) ==> False
          Applies to:  Eq (Char -> Bool)    (from line 3, col.  24)
Conflict: test f g = f 'a' || f == g
```

□

We can make things a little clearer by attaching a plain-language reason for the propagation rule involved. In this case we can associate a message with the `False` constraint, just like we did earlier with the Herbrand constraints.

Example 56 Consider the following program.

```
rule Eq (a -> b) ==> False ; cannot test functions for equality

test f g = f 'a' || f == g
```

We now report the following slightly more helpful version of the earlier type error message.

```
eq.ch:7:  ERROR: Type error - rule(s) involved
Reason  : cannot test functions for equality
Problem : Rule:  Eq (a -> b) ==> False
          Applies to:  Eq (Char -> Bool)     (from line 3, col.  24)
Conflict: test f g = f 'a' || f == g
```

□

The method we used for selecting which text message to report, if there are multiple, is essentially the same as the one we outlined in the previous section. The only difference is that instead of just looking for message-annotated constraints in type signatures, we must now also look in the bodies of propagation rules.

6.3 Summary

In this chapter we have expanded our type error reporting capabilities for programs containing type conflicts (unsatisfiability errors.) We have gone beyond the simple scheme of Chapter 5, and described a system which can generate informative text error messages that 1) can focus on any location involved in an error, 2) provide a reason (in terms of highlighted locations) for the reported conflicting types, 3) can report more than two conflicting types when necessary.

We have also described a method for systematically attaching user-defined messages to typing constraints. When one of these decorated constraints appears in a type conflict, the associated message can be reported as a possible reason for the failure.

Chapter 7

All minimal unsatisfiable sets

In Chapter 3 we introduced an algorithm for finding a single minimal unsatisfiable subset of constraints. We then used this in Chapters 5 and 6 to diagnose type errors. We begin in this chapter by considering the possible advantages, for error reporting, of having more than one minimal unsatisfiable subset (Section 7.1). We then present a number of procedures for efficiently finding *all* minimal unsatisfiable subsets of a constraint (Section 7.2). The algorithms we describe represent the most efficient known methods, for this purpose, at the time we published them [25]. The description and development of the algorithms presented here is based entirely on that publication. Proofs related to the content of this chapter can be found in Appendix B.

7.1 Using multiple minimal unsatisfiable sets

When the typing constraints arising from a source program are unsatisfiable, that program cannot be typed. To diagnose this, we would find a single minimal unsatisfiable subset of the typing constraints, and show where they came from. In general, it is possible that there could be multiple minimal unsatisfiable subsets; so far we have simply used whichever one our procedure happens to find first. Although multiple subsets may correspond to essentially the same programming mistake, it is likely that some subsets lend themselves to better explanations of the error than do others.

Example 57 Consider the following program. The functions `ord` and `null` are the standard Haskell functions of the same name with types $\mathit{Char} \to \mathit{Int}$ and $\forall a.[a] \to \mathit{Bool}$ respectively.

```
f x ys = ord (if x then x else null ys == x)
```

This program is ill-typed, since `ord` expects a *Char*, but the result of the conditional is always a *Bool*. As we will see, there are a number of possible explanations for this error, some of which are more straightforward than others. When we desugar this program and explicitly label all locations, we obtain the following.

$$\begin{array}{l}\texttt{(f } \texttt{x}_2 \texttt{ ys}_3 \texttt{ = (ord}_5 \texttt{ (case x}_7 \texttt{ of} \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{True}_8 \texttt{ -> x}_9 \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{\_ -> (((==)}_{13} \texttt{ (null}_{15} \texttt{ ys}_{16}\texttt{)}_{14}\texttt{)}_{11} \texttt{ x}_{12}\texttt{)}_{10}\texttt{)}_6\texttt{)}_4\texttt{)}_1\end{array}$$

We omit the CHR rule-generation and solving steps, since in this case they are trivial, and show only the final set of constraints. The variables t_t and t_f correspond to the *True* and *False* case alternatives, while $t_{\_}$, an unbound variable, stands for the 'don't-care' pattern.

$$\begin{array}{c}(t_1 = t_2 \to t_3)_1, (t_2 = t_x)_2, (t_3 = t_{ys})_3, (t_5 = t_6 \to t'_4)_4, (t_4 = t'_4)_4, \\ (t_5 = \mathit{Char} \to \mathit{Int})_5, t_t = t_8 \to t_9, t_f = t_{\_} \to t_{10}, (t_t = t_7 \to t'_6), \\ (t_f = t_7 \to t'_6), (t_6 = t'_6)_6, (t_7 = t_x)_7, (t_8 = \mathit{Bool})_8, (t_9 = t_x)_9, \\ (t_{11} = t_{12} \to t'_{10})_{10}, (t_{10} = t'_{10})_{10}, (t_{13} = t_{14} \to t'_{11})_{11}, (t_{11} = t'_{11})_{11}, \\ (t_{12} = t_x)_{12}, (t_{13} = a \to a \to \mathit{Bool})_{13}, (\mathit{Eq}\ a)_{13}, (t_{15} = t_{16} \to t'_{14})_{14}, \\ (t_{14} = t'_{14})_{14}, (t_{15} = [b] \to \mathit{Bool})_{15}, (t_{16} = t_{ys})_{16}\end{array}$$

There are a number of minimal unsatisfiable subsets here, each of which would yield a slightly different error report. One such subset is:

$$\begin{array}{c}(t_5 = t_6 \to t'_4)_4, (t_5 = \mathit{Char} \to \mathit{Int})_5, t_t = t_8 \to t_9, (t_t = t_7 \to t'_6), \\ (t_6 = t'_6)_6, (t_7 = t_x)_7, (t_8 = \mathit{Bool})_8, (t_9 = t_x)_9\end{array}$$

This subset essentially represents the conflict between `ord` and the returned `x` (at location 9), which is forced to be a *Bool* by the case pattern. An error report based on this subset would feature locations 4,5,6,7,8 and 9. Using the text message generation procedure of Chapter 6, we could report this (in terms of the original program) as follows.

```
f.hs:1:  ERROR: Type error
Problem : Function can't be applied to that argument
```

```
Types   : Char -> Int (function)
          Bool        (argument)
Conflict: f x ys = ord (if x then x else null ys == x)
```

Another minimal unsatisfiable subset, just as likely to be found as any other, is the following.

$$(t_5 = t_6 \to t'_4)_4, (t_5 = Char \to Int)_5, t_t = t_8 \to t_9, (t_t = t_7 \to t'_6),$$
$$(t_6 = t'_6)_6, (t_9 = t_x)_9, (t_{11} = t_{12} \to t'_{10})_{10}, (t_{13} = t_{14} \to t'_{11})_{11},$$
$$(t_{11} = t'_{11})_{11}, (t_{12} = t_x)_{12}, (t_{13} = a \to a \to Bool)_{13}, (t_{15} = t_{16} \to t'_{14})_{14},$$
$$(t_{14} = t'_{14})_{14}, (t_{15} = [b] \to Bool)_{15},$$

This subset again represents a conflict between `ord`'s expected argument type *Char*, and the type of its actual argument, *Bool*. In this case the reason that the argument is a *Bool* is more complicated. The result of the `null` call must be a *Bool*, which is propagated to `x`, via the use of `==`. This `x` shares the same type as the `x` in the 'then' branch, which here is responsible for the type of the argument to `ord`. In this case we have locations 4,5,6,9,10,11,12,13,14 and 15 to consider. The error we would report is as follows.

```
f.hs:1:  ERROR: Type error
Problem : Function can't be applied to that argument
Types   : Char -> Int (function)
          Bool        (argument)
Conflict: f x ys = ord (if x then x else null ys == x)
```

Both subsets provide valid reasons for the error, and both are equally likely to be found. The first subset provides for a clearer, more succinct explanation of the type error; far fewer locations are involved. In the second error explanation, the sudden jump from one branch of the if-then-else to the other, is particularly hard to follow.

There are also other minimal unsatisfiable subsets here, and many other possible ways to report these conflicts (see Chapter 6), but the point is simply that some subsets result in clearer error messages, with less intermediate reasoning required to understand them, than others do. □

It is our conjecture that, all else equal, a minimal unsatisfiable subset which refers to fewer program locations is more likely to result in a more succinct,

comprehensible error message. The reason is simply that the fewer locations highlighted, the easier it ought to be for a programmer to understand their interaction, and identify the mistake that caused the error. Given all minimal unsatisfiable subsets of an inference result, we propose the following strategy for selecting which subsets to report.

1. Let C be an unsatisfiable type inference constraint.

2. Find M, the set of all minimal unsatisfiable subsets of C.

3. Split M into n non-overlapping sets, $M_1, ..., M_n$. i.e. $M = \bigcup_{i=1}^{n} M_i$, and $M_i \cap M_j = \emptyset$, for all i, j where $i \neq j$

4. For each M_i: find a single $m_i \in M_i$ such that for all $m \in (M_i - m_i)$, $|just(m_i)| \leq |just(m)|)\}$, i.e. we pick one set from each M_i, which refers to the least number of program locations.[1]

The idea is simply that each distinct, non-overlapping subset should be reported as a single error. Furthermore, when subsets overlap, we select one which refers to the fewest program locations. If there are multiple sets which contain the same minimal number of references, we pick one at random. Each subset can then be reported in the style of Chapter 6.

In Section 3.3.1 we described a method for finding the intersection of all minimal unsatisfiable subsets. This information can provide a useful hint in debugging, since locations common to multiple subsets are more likely to be the true source of a type error. If there are multiple, non-overlapping minimal unsatisfiable subsets, unfortunately the result will always be empty. When we have all minimal unsatisfiable subsets, though, we can avoid the problem by first partitioning them into groups of overlapping sets, as in the process above. Finding the intersection of all sets in a group would be straightforward, and would prove just as useful as before.

In the following section we describe a number of relatively efficient procedures for calculating all minimal unsatisfiable subsets. It should be noted that these algorithms could also be modified to find all minimal implicants, just as we modified the procedure for finding a single minimal unsatisfiable subset to instead find

[1]Recall that $just(c)$ returns the concatenation of the justifications attached to all primitive constraints in c, and that $|J|$ represents the size of (i.e. number of locations in) a justification J.

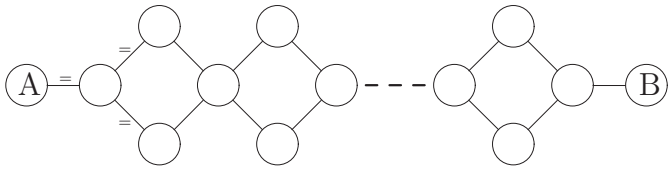

Figure 7.1: An exponential number of minimal unsatisfiable subsets

a minimal implicant in Section 3.4. The above reasoning could also be applied to select the best implicants to use.

7.2 Finding all minimal unsatisfiable subsets

Finding all minimal unsatisfiable subsets is a difficult problem, since there may be an exponential number of them. Consider the diagram in Figure 7.1. Nodes represent terms — unlabelled nodes are distinct variables — and edges represent constraints equating two terms. Each path in the graph from the type constructors A and B constitutes a different unsatisfiable subset, and every path that proceeds directly from one end to the other is minimal. Since the path branches periodically, the number of minimal unsatisfiable subsets is clearly exponential in the number of constraints.

We will first present a basic algorithm which enumerates subsets in a useful order, allowing us to easily ignore subsets of constraints already known to be minimal and unsatisfiable. We follow that by describing a number of modifications to this algorithm which can be used to reduce the search space further.

7.2.1 The basic algorithm

The problem we face is that given a set of constraints C, we somehow need to consider each subset of C, and determine whether it is unsatisfiable, and if so, if it is minimal. Since C has $2^{|C|}$ subsets, checking each soon becomes infeasible. We need a way to quickly reduce the number of sets under consideration.

Han and Lee [29] describe a procedure for enumerating subsets of a constraint C, which guarantees that no subset is visited more than once. Their algorithm generates a *CS-tree*, a tree where each node represents a subset of C, and the immediate children of a node of constraints D represent sets $D - d$, for all $d \in D$. The algorithm is presented below.

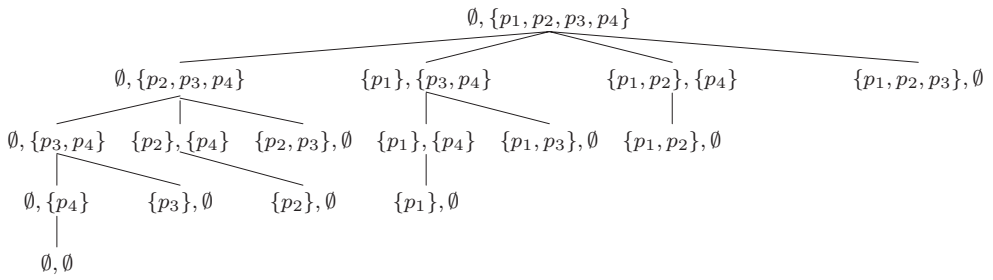

Figure 7.2: Visiting all subsets of $\{p_1, p_2, p_3, p_4\}$

```
all_subsets(D, P, A)
    A := A ∪ {D ∪ P}
    while ∃c ∈ P
        P := P − {c}
        A := all_subsets(D, P, A)
        D := D ∪ {c}
    endwhile
    return A
```

The procedure all_subsets takes three arguments: D which contains constraints which will *definitely* appear in all of the subsets; P which represents constraints that may *possibly* appear in some subsets, but not others; and $\mathbf{A}$ an accumulator which retains all of the subsets found so far. We can generate all subsets of a constraint C, by invoking the algorithm as all_subsets$(\emptyset, C, \emptyset)$. The algorithm's progress can be viewed as a CS-tree where the constraint at the node currently being visited is $D \cup P$.

Consider a set of constraints $C = \{p_1, p_2, p_3, p_4\}$, where the p_is are primitive equations. The tree traced out by all_subsets$(\emptyset, C, \emptyset)$ is shown in Figure 7.2. Each node corresponds to a call to all_subsets, and is labelled with the D and P constraints in that call, and represents the subset $D \cup P$ of C. Every node has $|P|$ children, each of which is formed by removing the constraints from P one by one, and adding them to the child's right siblings. For example, the parent node $\emptyset, \{p_2, p_3, p_4\}$, has a left-most child labelled $\emptyset, \{p_3, p_4\}$, and p_2 appears in the first component of all of its right siblings. The next child is again formed by taking the next constraint out of the P component, p_3 in this case, and adding it to the D part of all children to the right. In this way, every subset is enumerated exactly once, and the children of a node represent its immediate subsets.

We have a method for systematically calculating all subsets of a constraint,

we now need a way to discover all those which are minimal unsatisfiable subsets. We can make the following observations:

1. If $sat(C)$ then for every $C' \subset C$, $sat(C')$.
2. If $\neg sat(C)$ and for every $p \in C$, $sat(C - p)$, then C is minimal.
3. If $\neg sat(C)$ and C is *minimal*, then $C \wedge D$ (where $D \neq \emptyset$) is not minimal.

The first point states that all subsets of a satisfiable constraint must be satisfiable, and the second is essentially the definition of minimality of an unsatisfiable constraint. The third just says that a strict superset of a minimal unsatisfiable subset cannot be minimal.

Below, we define `all_min_unsat1` as a simple modification of `all_subsets` which returns in **A** only those nodes in the CS-tree which are labelled by a minimal unsatisfiable subset. It exploits the first point above.

```
all_min_unsat1(D, P, A)
   if (sat(D ∪ P)) return A
   while ∃c ∈ P
      P := P − {c}
      A := all_min_unsat1(D, P, A)
      D := D ∪ {c}
   endwhile
   if (¬∃A ∈ A such that A ⊂ D) A := A ∪ {D}
   return A
```

The above algorithm avoids visiting some nodes in the CS-tree; children of consistent nodes are not visited. The 'minimality' of all subsets in **A** is ensured by the final test. If at that point D contains any minimal unsatisfiable subsets, they will already appear in **A**, thereby eliminating D.

By making use of points 2 and 3, above, we can avoid visiting other, fruitless subsets. These ideas are implemented in the algorithm all_min_unsat2, below, which again is a modification of all_subsets.

```
all_min_unsat2(D, P, A)
   if (¬∃A ∈ A such that A ⊂ D ∪ P)
      if (sat(D ∪ P)) return A
   while ∃c ∈ P
      P := P − {c}
```

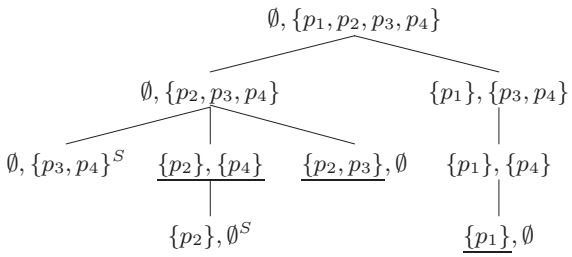

Figure 7.3: all_min_unsat2 finding $\{p_1\}, \{p_2, p_4\}, \{p_2, p_3\}$ in $\{p_1, p_2, p_3, p_4\}$

```
    A := all_min_unsat(D, P, A)
    D := D ∪ {c}
    if (∃A ∈ A such that A ⊂ D)
       return A
endwhile
return A ∪ {D}
```

The return result, and the $\mathbf{A}$ parameter, however consist only of minimal unsatisfiable subsets. There is a conditional test at the beginning which immediately skips this node if $D \cup P$ is satisfiable, since we could not possibly find any unsatisfiable subsets within those constraints. That particular *sat* test is avoided if there is a known minimal unsatisfiable subset within $D \cup P$.[2] The process of enumerating subsets by moving constraints from P to D is the same. If a minimal unsatisfiable subset is found that is a subset of D, then we can also immediately return, skipping all nodes to the right of the current one, since they must be supersets of D and therefore not minimal. Finally, if no unsatisfiable subsets were found, then the current D (which is equivalent to $D \cup P$, for the original values of D and P), must be minimal. This is essentially the algorithm of Han and Lee [29].

Example 58 Consider once again the set of constraints $C = \{p_1, p_2, p_3, p_4\}$, with minimal unsatisfiable subsets $\{p_1\}$, $\{p_2, p_3\}$ and $\{p_2, p_4\}$, and the call to all_min_unsat2$(\emptyset, C, \emptyset)$ traces out the CS-tree of Figure 7.3. Nodes marked with an S superscript are found to be satisfiable, and underlined nodes represent minimal unsatisfiable subsets. This tree is a pruned version of the CS-tree of Figure 7.2. For example, no sub-nodes of $\emptyset, \{p_3, p_4\}$ are explored, since it is found to be satisfiable. Note also that nodes to the right of $\{p_1\}, \emptyset$ are similarly ignored,

[2] As we will see, in the case of Herbrand constraints, for which an inexpensive *sat* test is available, this may not be a worthwhile optimisation.

since they must all be supersets of $\{p_1\}$, a minimal unsatisfiable subset (because it is unsatisfiable and has no potentially unsatisfiable subsets.) □

In work with Stuckey and García de la Banda [25] we have looked at introducing various optimisations into the basic algorithm described above, i.e. all_min_unsat2. We describe these optimisations individually in the following sections.

In the algorithms presented, we use Min as a placeholder for any of the all_min_unsat* versions defined herein. By letting one version of the algorithm call any other, we can chain together and stage different optimisations.

7.2.2 Preprocessing

As first mentioned in Section 3.3.1, it is straightforward to determine the intersection of all minimal unsatisfiable subsets. Given a set C, we can calculate this intersection $I = \{c \mid c \in C, sat(C - c)\}$.

By calculating this first, we can trivially modify any of the `all_min_unsat` procedures such that all subsets it enumerates and tests are supersets of I. We effectively reduce the size of the starting set by $|I|$ elements. Obviously if $I = \emptyset$, we gain nothing by performing this pre-processing step.

The following code uses the above observation to define the all_min_unsat3, which moves constraints in all unsatisfiable subsets into the first argument D, since we do not need to consider sets which do not include them. Note that we have modified the in_all_min_unsat algorithm of Section 3.3.1 so that it can be used to only test the constraints in P, since those in D are already known to be in subsets that will be examined.

```
all_min_unsat3(D, P, A)
    D' := in_all_min_unsat2(D ∪ P, P)
    return Min(D ∪ D', P − D', A)

in_all_min_unsat2(C, P)
    D := ∅
    foreach c ∈ P
        if sat(C − {c}) D := D ∪ {c}
    endfor
    return D
```

The cost of in_all_unsat(C, P) is $|P|$ calls to *sat*. If there are $m = |D'|$ constraints in every minimal unsatisfiable constraint, the subsequent call to Min$(D \cup D'$,$P - D'$,$\emptyset)$ may examine up to m less nodes for each non leaf call in the original call Min$(D$,P,$\emptyset)$. Therefore, if calls to *sat* are cheap and there is a significant probability of finding at least one constraint common to all unsatisfiable subsets, the cost is almost certainly worthwhile.

Although we could use this optimisation at any time during the search, it is most useful if performed once, at the beginning of the search. It is unlikely to discover new information later in the search, and the cost then becomes significant.

Example 59 Consider finding all minimal unsatisfiable subsets of $\{p_1, \ldots, p_5\}$ where the answers are $\{p_1, p_5\}$, $\{p_2, p_5\}$, $\{p_3, p_5\}$, $\{p_4, p_5\}$. Then

$$\text{all\_min\_unsat2}(\emptyset, \{p_1, p_2, p_3, p_4, p_5\}, \emptyset)$$

examines 31 subsets, requiring 23 calls to *sat*, while

$$\text{all\_min\_unsat2}(\{p_5\}, \{p_1, p_2, p_3, p_4\}, \emptyset)$$

examines 11 subsets, requiring 9 calls to *sat* to which we have to add another 5 calls from in_all_min_unsat2. □

7.2.3 Independence

For the following two improvements we use the notion of independence of constraints: constraints C_1 and C_2 are independent iff the solutions of $C_1 \cup C_2$ can be obtained by simply pairwise combining the solutions of C_1 and C_2 separately. In other words, if they do not influence each other's solutions.

A simple (and incomplete) test for constraint independence relies on examining the constraint graph. Given a set of constraints C, the *constraint graph*, $g(C)$ is a bi-partite graph with a *variable node* (labelled v) for each $v \in vars(C)$, a *constraint node* (labelled c) for each $c \in C$, and an edge (v, c) for each $v \in vars(c)$.

Lemma 1 *If C is a minimal unsatisfiable constraint set, then $g(C)$ is connected, i.e., the set of nodes cannot be partitioned into two sets such that there is no path connecting nodes in the two sets.*

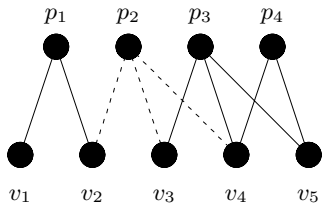

Figure 7.4: Constraint graph for $\{p_1, p_2, p_3, p_4\}$ with edges for p_2 dotted.

We can use this result to improve the all_min_unsat2 algorithm by first partitioning $C = D \cup P$ into connected subsets $X_1, \cdots X_n$ and then eliminating any partition which does not include D, since we are only interested in exploring supersets of D. The code is as follows:

```
all_min_unsat4(D, P, A)
   let X be the connected subsets of D ∪ P
   foreach X ∈ X
      if X ⊇ D
         P := X − D
         A := Min(D, P, A)
   endfor
   return A
```

where Min can again be any of the all_min_unsat* versions developed herein. Note that the only time when there can be more than one partition that passes the test $X \supseteq D$ is when $D = \emptyset$. Also note that partitioning the constraint graph is $O(||C||_c)$. Checking whether $X \supseteq D$ for all X is $O(|D|)$ since we need only traverse each constraint in D checking it belongs to the same partition. In the worst case all_min_unsat4 calls *sat* on each partition of $D \cup P$ which is certainly cheaper than a single call to $sat(D \cup P)$.

Note, however, that in the worst case all_min_unsat4 may indirectly cause the visitation of the node $(D, \emptyset)$ multiple times.

Example 60 Returning to the constraints of Example 58, suppose the variables of each constraint are $fv(p_1) = \{v_1, v_2\}$, $fv(p_2) = \{v_2, v_3, v_4\}$, $fv(p_3) = \{v_3, v_4, v_5\}$, and $fv(p_4) = \{v_4, v_5\}$. The constraint graph for this problem is shown in Figure 7.4.

Execution of all_min_unsat4$(\emptyset, \{p_1, p_2, p_3, p_4\}, \emptyset)$ (with Min = all_min_unsat2 in all_min_unsat4 and the recursive call in all_min_unsat2 in turn replaced by

all_min_unsat4) proceeds exactly as those for all_min_unsat2($\emptyset, \{p_1, p_2, p_3, p_4\}, \emptyset$) until we reach the call to all_min_unsat4($\{p_1\}$,$\{p_3, p_4\}$,$\mathbf{A}$). At this point the constraint set $\{p_1, p_3, p_4\}$ becomes disconnected and can be partitioned into two different sets, $\{p_1\}$ and $\{p_3, p_4\}$. Since only the first partition contains D, we do not address the second partition and the call all_min_unsat2($\{p_1\}$,$\emptyset$,$\mathbf{A}$) immediately adds $\{p_1\}$ to $\mathbf{A}$.

In total, the original all_min_unsat2 examines 9 subsets and makes 9 calls to *sat* while all_min_unsat4 examines only 8 subsets and makes 7 calls to *sat*. □

A more complex (and more complete) test for constraint independence disregards connections with variable nodes whose associated variables have been set to a fixed value. Intuitively, these connections are "dead" and thus cannot be the cause for dependency. Formally, a set of constraints C fixes a variable $v \in vars(C)$ if there exists a value $d \in \mathcal{D}$ such that for every solution θ of C: $\theta(v) = d$.

Lemma 2 *Let C be a minimal unsatisfiable constraint set with subset $D \subseteq C$ which fixes variables W. Then, the constraints $C - D$ are connected in $g(C)$ even after the variable nodes associated to variables W are removed from the graph.*

We can use the above to increase the partitioning performed by all_min_unsat4 by eliminating the connections in the graph of $D \cup P$ from variables fixed by D. This is managed by the code below.

```
all_min_unsat5(D, P, A)
   let X be the connected subsets of D ∪ P
         with variables fixed by D removed
   foreach X ∈ X
      if X ∪ D is connected
         P := X − D
         A := Min(D, P, A)
   endfor
   return A
```

Note that we now need to check that the partition X when unioned with D is connected (as opposed to checking that $X \supseteq D$), since D might not be a subset of X due to the removal of fixed variables. Also note that, as in all_min_unsat4, we may visit the node $(D, \emptyset)$ multiple times.

In order to use this optimisation effectively we need to efficiently determine variables that are fixed by a constraint D.

7.2.4 Always-satisfiable constraints

Another optimisation we can perform depends on detecting constraints that can never take part in a minimal unsatisfiable set, since their addition to any satisfiable subset will always give a satisfiable set. These "always satisfiable" constraints can be eliminated.

Lemma 3 *Let $S_1 \cup S_2$ be a set of constraints such that for any subset $U \subset S_2$ where $sat(U)$, then $sat(U \cup S_1)$. Then all minimal unsatisfiable subsets of $S_1 \cup S_2$ are subsets of S_2.*

This concept is in some sense related to independence since such constraints will also be independent of the minimal unsatisfiable sets. Unfortunately, most incomplete independence tests (like the two used in the previous section) are not powerful enough to detect this kind of independence.

A common approach to detect these kind of constraints is to use an analysis based on "degrees of freedom". This kind of analysis locates constraints which contain at least one variable that is completely "free", i.e., it can take any value in the domain.

Example 61 Consider the following set of Herbrand constraints

$$a = (b, c),\ b = d \rightarrow Int,\ c = Int,\ b = f \rightarrow f,\ d = Char$$

Since $a = (b, c)$ can always be satisfied, by choosing an appropriate pair type for a (a appears nowhere else), we can remove it from consideration. Consequently, we notice that c appears only in $c = Int$, so we eliminate it too. This leaves $\{b = d \rightarrow Int,\ b = f \rightarrow f,\ d = Char\}$, which has the same minimal unsatisfiable subsets as the original. □

We can use the above result in the algorithm by searching each set $D \cup P$ looking for constraints which are always satisfiable. If they appear in D, we can finish since D cannot then be a subset of a minimal unsatisfiable subset. Otherwise, we can remove them from P. Note that the symbol $\supset$ below refers to logical implication (it is not a set operation.)

```
all_min_unsat6(D, P, A)
  let Z ⊆ D ∪ P be the subset such that
        ∀U' ⊆ D ∪ P   sat(U') ⊃ sat(U' ∪ Z)
```

```
if (Z ∩ D ≠ ∅) return A
P := P − Z
return Min(D, P, A)
```

The implementation of the above optimisation uses a simple degrees of freedom analysis to detect always satisfiable constraints: constraints of the form $v = t$ are always satisfiable if variable v appears nowhere else. Such constraints are removed and the check is repeated until no constraints of this form appear. By keeping a list of the occurrences of each variable on the left hand side of an equation, and a count of other occurrences, our (incomplete) analysis can determine the always satisfiable constraints (of this form) in linear time.

7.2.5 Entailment

Any constraint $p \in P$ which is entailed (and thus redundant) with respect to D can be removed from P, since it cannot be part of a minimal unsatisfiable subset.

Lemma 4 *Let $U \subseteq P$ be a set of constraints such that $D \supset U$, then no minimal unsatisfiable subset of $D \cup P$ which is a superset of D intersects with U.*

Checking for entailment can be as expensive as the satisfiability checks it may avoid. However, some simple (and incomplete) kinds of entailment checking are straightforward. For example, by determining which variables are fixed by D and their values, we can cheaply check for entailment of constraints by simply evaluating the constraint with those values.

Example 62 Consider $D = \{(a, b) = (c, c), a = [Int], d = (e, f), e = c\}$ and $P = \{b = [g], f = Bool\}$. We can remove $b = [g]$ from P, since in all solutions of D, we have $b = [Int]$. The constraint $b = [g]$ is redundant and will never be part of a minimal unsatisfiable set. □

7.2.6 Cheap solvers

The original algorithm of Han and Lee [29] was developed for diagnosis of circuits where the satisfiability test *sat* is very expensive. Dealing with cheap solvers, such as the linear unification solver for Herbrand constraints, raises other considerations. First, checking that $D \cup P$ is a superset of a previous answer in order to avoid the call to *sat*, may not be beneficial since the *sat* check will usually be

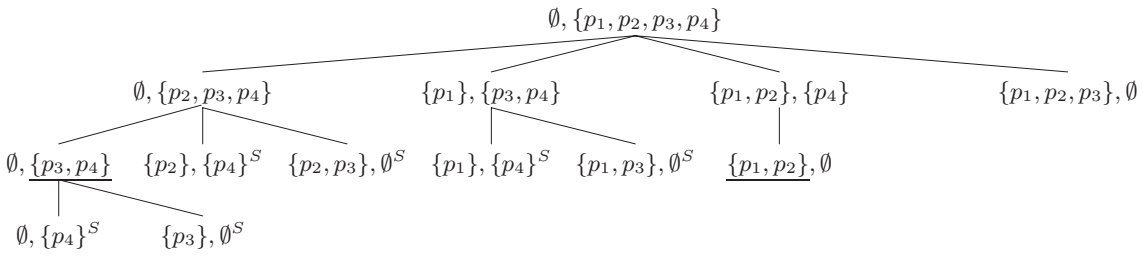

Figure 7.5: all_min_unsat2 finding $\{p_1, p_2\}, \{p_3, p_4\}$ in $\{p_1, p_2, p_3, p_4\}$

cheaper than detecting whether it is a superset or not. Similarly, we can replace the test which checks that D is not a superset of a previous answer by $sat(D)$. Finally, if D is unsatisfiable then we do not need to check any other subsets of $D \cup P$, since only D can be a minimal unsatisfiable subset.

This leads to the code below which replaces that of all_min_unsat2. Note that due to the changes to the loop break (when D becomes unsatisfiable then we add it to the set $\mathbf{A}$ of answers, if it is not subsumed, and return) execution will never reach the final line.

```
all_min_unsat7(D, P, A)
   if (sat(D ∪ P)) return A
   while ∃c ∈ P
      P := P − {c}
      A := all_min_unsat7(D, P, A)
      D := D ∪ {c}
      if (¬sat(D))
         if (¬∃A ∈ A such that A ⊂ D)) A := A ∪ {D}
         return A
   endwhile
   return A
```

Example 63 The extra *sat* check can discover answers earlier. Imagine we have the set of constraints $\{p_1, \ldots, p_4\}$ with minimal unsatisfiable sets $\{p_1, p_2\}$ and $\{p_3, p_4\}$, and we traverse the sets as in Figure 7.2. Then the nodes traversed by all_min_unsat2 are shown in Figure 7.5 while all_min_unsat7 will discover the unsatisfiable set $\{p_1, p_2\}$ at the root node, before calling all_min_unsat7$(\{p_1, p_2\}, \{p_4\}, \mathbf{A})$. Thus, it will not explore this subtree. □

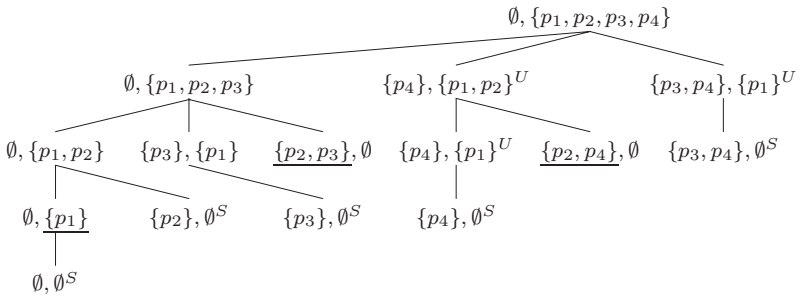

Figure 7.6: all_min_unsat2 finding $\{p_1\}, \{p_2, p_4\}, \{p_2, p_3\}$ in $\{p_4, p_3, p_2, p_1\}$

7.2.7 Incremental solving

As Han and Lee pointed out, the order in which subsets of each node are investigated can have significant impact on the total search space.

Example 64 Returning to the constraints of Example 58, if we build the tree by reversing the ordering of the constraints, we obtain a tree of calls as shown in Figure 7.6. Nodes marked U are determined as unsatisfiable without calling *sat*, since they are supersets of already determined minimal unsatisfiable sets.

For this order there are 14 nodes and 11 satisfiability checks compared to 9 nodes and 9 satisfiability checks for the order shown in Figure 7.3.

As the size of constraint sets increases, the differences get bigger. For example, for 5 constraints $\{p_1, \ldots, p_5\}$ with minimal unsatisfiable subsets $\{p_1\}$, $\{p_2, p_3\}$, $\{p_2, p_4\}$ and $\{p_2, p_5\}$ the first order requires 14 nodes and calls to *sat* while the reverse order leads to 28 nodes and 17 calls to *sat*. □

It is then important to determine orders which are likely to reduce the search. *Incremental* constraint solvers can help us in this task. This is not only because we can reuse some of the work performed by the incremental solver, thus reducing its cost, but also because we can gain important information from the constraints by adding them one by one. An incremental solver *isat* is a function from a constraint c, and a state *state* (representing the unification of some previous constraints), to a new state, *state'*, which includes the effect of c, and a result which indicates whether unification succeeded. One possible encoding of *state* is as a substitution. The idea is to start with a state representing D, and add constraints from P one by one, until we discover unsatisfiability. The last added constraint c must be part of a minimal unsatisfiable set and it should therefore be considered next in the order.

To take maximum advantage of the incremental solver, we will pass three extra arguments to our function: $T \supseteq D$ representing the subset of $D \cup P$ currently proved to be satisfiable by the incremental solver, $state_T$ the state representing the constraints in T, and $state_D$ the state representing the constraints in D. Intuitively, $state_T$ is obtained from $state_D$ by adding some constraints from P while looking for the next constraint to be chosen.

```
all_min_unsat8(D, state_D, P, T, state_T, A)
   result := true
   while (result && ∃c ∈ P − T)
      T := T ∪ {c}
      last_T := state_T
      (result, state_T) := isat(c, state_T)
   endwhile
   if (result) return A
   A := all_min_unsat8(D, state_D, P − {c}, T − {c}, last_T, A)
   D := D ∪ {c}
   (result, state_D) := isat(c, state_D)
   if (¬result)
      if (¬∃A ∈ A such that A ⊂ D))
         A := A ∪ {D}
      return A
   A := all_min_unsat8(D, state_D, P − {c}, D, state_D, A)
   return A
```

Example 65 Consider the constraints of Example 58 and the execution of (a) all_min_unsat8$(\emptyset, true, \{p_1, p_2, p_3, p_4\}, \emptyset, true, \mathbf{A})$ with the reverse (bad) ordering as the underlying one.

Initially, $D = \emptyset$ and $P = P_0 = \{p_1, p_2, p_3, p_4\}$. We add the constraints one by one, in reverse order, and discover unsatisfiability when $c = p_2$, leaving us with $T = \{p_2, p_3, p_4\}$. This leads to (b) all_min_unsat8$(\emptyset, _, \{p_1, p_3, p_4\}, \{p_3, p_4\}, _, \mathbf{A})$ (where _ represents some state information). This call adds p_1 to T which causes unsatisfiability, thus ending the while loop and resulting in the call to (c) all_min_unsat8$(\emptyset, _, \{p_3, p_4\}, \{p_3, p_4\}, _, \mathbf{A})$ which immediately returns since $P - T$ is empty.

Returning to call (b) we add $\{p_1\}$ to D detecting unsatisfiability, so $\{p_1\}$ is added to $\mathbf{A}$ and we return execution to call (a). We add p_2 to D which remains satisfiable and call (d) all_min_unsat8$(\{p_2\}, _, \{p_1, p_3, p_4\}, \{p_2\}, _, \mathbf{A})$. This call adds

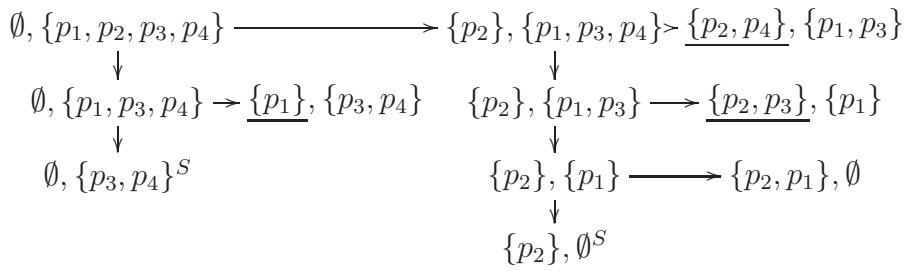

Figure 7.7: Finding $\{p_1\}, \{p_2, p_3\}, \{p_2, p_4\}$ in $\{p_4, p_3, p_2, p_1\}$ using an incremental solver

p_4 which immediately causes failure finishing the while loop and invoking (e) all_min_unsat8($\{p_2\}, \_, \{p_1, p_3\}, \{p_2\}, \_, \mathbf{A}$). This call adds p_3 which again causes failure, ending the loop, and invoking (f) all_min_unsat8($\{p_2\}, \_, \{p_1\}, \{p_2\}, \_, \mathbf{A}$). This call adds p_1 which again fails and calls (g) all_min_unsat8($\{p_2\}, \_, \emptyset, \{p_2\}, \_, \mathbf{A}$) which immediately succeeds. Returning to invocation (f) we add p_1 to p_2 and discover an unsatisfiable subset which is not added to $\mathbf{A}$ since it is a superset of an existing answer. Returning to invocation (e) we add p_3 to p_2, which is also unsatisfiable and the new answer is added. Finally, returning to invocation (d) we add p_4 to p_2, which again is unsatisfiable and the new answer is added.

The tree of calls is shown in Figure 7.7. Invocations dealing with the same set $D \cup P$ are shown horizontally aligned. Overall there are 12 calls to *isat* and 6 subsets (nodes of the CS-tree) are explored.

If we count the size of constraint sets passed to *sat* in Example 58 there are effectively 18 calls to *isat* and 9 subsets explored. In the bad order (illustrated by Example 64) there are effectively 21 calls to *isat* and 15 subsets explored. □

Note that the incremental version is far less sensitive to the underlying constraint order since it dynamically determines its own.

Example 66 The call all_min_unsat8($\emptyset, \mathit{true}, \{p_1, p_2, p_3, p_4\}, \emptyset, \mathit{true}, \mathbf{A}$) using the natural (good) order leads to the tree of calls in Figure 7.8. Overall there are 11 calls to *isat* and 6 subsets explored. □

Theorem 1 *The call* all_min_unsat8($\emptyset$, true, $C, \emptyset$, true, $\emptyset$) *returns all minimal unsatisfiable subsets of* C. *(See Appendix B.)*

Note that although it may appear that we need to keep two (or even three) states of solvers for this algorithm, in fact we can recover $state_D$ (and $last_T$) from $state_T$

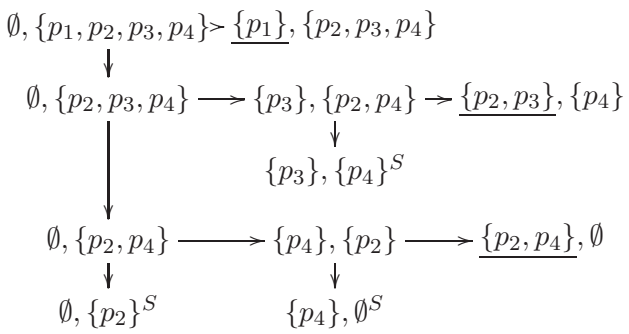

Figure 7.8: Finding $\{p_1\}, \{p_2, p_3\}, \{p_2, p_4\}$ in $\{p_1, p_2, p_3, p_4\}$ using an incremental solver

by backtracking, a facility available with all the Herbrand solvers used and, hence, we only need one solver state.

We can use the incremental version together with independence, always satisfiable and redundant improvements by extending the calls to $\mathsf{Min}(D, P, \mathbf{A})$ occurring in those versions to $\mathsf{all\_min\_unsat8}(D, state_D, P, D, state_D, \mathbf{A})$, since none of these change the set D.

7.2.8 Evaluation

In order to investigate the benefits of the optimisations discussed in previous sections, we have built a prototype system in SICStus Prolog [75] which implements each of the optimisations and their possible combinations. In this prototype the algorithms for checking the connectedness of constraints graphs, and for detecting always unsatisfiable and redundant (by fixed variables) constraints are implemented naively. Similarly, the full incremental algorithm is mimicked rather than implemented in full, since the internal state of the constraint solver is not available to the programmer. Hence, we will not compare the results on execution times but, rather, compare the number of different subsets $D \cup P$ explored by the various versions, and the number of calls to *isat* (where we implement *sat* in terms of *isat*). This allows us to better compare incremental and non-incremental versions.

The evaluation is based on sets of constraints generated by Chameleon (as described in Chapter 4) from a number of small, ill-typed programs. It uses the efficient satisfiability procedure for solving Herbrand equations provided by

Benchmark	$\|C\|$	$\|\mathbf{A}\|$
`suml`	20	2
`sum2`	22	2
`foldl1`	6	1
`foldl2`	8	2
`palin-l`	10	2
`palin-g`	260	5
`list`	12	9
`pop2`	24	1
`const`	2891	6
`tup`	36	1
`ite`	18	1
`rotate`	1711	8

Table 7.1: Benchmark statistics: number of constraints, and number of minimal unsatisfiable subsets.

SICStus Prolog. In the original paper [25], we also include results of benchmarks for a number of circuit diagnosis problems, which are phrased in terms of Boolean constraints.

Many different algorithms can be obtained by chaining together the various improvements suggested herein. To identify the different improvements in the algorithm of interest we will use the notation $x_1 \cdots x_n.x_{n+1} \cdots x_m$ to represent each algorithm, where each x_i represents the digit n of a all_min_unsat$n$ version in which the call to Min (or the recursive calls in all_min_unsat2, all_min_unsat7 and all_min_unsat8) is replaced by a call to the version indicated by x_{i+1}. The exception appears when $i = m$ in which case the call to Min is replaced by a call to the version appearing after the dot (x_{n+1}). For example, the algorithm described by 3.658 starts with the preprocessing step defined by version all_min_unsat3, where its call to Min is replaced by all_min_unsat6, that of all_min_unsat6 is replaced by all_min_unsat5, that of all_min_unsat5 is replaced by all_min_unsat8, and that of all_min_unsat8 is replaced by all_min_unsat6. Also, whenever the 'don't care' optimisation is used, we prepend a d to the algorithm description.

Table 7.1 shows the sizes of these benchmarks in terms of total number of constraints, and number of minimal unsatisfiable subsets. Table 7.2 shows the number of subsets $D \cup P$ examined by a number of selected algorithms (ignoring any subsets checked by the preprocessing steps). This gives a reasonable comparison of the size of the search space for each algorithm. We limited the number

of subsets to be searched to 10,000 for each algorithm. The symbol — indicates entries in which the search required more subsets to be examined. In order to select the algorithms that would appear in the table, we first evaluated all possible combinations and then summarised the results by eliminating those whose data did not add much to the discussion. As a result, the table is divided into two parts. The first compares the results for each individual optimisation (preprocessing, independence, always satisfiable, etc). The second part selects the best individual optimisation (incrementality) and compares the results of adding others to it. The last part evaluates the effect of the 'don't care' optimisation on the basic algorithm and on the best algorithm found in each of the two previous parts. Note that the redundancy elimination optimisation never occurs for the type error problems, and is only rarely applicable for the circuit error diagnosis problems. Thus, these results were also eliminated from the table. Table 7.3 shows the number of calls to *isat* examined by each selected algorithm (including preprocessing calls). Since the cost of satisfiability is almost constant, this gives a very accurate estimate of the work performed by the solver.

When comparing all_min_unsat1 (algorithm .1) and all_min_unsat2 (.2), it seems that for Herbrand equations the pruning improvements are only moderate. They make `pop2` solvable and occasionally reduce the number of subsets examined by around 1/3. They also reduce the number of calls to *isat* more frequently (once even by 2/3) although usually by less than 1/3.

The preprocessing step all_min_insat3 (3.2), which removes constraints in all unsatisfiable subsets, is more beneficial. For the benchmarks with only a single unsatisfiable subset, `foldl1`, `pop2`, `tup` and `ite`, it solves the entire problem. For other benchmarks, it substantially reduces the number of subsets visited, when there are any constraints found via the method, and makes `palin-g` solvable. The extra *isat* checks required for this optimisation are almost always repaid, except for the case of `palin-l` and `list`, where the optimisation finds no constraints in all minimal unsatisfiable subsets, and the number of *isat* checks is slightly increased. However, considering the low cost of the Herbrand *isat* checks, the increase seems reasonable.

The independence based optimisation all_min_unsat4 (.42) is clearly useful. In all but one example it reduces the number of subsets considered, often by around 1/2 and sometimes by more than an order of magnitude. The more sophisticated call-graph approach all_min_unsat5 (.52), which takes into account fixed variables, only occasionally improves on the simple call graph approach (`list` and `tup`), and

often worsens the situation. The reason is that whenever one of the optimisations applies, we may visit the same set twice. Since the more sophisticated version finds more places to apply the optimisation, this happens more often.

The 'always satisfiable' constraints optimisation all_min_unsat6 (.62) are also highly effective, often reducing the number of sets examined by an order or magnitude or more. Applying this optimisation makes two more examples (`palin-g` and `ite`) solvable where before (using .2) they were not.

Replacing subset checks by calls to *isat* in all_min_unsat7 (.7) reduces the number of subsets examined very slightly and only in a few cases. Furthermore, it increases the number of *isat* checks quite dramatically. Surprisingly, the comparison of the execution times (not shown) of .2 versus .7 indicates that the benefit in terms of reduced subset checks is not repaid. For these two simple algorithms the prototype times should be reasonably accurate.

The most benefit comes from the use of the incremental solver in all_min_unsat8 (.8). The number of subsets explored is dramatically reduced, since the incremental approach quickly finds large satisfiable subsets. Improvements of two orders of magnitude are common and all the benchmarks now require examining less than 3000 different subsets.

When the optimisations are combined the benefits are not always cumulative. For example, comparing 3.8 to .8 in terms of calls to *isat* shows that using the incremental solver the benefit of preprocessing step is usually not paid off, although for the hardest benchmarks `const` and `rotate` it is beneficial. On the other hand, comparing the results of .68, .8 and .62, the benefits of the combination .68 do seem cumulative. In any case, the combination 3.648 visits the least number of subsets in all cases, and overall is the most efficient in terms of the use of the solver. Hence, there is definite synergy between the optimisations.

7.3 Summary

In this chapter we considered the advantages that having all multiple minimal unsatisfiable subsets offer for type error diagnosis. We also described the development of a flexible algorithm for efficiently calculating all minimal unsatisfiable subsets.

In more recent work Bailey and Stuckey [40] have had some success in applying an algorithm used for discovering frequent patterns in the field of data mining to

the problem of finding all minimal unsatisfiable subsets. Their benchmarks show a significant reduction in running time as compared to the methods we outlined here.

Unfortunately in practice, even with the improvements described here, finding all minimal unsatisfiable subsets, for simple error reporting purposes, seems infeasible. Nevertheless these optimisations would prove useful in an interactive system where further minimal unsatisfiable subsets could be silently calculated in the background, while the programmer proceeds with the information found so far.

Alg	sum1	sum2	foldl1	foldl2	palin-l	palin-g	list	pop2	const	tup	ite	rotate	TOTAL
.1	4614	6020	9	41	256	—	1949	—	—	5808	—	—	18697
.2	4614	5509	9	41	256	—	1292	6240	—	5808	—	—	23769
3.2	1793	2562	0	10	256	3703	1292	0	—	0	0	—	9616
.42	1885	2280	10	33	11	—	606	4385	—	430	4663	—	14303
.52	1882	2281	9	32	11	—	484	4859	—	308	5012	—	14878
.62	182	204	9	22	8	3089	190	704	—	38	149	—	4595
.7	4614	5508	9	41	255	—	1283	6234	—	5808	—	—	23752
.8	22	24	6	10	15	278	123	24	2884	31	13	1713	5143
3.8	5	7	0	5	15	93	123	0	972	0	0	127	1347
.68	16	17	6	10	8	124	31	24	455	31	13	293	1028
3.648	4	5	0	5	8	34	31	0	102	0	0	29	218

Table 7.2: Number of subsets examined for type benchmarks.

Alg	sum1	sum2	foldl1	foldl2	palin-l	palin-g	list	pop2	const	tup	ite	rotate	TOTAL
.1	56803	79110	44	239	1023	—	12844	—	—	202128	—	—	352191
.2	40315	55210	44	213	483	—	3998	143889	—	166673	—	—	410825
3.2	9417	19952	35	113	539	138400	4110	1487	—	1590	948	—	176591
.42	1695	2692	38	96	38	—	367	32128	—	6160	20652	—	63866
.52	1738	2715	39	97	38	—	284	44129	—	4905	24747	—	78692
.62	478	704	15	61	15	33120	111	3300	—	454	178	—	38436
.7	108396	135242	74	429	1591	—	16184	362714	—	373128	—	—	997758
.8	428	515	37	72	91	8421	887	1058	147945	1406	502	81440	242802
3.8	371	440	35	76	147	4281	999	1487	47721	1590	948	8546	66641
.68	195	246	24	55	43	2627	152	622	12028	671	168	11156	27987
3.648	341	377	35	74	99	3031	264	1487	7039	1590	948	6934	22219

Table 7.3: Number of calls to incremental solver *isat* for type benchmarks

Chapter 8

Advanced type system extensions

In previous chapters we described a type system, CHR-based inference framework, and type error reporting for a Haskell-like language based on Hindley/Milner with algebraic data types, type classes and type annotations. In this chapter we extend our source language with the addition of extended algebraic data types, and lexically scoped annotations (Section 8.1.) To support these features, we broaden our type system, and generalise the inference system by adding implication constraints (Section 8.2.) We describe a solver for the implications that inference generates (Section 8.3), and show that the constraint-reasoning technology of earlier chapters can also be used to report new kinds of errors in this more advanced setting (Section 8.4.) Finally, we discuss a possible modification to the implication solver which in practice may make error reports easier to understand (Section 8.5.) The inference scheme and solver presented in this chapter are based on [85], where further results not directly relevant to the issue of type error diagnosis may be found. This chapter serves primarily to demonstrate that the idea of constraint reasoning for type error diagnosis scales to a more demanding setting.

8.1 Extending the source language

We begin by describing an extended version of the core language first presented in Chapter 4. The complete language appears in Figure 8.1.

One important difference is that we now extend the declaration of data types to allow for local variables. These *existential* data types, first considered by Läufer and Odersky [53], add considerably to the expressiveness of ordinary algebraic

$$
\begin{array}{llll}
\text{Declaration} & d & ::= & \mathsf{class}\ (Ctx \Rightarrow (U\ t_1\ \ldots\ t_n))_{l_1} \mid fd_1, ..., fd_n \\
 & & & \quad \mathsf{where}\ f :: (C \Rightarrow t)_{l_2} \\
 & & & \mid \mathsf{instance}\ Ctx \Rightarrow (U\ t_1\ \ldots\ t_n)_{l_0} \\
 & & & \mid \mathsf{data}\ T\ \bar{a} = \forall\ \bar{a}.Ctx \Rightarrow K_1\ \bar{t} \mid ... \mid \forall\ \bar{a}.Ctx \Rightarrow K_n\ \bar{t} \\
\text{Context} & Ctx & ::= & U_1\ \overline{t_1}, ..., U_n\ \overline{t_n} \\
\text{FD} & fd & ::= & \bar{a}\text{->}_l\ \bar{a} \\
\text{Expressions} & e & ::= & f_l \mid x_l \mid (\lambda x_l.e)_l \mid (e\ e)_l \mid \mathsf{let}\ f = e\ \mathsf{in}\ e \mid \\
 & & & \mathsf{let}\ \begin{array}{l}(f :: \sigma)_l \\ f = e\ \mathsf{in}\ e\end{array} \mid \mathsf{let}\ \begin{array}{l}(f ::: \sigma)_l \\ f = e\ \mathsf{in}\ e\end{array} \mid \\
 & & & (\mathsf{case}\ e\ \mathsf{of}\ [p_i \rightarrow e_i]_{i \in I})_l \\
\text{Patterns} & p & ::= & x_l \mid (K_l\ p_1\ ...\ p_n)_l \\
\text{Types} & t & ::= & a \mid t \rightarrow t \mid T\ \bar{t} \\
\text{Type Schemes} & \sigma & ::= & t \mid \forall \bar{\alpha}.C \Rightarrow t
\end{array}
$$

Figure 8.1: The core language extended

---

data types. We also allow arbitrary context constraints to be associated with data types. This includes type class constraints, giving us the language feature described later by Läufer [52], as well as equations, which allow us to directly encode the guarded recursive data types (GRDTs) of Xi, et al.[90]. Chapter 2 contains a number of examples of these features.

We retain the `::` (universal) annotations of earlier, while adding a new kind of type annotation, and lifting the restriction that declared types must be closed. Types declared with a `:::` annotation can be used to existentially bind type variables, and to introduce partial type information. Type variables now scope over let-bound definitions, i.e. type annotations can no longer be assumed to be implicitly fully quantified. A variable in a nested annotation which shares the same name now refers to the same type. In our language `::` and `:::` bound variables share the same scope. For example, a variable of the same name first bound in a `:::` and then re-appearing in a nested `::` annotation is the very same variable, and remains existentially quantified by the outer annotation.

Example 67 The following program makes use of lexically scoped type variables, and combines `::` and `:::` annotations.

```
replace :: Eq a => a -> a -> [a] -> [a]
replace x y xs = let rep ::: [a] -> b
                     rep [] = []
```

```
        rep (z:xs) = if z == x then y : rep xs
                               else x : rep xs
in  rep xs
```

The `replace` function is given a `::` annotation, which says replace has type $\forall a.Eq\ a \Rightarrow a \rightarrow a \rightarrow [a] \rightarrow [a]$. This annotation has the same meaning in Haskell.

The nested function `rep` has been given a `:::` annotation; a feature unique to our language. The `a` in `rep`'s type refers to the same type variable as the `a` in `replace`'s declared type. In our system, `rep`'s type is interpreted as $\exists b.[a] \rightarrow b$.

Note that had we written `rep`'s annotation as `rep :: [a] -> b`, this would be interpreted in our system as $\forall b.[a] \rightarrow b$, which is clearly too polymorphic, as `rep` must return a $[a]$. The same annotation in Haskell, would stand for $\forall a, b.[a] \rightarrow b$, which is even more polymorphic, and therefore even less correct.

□

8.2 A more-general inference framework

The typing rules for our language with lexically-scoped type annotations and extended algebraic data types can be found in Figure 8.2. We have only presented the new rules and the more interesting of those which have been updated from Chapter 4. Those which have been omitted are only trivially modified versions of the original rules.

We extend the typing judgements of Chapter 4 to accommodate an additional parameter V. We now write $C, V, \Gamma \vdash e : \sigma$, where C is a set of scoping constraints, V contains all type variables in scope, Γ maps term variables to type schemes, e is an expression, and the result σ is a type scheme. We again assume that all data constructors have been pre-processed and that their types are known in advance. The type of a constructor K is represented by $K : \forall \bar{a}, \bar{b}.D \Rightarrow t'_1 \rightarrow ... \rightarrow t'_n \rightarrow T\ \bar{a}$, where $\bar{a} \cap \bar{b} = \emptyset$. All $\bar{a}$ variables are required to appear in the constructor's result type, while $\bar{b}$, which are the local 'existential' variables, must not. The constraint D comes straight from the data constructor's context in the data declaration (which is how we supply 'guard' equations and type class constraints.)

The (Eq) rule allows us to interchange types which are equivalent with respect to the current constraint C and 'program theory' P; the symbol $\supset$ stands for logical implication. The (Case) and (Alt) rules are the same as before, except

$$\text{(Eq)}\quad \frac{C, V, \Gamma \vdash e : t_1 \quad P \models C \supset t_1 = t_2}{C, V, \Gamma \vdash e : t_2} \qquad \text{(Alt)}\quad \frac{p : t_1 \vdash \forall \bar{b}.(D|\Gamma_p) \quad fv(C, V, \Gamma, t_2) \cap \bar{b} = \emptyset \quad C \wedge D, \Gamma \cup \Gamma_p \vdash e : t_2}{C, V, \Gamma \vdash (p \to e)_l : t_1 \to t_2}$$

$$\text{(Case)}\quad \frac{C, V, \Gamma \vdash e : t_1 \quad C, V, \Gamma \vdash (p_i \to e_i)_{l_i} : t_1 \to t_2 \quad \text{for } i \in I}{C, V, \Gamma \vdash (\mathsf{case}\ e\ \mathsf{of}\ [(p_i \to e_i)_{l_i}]_{i \in I})_l : t_2}$$

$$\text{(Pat-K)}\quad \frac{K : \forall \bar{a}, \bar{b}.D \Rightarrow t'_1 \to ... \to t'_n \to T\ \bar{a} \quad \bar{b} \cap \bar{a} = \emptyset \quad p_i : [\bar{t}/\bar{a}]t'_i \vdash \forall \bar{b'_i}.(D'_i|\Gamma_{p_i}) \quad \text{for } i = 1, ..., n}{K\ p_1 ... p_n : T\ \bar{t} \vdash \forall \bar{b'_1}, ..., \bar{b'_n}, \bar{b}.(D'_1 \wedge ... \wedge D'_n \wedge [\bar{t}/\bar{a}]D|\Gamma_{p_1} \cup ... \cup \Gamma_{p_n})}$$

$$\text{(Let)}\quad \frac{C_1, V, \Gamma \vdash e_1 : t_1 \quad \bar{a} = fv(C_1, t_1) - fv(C_2, V, \Gamma) \quad C_2, V, \Gamma.g : \forall \bar{a}.C_1 \Rightarrow t_1 \vdash e_2 : t_2}{C_2, V, \Gamma \vdash \mathsf{let}\ g = e_1\ \mathsf{in}\ e_2 : t_2}$$

$$\text{(LetA)}\quad \frac{\bar{a} = fv(C_1, t_1) - fv(C_2, V, \Gamma) \quad C_2 \wedge C_1, V.\bar{a}, \Gamma.(g : \forall \bar{a}.C_1 \Rightarrow t_1) \vdash e_1 : t_1 \quad C_2, V, \Gamma.(g : \forall \bar{a}.C_1 \Rightarrow t_1) \vdash e_2 : t_2}{C_2, V, \Gamma \vdash \mathsf{let}\ \begin{array}{l} g :: C_1 \Rightarrow t_1 \\ g = e_1 \end{array}\ \mathsf{in}\ e_2 : t_2}$$

$$\text{(LetA2)}\quad \frac{\bar{a} = fv(C, t) - fv(V, \Gamma) \quad C_1, V.\bar{a}, \Gamma \vdash e_1 : t_1 \quad P_p \models C_1 \supset \exists \bar{a}.(C \wedge t = t_1) \quad \bar{b} = fv(C_1, t_1) - fv(V, \Gamma) \quad C_2, V, \Gamma.(g : \forall \bar{b}.C_1 \Rightarrow t_1) \vdash e_2 : t_2}{C_2, V, \Gamma \vdash \mathsf{let}\ \begin{array}{l} g ::: C \Rightarrow t \\ g = e_1 \end{array}\ \mathsf{in}\ e_2 : t_2}$$

Figure 8.2: Hindley/Milner with LSAs and EADTs

that they now make use of a slightly more involved auxiliary judgement for typing patterns. We write $p : t \vdash \forall \bar{b}.(D|\Gamma_p)$ to indicate that pattern p has type t, and introduces some constraints D and locally quantified variables $\bar{b}$, as well as an extended type environment Γ_p. The $\bar{b}$ correspond to local, 'existential' arguments of a data constructor. In the (Alt) rule, the statement $fv(C, V, \Gamma, t_2) \cap \bar{b} = \emptyset$

$$\begin{array}{lrcl} \text{Primitive} & p & ::= & (t_1 = t_2)_J \mid (U\ \overline{t})_J \mid True_J \\ \text{Constraint} & c & ::= & p \mid c \wedge c \mid \forall \bar{a}(\bigwedge \overline{p} \supset \exists \bar{b}.c)_J \end{array}$$

Figure 8.3: Extended constraint domain

prevents instantiation of any of the quantified variables. The same is achieved in (Pat-K) by limiting the variables which can be instantiated to the $\bar{a}$ variables, in the judgement $p_i : [\bar{t}/\bar{a}]t'_i \vdash \forall \bar{b'_i}.(D'_i|\Gamma_{p_i})$.

In the Let rules, type scheme generalisation has been modified so that type variables bound outside of the current let binding are not quantified, i.e. we remove V from $\bar{a}$ (the set of universally quantified variables.) The rules (LetA) and (LetA2), for handling :: and ::: annotations respectively, both update V with any fresh type variables. Both forms of type annotation share the same scope. A further restriction of the (LetA2) rule is that we must ensure that the annotation is satisfiable, this is captured by the logical implication $P_p \models C_1 \supset \exists \bar{a}.(C \wedge t = t_1)$, where P_p is some 'program theory' in our logic (which in practice we will encode as CHR rules.) Note that this extra step is not required in the (LetA) case, since there the type that is declared is the type that will actually be used. The Let rules are otherwise the same as those of Chapter 4.

8.2.1 Constraint and CHR generation

It is well understood that the typing problem for EADTs can be represented by implication constraints [76, 85]. In this section we extend the constraint and CHR based inference scheme of Chapter 4 with implications which we use to introduce temporary assumptions about types in the program, in particular declared types and guard constraints. In a later section we will describe a solving scheme based on an underlying CHR solver which we will use to solve these implications.

We extend our constraint domain, as shown in Figure 8.3. All other constraints, and associated operations, from Chapter 3 have been retained. We write implication constraints with an infix $\supset$ (implication) symbol, and optional sets of universally and existentially quantified type variables. We assume that $\bar{a} \cap \bar{b} = \emptyset$. Often we will omit the explicit existential quantifier, in which case it can be assumed that any variables unique to the right-hand side are existentially quantified there. Note that the left-hand side of an implication constraint can consist only of primitives, while the right may also contain further nested

implications.

Logically, a lone implication constraint $\forall \bar{a}.(C \supset \exists \bar{b}.D)$ has the same meaning as the equivalent statement in first order logic, i.e. $\models \forall \bar{a}.(C \supset \exists \bar{b}.D)$. Note that, in practice, we will impose the additional restriction that both sides of the implication are themselves satisfiable. This condition will be necessary since we still wish to reject ill-typed expressions even if their constraints are part of an implication.

One important difference between our new constraint generation scheme and the old is that we now use ternary predicate symbols to represent the types of functions. Where we once wrote $f(t, x)$, we now use $f(t, x, v)$. The v component corresponds to the V of the typing rules in the previous section. We will use v in much the same way as we used x, i.e. to keep track of variables in scope. In this case we record type variables, rather than λ-bound variables.

The constraint generation process, which we will now describe, is presented in its entirety in Figure 8.4. We have split this figure into three parts, grouping related rules. The first group contains versions of the standard Hindley/Milner rules for expressions other than let and case.

Once again we formulate constraint generation as a logical deduction system, this time with clauses of the form $V, E, \Gamma, e \vdash_{Cons} (C \mid t)$ where environments V, E and Γ, and expression e are input parameters and constraint C and type t are output parameters. Environments E and Γ perform the same functions as in the corresponding rules of Chapter 4, i.e. environment E consists of all let-defined and pre-defined functions, and environment Γ contains all lambda-bound variables. As earlier, the predicate symbol f refers to the inferred type of a let-bound variable `f`, while f_a represents `f`'s programmer-annotated, 'universal' type (if one is provided.), i.e. a type introduced by a `::` annotation. We will not generate an annotation-specific CHR rule in the case of a `:::` annotation.

To these we add the environment V which plays a similar role to Γ, except that it keeps track of all bound *type* variables in scope. Like Γ, we depend on the order of elements in V and thus treat it as a list, though we occasionally represent it as a set. New entries are always added to the end of V. i.e. if $V = \{a_1, ..., a_n\}$, then $V.a = \{a_1, ..., a_n, a\}$; the same goes for $V \cup \{a\}$. Since all type variables in our language share the same scope, regardless of where they are bound, V will contain both universally (::) and existentially (:::) bound variables.

Again, each sub-expression gives rise to a constraint which is justified by the location attached to it, and we enforce the invariant that for each location l a

$$\text{(Var-x)}\quad \frac{(x:t)\in\Gamma \quad t_l \text{ fresh}}{V,E,\Gamma,x_l \vdash_{Cons} (t_l = t \mid t_l)}$$

$$\text{(Var-f)}\quad \frac{\begin{array}{c} f\in E \quad \Gamma = \{\overline{x:t_x}\} \quad V=\bar{a} \quad t_l,x,v \text{ fresh} \\ C = \{f(t_l,x,v)_l, x = \langle\overline{t_x}\rangle, v = \langle\bar{a}\rangle\}\end{array}}{V,E,\Gamma,f_l \vdash_{Cons} (C \mid t_l)}$$

$$\text{(VarA-f)}\quad \frac{\begin{array}{c} f_a\in E \quad \Gamma = \{\overline{x:t_x}\} \quad V=\bar{a} \quad t_l,x,v \text{ fresh} \\ C = \{f_a(t_l,x,v)_l, v = \langle\bar{a}\rangle\}\end{array}}{V,E,\Gamma,f_l \vdash_{Cons} (C \mid t_l)}$$

$$\text{(Abs)}\quad \frac{\begin{array}{c} V,E,\Gamma.x:t_{l_1},e \vdash_{Cons} (C \mid t) \quad t_{l_1},\, t_{l_2},\, t' \text{ fresh} \\ C' = C \wedge (t_{l_2} = t' \to t)_{l_2} \wedge (t_{l_1} = t')_{l_1}\end{array}}{V,E,\Gamma,(\lambda x_{l_1}.e)_{l_2} \vdash_{Cons} (C' \mid t_{l_2})}$$

$$\text{(App)}\quad \frac{\begin{array}{c} V,E,\Gamma,e_1 \vdash_{Cons} (C_1 \mid t_1) \\ V,E,\Gamma,e_2 \vdash_{Cons} (C_2 \mid t_2) \\ C = C_1 \wedge C_2 \wedge t_1 = (t_2 \to t)_l \wedge (t_l = t)_l \quad t,t_l \text{ fresh}\end{array}}{V,E,\Gamma,(e_1\ e_2)_l \vdash_{Cons} (C \mid t_l)}$$

Figure 8.4: Justified constraint generation for Hindley/Milner with LSAs and EADTs

constraint of the form $(t_l = t)_l$ for some t, is generated.

The (Var-x) rule simply looks up the type of a λ-bound variable x_l in the environment Γ, and returns a new variable standing for the type at l, constrained to the type found in the environment. We use (Var-f) for calls to unannotated let-bound variables. As usual, we pass in the current Γ environment to the called CHR. In addition, and for much the same reason, since type variables now scope over definitions, we also pass in the V environment. (VarA-f) is applicable when a universally annotated function is being called. We call the corresponding annotation CHR with the current list of type variables in scope. The (Abs) and (App) rules are the same as earlier, except that we additionally thread through the variable environment V.

The next batch of rules concerns pattern matching. In the (Case) rule we collect all constraints on the alternatives, and unify their pattern types with the type of the scrutinised expression.

$$\text{(Case)}\quad \frac{\begin{array}{c} V, E, \Gamma, e \vdash_{Cons} (C_e \mid t_e) \\ V, E, \Gamma, (p_i \to e_i)_{l_i} \vdash_{Cons} (C_i \mid t_i) \quad \text{for } i \in I \\ C = \left\{\bigcup_{i \in I}(t_i = t_e \to t' \wedge C_i)\right\} \wedge (t_l = t')_l \wedge C_e \quad t', t_l \text{ fresh} \end{array}}{V, E, \Gamma, (\mathsf{case}\ e\ \mathsf{of}\ [p_i \to e_i]_{i \in I})_l \vdash_{Cons} (C \mid t_l)}$$

$$\text{(Alt)}\quad \frac{\begin{array}{c} p \vdash_{Cons} \forall \bar{b}(D \mid \Gamma_p), (C_p \mid t_p) \\ V, E, \Gamma', e \vdash_{Cons} (C_e \mid t_e) \qquad \Gamma' = \Gamma \cup \Gamma_p \\ C = \{\forall \bar{b}.(D \supset C_p \wedge C_e) \wedge (t_l = t_p \to t_e)_l\} \end{array}}{V, E, \Gamma, (p \to e)_l \vdash_{Cons} (C \mid t_l)}$$

$$\text{(Pat-Var)}\quad \frac{t_l, t_x \text{ fresh}}{x \vdash_{Cons} \forall \emptyset(True \mid \{x : t_x\}), ((t_l = t_x)_l \mid t_l)}$$

$$\text{(Pat-K)}\quad \frac{\begin{array}{c} K : \forall \bar{a}, \bar{b}.D \Rightarrow t_K \qquad I = \{1, ..., n\} \\ p_i \vdash_{Cons} \forall \bar{b_i}.(D_i \mid \Gamma_i), (C_i, t_i) \quad \text{for } i \in I \\ \Gamma' = \bigcup_{i \in I} \Gamma_i \quad B = \bar{b} \cup (\bigcup_{i \in I} \overline{b_i}) \\ D' = D \cup (\bigcup_{i \in I} D_i) \quad C' = C \cup (\bigcup_{i \in I} C_i) \\ C'' = C' \cup \left\{ \begin{array}{c} t_K = t_1 \to ... \to t_n \to t'_{l_2} \wedge \\ (t_{l_1} = t_K)_{l_1} \wedge (t_{l_2} = t'_{l_2})_{l_2} \end{array} \right\} \end{array}}{(K_{l_1}\ p_1\ ...\ p_n)_{l_2} \vdash_{Cons} \forall B.(D' \mid \Gamma'), (C'' \mid t_{l_2})}$$

Figure 8.4: *(continued)*

For the Pat rules we make use of a slightly modified judgement of the form $p \vdash_{Cons} \forall \bar{a}.(D \mid \Gamma), (C \mid t)$, where p is a pattern, $\bar{a}$ is a set of polymorphic variables, D is a set of 'local' constraints, Γ is an environment binding variables in p, C is a set of 'global' constraints, and t is a type. Note that since patterns can only bind new variables, and not refer to variables already in scope, there is no need for any 'input' environments. The distinction between C and D is important, as we will see, since there may be some constraints which we require to remain within the scope of the pattern binding, while others are allowed to float outwards.

In the (Pat-Var) rule we simply generate a fresh variable for the newly-bound pattern variable, as well as returning this binding in a type environment. Since λ-bound variables are monomorphic, there are no polymorphic variables to account for in this case. In (Pat-K) we generate constraints out of a constructor

$$\text{(Let)} \quad \frac{V, E.f, \Gamma, e_2 \vdash_{Cons} (C \mid t)}{V, E, \Gamma, \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \vdash_{Cons} (C \mid t)}$$

$$\text{(LetA2)} \quad \frac{V, E.f, \Gamma, e_2 \vdash_{Cons} (C' \mid t')}{V, E, \Gamma, \mathsf{let}\ \begin{array}{l}(f ::: C \Rightarrow t)_l \\ f = e_1\end{array}\ \mathsf{in}\ e_2 \vdash_{Cons} (C' \mid t')}$$

$$\text{(LetA)} \quad \frac{\begin{array}{c}V, E.f_a, \Gamma, e_2 \vdash_{Cons} (C' \mid t') \\ \Gamma = \overline{x : t} \quad V = \bar{a} \quad \bar{b} = fv(C'', t'') - V \quad t, l, v \text{ fresh} \\ C = \{C', \forall \bar{b}.(C''_l, t = t''_l, l = \langle \bar{t} \rangle, v = \langle \bar{a}, \bar{b} \rangle \supset f(t, l, v))\}\end{array}}{V, E, \Gamma, \mathsf{let}\ \begin{array}{l}(f :: C'' \Rightarrow t'')_l \\ f = e_1\end{array}\ \mathsf{in}\ e_2 \vdash_{Cons} (C \mid t')}$$

Figure 8.4: *(continued)*

pattern. Most of the complexity of the rule arises from the fact that we allow constructors to have an arbitrary number (n in the rule) of sub-patterns. The rule is otherwise straightforward, in that it processes all sub-patterns, accumulates all constraints and variables, and unifies the types of the individual patterns with the constructor's arguments.

We use the (Alt) rule to process an individual case alternative. We generate all constraints, variables and the type of the pattern, and then proceed to translate the right-hand side expression under the new extended type environment. We produce an implication constraint in order to keep the D constraints local to this expression. The accumulated set of polymorphic variables is also put to use here.

Finally, we come to the third group of rules which concern let expressions. The (Let) rule is essentially the same as in Chapter 4, extended with the V parameter. We use (LetA2) in the case of ::: annotated let-bindings. Note that the effect of the annotation is not significant here, since we are only interested in the body ($e2$ of the let-expression.) We will take care of the annotation when generating the CHR rule for this binding. Finally, the (LetA) rule is used for handling :: annotated bindings. We make use of an implication to encode the subsumption condition on the annotated and inferred types. The variables $\bar{b}$ are newly bound, and because it is a :: annotation, are universal. These are passed in via the v component in the call $f(t, x, v)$, and are polymorphic at the point of

$$\text{(Var)} \qquad V, E, \Gamma, v \vdash_{Def} \emptyset$$

$$\text{(App)} \qquad \frac{V, E, \Gamma, e_1 \vdash_{Def} P_1 \quad V, E, \Gamma, e_2 \vdash_{Def} P_2}{V, E, \Gamma, e_1\ e_2 \vdash_{Def} P_1 \cup P_2}$$

$$\text{(Abs)} \qquad \frac{V, E, \Gamma.x : t, e \vdash_{Def} P \quad t \text{ fresh}}{V, E, \Gamma, \lambda x.e \vdash_{Def} P}$$

$$\text{(Case)} \qquad \frac{V, E, \Gamma, e \vdash_{Def} P \quad p_i \vdash_{Cons} (\_ \mid \_ \mid \Gamma_{p_i}) \quad V, E, \Gamma \cup \Gamma_{p_i}, e_i \vdash_{Def} P_i \quad \text{for } i \in I}{V, E, \Gamma, (\mathsf{case}\ e\ \mathsf{of}\ [p_i \to e_i]_{i \in I})_l \vdash_{Def} P \cup \bigcup_{i \in I} P_i}$$

Figure 8.5: CHR rule generation for Hindley/Milner with LSAs and EADTs

the implication.

We now turn to the CHR generation rules. As before, we generate a single simplification rule which represents the inferred type of a let-bound variable, as well as a simplification rule for the (optional) annotated universal type. We use the same $\vdash_{Def}$ judgements, as in Chapter 4, but similarly extended with a parameter V for tracking type variables. The rules for generating CHRs out of expressions are presented in Figure 8.5 (which has also been split into a number of parts.)

The initial couple of rules are trivial. No CHR rule can be generated from a variable expression (Var), and in the case of application (App) we simply collect any rules generated from sub-expressions. The situation for abstraction (Abs) and case expressions (Case) is similar, except that we must take care to extend the type environment Γ with any freshly-bound variables, before descending to sub-expressions.

The rules for dealing with let expressions are more interesting. The cases for unannotated (Let) and $::$ annotated (LetA2) bindings are almost the same. In both cases we generate constraints from the definition expression under an E environment extended with f, since we will refer to this definition by the predicate symbol f. In the annotated case we also extend V with the newly-introduced type variables $\bar{b}$, and also add the constraints represented by the declared type scheme to the body of the simplification rule. As usual, we add a constraint

$$\text{(Let)}\quad \frac{\begin{array}{c}\Gamma = \overline{x : t} \quad V = \bar{a} \quad t, x, v, xr, vr \text{ fresh} \\ V, E.f, \Gamma, e_1 \vdash_{Def} P_1 \quad V, E.f, \Gamma, e_2 \vdash_{Def} P_2 \\ V, E, \Gamma, e_1 \vdash_{Cons} (C_1 \mid t_1) \\ P_3 = \{f(t, x, v) \iff x = \langle \bar{t} \mid xr \rangle, v = \langle \bar{a} \mid vr \rangle, t = t_1, C_1\}\end{array}}{V, E, \Gamma, \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \vdash_{Def} P_1 \cup P_2 \cup P_3}$$

$$\text{(LetA2)}\quad \frac{\begin{array}{c}\Gamma = \overline{x : t} \quad V = \bar{a} \quad \bar{b} = fv(C', t') - V \\ V.\bar{b}, E.f, \Gamma, e_1 \vdash_{Def} P_1 \quad V, E.f, \Gamma, e_2 \vdash_{Def} P_2 \\ V.\bar{b}, E, \Gamma, e_1 \vdash_{Cons} (C'' \mid t'') \quad t, x, v, xr, vr \text{ fresh} \\ P_3 = \{f(t, x, v) \iff x = \langle \bar{t} \mid xr \rangle, v = \langle \bar{a} \mid vr \rangle, t = t'', C'', (t = t')_l, C'_l\}\end{array}}{V, E, \Gamma, \mathsf{let}\ \begin{array}{l}(f ::: C' \Rightarrow t')_l \\ f = e_1\end{array}\ \mathsf{in}\ e_2 \vdash_{Def} P_1 \cup P_2 \cup P_3}$$

$$\text{(LetA)}\quad \frac{\begin{array}{c}\Gamma = \overline{x : t} \quad V = \bar{a} \quad \bar{b} = fv(C', t') - V \\ V.\bar{b}, E.f_a, \Gamma, e_1 \vdash_{Def} P_1 \quad V, E.f_a, \Gamma, e_2 \vdash_{Def} P_2 \\ V.\bar{b}, E.f_a, \Gamma, e_1 \vdash_{Cons} (C'' \mid t'') \quad t, x, v, xr, vr \text{ fresh} \\ P_3 = \left\{ \begin{array}{lcl} f_a(t, x, v) & \iff & v = \langle \bar{a} \mid vr \rangle, (t = t')_l, C'_l \\ f(t, x, v) & \iff & x = \langle \bar{t} \mid xr \rangle, v = \langle \bar{a}, \bar{b} \mid vr \rangle, t = t'', C'' \end{array} \right\}\end{array}}{V, E, \Gamma, \mathsf{let}\ \begin{array}{l}(f :: C' \Rightarrow t')_l \\ f = e_1\end{array}\ \mathsf{in}\ e_2 \vdash_{Def} P_1 \cup P_2 \cup P_3}$$

Figure 8.5: *(continued)*

$x = \langle \bar{t} \mid xr \rangle$ to each rule to project out the scoped λ-bound variables which will be passed in from instantiation/call sites. We now also add an analogous constraint $v = \langle \bar{a} \mid vr \rangle$ in the annotated case.

In the (LetA) rule we handle :: annotated bindings. For these we generate two separate CHR rules as before: one rule for the inferred type, and another for the provided. Note that as in the (LetA2) rule, we include the declared type in the inference CHR rule. The projection constraints on x and v are as usual, but since the annotation cannot contain references to λ-bound variables, we omit the x constraint from that rule.

We treat class and instance declarations as we did in Section 4.3.3, generating the appropriate CHR rules for super-class relations and functional dependencies.

8.3 Implication solving

We now describe an implication solving scheme in terms of our existing CHR solver (defined in Chapter 3.) The scheme places necessary restrictions on the form of solutions, but does not specify how such solutions are to be found. Thus completeness depends on the details of a specific solver, that is, a specific instantiation of the presented scheme. Although the detail of the solving step is omitted, we will refer to the scheme as a 'solver', even when we actually mean some specific implementation. We denote a single implication solving step from constraint C to C' by $C \gg_P C'$, where P is a set of CHR rules. Solving C successfully requires repeated application, $C \gg_P^* C''$, such that C'' contains only primitives.

Let P be a set of CHRs and F an implication constraint, and consider the following cases.

Primitive: If $F \equiv \exists \bar{a}.C$, then $F \gg_P C'$ where $C \longrightarrow_P^* C'$.

General: Otherwise $F \equiv C_0, (\forall \bar{a}.D \supset \exists \bar{b}.F_1), F_2$ where C_0 is a conjunction of primitive constraints, D is a set of assumptions and F_1 and F_2 are implication constraints.

We compute (1) $C_0, D \longrightarrow_P^* D'$ and (2) $C_0, D, F_1 \gg_P^* C'$ for some D' and C', and distinguish among the following cases.

Solved: We define $F \gg_P C_0, F_2$ if $\models (\bar{\exists}_V.D') \leftrightarrow (\bar{\exists}_V.C')$ where $V = fv(C_0, D, \bar{a})$.

Add: We define $F \gg_P S, F$ if $\models (\bar{\exists}_V.D') \not\leftrightarrow (\bar{\exists}_V.C')$ where $V = fv(C_0, D, \bar{a})$ and $\mathit{False} \notin C'$ and S is a conjunction of primitive constraints such that (3) $P \models C' \supset S$, (4) $fv(S) \cap \bar{a} = \emptyset$ and (5) $P \models D' \not\supset S$.

Failure: We define $F \gg_P \mathit{False}$ in all other cases.

In the **Primitive** step we apply standard CHR solving, since F contains no implications. In the **General** step, we split the constraint store into: C_0 containing sets of primitive constraints, a single implication constraint $(\forall \bar{a}.D \supset \exists \bar{b}.F_1)$, and F_2 containing the remaining implication constraints.

The **Solved** step applies if the equivalence check succeeds. Consequently, the constraint $(\forall \bar{a}.D \supset \exists \bar{b}.F_1)$ is removed from the store. Note that w.l.o.g. we assume

that $\bar{b} = fv(F_2) - fv(\bar{a}, C_0)$. Any variable not bound by a universal quantifier is (implicitly) existentially bound.

The **Add** step is the most interesting and applies if equivalence checking failed. We perform some abductive reasoning [57] to find a constraint S which may help us to eventually solve the implication. We refer to S as a *partial solution*. Conditions (3-5) give only a characterisation of the primitive constraints implied by the final store of derivation (2) which may prove useful in solving the implication $(\forall \bar{a}.D \supset \exists \bar{b}.F_1)$. Any partial solution must satisfy at least the following criteria. Condition (3) ensures that we only add in useful constraints. Condition (4) prevents the universally bound variables $\bar{a}$ from escaping. Condition (5) stops us from adding redundant constraints.

We can state that any solver satisfying the above description is sound [85]. If P is a set of CHRs, F an implication constraint and C a set of primitive constraints such that $F \gg^*_P C$, then, $P \models C \supset F$.

Termination is another important property. Obviously, the underlying CHR program must be terminating, but this is not sufficient to guarantee termination of the implication solver. A subtle problem is that the above CHR implication solving scheme allows the addition of repeated, renamed copies of essentially the same constraints. Consider solving $(Bar\ a \supset Foo\ a)$ where $P = \{Foo\ a \Longrightarrow a = [b]\}$. We will find that $Bar\ a \supset Foo\ a \gg a = [b_1], (Bar\ a \supset Foo\ a) \gg a = [b_1], a = [b_2], (Bar\ a \supset Foo\ a), ...$ is a valid derivation.

We can state termination under the condition that partial solutions are variable-restricted. Formally, partial solution S is *variable-restricted* iff (6) $fv(S) \subseteq fv(C_0, D) - \bar{a}$. In essence, we prevent the addition of any fresh variables.

Type inference is performed as described in Section 4.2.2, with the one difference being that we use the implication solver $\gg$, rather than the CHR solver $\longrightarrow$. From an expression e we generate constraints C, and CHR rules P, and perform $C \gg^*_P D$. The inferred type of e can be obtained from D as before. It can be shown that implication solving yields sound type inference for extended algebraic data types, though we omit details here. Please consult [85] for more information.

Implication solving may fail for a number of reasons. In the simplest case, any of the underlying CHR solving steps may fail, i.e. in the **Primitive** case, or (1) or (2) of the **General** case. More interestingly, solving may also fail if the implication is not trivially **Solved**, and no partial solution can be found that satisfies the criteria of the **Add** step. As we will see, this failure can be diagnosed

in a number of different ways.

8.3.1 Justifying newly added constraints

A partial solution S to some implication constraint may contain constraints which are implied by the right-hand side of an implication, but do not actually appear therein. Consider the following justified implication constraint

$$F \equiv \forall a.((Bar\ a\ b\ c)_2 \supset (a = b)_3, (a = c)_4)_1$$

and the CHR rule

$$Bar\ a\ b\ b \iff (a = b)_5$$

We assume that F is nested within some other implication, where b and c are quantified, and that all of these variables are otherwise unconstrained. After derivation (1) of the **General** solver case, we have $(Bar\ a\ b\ c)_2$, and after (2) we have $(Bar\ a\ b\ c)_2 \wedge (a = b)_3 \wedge (a = c)_4$. At first glance it may not appear possible to solve this implication since neither $(a = b)_3$ nor $(a = c)_4$ can be in S, since both mention the universally bound variable a. A less obvious candidate for S, however, is the constraint $b = c$, which is clearly implied by $(a = b)_3 \wedge (a = c)_4$, and meets the conditions of the **Add** step. If we add $b = c$ to the (invisible) context, on our return to F we will find F **Solved**: derivation (1) will yield $b = c \wedge (a = b)_5$ (since the *Bar* rule will apply), and derivation (2) will result in $b = c \wedge (a = b)_5 \wedge (a = b)_3 \wedge (a = c)_4$, which is clearly logically equivalent.

The constraint $b = c$ does not itself appear directly in F, it was freshly generated during solving. This constraint is currently unjustified, which means that if it appears in a type explanation, either in a minimal unsatisfiable subset, or a minimal implicant, we will have no source locations to link it back to. This would be problematic, since the value of our type explanations lies primarily in their completeness.

We amend the **Add** step of the implication solver as follows, adding a new condition to ensure that new constraints are properly justified. (We include the termination condition (6), though it is not related to re-justification.)

Add: We define $F \gg_P S, F$ if $\models (\bar{\exists}_V.D') \not\leftrightarrow (\bar{\exists}_V.C')$ where $V = fv(C_0, D, \bar{a})$ and $False \notin C'$ and S is a conjunction of primitive constraints such that (3) $P \models C' \supset S$, (4) $fv(S) \cap \bar{a} = \emptyset$, (5) $P \models D' \not\supset S$, (6) $fv(S) \subseteq fv(C_0, D) - \bar{a}$,

and (7) for every $s_J \in S$ if $s_J \notin C'$ then, let $M = \mathsf{min\_impl}(C', s_J)$, and $J = just(M)$.

Condition (7) requires that for each new constraint s_J, J should be derived from a minimal subset of C' which implies s_J. Recall that the function *just* concatenates the justifications of its argument constraints, and that min_impl is the minimal implicant procedure of Section 3.4.

Returning to our example above, under our amended **Add** rule, the newly added constraint $b = c$, would actually be $(b = c)_{[3,4]}$, since it is implied by $(a = b)_3$ and $(a = c)_4$.

In examples where explicit justifications are omitted for clarity, we will assume that condition (7) is silently enforced, allowing us to always establish a link between constraints and their contributing source locations.

8.4 Error reporting

We will now consider the sorts of errors that may arise during implication solving, and how they can be presented to the programmer. The following sections should be understood within the context of the solving scheme of Section 8.3.

8.4.1 Derivation failed

While attempting to solve a single implication constraint, we perform at least one, and possibly more CHR derivations. Programs which give rise to unsatisfiable constraints when typed under the system of Chapter 4 will cause failure in the **Primitive** case of the $\gg$ solver, since there are no implications involved.

In the **General** case, there are two possible sources of unsatisfiability. Given a set of CHRs P and a goal $C_0, (\forall \bar{a}.D \supset \exists \bar{b}.F_1), F_2$, the CHR derivation $C_0, D \longrightarrow_P^* D'$ may fail, with $D' = \mathit{False}$. If so, we stop and report the error in the same way that we would report any type conflict (see Chapter 6.) Alternatively, it is possible that $C_0, D, F_1 \gg_P^* C'$ could fail, with $C' = \mathit{False}$. Note that if the first derivation should fail, then we can stop immediately, since the second will also fail.[1]

Example 68 Consider the following simple program.

[1] Recall that we wish to exclude solutions that rely on unsatisfiability (See Section 8.2.)

```
class C a b | a -> b where c ::  a -> b
instance C Int Int

f ::  C Int Bool => Int -> Bool
f n = c n
```

The CHR rules which represent the functional dependency, and the instance are as follows:

$$\begin{aligned} C\ a\ b,\ C\ a\ b' &\implies b = b' \\ C\ Int\ b &\implies b = Int \\ C\ Int\ Int &\iff True \end{aligned}$$

The implication representing the type subsumption condition for this program would be of the form $(C\ Int\ Bool\ \supset\ f(t, x, v))$, where the predicate $f(t, x, v)$ represents the inferred type of `f`. Clearly, in attempting to solve this implication, the left-hand side will be found to be equivalent to *False*, which we must report as a type error.

```
f.hs:4: ERROR: Functional dependency causes type error
Types   : Bool
          Int
Problem : class C a b | a -> b ...
          instance C Int Int ...
          Enforce: C Int a ==> a = Int
          On constraint:
             C Int Bool (from line 4, col. 6)
Conflict: f ::  C Int Bool => Int -> Bool
```

□

If the first derivation succeeds, we continue on to the second, $C_0, D, F_1 \gg^*_P C'$. This is a recursive invocation of the implication solver, and as such, it will not fail directly itself, but only if some sub-derivation fails. It will fail with an unsatisfiability error if it depends on an unsatisfiable **Primitive** step, or an implication solving step where the first derivation fails, as above. Ill-typed expressions of Chapter 4, when placed on the right-hand side of some EADT pattern, or an annotation,[2] will cause unsatisfiability in this step.

[2]It is also possible that the addition of an incorrect annotation could lead to an error. The combination of an annotation and an ill-typed expression could also result in a slightly different error being reported than if the expression were alone.

Example 69 Consider the following definition of Haskell's `flip` function. The declared type is correct, but the body of `flip` is wrong; it ought to be `f y x`.

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = x y x
```

The implication constraint we generate from the above program, is of the form $\forall a, b, c.(t = (a \to b \to c) \to b \to a \to c \supset flip(t, x, v))$, where $flip(t, x, v)$ yields an unsatisfiable result. The error reported for the type-annotated version of this program is the same as that reported for the unannotated version.

```
flip.hs:2:  ERROR: Type error - one error found
Problem : Pattern variable 'x'(bound at line 2, col.  8)
          used with multiple types
Types   : a -> b -> c
          b
Conflict: flip :: (a -> b -> c) -> b -> a -> c
          flip f x y = x y x
```

□

8.4.2 Subsumption failure

Assume we are solving the goal $C_0(\forall \bar{a}.D \supset \exists \bar{b}.F_1)$, under CHR rules P, and that the derivations $C_0, D \longrightarrow_P^* D'$ and $C_0, D, F_1 \gg_P^* C'$ succeed. The implication solver can fail if this particular implication is not already solved, i.e. if the **Solved** case does not apply, and we cannot contribute a partial solution. If we look to the definition of the **Add** step, we see that it is limited to adding constraints which do not affect any of the universally quantified variables. For this step to fail, there must be constraints in C', which are not in D', that concern the universal variables $\bar{a}$. If this is the case then, the **Failure** case is inevitable.

One possible cause of this is some $C'' \subseteq C'$ which constrains the variables $\bar{a}$ further than D'. These C'' constraints (as we will also refer to them later) could be equations or type class constraints. We address error diagnosis for these two possibilities separately.

This kind of failure corresponds directly to the subsumption failure of Section 5.2. In this more general system, with explicit implication constraints, essentially the same error diagnosis applies equally to cases involving extended algebraic data types as it does to incorrect type annotations.

Unmatched user constraint

It is possible that there may be user constraints in C', not appearing in D', which would therefore have to be part of any solution, but cannot be because of the (necessary) conditions we have placed on the form of solutions. If such constraints exist, then clearly the implication cannot be solved, and we must report an error.

If there exist any type class constraints $U = \{u \mid u \in C'_u - D'_u, fv(\phi_{C'}(u)) \cap fv(\phi_{C'}(\bar{a})) \neq \emptyset\}$, then we can say that $C'' = \{u\}$, $u \in U$. That is, the minimal difference between C' and D' is a single type class constraint in C' which constrains a universal variable. We can report this as an 'unmatched constraint'.

Example 70 Consider the following program for testing whether two values are not equal.

```
notEq :: a -> a -> Bool
notEq x y = not (x == y)
```

The Haskell function `==` has type $\forall a.Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$, and `not` has the type we assigned it in the previous example.

In typing `notEq` we will need to solve an implication of the following form:

$$\begin{aligned}\forall a.(\ & a \rightarrow a \rightarrow Bool = t_1 \supset \\ & t_{eq} = b \rightarrow b \rightarrow Bool, Eq\ b, \\ & t_{eq} = t_x \rightarrow t_y \rightarrow t_2, t_{not} = Bool \rightarrow Bool, \\ & t_{not} = t_2 \rightarrow t_3, t_1 = t_x \rightarrow t_y \rightarrow t_3\ )\end{aligned}$$

We can immediately see that the right-hand side of the implication contains a type class constraint $Eq\ b$, and that in C', $b = a$. We can report $Eq\ a$ as an unmatched constraint, and suggest to the programmer that it should be added to `notEq`'s type annotation.

```
notEq.hs:1: ERROR: Inferred type does not subsume
                   declared type
Declared  : forall a. a -> a -> Bool
Inferred  : forall a. Eq a => a -> a -> Bool
Problem   : Constraint Eq a, from following location,
            is unmatched
            notEq :: a -> a -> Bool
            notEq x y = not (x == y)
Suggestion: Try adding Eq a to notEq's annotation
```

This is the same error message that the subsumption error reporting procedure of Section 5.2 would report. □

Example 71 Consider the following simple program which makes use of an existential data type.

```
class Foo a b where foo::a->b->Int
instance Foo a Int where foo _ _ = 1
data Bar = forall b. Mk b
f x z = (x, case z of (Mk y) -> foo y x)
```

In typing `f` we will need to solve a constraint of the following form:

$$t_f = t_x \rightarrow Bar \rightarrow t_1, t_1 = (t_x, t_2),$$
$$\forall t_y.(True \supset Foo\ t_y\ t_x, t_2 = Int)$$

We can immediately see that the right-hand side of the implication contains a type class constraint $Foo\ t_y\ t_x$, which does not appear on the left. We can report $Foo\ t_y\ t_x$ as an unmatched constraint, and suggest to the programmer that it should be added to `Mk`'s constraint context.

```
notEq.hs:1: ERROR: Unmatched user constraint
Problem   : Constraint Foo b, from following location,
            is unmatched
            f x z = (x::Int, case z of (Mk y) -> foo y x)
Suggestion: Try adding Foo b to Mk's context
```

□

If we include the termination condition (6), failure may also occur if there exists a user constraint $c \in C'$, and $c \notin D'$, such that $fv(c) - fv(C_0, D) \neq \emptyset$. This constraint c will be rejected because doing so will ensure that we do not add any new variables to the result. This in turn prevents the implication solver from getting stuck in a loop in which it simply adds renamed copies of c, without ever actually getting closer to a stable result.[3]

In order to resolve such a problem, the programmer may choose to supply additional type information, in the form of annotations, to ground any of these

[3]The same problem also occurs if there are equations in C' which contain new, non-C_0, non-D variables. Since we can handle this in exactly the same way as we do the case of user constraints, we do not consider it separately. i.e. it still amounts to finding the locations that contain these new variables, and suggesting the programmer should annotate them.

new, problematic variables. Thus, we report this kind of error by identifying where such annotations could be applied. This is essentially the same strategy we have already applied to reporting ambiguity errors in Section 5.3.

Example 72 Consider the following program.

```
class G a b where
      g :: a -> b


f :: (G a (b,c)) => a -> b
f x = fst (g x)
```

To represent the subsumption condition for `f`'s declared type, we would generate an implication constraint like the following $\forall a, b.(G\ a\ (b, c)\ , t = a \rightarrow b \supset fst(t))$ along with the simplified (x,v arguments omitted) CHR rules representing `f`'s inferred type, and `fst`:

$$\begin{aligned} f(t) &\iff t = t_x \rightarrow t_r, g(t_g), t_g = t_x \rightarrow t_p, fst(t_f), t_f = t_p \rightarrow t_r \\ fst(t) &\iff t = (a, b) \rightarrow a \end{aligned}$$

The problem here is that the right-hand side of the implication gives rise to a constraint $G\ a\ (b, c')$, which differs from the declared $G\ a\ (b, c)$ in the type of the second component of its tuple argument. Note that c' is existentially quantified.

We report the following

```
unmatched.ch:5: ERROR: Unmatched user constraint
Problem   : User constraint G a (b,c) from the following
            location is unmatched
            f x = fst (g x)
Suggestion: Add type information at one of the following locations
            f x = fst (g x)
```

Since we have lexically-scoped type variables in our language, one possible fix is:

```
f :: (G a (b,c)) => a -> b
f x = (fst::(b,c)->b) (g x)
```

In terms of the constraints mentioned earlier, this annotation instantiates the c' type variable of the inferred type class constraint, equating it with c. The declared and inferred constraints will then match.

□

Polymorphic variable instantiated

We return to our $C'' \subseteq C'$, which constrains universal variables $\bar{a}$ further than D'. It is possible that C'' may contain equations.

We need to check whether any universal variable which is uninstantiated in D' has been further instantiated in C'. Specifically, let $A = fv(\phi_{D'}(\bar{a}))$, where $\phi_{D'}$ is the mgu of D', and check if there exist any variables $\{a | a \in A \wedge inst(\phi_{C'}(a), A - a)\}$, where the predicate $inst(x, Y)$ is *True* if x is not a variable or $x \in Y$, i.e. if the first argument is ground at all or appears in the second. For a universal variable a, instantiated to t, we need to find a minimal $C'' \subseteq C'$ which is responsible. To do this, we employ the minimal implicant algorithm of Section 3.4, i.e. $C'' = \mathsf{min\_impl}(C', \{a = t\})$.

The constraints C'' can be blamed as the cause of the failure since 1) they distinguish C' from D', and 2) they cannot form a partial solution since they constrain variables in $\bar{a}$. We can then highlight the locations that gave rise to those constraints, which suggests to the programmer that those are the locations to consider when debugging.

Example 73 Consider the following program for testing whether two values are not equal.

```
notEq ::  Eq a => c -> a -> Bool
notEq x y = not (x == y)
```

We generate the following (simplified) implication in attempting to type this version of `notEq`. We have expanded, in-place the call to `not`, which has type $Bool \rightarrow Bool$.

$$\begin{aligned} \forall a, c.(\; & Eq \; a, c \rightarrow a \rightarrow Bool = t_1 \supset \\ & t_{eq} = b \rightarrow b \rightarrow Bool, Eq \; b, \\ & t_{eq} = t_x \rightarrow t_y \rightarrow t_2, t_{not} = Bool \rightarrow Bool, \\ & t_{not} = t_2 \rightarrow t_3, t_1 = t_x \rightarrow t_y \rightarrow t_3 \;) \end{aligned}$$

The problem here stems from the fact that C' contains constraints that imply $a = c$, but D' does not. We can diagnose this error by finding precisely those constraints, and identifying the source locations that gave rise to them.

```
notEq.hs:1: ERROR: Inferred type does not subsume
                   declared type
```

```
Declared: forall a.  Eq a => c -> a -> Bool
Inferred: forall a.  Eq a => a -> a -> Bool
Problem : The variable ‘c’ makes the declared type
          too polymorphic, wrt locations
          notEq ::  Eq a => c -> a -> Bool
          notEq x y = not (x == y)
```

□

Example 74 In the following simple program, the argument of an existential data type is instantiated in the body of the function `h`.

```
data K = forall a.  K a

h (K x) = not x
```

We would generate a CHR of the following, slightly simplified form to type `h`. Again, we have used `not`'s type directly.

$$h(t) \Longleftrightarrow t = K \rightarrow t_r, \forall a.(\mathit{True} \supset t_n = \mathit{Bool} \rightarrow \mathit{Bool}, t_x = a, t_n = t_x \rightarrow t_r)$$

It is clear from the above that the constraints arising from the call to `not` will instantiate t_x and therefore the polymorphic variable a.

We report this error as follows.

```
notEq.hs:1: ERROR: Existential type variable instantiated
Problem : The variable ‘x’ has an existential type which is
          instantiated by the following locations
          h (K x) = not x
```

□

8.4.3 Polymorphic variable escaped

The one remaining cause of failure, not addressed by the preceding sections, occurs when a universal variable $a \in \bar{a}$ is bound to some other variable which appears outside of the implication. This binding constraint cannot be let out during an **Add** step, as it contains an a, and so again **Failure** is inevitable. Programmers who are familiar with existential data types would be aware of this

restriction that existential variables are not allowed to 'escape'; i.e. the existential types representing components of some abstract data type cannot appear outside of the scope of the pattern in which they are bound.

We can identify when a set of unsolvable constraints represents this sort of error, and report it as such. Assume we are inferring the type of `f`, and therefore begin with an initial goal $f(t, x, v)$. After some number of **Primitive** CHR steps, we reach the **General** case of our solver. We have the goal $C_0, (\forall \bar{a}.D \supset \exists \bar{b}.F_1), F_2$, and perform two CHR derivations, $C_0, D \longrightarrow_P^* D'$ and $C_0, D, F_1 \gg_P^* C'$. Assume that the type we are inferring is represented by variable t, and let $\phi_{C'}$ be the mgu of C'. The set of existential variables which escape is $E = fv(\phi_{C'}(t)) \cap fv(\phi_{C'}(\bar{a}))$. If $E \neq \emptyset$ then the implication cannot be solved. We will inevitably arrive at the **Failure** step of our algorithm, since none of the constraints in C' which unify any of $\phi_{C'}(t)$'s variables with a quantified variable will ever appear in a partial solution generated by the **Add** step.

In reporting such an error, we should be able to identify precisely which program locations are responsible for the variable escape. For a variable $e \in E$, we need to find a minimal $C'' \subseteq C'$, with mgu $\phi_{C''}$, such that $e \in fv(\phi_{C''}(t))$. We can then report that the locations which gave rise to C'' are the cause of the error. We can calculate C'' using our minimal implicant algorithm.

Let ϕ_{C_0} be the mgu of C_0. First we find $V = fv(\phi_{C_0}(t))$, the set of variables appearing in the inferred type so far (before the implication is entered.) We need to narrow V down to just the variables which are eventually bound to a universal variable, $V' = \{v \mid v \in V, (\phi_{C'}(v) \cap \phi_{C'}(\bar{a})) \neq \emptyset\}$. We pick a single $v \in V'$ and let $C'' = \mathsf{min\_impl}(C', s)$, where $s = abstract(E, \phi_{C'}(v))$, and the *abstract* function is defined below.

$$abstract(E, t) = \begin{cases} t & \text{if } t \in E \\ a & \text{if } t \text{ is a variable } (t \notin E), \\ & \text{or } t \text{ is a constructor} \qquad (a \text{ fresh}) \\ t'_1\ t'_2 & \text{if } \exists t_1, t_2.t = t_1\ t_2, \text{ and} \\ & t'_1 = abstract(E, t_1),\ t'_2 = abstract(E, t_2) \end{cases}$$

The function *abstract* is defined inductively on the structure of types (not type schemes), and the call $abstract(E, t)$ simply returns the type t with all

constructors, and all non-E types replaced by fresh variables. The effect of this, in our use above, is to ensure that C'' contains only constraints which cause escaping variables E to appear in the inferred type. We are not interested in any other constraints which might also shape the type in which the escaping variable appears.

Example 75 Consider the following simplistic program. The `const` function has type $\forall a, b.a \rightarrow b \rightarrow a$.

```
data Key = forall a.  MkKey a
f (MkKey k) = const (k,'m') True
```

In inferring `getKey`'s type, we need to solve the following (simplified) constraint. The implication therein comes from the match against `MkKey`.

$$t_1 = Key\ a \rightarrow t_2,$$
$$\forall b.(\mathit{True} \supset t_c = a' \rightarrow b' \rightarrow a', t_c = (a, \mathit{Char}) \rightarrow \mathit{Bool} \rightarrow t_2)$$

The right-hand side of the implication implies $t_2 = b$, but this cannot be present in any partial solution. Our diagnosis of this error is based on identifying the locations whose constraints led to this implication.

```
key.hs:3: ERROR: Existential type variable escapes
Problem : Existential variable escapes via locations
          getKey (MkKey k) = const (k,'m') True
```

□

8.5 Discussion

Here we discuss a practical variation of the implication solver described earlier, and consider the impact on error reporting.

8.5.1 Solving order is significant

The order in which implication constraints are solved can have an effect on the error which is ultimately reported. Consider a goal of the following form, where

b is bound somewhere outside of this implication.

$$\forall a.\left(a = b \supset \left\{ \begin{array}{l} (b = Int \supset b = Int), \\ (True \supset b = Bool) \end{array} \right\}\right)$$

In solving this, we must attempt to solve both of the nested implications, which we will call F_1 (top) and F_2 (bottom.) If we attempt F_1 first, we find it to be trivially **Solved**, and can remove it. Moving to F_2, we will find that we can solve it after an **Add** step, which contributes $b = Bool$. With the addition of this new constraint, however, we will find the outermost implication unsolvable. This will be diagnosed as an 'existential variable instantiated' (subsumption) style of error.

If instead, we try F_2 first, we will first **Add** $b = Bool$. Then, upon attempting F_1, normal CHR solving will fail with the addition of $b = Int$. In this case, the error is found to be due to an unsatisfiable constraint.

Example 76 Consider the following simple example of an EADT program.

```
class C a b
data T a = (a = Int) => K1 a
         | (a = Bool) => K2 a

f :: C a b => T a -> b
f (K1 1) = 'a'
f (K2 True) = False
```

In type checking `f` we will need to solve an implication constraint of the following form.

$$\forall t, a.\left(\begin{array}{l} t = T\ a \rightarrow b, C\ a\ b \end{array} \supset \left\{ \begin{array}{l} t = T\ a \rightarrow t', \\ (a = Int \supset t' = Char) \\ (a = Bool \supset t' = Bool) \end{array} \right\} \right)$$

If we attempt to solve $(a = Int \supset t' = Char)$ first, we add $t' = Char$ as a partial solution, which then clashes with the $t' = Bool$ of the other nested implication. We would report the following.

```
f.ch:5: ERROR : Type error
Problem : Definition clauses not unifiable
```

```
Types   : T Int -> T Char
          T Bool -> T Bool
Conflict: f :: C a b => T a -> b
          f (K1 1) = 'a'
          f (K2 True) = False
```

On the other hand, if we begin with $(a = Bool \supset t' = Bool)$, we find it to be **Solved**. The other implication, $(a = Int \supset t' = Char)$, can also be solved after contributing $t' = Char$. This leaves us with the outer implication, which has been reduced to $\forall t, a.(t = T\ a \rightarrow b,\ C\ a\ b \supset t = T\ a \rightarrow t', t' = Char)$. We will not be able to solve this, since $a = Char$, which is implied by the right-hand side, cannot contribute to any solution. The error reported is as follows.

```
f.ch:4: ERROR: Inferred type does not subsume declared type
Declared: forall a, b.  C a b => T a -> b
Inferred: forall a.  C a Char => T a -> Char
Problem : The variable ‘b’ makes the declared type too polymorphic,
          wrt locations
          f :: C a b => T a -> b
          f (K1 1) = 'a'
          f (K2 True) = False
```

We can transcribe this program into GHC's syntax for EADTs/GADTs, by replacing the above data declaration with the following.

```
data T a where
     K1 ::  Int -> T Int
     K2 ::  Bool -> T Bool
```

If we attempt to compile the resulting program, GHC reports the following.

```
gadt.hs:7:11:
    Couldn't match the rigid variable ‘b’ against ‘Char’
      ‘b’ is bound by the type signature for ‘f’
      Expected type: b
      Inferred type: Char
    In the definition of ‘f’: f (K1 1) = 'a'
```

This is essentially the same as the second error we reported above, but as expected lacks reference to the specific locations involved. □

8.5.2 Removing the Add step

As we saw, a set of inter-related implication constraints can offer different possibilities for error reporting, since different failures may be detected depending on the solving order. In general, the more reasoning that is required to explain an error, the harder it will be to report as a comprehensible, static error message. The **Add** step of our implication solver poses a particular problem in that it introduces inferred type information which may not have been present in the same form in the original program.

If we remove the **Add** step of our solver, failure will occur whenever an implication is not immediately found to be true. This means that all error diagnosis can be performed in terms of the original constraints, which may simplify the error report. Assume we have CHR rules P and attempt to solve $C_0(\forall \bar{a}.D \supset \exists \bar{b}.F_1)$, P, and that the derivations $C_0, D \longrightarrow_P^* D'$ and $C_0, D, F_1 \gg_P^* C'$ succeed. Solving will fail if the **Solved** step does not apply, i.e. if $\models (\bar{\exists}_V.D') \not\leftrightarrow (\bar{\exists}_V.C')$, where $V = fv(C_0, D, \bar{a})$. We can diagnose this failure by finding a minimal $C'' \subset C$, which further instantiates some variable in V. In terms of the minimal unsatisfiable subset algorithm of Section 3.3, this means finding $C'' = \mathsf{min\_unsat}(skol_V(C'))$, where $skol_V$ is a substitution mapping each variable in V to a unique (Skolem) constant.

Example 77 Returning briefly to Example 76, one possible cause of failure was that the implication $(a = Int \supset t' = Char)$ would cause $t' = Char$ to be added, which would only lead to an error later on.

Removing the **Add** step from the solver, we could immediately generate an error like the following.

```
f.ch:4: ERROR: Inferred type too strong
Problem : The following locations contribute new,
          unexpected type information
          f (K1 1) = 'a'
          f (K2 True) = False
```

□

Note that our inference algorithm is complete for fully annotated programs [85]. A 'fully annotated' program is one in which, by definition, all type information will have been provided by the user in the form of type annotations before any

implication is solved, i.e. as long as the program is well-typed, implication solving simply reduces to implication checking. In such a case, the **Add** step will never be required for checking a well-typed program, and so we could adopted the more-direct error reporting strategy of this section.

8.6 Summary

In this chapter we have extended our Haskell-like source language with support for lexically scoped type annotations and extended algebraic data types. We updated our CHR-based inference framework with implication constraints, which allow us to introduce temporary assumptions, like the equations of a GRDT. We also showed that type error reporting in this new setting is still feasible, and that the constraint reasoning operations developed earlier can be applied to useful effect.

No language implementation that we are aware of (other than Chameleon) has support for fully extended algebraic data types, which work naturally alongside type classes and functional dependencies. Moreover, our work represents the first serious attempt at formalising type error reporting in such a system. The techniques we describe can naturally be applied to languages which contain only a subset of the features described herein, like existential types with or without type classes.

Chapter 9

Interactive debugging

In the previous chapters we described our CHR-based type inference systems, and looked at ways of interpreting the resulting constraints to diagnose type errors. We presented methods for generating a single static error message for each error detected in a program. In this chapter we will introduce the Chameleon type debugger, an interactive type debugging tool, its features and their implementation. The Chameleon type debugger has been implemented, and is available as part of the original, prototype Chameleon compiler [84].

9.1 The Chameleon type debugger

In this section, we give an overview of the Chameleon type debugger. We present the debugger as an example of a simple application of the type debugging technology we have developed in earlier chapters. Note that because the type debugger was directly integrated into the original Chameleon compiler itself, and was not a stand-alone application, we never gave it a separate name. For that reason, we will refer to it simply as 'the type debugger', or just 'the debugger.'

The type debugger's primary use is to identify locations within a source program which are involved in a type error. By further examining these potentially problematic program locations, users gain a better understanding of their program and are able to work towards the actual mistake which was the cause of the type error. The debugger is interactive, allowing the user to explore the program beyond what might be presented in a single type error message. One of the novel aspects of the debugger is the ability to explain erroneous-looking types. In the event that an unexpected type is inferred, the debugger can highlight program

locations which contributed to that result.

A debugging session can be initiated by running Chameleon with the appropriate flag, and supplying a filename to load. On starting, the system type checks the nominated file, and reports any errors found. It then presents a prompt, and the user interacts with it by typing commands which the system will execute. In our examples, lines with a prefix *filename*> represent the debugger's prompt followed by a user command. The lines immediately following are the debugger's response.

Since the debugger was part of the original Chameleon implementation, it is not capable of handling the more advanced type system extensions of Chapter 8.

9.1.1 Overview

We present an example of a simple debugging session, which demonstrates the Chameleon type debugger in action.

Example 78 This program is supposed to print a graph of a mathematical function on the terminal. The `plot` function takes as arguments: a) the function to plot, b) the width (in graph units) of each terminal character, c) the height (in graph units) of each character, and d) the vertical distance (in characters) of the origin from the top of the screen.

```
plot f dx dy oy =
     let fxs = getYs f dx
         ys = map (\y-> fromIntegral (y-oy)*dy) [maxY,maxY-1..minY]
         rows = map (doRow fxs) ys
     in unlines rows
     where
         doRow [] r = ""
         doRow (y:ys) r = (if y < r && y > (r-dy) then '*'
                           else ' ') : doRow r ys
         getYs f dx = [ f ((centre x * dx)) | x <- [minX..maxX] ]
               where centre = (+) .5

minX = 0
maxX = 79
minY = 0
maxY = 19
```

```
test = plot sin (2*pi/maxX) 0.2 10
```

Curious to see our test function plotted, we load this program in GHC and are met with the following:

```
plot.hs:8:
    Occurs check: cannot construct the infinite type: a = [a]
        Expected type: [a]
        Inferred type: a
    In the first argument of ‘doRow’, namely ‘r’
    In the second argument of ‘(:)’, namely ‘doRow r ys’
```

We launch the Chameleon type debugger, and ask for the type of `doRow`. We use the syntax `;` to refer to local definitions of `plot`. No Haskell implementation that we are aware of would allow us to examine the type of a locally bound identifier, like `doRow`.

```
plot.hs> :type plot;doRow
type error - contributing locations
doRow (y:ys) r = (if y < r && y > r - dy then '*' else ' ') : doRow r ys
```

We see that `doRow`'s first argument is the pattern `(y:ys)`, whereas in the body of the definition it is first applied to `r`. This is a problem since the function `<` equates the types of `r` and `y`, and yet by reversing the order of `doRow`'s arguments we are also equating the types of `r` and `ys`. This is the source of the "infinite type" error message of GHC's that we saw earlier. To resolve this problem we simply reorder the arguments in the recursive call. The new clause is:

```
doRow (y:ys) r = (if y < r && y > (r-dy) then '*'
                  else ' ') :  doRow ys r
```

As an aside, if we had invoked a newer version of Chameleon with the more advanced text error generation feature of Chapter 6, it would have reported the following.

```
plot.hs:1: ERROR: Type error - some locations are also part of
                  other errors
Problem : Pattern variable ‘y’(bound at line 2, col.  8) used
          with multiple types
```

```
Types   : a
          [a]
Conflict: doRow (y:ys) r = (if (y < r) && y > r - dy then '*' else ' ')
                         : doRow r ys
Common  : doRow (y:ys) r = (if (y < r) && y > r - dy then '*' else ' ')
                         : doRow r ys
```

Note that the locations highlighted in the 'conflict' differ slightly from those reported by the debugger. The debugger's report includes the application of `doRow` to `r`, whereas the text error message system has found a different minimal unsatisfiable subset which instead involves the application to `ys`. The reversal of these arguments in the original program has meant that both have conflicting types. Obviously, there are multiple minimal unsatisfiable subsets, and the system has found their intersection, which it reports in the 'common' part of the error message. Unfortunately, none of the arguments in the `doRow` recursive call are highlighted, because as we can see, neither the `r` nor the `ys` are present in all minimal unsatisfiable subsets.

Having fixed this bug, we return to GHC, and attempt to load the corrected program. GHC reports, amongst other things, the following, which we have had to reformat slightly to fit.

```
plot.hs:18:
    No instances for (Enum (b -> b),
                      Floating (b -> b),
                      Num ((b -> b) -> b),
                      Ord (b -> b))
      arising from use of 'plot' at plot.hs:18
    Possible cause: the monomorphism restriction applied
    to the following:
      plot :: forall b1 a1.
              (Num (b1 -> b1), Num b1, Num ((b -> b) -> b1), Ord a1,
               Num a1) =>
              ((b1 -> b1) -> a1) -> (b1 -> b1) -> a1 -> a -> String
        (bound at plot.hs:1)
      minX :: b -> b (bound at plot.hs:13)
      maxX :: b -> b (bound at plot.hs:14)
    Probable fix: give these definition(s) an explicit
                  type signature
```

```
    In the definition of ‘test’:
        test = plot sin ((2 * pi) / maxX) 0.2 10
```

The problem seems to be the constraints `Enum (b -> b)`, `Floating (b -> b)`, etc., which certainly were unexpected. We need to find out where these strange constraints are coming from in the definition of `getYs`. The *explain* command allows us to ask why a variable or expression has a type of a particular form. Note that we can write `_` to stand for types we are not interested in; each `_` represents a different, fresh type variable. We decide to investigate the `Num` constraint, and ask the system, why it has an argument which is a function type.

```
plot.hs> :explain (plot) (Num (_ -> _) => _)
let fxs = getYs f dx
    ys = map (\y-> fromIntegral (y - oy) * dy) [maxY,maxY - 1..minY]
    rows = map (doRow fxs) ys
in  unlines rows

plot.hs> :explain (plot;getYs) (Num (_ -> _) => _)
centre x * dx

plot.hs> :type plot;getYs;centre
forall a,b.  (Num b, Num (a -> b)) => a -> b -> b
```

The locations highlighted in response to the first query represent calls from `plot` to `rows`, then to `fxs`, and finally to `getYs`. The erroneous `Num` constraint must come from somewhere within `getYs`. We follow this up with the same query, but re-targeted at `getYs`. This shows us that the constraint must have, in turn, come from `centre`. Indeed, when we ask for `centre`'s type in the next command, we see the `Num (a -> b)` right there. Furthermore, it looks like `centre` actually takes two arguments, when we had intended it to take only one and add `.5` to it. We inquire about this in the following step.

```
plot.hs> :explain (plot;getYs;centre) (_ -> _ -> _)
centre = (+) . 5
```

The source of the error suddenly becomes clear. It was our intention that the expression `.5` be a floating point number. In fact, the `.` is parsed as a separate

identifier — the function composition operator, which has type $\forall a, b, c.(a \to b) \to (c \to a) \to c \to b$. The fix is to rewrite this as a floating point value that Haskell will understand as such. The corrected definition of `centre` is:

```
centre = (+) 0.5
```

Although we were able to reason our way towards the error, it is interesting to note that if we had instead used the missing instance error reporting scheme of Section 5.4, from the beginning, we would have been taken directly to the source of the error.

```
plot.hs: 11: ERROR: Missing instance
Instance:Num (a->b): centre = (+) . 5
```

Having fixed these two bugs, we return to GHC and are relieved to find that our program now loads and is accepted.

□

The following subsections broadly outline the type debugger's key features. See Appendix C for a brief summary of all the debugger's commands.

9.1.2 Type error explanation

The Chameleon type debugger primarily serves to direct users toward the source of type errors within their programs. Thus, it would typically be invoked when a type error is already known to be present within a program.

The Chameleon type debugger reports errors by highlighting all source locations which contribute to an error. The system highlights a minimal set of program locations which lead to a type error. There may be multiple such sets — for reasons of efficiency, only one set is found. The implementation of type inference used in the Chameleon type debugger follows the CHR-based scheme of Chapter 4, with all extensions. Like the error reporting schemes of earlier chapters, most of the debugger's functionality is implemented in terms of the constraint reasoning operations of Chapter 3.

Example 79 Consider the following simple program.

```
f g x y = (g (if x then x else y), g "abc")
```

GHC reports:

```
realprog.hs:1:
    Couldn't match 'Bool' against '[Char]'
        Expected type: Bool
        Inferred type: [Char]
    In the first argument of 'g', namely '"abc"'
    In the definition of 'f':
        f g x y = (g (if x then x else y), g "abc")
```

This indicates that GHC has inferred a type of shape $Bool \rightarrow a$ for `g`, but that it has been applied to `"abc"`, which has type $[Char]$.

In Chameleon we can ask the system for the inferred type of some function f using the command `:type` f. If the function has no type (contains a type error), then an explanation is printed in terms of locations that contribute to the error. The system essentially reports an error in the simple style of Chapter 5.

Using the Chameleon debugger, on the example above:

```
realprog.ch> :type f
f g x y = (g (if x then x else y), g "abc")
```

The highlighted locations are those which are implicated in the error. There is no attempt to identify any specific location as the "true" source of the error. No claim is made that the if-then-else expression must have type `[Char]`. The source of all the conflicting types is highlighted and it is up to the user to decide where the actual error is. It may be that for this particular program, the mistake was in using `x` in the conditional part of the if-then-else — a possibility that the GHC message ignores.

By comparison, if we had used the error generating features of Chapter 6 instead, the system could have reported the following.

```
realprog.ch:1:  ERROR: Type error - one error found
Problem : Pattern variable 'g'(bound at line 1, col. 3) used
          with multiple types
Types   : Bool -> a
          [Char] -> b
Conflict: f g x y = (g (if x then x else y), g "abc")
```

Since there is only a single minimal unsatisfiable subset present, the locations highlighted are the same, this error report is clearly more informative, though, since it immediately answers a number of questions that the user may have about the error.

□

The obvious drawback of this approach to error reporting is its dependence on the user to properly interpret the effect of each location involved. Indeed, it may not always be clear why some location contributes to the type error. Of course, the user is free to follow up his initial query with additional commands to further uncover the types within the program.

9.1.3 Local and global explanation

The Chameleon type debugger employs the CHR-based inference scheme of Chapter 4, which works by generating constraints and constraint handling rules out of a source program. The more constraints that are involved, the slower this process may be. In general, the number of constraints present is proportional to the size of the call graph in a definition. To reduce the amount of overhead, which is particularly important for an interactive system, we provide two distinct reporting modes: *local* and *global*.

In the local mode, we restrict error/type explanations to locations within a single definition. The consequence of this is that we are then free to simplify the constraints which arise from outside of this definition. What this means in terms of our inference scheme is that we simplify the bodies of CHR rules. Given a set of rules P, for each rule r of form $f(t,x) \iff C$, we run $f(t,x) \longrightarrow_P^* C'$, and replace r by $f(t,x) \iff \phi(C'_u), t = \phi(t), x = \phi(x)$, where ϕ is the mgu of C'. By doing this, we lose the detailed constraint and justification information present in each rule, and obviously we cannot simplify any rule which corresponds to an ill-typed definition, i.e. since C' will be unsatisfiable. The loss of fidelity will not matter, though, if we are not interested in locations arising from other definitions, as is the case for *local* explanations.

Example 80 Consider the following simple program which is supposed to model a stack data structure using a list.

```
idStack stk = pop (push undefined stk)
push top stk = (top:stk)
pop (top,stk) = stk
empty = []
```

There is obviously a problem here. GHC reports the following.

```
stack.hs:1:
    Couldn't match `(t, t1)' against `[a]'
        Expected type: (t, t1)
        Inferred type: [a]
    In the application `push undefined stk'
    In the first argument of `pop', namely `(push undefined stk)'
```

Launching the debugger in local mode (the default), the following is reported:

```
type error - contributing locations:
idStack stk = pop (push undefined stk)
```

The problem must lie in the interaction of the two highlighted functions. We examine both of the culpable functions. Note that neither GHCi (the interactive version of GHC) nor Hugs would allow the following two queries. Both systems will completely reject any file containing a type error, preventing us from querying the types of even the well-typed parts of the module.

```
stack.hs> :type push
a -> [a] -> [a]
stack.hs> :type pop
(a,b) -> b
```

The error is revealed. In the definition of `idStack`, `push` returns a list, where `pop` expects a pair. □

In the global explanation mode, we place no restrictions on the locations which can be highlighted. Since we no longer restrict ourselves to any particular locations, we can no longer simplify away any of the constraints involved.

Example 81 We return to the previous example, but this time we use the global explanation mode.

```
stack.hs> :set global
stack.hs> :type idStack
type error - contributing locations:
idStack stk = pop (push undefined stk)
push top stk = (top:stk)
pop (top,stk) = stk
```

In this mode, all program locations contributing to the error are revealed immediately. The most interesting locations are the newly highlighted ones, i.e. those which were not already highlighted in the local error report. In this case, the conflict between the list and pair is immediately revealed. □

9.1.4 Type explanation

It is possible to write a function which, though well-typed, does not have the type that was intended. A novel feature of our debugger is its ability to explain such unexpected types. The command `:explain` (e) $(C \Rightarrow t)$ asks the system to underline a set of locations which force expression e to have type of the form $C \Rightarrow t$. Note that this type scheme is implicitly existentially quantified, and any universal quantifiers in inferred types are ignored for the purpose of this command. As is the case for type errors, explanations consist of references to the responsible source locations.

Type explanations are particularly useful for explaining errors involving malformed type class constraints. A typical type error message would merely complain about the existence of such a constraint, without providing any information to further investigate the problem.

Type explanation is implemented as follows. Assume we have already loaded a program, and generated the CHR rules P, and initial environment E containing all let-bound identifiers. Given a query `:explain` e $(C \Rightarrow t)$, we first generate the appropriate type, constraints, and CHR rules out of e, i.e. $E, \Gamma, e \vdash_{Cons} (C' \mid t')$ and $E, \Gamma, e \vdash_{Def} P'$ (see Section 4.2.1), perform the CHR derivation $C' \longrightarrow^*_{P \cup P'} C''$. We then use the minimal implicant algorithm of Section 3.4, to find $M = \mathsf{min\_impl}(C'', C \wedge t' = t)$, where M is a minimal subset of C'' which explains the queried type. As usual, we then read the justifications of M to find the program locations which form an explanation.

Example 82 The following program scales all elements of a list, such that the biggest element becomes 1. It contains a type class error.

```
normalise xs = scale biggest xs
scale x ns = map (/x) ns
biggest (x:xs) = max x xs
        where max x [] = x
              max x (y:ys) | y > x = max y ys
```

```
                          | otherwise = max x ys

test = normalise [1,2,3,4]
```

GHC reports the following:

```
normalise.hs:8:
    No instance for (Fractional ([a] -> a))
      arising from use of ‘normalise’ at normalise.hs:8
    Possible cause: the monomorphism restriction applied
                    to the following:
      test :: [[a] -> a] (bound at normalise.hs:8)
    Probable fix: give these definition(s) an explicit
                  type signature
    In the definition of ‘test’: test = normalise [1, 2, 3, 4]
```

As we discussed in Section 5.4, in Haskell, context type class constraints must have an argument of the form α, or $\alpha\ t_1 \ldots t_n$, there α is a type variable, and each t_i is an arbitrary type. The inferred constraint, *Fractional*$([a] \rightarrow a)$, obviously violates this condition.[1]

To solve this problem, we will need to know why *Fractional*'s argument is a function. Since this is a fairly compact program, we will use the debugger in global mode. We proceed as follows, first finding out precisely where the *Fractional* constraint comes from.

```
normalise.hs> :set global
normalise.hs> :explain (normalise) ((Fractional _) => _)
normalise xs = scale biggest xs
scale x ns = map (/x) ns
```

[1]GHC is slightly more liberal about detecting missing instances than some other implementations, like Hugs. GHC ignores this problem until a class constraint is monomorphic, whereas Hugs seems to check as soon as a type is generalised. If we omit the `test` binding from our example, then GHC will not report an error immediately. When `normalise` is eventually used monomorphically somewhere, if there is no instance then the error will be reported there. The obvious downside is that if the strange context is the result of a mistake, then that bug may only be discovered at great distance from the original definition.

Strictly speaking however, GHC will admit slightly more programs than Hugs; it is possible that even though no instance exists in the module containing the definition, it may be that the value is always used in modules where an appropriate instance is available. e.g. We might only call `normalise` from a module where an instance for *Fractional* $([a] \rightarrow a)$ is declared.

As expected, we see that *Fractional* comes from the use of the division function `/`. We still need to find out why its argument is a function, though. In the command below, the queried type `(Fractional (_ -> _)) => _`, expresses precisely this; we are not interested in any of the component types, only that `Fractional`'s argument is a function of some sort.

```
normalise.hs> :explain (normalise) ((Fractional (_ -> _)) => _)
normalise xs = scale biggest xs
scale x ns = map (/x) ns
biggest (x:xs) = max x xs
        where max x [] = x
              max x (y:ys) | y > x = max y ys
                           | otherwise = max x ys
```

This indicates that the problem obviously has to do with `scale` and its application to `biggest`. We infer their types to see if we can make something of this.

```
normalise.hs> :type scale
forall a.  Fractional a => a -> [a] -> [a]

normalise.hs> :type biggest
forall a.  [a] -> a
```

We see now that `scale`'s first argument shares the same type as `Fractional`'s argument. The function `biggest` has type `[a] -> a`, and since `scale` is applied directly to `biggest`, this same type makes its way into the `Fractional` constraint. Obviously this application is incorrect.

Exercising our curiosity, we perform a couple of additional queries, just to confirm our suspicions.

```
normalise.hs> :explain (normalise) ((Fractional ([_] -> _)) => _)
normalise xs = scale biggest xs
scale x ns = map (/x) ns
biggest (x:xs) = max x xs
        where max x [] = x
              max x (y:ys) | y > x = max y ys
                           | otherwise = max x ys
```

```
normalise.hs> :explain (normalise) ((Fractional ([a] -> a)) => _)
normalise xs = scale biggest xs
scale x ns = map (/x) ns
biggest (x:xs) = max x xs
        where max x [] = x
              max x (y:ys) | y > x = max y ys
                           | otherwise = max x ys

normalise.hs> :type biggest;max
forall a,b.  a -> [b] -> a
```

The first query above is slightly more specialised than our original, in that we are now asking for an explanation of a constraint of form `Fractional ([_] -> _)`, i.e. we now also want to know why the function must take a list argument. As compared to the previous, less specific query, there is one additional location highlighted in the result, the `:` pattern constructor in `biggest`; all others are the same.

Finally, we ask for an explanation of `Fractional ([a] -> a)` (for some `a`), which is even more specific, in that the two unknown types must now be the same. The previous explanation now grows to encompass: `biggest`'s `x` argument, a call to `max`, and `max` returning its first argument in its first clause. We can see that the `a`s in the `Fractional` constraint must be the same, because `biggest` calls `max`, and its return type must match the type of its first argument.

These last few queries have yielded new highlighted locations which are confined, as we originally suspected, to the definitions of `scale` and `biggest`. It is reassuring to know that no other function, e.g. `map` is at all responsible for this typing problem.

The following corrected version of `normalise`, first applied `biggest` to `xs`, and will work as intended.

```
normalise xs = scale (biggest xs) xs
```

Note that had we generated a missing instance error in the style of Section 5.4, the error message would have reported the same locations as highlighted above. In this case however, the explicit global explanation feature of the debugger proves especially useful.

```
normalise.hs: 1: ERROR: Missing instance
Instance:Fractional ([a]->a): normalise xs = scale biggest xs
```

□

9.1.5 Most-likely explanations

Since it is technically infeasible to present all explanations for a particular type error or `:explain` query, the debugger will only display one. However, as we saw in Section 3.3.1, it is possible to cheaply find the intersection of all explanations. When non-empty, this information is useful since in many cases it would be reasonable to assume that locations which are involved in more errors are more likely to be the real cause of the problem.[2] Indeed fixing a location involved in all errors will immediately resolve them all.

If the debugger finds that there is a common subset of all errors, it will automatically report it, along with a specific error. The same functionality was described in Section 5.1.1.

Example 83 Consider the following program: `merge` takes two sorted lists as arguments and combining all elements, returns a single sorted list.

```
merge [] ys = ys
merge xs [] = [xs]
merge (x:xs) (y:ys) | x < y = [x] ++ merge xs (y:ys)
                    | otherwise = [y] ++ merge (x:xs) ys
```

This program contains a type error. Loading it into the debugger, we get the following error report.

```
type error - conflicting locations:
merge [] ys = ys
merge xs [] = [xs]
merge (x:xs) (y:ys) | x < y = [x] ++ merge xs (y:ys)
                    | otherwise = [y] ++ merge (x:xs) ys

locations which appear in all conflicts:
merge [] ys = ys
merge xs [] = [xs]
merge (x:xs) (y:ys) | x < y = [x] ++ merge xs (y:ys)
                    | otherwise = [y] ++ merge (x:xs) ys
```

[2] See the `doRow` function of Example 78 for an example where this is *not* the case.

The first display of the program above is a typical debugger error report. A single error is reported and all responsible program locations are highlighted.

The second program printout indicates those locations which appear in all errors. Note that these locations alone do not themselves constitute a type error.[3] An actual error occurs when these locations are considered alongside (either of) the recursive calls to `merge` in the third clause. Nevertheless, this provides us a view of the most likely source of error, and makes clearer the mistake in this program.

□

9.1.6 Type inference for arbitrary bindings

Most existing implementations/interpreters for Haskell, like Hugs and GHCi, allow users to ask for the type of variables in their program. These systems, however, restrict queries to variables bound at the top level of their program. It is not possible for users to enquire about the types of variables bound within a `let` or `where` clause. This can be frustrating since, in order to uncover the types of those definitions, the source program would need to be modified by lifting sub-definitions to the top level.

Furthermore, these systems restrict type queries to programs which are well-typed. Obviously this is of little help when debugging a program, since it must somehow be made typeable while maintaining the essence of the error - for example by replacing expressions with calls to `undefined`, which has type $\forall a.a$. Once we start making these sorts of modifications, however, we are no longer debugging the same program.

By contrast, the Chameleon type debugger allows users to infer types of arbitrary variables and expressions within a program, regardless of whether it is typeable. (See Section 4.2.3 for details of how this can be done.)

Example 84 Consider the following program, where most of the work is being done in a nested-definition. We will find that the type of `reverse` is not what we expect.

```
reverse = rev []
```

[3] If there is only a single minimal unsatisfiable subset, then the same constraints will obviously also represent the intersection of all minimal unsatisfiable subsets. In such a case, the debugger does not make mention of this trivial intersection.

```
        where rev rs [] = rs
              rev rs (x:xs) = rev (x ++ rs) xs
```

We run the debugger as follows.

```
reverse.hs> :type reverse
[[a]] -> [a]
```

Since the problem is mostly due to the definition of `rev`, we continue by examining the type of `rev`. We can refer to nested bindings, like `rev`, via their enclosing binding by separating their names with `;`s. We can refer to a specific clause of a definition by following it with a `#` and the number of the clause. Otherwise, by default, the reference is to the entire declaration, including all clauses.

```
reverse.hs> :type reverse;rev
[a] -> [[a]] -> [a]
reverse.hs> :type reverse;rev#1
a -> [b] -> a
reverse.hs> :type reverse;rev#2
[a] -> [[a]] -> b
```

We see above that the second clause of `rev` introduces the erroneous `[[a]]` type. Having narrowed our search to a more easily understood fraction of the original code, we proceed as follows:

```
reverse.ch> :explain (reverse;rev#2) (_ -> [[_]] -> _)
rev rs (x:xs) = rev (x ++ rs) xs
```

This result identifies the cause of the problem. The variable `x` shares the type of the accumulator argument, while also being an element of the second argument list. □

Certainly, a text-based interface places restrictions on how easily one may refer to arbitrary program fragments, however this is merely a limitation of the current debugger implementation. In principle, we feel that the ability to infer types anywhere within any program is of invaluable benefit when debugging type errors.

9.1.7 Source-Based Debugger Interface

Being interactive, the debugger's text-based interface provides users with immediate feedback which they can use to iteratively work their way toward the source of a type error. Although flexible, such an interface can at times be slightly stifling and a distraction from the source code being edited separately by the programmer. As an alternative, the original Chameleon system allows programmers to pose debugger queries directly in their program's source.

The command '`:type` e', where e is an expression, can be written directly in the program as e`::?`. Also, `:explain` (e) $(D \Rightarrow t)$, where e is an expression and $D \Rightarrow t$ is a type scheme, can be expressed as e`::?`$D \Rightarrow t$ within the program itself. As well as individual expressions, entire declarations can be queried by writing such a command at the same scope (with a declaration name in place of an expression.)

The debugger responds to these embedded commands by collecting and processing them in textual order. They do not otherwise affect the working of the compiler, and do not change the meaning of programs they are attached to.

Example 85 Returning to Example 80, we modify it by adding a `type` to the definitions of `push` and `pop`.

```
idStack stk = pop (push undefined stk)
push ::?
push top stk = (top:stk)
pop ::?
pop (top,stk) = stk
empty = []
```

Compilation would still fail as usual, but the Chameleon system will print the following before stopping:

```
push :: a -> [a] -> [a]
pop :: (a, b) -> b
```

□

9.2 Summary

In this Chapter we have introduced the Chameleon type debugger, which is yet another application of the constraint reasoning operations of Chapter 3 and the

CHR-based inference framework of Chapter 4. The debugger is a text-based interactive system which reports type errors as sets of conflicting locations, and allows the user to query the types within programs. Most importantly, and distinctively, the debugger supports a form of automatic type explanation. The user can ask why the program gives rise to some inferred type, and the system obliges by indicating a minimal set of locations which is responsible. By repeated application of these steps, the user can gain a better understanding of the types in his program, and work towards any mistakes.

It is interesting to compare the type error facilities of the Chameleon type debugger with the information reported in the more expressive error messages of Chapter 6. In many cases, for simple programs at least, the more expressive messages often capture the first few debugging steps that a user would typically take when using the debugger. It performs the task of inferring some initial types, and generating their explanations, in the style of the debugger's `:explain` command.

Chapter 10

Related work and discussion

In this chapter we take a look at other research which shares the same goal of assisting programmers to deal with type errors. The number of different techniques employed is quite varied, so we will attempt to distil this existing work into a number of distinct categories, and discuss the specific problems tackled and the effectiveness of the ideas presented in addressing those problems.

Unfortunately, as far as we know, little of this work has been adopted for use in practical, modern compilers. This could be for a number of reasons. In some cases the benefit of a particular approach is not always clear and obvious. In others, the ideas are only presented in terms of a simple λ calculus, and never extended to a full-scale, practical language. Almost none of the approaches we discuss below have been modified to work in a setting that supports ad hoc overloading, like Haskell's type class overloading, or more advanced extensions like extended algebraic data types (EADTs.)

10.1 Improved conventional type error reports

By far the most work we are aware of in assisting type error debugging has gone towards improving the quality of the conventional type error messages which most compilers already generate. A 'conventional' type error message is one which identifies a single point of type inference failure in the program, and reports the error as a conflict between two distinct, non-unifiable types. The reason for the focus on this style of error reporting is probably due to the fact that these schemes can often be implemented as conservative extensions, or minor modifications of traditional inference algorithms, like $\mathcal{W}$.

As we have seen, there are some serious limitations when type errors are reported in the conventional style. All systems we are aware of, other than our own, are restricted to reporting errors as conflicts between two nonunifiable types. Furthermore, since they focus on a single program location, most type error messages represent only a single reasoning step in a possibly long and complicated type inference derivation. Again, this is something we have tried to address in Chapter 6, by highlighting the source of the types involved in the conflict.

The fundamental problem with conventional type error messages is that they do not scale with the size of type error problems. If we define the 'size' of a type error as the number of constraints appearing in the minimal unsatisfiable subset then, as we saw in Chapter 5, the number of locations which we could blame the error on is (approximately) equal to its size. Clearly, as the size of the error increases, a conventional error message is less and less likely to pinpoint the site of the actual mistake, making such error reports less insightful than they could be. Moreover, for larger problems, the context that the programmer has to consider when reasoning about the conflicting types that were reported also grows. Conventional type error messages only ever report a single erroneous location, regardless of the complexity or size of a type error. This is also why the heuristics employed by systems like Helium [39] and that of Johnson and Walz [87] are usually extremely simple.

Despite this, there are also benefits to employing a conventional type error reporting style. Such type error diagnoses are often succinct, efficient to generate, and familiar to current programmers. Although they allow for more thorough exploration, interactive systems often take a great deal more effort on the behalf of the programmer than reading and deciphering a simple type error message. We will now take a look at some related research which aims to generate more informative conventional type error messages.

10.1.1 Modifying $\mathcal{W}$ to eliminate bias

The most straightforward attempts to generating more informative error messages involve modifying the order in which algorithm $\mathcal{W}$ performs type unification as it traverses the abstract syntax tree of a program. As we noted in Section 2.2.2, algorithm $\mathcal{W}$ discovers type errors when unification fails at expression application sites. In a single step, for an expression $e_1\ e_2$, it attempts to unify the type of t_1 with $t_2 \to \beta$, where t_1 and t_2 are the types of expressions e_1 and e_2, and β is a

fresh variable. If e_1's argument type and t_2 are basic types which clash, then the application site is indeed a reasonable place to report the error from. For instance, consider the expression `\x -> not (toUpper x)`, where `not` : $Bool \rightarrow Bool$ and `toUpper` : $Char \rightarrow Char$. In more complex cases, when the two sub-expressions' types interact, the most useful location to report the error from is less obvious. For example, for the ill-typed expression `\f x -> (f x, x f)`, there is nothing that suggests that the mistake is more likely to lie in the first part of the tuple than the second.

Lee and Yi [54] formalise a folklore variant of $\mathcal{W}$, which they dub $\mathcal{M}$, and prove that $\mathcal{M}$ finds a strictly smaller ill-typed expression than $\mathcal{W}$, if one exists.[1] They claim, as do others [59, 92], that errors that refer to a smaller fragment of the program may be easier for programmers to comprehend, since they have fewer sub-expressions to consider. It is clear from our work, however, that the set of program locations which all contribute to a type error can be many and scattered. No error can be fully explained by a single program location. Indeed, an error message which focuses on a small, deeply nested part of an ill-typed expression might even be considered misleading, since to understand it fully, the programmer must still examine locations outside of that sub-expression; locations that the error report omits any mention of.

In later work Eo, Lee and Yi [20] generalise the same method they used to formalise $\mathcal{M}$, coming up with what they call algorithm $\mathcal{G}$. This meta-algorithm generalises algorithms $\mathcal{W}$ and $\mathcal{M}$ (which are shown to be specific instances of $\mathcal{G}$), and offers more freedom to unify types at different times, which can lead to the discovery of errors at different program locations. It is based on the same idea as $\mathcal{M}$, in that the algorithm propagates a constraint on the type of an expression, which it may introduce earlier than $\mathcal{W}$ would normally enforce that same condition. The key difference between $\mathcal{M}$ and $\mathcal{G}$ is that $\mathcal{G}$ allows for some variation in the constraint being imposed; $\mathcal{M}$ can be thought of as an instance of $\mathcal{G}$ which always imposes the strongest possible constraint as early as possible. A number of other researchers, ourselves included, have independently recognised that the order in which an inference algorithm traverses a program, and performs unification affects the type error that is discovered, and have been motivated to generalise this [61, 32].

We feel that viewing type inference in terms of stand-alone constraints is

[1] It is equivalent to say that $\mathcal{M}$ always finds errors earlier than $\mathcal{W}$, if it is possible.

a good starting point when considering generalisation. Although Lee and Yi's algorithm $\mathcal{G}$ does indeed generalise $\mathcal{W}$ to some extent, it still requires that some types are always inferred before others – an inherent bias. As such, it is not as general as it could be. When we represent inferred types as constraints, on the other hand, we are free to solve them in absolutely any order we choose. Another advantage of this view is that it naturally supports the discovery of multiple errors.

McAdam [58] has written much about type error reporting in the Hindley/Milner system. In an early paper he identifies the inherent left-to-right bias of algorithm $\mathcal{W}$ [59]. This bias is a consequence of the fact that, for application sites, algorithm $\mathcal{W}$ (see Figure 2.2) first processes the left expression, then applies the resulting substitution to the environment used to type the right expression. This means that if there is any erroneous interaction between the types of those two expressions, other than through the application, it will always be detected on the right. McAdam suggests an alternative inference algorithm which avoids bias by first inferring the types of the sub-expressions in isolation, and then combining their resulting substitutions. Modifying an existing real-world, implementation of algorithm $\mathcal{W}$ to work like this would take some effort since practical implementations of unification update variables directly, rather than returning substitutions. In our system, constraints can be considered independently of each other and we are free to pick any to base an error message on, so there is no inherent bias.

Jun, et al. [92, 47] describe a number of type inference algorithms, based on modifications of $\mathcal{W}$ which they claim produce better type error messages. Much of their work is based on a trial they performed in which they studied how people manually infer and check types [91]. One algorithm they describe, called $\mathcal{U}_{\text{AE}}$, avoids left-to-right bias by inferring types of sub-expressions in isolation, and then only unifying assumption environments at the root of those sub-expressions. This is essentially the same idea as McAdam's (described above) for avoiding bias [59]. They describe another algorithm, $\mathcal{IEI}$ which is basically the same as $\mathcal{U}_{\text{AE}}$, except that when an error is detected, it switches to algorithm $\mathcal{M}$, in order to try to find a smaller sub-expression to base the error report on.

10.1.2 Alternative models of type inference

Other researchers have proposed more dramatic changes to type inference in order to pinpoint and report type errors more accurately. The systems proposed range

from alternative algorithms which can infer types of open expressions, or expressions containing type errors, to views of type inference in terms of constraints and graphs. Some have also considered the problem of employing heuristics to automatically generate 'probable fixes' which can then be suggested to the programmer as a remedy to the immediate problem.

Bernstein and Stark [4] outline a method for type inference of open expressions. An 'open expression' is one that contains unknown, unbound variables. The compiler automatically infers the most general type of each unbound variable, which it then reports. They argue that this feature could be used by programmers to inspect types within a program; by replacing expressions with free variables, the user can find the type of that expression. To debug a type error in this system, the user is required to iteratively work towards the source of the error by manually inserting such "break points", and re-running the compiler. This debugging technique is essentially included in our debugger. There is no need to modify the program source and recompile; the type of any sub-expression can be reported in combination with or independently of any other program location. Moreover, we feel it is greatly strengthened by the fact that from within the same system we can not only query types, but ask for explanations of them.

Braßel [5] employs a similar idea in his TypeHope system. He suggests that programmers often deal with type errors by first commenting out potentially erroneous code, or replacing it with a dummy variable with a fully universal type ($\forall\alpha.\alpha$). If the resulting type is 'correct' (how this is determined is unclear), then he concludes that the code which was removed was the source of the error. Braßel's system works by automating this process to select a candidate. Compared with our approach, it is clear that his system will essentially select one of the locations that we would normally highlight in reporting an error. He does not describe, however, how his system deals with expressions that may contain multiple, distinct type errors. e.g. `\x -> (x x, True 'a')`

Neubauer and Thiemann confront the problems associated with the loss of source information during type unification by using a system of discriminative sum types [67]. Roughly, the result of unifying two (normally) nonunifiable types in their system is a sum type which records both of those types within it. They keep track of the source of such types and at the end, if the result is a sum of conflicting types, they report those types and their source locations. Unfortunately, their algorithm does not necessarily find minimal sets of conflicting locations, and so may report program sites that are not involved in the error at all. This

could happen, for example, if an otherwise unrelated type variable is unified with an erroneous sum type. One interesting refinement of their system is that they record "source" and "target" expressions. i.e. producers and consumers of types. Building on this type system implementation, they have also implemented a type browsing tool [68], which exports the type information in a form that can be presented and explored in a web browser.

In early work, Johnson and Walz describe a type inference system, in the context of an interactive program editor, which is able to identify and highlight program locations that are most likely to be the cause of a type error [87]. Their system works by normalising a set of equations that they generate during inference, and looking for type variables which are bound to multiple, conflicting types. In some ways this is similar to the approach of Neubauer and Thiemann [67], except that in their system the conflicting types of a particular location are bound together in one discriminative sum type, rather than appearing on the left-hand sides of separate normalised equations. The system of Johnson and Walz assigns weights to the different types associated to a variable. They assume, quite reasonably, that a type which is assigned more frequently to some variable is more likely to be its correct type.

McAdam also describes a graph data structure for representing type relations between expressions within a program [61]. He demonstrates by example that it is possible to extract from his graphs much the same information as can Bernstein and Stark [4], Duggan [18] and Wand [88], from their specialised techniques. This is not surprising since these graphs are essentially just a different representation of Herbrand constraints. The graph traversal techniques that McAdam describes could be straightforwardly recast in terms of type equations. Furthermore, the practicality of the graph representation is questionable, since there is no obvious way to encode the types of recursive definitions.[2]

In other work McAdam describes a technique for suggesting fixes for some kinds of type errors [60]. The approach he describes can recognise when a function's arguments have been passed to it in the wrong order. By finding a 'type isomorphism', the system can suggest a reorder of those arguments such that the resulting application would be typeable.

Haack and Wells propose a scheme for discovering type errors which has much in common with ours [27]. Like us, they generate labelled (justified) equations

[2] A cycle in one of McAdam's graphs represents a type error.

from source programs and they also minimise the set of erroneous constraints that their inference algorithm generates. They use a special-purpose unification algorithm which associates a set of location labels along with each variable that is influenced by the so-labeled constraint. Even if this set of labels is not minimal, it allows them to start from a smaller subset of constraints when attempting to find a minimal subset. The same optimisation could be applied in our system, using justifications. One difference between our approaches is that where we use CHR rules to represent polymorphic types, they simply copy and rename constraints associated with let bindings. Since their constraint domain consists only of equations, they would need to extend it to support language features like ad hoc overloading (whether Haskell or ML style), or EADTs, whereas in our system, support for type classes was present from the beginning. As far as we know, Haack and Wells never pursued this work any further. The style of type error messages produced by their system is similar to the simple type error reports of Chapter 5, which consist only of the original source program with conflicting locations highlighted, and no further explanation.

Dinesh and Tip [15] present a system which also reports type errors as program slices. Their work differs from ours in that it is based on a language which requires exhaustive type annotations, and only has basic (non-variable) types. This greatly simplifies the problem of identifying type conflicts. Their approach is also formulated quite differently to ours, in that it is based on a term rewriting system. They use 'dependence tracking' [22] to calculate a minimal slice of a program which leads to the error that was eventually found. Solving our Herbrand constraints relies on the unification of types containing variables. During such a unification step, it is possible that variables can become aliased, and we may not be able to differentiate between a variable that makes a real contribution to an error, and one that simply looks like it might (because it has been unified with one that does.) We cannot imagine a type error diagnosis system, that also performs type inference, based on term rewriting that could find minimal error sites. Essentially, we would run into the same problem faced by other systems of discovering non-minimal errors [88, 87, 67, 33].

The Helium language [39] was designed to make learning functional programming, in a Haskell style, easier for beginners. It specifically addresses the problem of reporting understandable and helpful error messages for learners. To make things more manageable, they simplify the language slightly, removing features, chiefly type classes, which are likely to confuse beginners.

Heeren, et al. [33] describe the basis of the type inference and type error reporting system of Helium. Essentially, the idea, as in our work and many others, is to generate type equations from the original source program that can then be solved in a separate step. One difference between our work and theirs, is that they use 'generalisation' and 'instantiation' constraints (which they call "instance" constraints) to provide support for let-polymorphism, whereas we use separate CHR rules and generate user constraints at instantiation sites. Instead of finding a minimal unsatisfiable subset of constraints, Heeren, et al. attempt to solve the generated constraints in order, until an unsatisfiable prefix is found. To avoid some of the difficulties associated with a fixed-order, program traversal inference algorithm, like $\mathcal{W}$, they first reorder their constraints in order to influence the location at which the error is found. Nevertheless, their system still suffers from the problem of reporting type errors as conflicts at a single program location, overlooking the effect of contributing locations. They also describe a number of simple heuristics that their system employs in order to try to suggest probable program fixes to the programmer. One strategy they use involves replacing a function which appears in the context of a type error with a related or 'sibling' function, in the hope of discovering that the programmer may have mistakenly applied one function, when he actually meant the other.

In later work, Heeren, Hage, et al. [34] describe a scripting language which can be used to precisely specify the order in which constraints are solved when typing an expression containing some known identifiers. This was the inspiration for the 'user-specified type error messages' which we describe in Section 6.2. These systems allow programmers to attach specific error messages to the constraints used in typing some variable. If those constraints are part of an error, the specialised message can be displayed. The difference between our systems is chiefly that in theirs, the order in which the constraints are solved is a fundamental part of their error reporting algorithm, whereas we simply find a minimal unsatisfiable subset. They also use their system to generalise the notion of 'sibling' functions, mentioned above; they can be specified as part of the script. It would be expected that these facilities would be of most use to writers of reusable, library code, since the messages could be tailored to reflect the specific domain that the library represents. Code in libraries could be used by many other programmers who would all benefit from the extra time spent manually improving the quality of the error messages.

In other, more recent work, Heeren and Hage [31] turn to the problem of

generating reasonable error reports for programs involving type classes. They define a number of 'type class directives' which place various restrictions on the shape of type classes. The idea is to detect and report such constraints earlier than they otherwise would be. For example a 'never' directive allows them to specify type class instances that should never be allowed. Most of their directives (apart from 'default' which is a kind of generalisation of Haskell type class defaulting) can be encoded directly using CHRs. e.g. the directive `never (C Int)` could be represented by the rule $C\ Int \Longrightarrow False$. Their system is presented informally, and they conclude that any future work ought to be based on a sound, formal understanding of type class semantics. Interestingly, they suggest that Constraint Handling Rules would be a good starting point for just such a formalism.

10.2 Inference explanation systems

A number of systems avoid the problems associated with conventional type error reports altogether, by presenting not only a type conflict, but also a reason for that conflict. We call these *inference explanation systems*. Note that these are occasionally referred to as *type* explanation systems, but we feel that since they typically present explanations based on the operational behaviour of an underlying inference algorithm, calling them *inference* explanation systems is more appropriate.

Usually, the explanations generated by these systems take the form of a sequence of inference steps which lead to the failure. These steps can be recorded as the normal inference procedure works. Such explanations can be presented to the user in an interactive fashion. i.e. the system presents an inference step, and the user can prompt it to continue with a particular sub-expression. Or, alternatively, they may form part of a fixed error message. Unfortunately, these explanations tend to grow extremely quickly, and often contain many small, repetitive steps which could confuse and overwhelm programmers.

Wand [88] describes a simple modification to the unification algorithm employed during type checking which allows him to record an explanation for each unification step. He represents types as a tree and a substitution, where each element of the substitution has a reason, or explanation. In the event that unification fails, explanations for the two conflicting types can be read out of this environment. New explanations are only generated as a side-effect of unification

at application sites. Consequently, unlike in our approach, identifiers which are present in the initial type environment (like constants) are never part of explanations. One other shortcoming of this scheme is that explanations are not minimal. That is, an explanation may contain references to expressions which merely alias a type variable, without ever truly contributing to the error.

Duggan and Bent[19] describe a system which, like that of Wand, records a reason for each unification step performed during inference. They begin with a standard, destructive unification algorithm; types are represented as (structured) nodes in some heap, and each type variable contains a pointer which indicates its current 'value'. Unifying a variable a with a type t requires first fully dereferencing a, yielding some a', and updating a' to point to t in the heap.[3] Each such pointer has a number of expressions associated with it; these are the expressions which led to the particular unification step that caused that pointer to be asserted. An explanation for why a variable (representing the type of a specific expression) has been assigned some type can be obtained by following the chain of pointers from the variable to its ultimate type, and reading off the expressions responsible. They address the issue of variable aliasing directly in their unification algorithm; reversing the direction of dereferenced pointers

Unfortunately, their system is not able to provide explanations for all possible types. For instance, if we consider the equations $a = b, b = c, d = e, b = e$, their system is not able to explain why $a = d$. This is because their unification algorithm builds the directed path represented by those equations, i.e. a pointer from a to b, then b to c, etc., which does not include a route from a to d. In comparison, we can use our minimal implication algorithm to find an explanation for *any* valid type, in terms of the responsible program expressions; the underlying representation is not an issue. Another problem with their system is that it does not always find all of the expressions responsible that are needed for a complete explanation. Given the following expression, which is adapted from their paper

```
\f x y z -> (f y, y == ('a',x), f (z,True))
```

an explanation for `z` having type *Char* consists only of the sub-expression `f`

[3] If dereferencing a leads to a non-variable type t', then the outermost type constructors of t and t' must be equal, and if so, the unification algorithm continues on to the sub-components of t and t'. This is standard.

`(z,True)`. In our approach, we would highlight the following.

```
\f x y z -> (f y, y == ('a',x), f (z,True))
```

Clearly, the character constant itself, as well as the other application site, and others, are also responsible. An explanation that does not mention them cannot be considered complete.

Beaven and Stansifer [3] argue that the most difficult thing about debugging type errors is that the long chain of deductions made during inference are lost. This makes the cause of the error hard to decipher. They propose an approach which maintains the deductive steps of inference up until the point of failure. They want to be able to ask 'how' the system arrived at a particular conclusion. A downside of their scheme, which is common to all type error explanation systems, is that the error reports it could generate would be extremely verbose; moreover some explanations are potentially infinite. Their system would be better suited to an interactive environment than for automatically generating error messages. Even so, it would probably be necessary to be able to 'switch off' some of the detail, since a lot of it is repetitive and uninteresting.

In their paper they present an example explanation for the following ill-typed ML expression.

```
(fn a=> +((fn b=>if b then b else a) true, 3))
```

Their system generates the following explanation, which we have taken directly from their paper, but have significantly shortened for brevity. The full version they present in their paper stretches to thirty-six lines of text, and even then a number of portions have been elided.

```
1 A type error was detected in the application >> (+ (#,3)) <<.
2 The domain of the function >> + << is not unifiable
3 with the type of the argument >> ((# true),3) <<.
4 Domain of function is >> (int*int) <<.
5 The argument has type >> (bool*int) <<.
6 **Why does the function >> + << have type >> ((int*int)->int) <<?
7   The identifier >> + << was assigned type >> ((int*int)->int) <<
8   as part of the initial environment.
...
```

If we transcribe the above program into Haskell, it is essentially

```
\a -> plus ((\b -> if b then b else a) True, 3)
```

where `plus` is addition for *Int*s and has type $(Int, Int) \rightarrow Int$. We can report the following for this program.

```
beaven94explaining.hs:1:  ERROR: Type error - one error found
Problem : Function can't be applied to that argument
Types   : (Int,Int) -> Int (function)
          (Bool,a)         (argument)
Conflict:  f = \a-> plus ((\b-> if b then b else a) True, 3)
```

This explanation is not as thorough, but gets to the point much more quickly, which is important for such a small expression.

The type inference framework we have described supports the notion of type explanation through the discovery of minimal implicants. This is demonstrated directly by Example 27, and is used for subsumption error reporting (see Sections 5.2 and 8.4.2.) The Chameleon type debugger (see Section 9.1) has a type explanation command which can be invoked selectively by the user on program fragments that interest him. The user can, at any stage, ask for the type of some sub-expression, and then ask for an explanation of it. This then directs the user's attention to precisely those locations which were involved. What our system does not report are the deduction steps themselves, but these are arguably fairly elementary and obvious from the relationship between the locations actually involved. If this is unclear to the user, further queries may be made of any unclear sub-expressions.

A common shortcoming of error explanation systems is the excessive size of explanations. Although complete (in the sense that they reflect all of the steps that a particular inference algorithm performed), such explanations are full of repetitive and redundant information which can be a burden to sort through. Furthermore, since these systems are layered on top of an existing inference algorithm, they suffer from the same left-to-right bias when discovering errors. Another problem common to all of these systems is that once run to completion, they never re-solve any constraints, or re-infer any types. Consequently, the view of inference that is presented is always based on a single valuation of the type variables in the program. i.e. you can explore the types in the program, but

only those which are present and fixed at the end of the derivation - you cannot selectively include/exclude any parts at will.

10.3 Inference exploration systems

Interactive tools exist which assist the user in finding type errors manually. These allow the user to uncover the type at any program location. Through examination of the types of sub-expressions, the user gradually works towards the source of the error. Identifiers with suspicious looking types are followed to their definition and further examined. As we did in naming the systems of the previous section, we use the word 'inference' instead of 'type' when talking about these exploration tools.

Neubauer and Thiemann [68] have implemented a type browsing tool, based on their discriminative sum type system [67]. The tool displays the result of type inference, whether successful or not, and allows the user to investigate the result. When the user selects some erroneous type, the system uses highlighting to indicate a reason for that type. The tool, because of the underlying type system, cannot present minimal explanations for bad types.

Chitil, Huch and Simon [36] describe a tool called TypeView which is designed to allow programmers to inspect the types of expressions within a program. In the case of a type error, their system supports a form of algorithmic debugging where the user can select an erroneous looking type, which will hopefully draw him closer to the source of the error. It is not clear how the proposed system would actually work, and precisely what capabilities it would support, since their paper omits formal technical details.

In later work, Chitil [10] presents a system for algorithmic debugging of type errors. The key idea behind his system is that it reports types within a program in a compositional manner. He is more interested in typings, rather than types alone, so that when reporting the type of an expression, his system also prints the types of all sub-expressions. In that way, the programmer can gain some insight as to the relationship between the types of the sub-expressions, and variables appearing in the types of sub-expressions can be named consistently. His system generates compositional typing graphs which do not depend on any polymorphic type environment; everything necessary to explain an expression's type is present. These graphs can then be traversed using a standard algorithmic/declarative

debugging approach that asks the user to classify the inferred types as correct or not. The source of the error is an erroneous expression, whose sub-expressions are all correct.

None of the systems described above allow the user to quickly and efficiently identify conflicting program locations; the process is directed at every step by the user. Furthermore, none of these allow the user to immediately locate precisely those sites which contribute to some suspicious looking type. Our type debugging tool is flexible enough to allow both a user-directed examination of the program's types as well as provide useful, succinct advice to direct the user's search.

10.4 Summary

In this chapter we have explored the space of existing research in the discovery and diagnosis of type errors. We first looked at systems which try to address the problems associated with algorithm $\mathcal{W}$ by modifying the order in which it performs unification while exploring different locations within the source program. Many of these alternatives are based on simple modifications of $\mathcal{W}$. We also looked at a number of alternative type inference systems which avoid the problems of $\mathcal{W}$ altogether by performing inference in terms of a graph, or as in our case, constraints. Following this, we looked at a number of inference explanation and exploration systems. These generate more type information, to give programmers a better understanding of how the inference process has led to an error, than do the schemes that generate more conventional type error messages.

A common shortcoming of almost all existing work is the exclusive focus on standard Hindley/Milner inference. Very few have considered the problem of extending their systems to work with ad hoc overloading (like type classes), which is prevalent in practical language implementations. As far as we are aware, we are also the first to have comprehensively considered the problem of type error reporting for extended algebraic data types.

Chapter 11

Conclusion

Traditional type inference algorithms, like algorithm $\mathcal{W}$, are adequate for confirming that a program is well-typed.[1] When run on ill-typed programs, however, they tend to produce error messages which are poorly focused, and lack the necessary detail to expose the actual cause of a type error. The main reason for this is that in processing a program's abstract syntax tree, these algorithms accumulate and consume type information which cannot easily be recovered. The result is an error message which only explains why the inference algorithm could not proceed any further. What it lacks, most critically, is any description of what it was that led to the impasse. Unfortunately, all known implementations of type inference used in practice employ such algorithms.

In this thesis we have described an implementation of type inference which avoids these problems. Our system is based on a translation of the typing problem for Hindley/Milner, with type annotations and Haskell-style overloading, into constraints and constraint handling rules (CHRs). We use justifications (constraint labels) to record the source locations which are responsible for individual constraints that arise during inference, and refine the standard CHR solver semantics to accurately propagate these justifications to new constraints. This inference system gives a means to accumulate all relevant typing constraints, rather than immediately solving them, as in algorithm $\mathcal{W}$.

An unsatisfiable set of inference constraints represents a type error (or satisfiability error.) To deal with this, we presented a number of constraint reasoning procedures which allow us to calculate minimal sets of culpable constraints, and

[1] We have seen in Chapter 8, where we used implication constraints to delay subsumption testing of declared types, that there are even some cases where a $\mathcal{W}$-based algorithm will needlessly fail because of some arbitrary ordering of program code.

in turn to identify a minimal set of responsible program locations in case of a type error or unexpected result. Building on this, we also described a method for selectively generating type error messages which can focus on any program location involved in a type error. An important feature of our error messages is that they indicate precisely all of the program locations which give rise to the error.

We then reformulated our type system, and inference, to support new, advanced language features. Our new scheme extends the original CHR-based system with implication constraints, which allow us to type programs that use extended algebraic data types and lexically scoped type annotations. We showed that our new inference system is capable of reporting expressive type errors for the new language features, and also demonstrated that our implication-based framework encompasses and extends our previous work. All of the error-reporting principles discussed earlier can be neatly incorporated into the one, more expressive system.

We feel that the constraint-based approach to type inference provides many benefits over systems which unify types immediately. The ability to retain and then reason precisely about typing information gives us much firmer ground to stand on when diagnosing type errors. Other recent work also seems to bear this out [39, 27]. The incorporation of a general-purpose, programmable constraint solver (in our case a CHR solver), which is a novel aspect of our work, has also had many benefits. It has allowed us to naturally extend our type error diagnosis facilities to handle programs that make use of type classes, functional dependencies, and extended algebraic data types, whereas almost all other efforts at type error reporting have focused on standard Hindley/Milner.

11.1 Future work

The systems we have described in this thesis, based on constraints and a programmable solver, are sufficiently generic that they could be extended to cover other language features whose typing can be expressed in terms of CHRs. Consider for example, implicit parameters [55], which behave like type class constraints, or associated type synonyms [8], a language extension which allows a very limited form of type-level programming, that could be implemented in CHR. Support for higher-rank polymorphism would also be desirable. Although we do

not directly represent type schemes in our constraint language, user constraints play the same role. It is not hard to imagine a modification of our system where each polymorphic function argument is given its own user constraint.

The idea of generating constraints, and diagnosing errors in terms of minimal unsatisfiable subsets, and minimal implicants can also be carried over to other program analysis settings. In recent work, Lam [51] has adopted our basic inference scheme to perform kind inference for Haskell. His implementation is part of the Chameleon system, and presents kind errors in the same style as we report type errors. In a similar vein, Lu and Sulzmann [56] plan to incorporate their regular expression pattern matching system into Chameleon, and also make use of the same basic error reporting techniques. They are able to encode regular expression pattern matching in terms of type classes, and expect that informative and intuitive error messages, related to the regular expressions involved, could be automatically generated from the resulting type inference constraints.

It is clear, however, that when we stray too far beyond the typical uses of Haskell type classes, and certainly once we allow for a fully programmable constraint domain, we can run into all sorts of difficulties reporting type errors concisely. As noted in Section 6.1.4, our justification system is not capable of recording and explaining arbitrary CHR derivations. For example, in that section, we limited ourselves to always explaining functional dependency errors in terms of the final propagation rule that fired. With the addition of extended algebraic data types, there are cases where it can be useful to be able to define arbitrary type-level relations/functions which encode various program properties that the type system could automatically check for us (see Example 12.) Fundamentally though, if we allow for arbitrary type-level CHR programs, we make possible at the type level all the sorts of problems we would normally consider 'run-time' errors in a stand-alone CHR interpreter. To handle those satisfactorily, we would ultimately need a dedicated CHR debugger. The integration of a CHR debugger into an interactive type debugging environment is something that could be explored in future.

The type error reporting schemes we described, apart from the ability to attach messages to constraints, are all generic. The systems we have developed have all focused on reporting the cause of type errors. We have generally avoided generating messages which try to guess much about the problem, and automatically suggest fixes. As we have discussed, it is not possible in general to precisely gauge the true source of a type error, and guesses which are incorrect might only

lead to confusion. Nevertheless, other systems [39, 60], have had some success in applying simple heuristics to suggest program fixes to beginning programmers. The execution of most of these heuristics involves simple trial and error changes to the source program, followed by a check to see if the modification has yielded a well-typed program. We think it is possible that such techniques could be improved by making use of minimal unsatisfiable subsets and minimal implicants. This could be investigated further.

The Chameleon type debugger, presented in Chapter 9 represents only a very simple application of our type error diagnosis system. In the future we might look into designing a more user-friendly debugger which allows the user to work in a more natural, and intuitive manner. One problem with the Chameleon type debugger is the effort required to refer to nested values; they are specified via a name which contains the path from the top-level down to that variable. A more intuitive tool would present a graphical interface to the programmer, who could then directly select and query parts of the loaded program. The system could generalise the automatic text message creation of Chapter 6, allowing the user to select different locations as the focus of the messages being generated. It could also allow the user to selectively 'turn-off' parts of the program, which would have the same effect as temporarily replacing them with `undefined`, and observe the result on the rest of the program. There is still a lot of scope for design of a more practical and useful type debugger that could be seamlessly integrated into the normal 'write-compile-debug' programming cycle.

Bibliography

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In Gert Smolka, editor, *Principles and Practice of Constraint Programming - CP97, Third International Conference, Proceedings*, volume 1330 of *LNCS*, pages 252–266. Springer, 1997.

[2] Slim Abdennadher and Thom W. Frühwirth. On completion of constraint handling rules. In *Principles and Practice of Constraint Programming - CP98, Proceedings*, volume 1520 of *LNCS*, pages 25–39. Springer, 1998.

[3] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. In *ACM Letters on Programming Languages*, volume 2, pages 17–30. ACM Press, December 1993.

[4] K. Bernstein and E. Stark. Debugging type errors (full version). Technical report, State University of New York at Stony Brook, November 1995.

[5] B. Braßel. Typehope: There is hope for your type errors. In *16th International Workshop on Implementation and Application of Functional Languages*, September 2004.

[6] Caml. `http://caml.inria.fr/`.

[7] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, FL, 1997.

[8] M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. To appear in Proceedings of The 10th ACM SIGPLAN International Conference on Functional Programming (ICFP 2005), 2005.

[9] J. Cheney and R. Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

[10] O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 193–204. ACM Press, 2001.

[11] Clean. `http://www.cs.ru.nl/~clean/`.

[12] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977.

[13] Luis Damas and Robin 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.

[14] B. Demoen, M. García de la Banda, and P. J. Stuckey. Type constraint solving for parametric and ad-hoc polymorphism. In *Proceedings of the 22nd Australian Computer Science Conference*, pages 217–228. Springer-Verlag, 1999.

[15] T. B. Dinesh and F. Tip. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology*, 10(1):5–55, 2001.

[16] G. J. Duck, S. Peyton Jones, P. J. Stuckey, and M. Sulzmann. Sound and decidable type inference for functional dependencies. In D. A. Schmidt, editor, *13th European Symposium on Programming, ESOP 2004 Proceedings*, volume 2986 of *LNCS*, pages 49–63. Springer-Verlag, 2004.

[17] G.J. Duck, M. Garcia de la Banda, P.J. Stuckey, and C. Holzbaur. The refined operational semantics of constraint handling rules. In B. Demoen and V. Lifschitz, editors, *Logic Programming, 20th International Conference, ICLP 2004, Proceedings*, volume 3132 of *LNCS*, pages 90–104. Springer, 2004.

[18] D. Duggan. Correct type explanation. In *Workshop on ML: ACM SIGPLAN, 1998*, pages 49–58. ACM Press, 1998.

[19] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.

[20] H. Eo, O. Lee, and K. Yi. Proofs of a set of hybrid let-polymorphic type inference algorithms. *New Generation Computing*, 22(1):1–36, 2004.

[21] K. F. Faxén. Haskell and principal types. In *Haskell '03: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 88–97. ACM Press, 2003.

[22] J. Field and F. Tip. Dynamic dependence in term rewriting systems and its application to program slicing. In *PLILP '94: Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming*, volume 844 of *LNCS*, pages 415–431. Springer-Verlag, 1994.

[23] T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*. Springer-Verlag, 1995.

[24] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, October 1998.

[25] M. García de la Banda, P.J. Stuckey, and J. Wazny. Finding all minimal unsatisfiable constraints. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 32–43. ACM Press, 2003.

[26] Glasgow Haskell compiler home page. `http://www.haskell.org/ghc/`.

[27] C. Haack and J. B. Wells. Type error slicing in implicitly typed, higher-order languages. In P. Degano, editor, *12th European Symposium on Programming, Proceedings*, volume 2618 of *LNCS*, pages 284–301. Springer-Verlag, 2003.

[28] C. V. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type classes in Haskell. In D. Sannella, editor, *Programming Languages and Systems - ESOP'94, 5th European Symposium, Proceedings*, volume 788 of *LNCS*, pages 241–256. Springer, April 1994.

[29] B. Han and S-J. Lee. Deriving minimal conflict sets by cs-trees with mark set in diagnosis from first principles. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 29(2):281–286, 1999.

[30] Haskell 98 language report. `http://www.haskell.org/definition/`.

[31] B. Heeren and J. Hage. Type class directives. In M. V. Hermenegildo and D. Cabeza, editors, *Practical Aspects of Declarative Languages, 7th International Symposium, PADL 2005, Proceedings*, volume 3350 of *LNCS*, pages 253–267. Springer, 2005.

[32] B. Heeren, J. Hage, and D. Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, Utrecht University, 2002.

[33] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse 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, September 2003.

[34] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 3–13. ACM Press, 2003.

[35] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(1):253–289, April 1993.

[36] F. Huch, O. Chitil, and A. Simon. Typeview: a tool for understanding type errors. In M. Mohnen and P. Koopman, editors, *Proceedings of 12th International Workshop on Implementation of Functional Languages*, pages 63–69. Aachner Informatik-Berichte, 2000.

[37] Hugs home page. `http://www.haskell.org/hugs/`.

[38] M. Huth and M. Ryan. *Logic in Computer Science: modelling and reasoning about systems*. Cambridge University Press, 2000.

[39] Arjan IJzendoorn, Daan Leijen, and Bastiaan Heeren. The Helium compiler. `http://www.cs.uu.nl/helium`.

[40] P.J. Stuckey J. Bailey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In M. V. Hermenegildo and D. Cabeza, editors, *Practical Aspects of Declarative Languages, 7th International Symposium, PADL 2005, Proceedings*, volume 3350 of *LNCS*, pages 253–267. Springer, 2005.

[41] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture, 1985, Proceedings*, volume 201 of *LNCS*, pages 190–203. Springer-Verlag Inc., 1985.

[42] M. P. Jones. Coherence for qualified types. Research Report YALEU/DCS/RR-989, Yale University, Department of Computer Science, September 1993.

[43] M. P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *9th European Symposium on Programming, ESOP 2000, Proceedings*, volume 1782 of *LNCS*, pages 230–244. Springer, March 2000.

[44] S. Peyton Jones, M. P. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.

[45] S. Peyton Jones and M. Shields. Lexically scoped type variables. `http://research.microsoft.com/~simonpj/papers/scoped-tyvars/scoped.ps`, 2004.

[46] S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types. `http://research.microsoft.com/~simonpj/papers/gadt/gadt.ps.gz`, 2004.

[47] Yang Jun, Greg Michaelson, and Phil Trinder. Explaining polymorphic types. *Computer journal*, 45(4):436–452, 2002.

[48] B.W. Kernighan and D.M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.

[49] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, 1993.

[50] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[51] Edmund Lam. A fresh look at kind validation. Honours thesis, National University of Singapore, 2005.

[52] K. Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.

[53] K. Läufer and M. Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 78–91. ACM Press, 1992.

[54] 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.

[55] Jeffrey Lewis, Mark Shields, Erik Meijer, and John Launchbury. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 108–118, Jan 2000.

[56] K. Zhuo Ming Lu and M. Sulzmann. An implementation of subtyping among regular expression types. In W-N. Chin, editor, *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Proceedings*, volume 3302 of *LNCS*, pages 57–73. Springer, 2004.

[57] M. J. Maher. Herbrand constraint abduction. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), Proceedings*, pages 397–406. IEEE Computer Society, 2005.

[58] B. McAdam. *Repairing Type Errors in Functional Programs*. PhD thesis, Laboratory for Foundatations of Computer Science, The University of Edinburgh, 2001.

[59] B.J. McAdam. On the unification of substitutions in type inference. Technical report, The University of Edinburgh, 1998.

[60] B.J. McAdam. How to repair type errors automatically. In *Trends in Funtional Programming*, 2001.

[61] Bruce J. McAdam. Generalising techniques for type debugging. In *SFP '99: Selected papers from the 1st Scottish Functional Programming Workshop (SFP99)*, pages 50–58. Intellect Books, 2000.

[62] Mercury. `http://www.cs.mu.oz.au/research/mercury/`.

[63] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[64] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, 1997.

[65] Miranda. `http://www.engin.umd.umich.edu/CIS/course.des/cis400/miranda/miranda.ht%ml`.

[66] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1985.

[67] M. Neubauer and P. Thiemann. Discriminative sum types locate the source of type errors. In C. Runciman and O. Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003*, pages 15–26. ACM Press, 2003.

[68] 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.

[69] Nhc98 home page. `http://www.haskell.org/nhc98/`.

[70] M. Odersky, M. Sulzmann, and M Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[71] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[72] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.

[73] T. Sheard. Omega. `http://www.cs.pdx.edu/~sheard/Omega/`.

[74] J.R. Shoenfield. *Mathematical Logic*. Addison-Wesley, 1967.

[75] Sicstus prolog. `http://www.sics.se/isl/sicstuswww/site/index.html`.

[76] V. Simonet and F. Pottier. Constraint-based type inference for guarded algebraic data types. Research Report 5462, INRIA, 2005.

[77] P. J. Stuckey, M. Sulzmann, and J. Wazny. The Chameleon type debugger. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, pages 247–258. Computer Research Repository (http://www.acm.org/corr/), 2003.

[78] 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. ACM Press, 2003.

[79] 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.

[80] P. J. Stuckey, M. Sulzmann, and J. Wazny. Improved inference for checking type annotations. Technical Report TRA2/05, The National University of Singapore, 2005.

[81] P. J. Stuckey, M. Sulzmann, and J. Wazny. Type error reporting in the Hindley/Milner system with extensions. *ACM Transactions on Programming Languages and Systems*, 2005. Submitted for review.

[82] P.J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems*, 2004. To appear.

[83] M. Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000.

[84] M. Sulzmann and J. Wazny. Chameleon. `http://www.comp.nus.edu.sg/~sulzmann/chameleon`.

[85] M. Sulzmann, J. Wazny, and P. J. Stuckey. A framework for extended algebraic data types. Submitted to POPL 2006.

[86] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM Press, 1989.

[87] J. A. Walz and G. F. Johnson. A maximum flow approach to anomaly isolation in unification-based incremental type inference. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 44–57. ACM Press, January 1986.

[88] M. Wand. Finding the source of type errors. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 38–43. ACM Press, 1986.

[89] H. Xi. ATS. `http://www.cs.bu.edu/~hwxi/ATS/ATS.html`.

[90] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 224–235. ACM Press, 2003.

[91] J. Yang, G. Michaelson, and P. Trinder. Helping students understand polymorphic type errors. In *1st Annual Conference of the LTSN Centre for Information and Computer Sciences*, pages 11–19, 2000.

[92] J. Yang, J. Wells, P. Trinder, and G. Michaelson. Improved type error reporting. In *Proceedings of 12th International Workshop on Implementation of Functional Languages*, volume 2011 of *LNCS*, pages 71–86. Springer, 2000.

Appendix A

CHR-based type inference proofs

A.1 Soundness and completeness

We relate our CHR-based inference scheme against HM(=) which is the system described in Figure 4.1 where only equations are allowed to appear in constraints. Note that Sulzmann [83], has formally verified that HM(=) is equivalent to the standard presentation of Hindley/Milner where equality constraints are resolved via unification. In fact, our results are even slightly stronger and apply to arbitrary instances of HM(X) (as long as constraint properties can be expressed in terms of CHRs).

For convenience, we will make use of a slightly different (but equivalent) formulation of rule (Let) from Figure 4.1. To simplify things we will directly couple quantifier introduction with let statements.

$$\frac{D'', \Gamma'' \vdash e_1 : t'' \quad \sigma = \forall fv(D'') - fv(\Gamma'').D'' \Rightarrow t'' \qquad D', \Gamma''.f : \sigma \vdash e_2 : t'}{D', \Gamma'' \vdash \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : t'}$$

First, we verify soundness. In preparation, we slightly generalise the relation introduced among CHRs P and environments E, Γ and Γ_λ. We assume that $\mathit{Termfv}(\{x_1 : \sigma_1, ..., x_n : \sigma_n\}) = \{x_1, ..., x_n\}$.

We define $P_E, E \sim \Gamma, \Gamma_\lambda$ iff

1. Γ_λ only consists of simple types, $\mathit{Termfv}(\Gamma) = E$, $fv(\Gamma) \subseteq fv(\Gamma_\lambda)$,

2. $P_E = \{f(t, x) \Longleftrightarrow C \mid f \in E\}$,

3. $\Gamma = \left\{ f : \forall \bar{a}.C' \Rightarrow t \mid \begin{array}{l} f \in E, fv(\Gamma_\lambda) = \bar{t}, f(t,x) \wedge x = \langle \bar{t} \rangle \longrightarrow^*_{P_E} C', \\ \bar{a} = fv(C',t) - fv(\Gamma_\lambda) \end{array} \right\}$. [1]

In the upcoming soundness proof we make use of the following lemma. A proof can be found in [83].

Lemma 5 *Let* $C, \Gamma \vdash e : t$ *and* $\models C' \supset C, t = t'$. *Then,* $C', \Gamma \vdash e : t'$.

Theorem 2 *Let* $P_E, E \sim \Gamma, \Gamma_\lambda$, *and* $E, \Gamma_\lambda, e \vdash_{Cons} (C \mid t)$, *and* $E, \Gamma_\lambda, e \vdash_{Def} P$ *such that* $C \longrightarrow^*_{P,P_E} D$ *and* e *is let-realisable. Then* $D, \Gamma \cup \Gamma_\lambda \vdash e : t$.

Proof: The proof proceeds by structural induction over e. We only show some of the more interesting cases.

Case (Abs): We have

$$\frac{E, \Gamma_\lambda.x : t_{l_1}, e \vdash_{Cons} (C \mid t) \quad t_{l_1}, t_{l_2}, t' \text{ fresh}}{E, \Gamma_\lambda, (\lambda x_{l_1}.e)_{l_2} \vdash_{Cons} (C, (t_{l_2} = t' \to t)_{l_2}, (t_{l_1} = t')_{l_1} \mid t_{l_2})}$$

$$\frac{t_{l_1} \text{ fresh} \quad E, \Gamma_\lambda.x : t_{l_1}, e \vdash_{Def} P}{E, \Gamma_\lambda, (\lambda x_{l_1}.e)_{l_2} \vdash_{Def} P}$$

Note that w.l.o.g. we can assume that fresh variable t_{l_1} is the same for both premises (otherwise we could simply replace one by the other by applying a substitution).

By assumption $C, (t_{l_2} = t' \to t)_{l_2}, (t_{l_1} = t')_{l_1} \longrightarrow^*_{P,P_E} D$. Hence, $C \longrightarrow^*_{P,P_E} D'$ such that $\models D \supset D', (t_{l_2} = t' \to t)_{l_2}, (t_{l_1} = t')_{l_1}$ (1). Application of the induction hypothesis yields $D', \Gamma \cup \Gamma_\lambda.x : t_{l_1} \vdash e : t$. By application of rule (Abs) we find that $D', \Gamma \cup \Gamma_\lambda \vdash (\lambda x_{l_1}.e)_{l_2} : t_{l_1} \to t$ (2). Lemma 5 applied to (1) and (2) yields $D, \Gamma \cup \Gamma_\lambda \vdash (\lambda x_{l_1}.e)_{l_2} : t_{l_2}$ and we are done.

[1] Note that in C' we can further simplify constraints such as $x = \langle \bar{t} \rangle$ by building the m.g.u. but this does not matter here. The important point is that the type represented by the CHR and recorded in the environment are (logically but not necessarily syntactically) equivalent.

Case (Let): We have that

$$\frac{E \cup \{f\}, \Gamma_\lambda, e_2 \vdash_{Cons} (C \mid t)}{E, \Gamma_\lambda, \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \vdash_{Cons} (C \mid t)}$$

$$\frac{\begin{array}{c} E, \Gamma_\lambda, e_1 \vdash_{Cons} (C' \mid t') \quad \Gamma_\lambda = \{x_1 : t_1, \ldots, x_n : t_n\} \quad x', r \text{ fresh} \\ E, \Gamma_\lambda, e_1 \vdash_{Def} P_1 \quad E \cup \{f\}, \Gamma_\lambda, e_2 \vdash_{Def} P_2 \\ P = P_1 \cup P_2 \cup \{f(t', x') \Longleftrightarrow C', x' = \langle t_1, \ldots . t_n | r \rangle\} \end{array}}{E, \Gamma_\lambda, \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 \vdash_{Def} P}$$

We assume $C \longrightarrow^*_{P_E,P} D$ and that the expression is let-realisable and, $C' \longrightarrow^*_{P_E,P} D'$ for some D'.[2] Application of the induction hypothesis to sub-expression e_1 yields $D', \Gamma \cup \Gamma_\lambda \vdash e_1 : t'$ (1). Let $\bar{a} = fv(D', t') - fv(\Gamma_\lambda)$ (recall that $fv(\Gamma) \subseteq fv(\Gamma_\lambda)$). We can easily verify that $P_{E\cup\{f\}}, E \cup \{f\} \sim \Gamma.(f : \forall \bar{a}.D' \Rightarrow t'), \Gamma_\lambda$, and hence are in position to apply the induction hypothesis to e_2 which yields $D, \Gamma.(f : \forall \bar{a}.D' \Rightarrow t') \cup \Gamma_\lambda \vdash e_2 : t$ (2). From (1) and (2) we conclude (via our reformulated (Let) rule) that $D, \Gamma \cup \Gamma_\lambda \vdash \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : t$ and we are done. □

Next, we consider completeness. In preparation, we formally define subsumption among type schemes. Let F be a first-order formula. We define $F \vdash (\forall \bar{a}.C \Rightarrow t) \preceq (\forall \bar{b}.C' \Rightarrow t')$ iff $F \models C' \supset \exists \bar{a}.(C, t = t')$ where universal quantifiers are left implicit and w.l.o.g. there are no name clashes between $\bar{a}$ and $\bar{b}$. In the case of $True \vdash \sigma \preceq \sigma'$ we write $\vdash \sigma \preceq \sigma'$ for short.

We extend the subsumption relation to environments. We define $F \vdash \Gamma \preceq \Gamma'$ iff and $F \vdash \sigma \preceq \sigma'$ for each $f : \sigma \in \Gamma$ and $f : \sigma' \in \Gamma'$. We assume that Γ' and Γ define the same set of variables.

We will make use of the following result stated in [23].

Lemma 6 (CHR Soundness) *Let P, C and D such that $C \longrightarrow^*_P D$. Then, $P \models C \leftrightarrow \bar{\exists}_{fv(C)}.D$.*

[2]Note that in the constraint-based formulation of Hindley/Milner realisability is not strictly necessary because we can always "type" ill-typed expressions under the *False* assumptions. Recall that our formulation of (∀ Intro) in Figure 4.1 is lazy. However, we silently assume that constraints appearing in intermediate judgements are satisfiable.

In the following, we prove a slightly stronger statement than necessary so that the induction will go through.

Theorem 3 *Let* $P_E, E \sim \Gamma', \Gamma_\lambda$*, and* $fv(\Gamma) \subseteq fv(\Gamma_\lambda)$*, and* $\vdash \Gamma' \preceq \Gamma$*, and* $D', \Gamma \cup \Gamma_\lambda \vdash e : t'$*. Then* $E, \Gamma_\lambda, e \vdash_{Cons} (C \mid t)$ *and* $E, \Gamma_\lambda, e \vdash_{Def} P$ *for some constraint* C*, type* t *and set of CHRs* P *such that* $P_E, P \models D' \supset \bar{\exists}_{fv(\Gamma_\lambda)}.(C, t = t')$*.*

Proof: The CHR Soundness Lemma guarantees that if $C_1 \longrightarrow_P^* C_2$ then $P \models C_1 \leftrightarrow \bar{\exists}_{fv(C_1)}.C_2$. Hence, we could equivalently express $P_E, P \models D' \supset \bar{\exists}_{fv(\Gamma_\lambda)}.(C, t = t')$ by stating that $\models D' \supset \bar{\exists}_{fv(\Gamma_\lambda)}.(C', t = t')$ where $C \longrightarrow_{P_E,P}^* C'$.

The proof proceeds by structural induction. We only show the case for let-defined functions.

Case (Let): We have that

$$\frac{D'', \Gamma'' \vdash e_1 : t'' \quad \sigma = \forall fv(D'') - fv(\Gamma'').D'' \Rightarrow t'' \qquad D', \Gamma''.f : \sigma \vdash e_2 : t'}{D', \Gamma'' \vdash \mathsf{let}\ f = e_1\ \mathsf{in}\ e_2 : t'}$$

where Γ'' is a short-hand for $\Gamma \cup \Gamma_\lambda$.

Application of the induction hypothesis to the left premise gives us $E, \Gamma_\lambda, e_1 \vdash_{Cons} (C_1 \mid t_1)$ and $E, \Gamma_\lambda, e_1 \vdash_{Def} P_1$ such that $P_E, P_1 \models D'' \supset \bar{\exists}_{fv(\Gamma_\lambda)}.(C_1, t_1 = t'')$. From that, we conclude that $\vdash (\forall \bar{b}.C_1' \Rightarrow t_1) \preceq (\forall fv(D'') - fv(\Gamma'').D'' \Rightarrow t'')$ where $C_1 \longrightarrow_{P_E,P}^* C_1'$ and $\bar{b} = fv(C_1', t_1) - fv(\Gamma_\lambda)$. We also find that $P_E \cup \{f(t_1, x) \iff C_1, x = \langle t_1', \ldots . t_n' | r \rangle\}, E \cup f \sim \Gamma'.(\forall \bar{b}.C_1' \Rightarrow t_1), \Gamma_\lambda$ where $fv(\Gamma_\lambda) = t_1', \ldots . t_n'$ and x and r are fresh. We are in position to apply the induction hypothesis to the right premise and find that $E \cup \{f\}, \Gamma_\lambda, e_2 \vdash_{Cons} (C \mid t)$ and $E \cup \{f\}, \Gamma_\lambda, e_2 \vdash_{Def} P_2$ such that $P_E \cup \{f(t_f, x) \iff C_1, t_f = t_1, x = \langle t_1', \ldots . t_n' | r \rangle\}, P_2 \models D' \supset \bar{\exists}_{\Gamma_\lambda}.(C, t = t')$. Note that the final statement still holds if we add P_1. An easy observation shows that constraint and rule generation for (Let) applies; hence we are done. □

A.2 Type annotations

We verify soundness and completeness for type inference in the presence of type annotations. First, we consider soundness. We will simply need to extend the proof of Theorem 2.

Theorem 4 *Let* $P_E, E \sim \Gamma, \Gamma_\lambda$, $E, \Gamma_\lambda, e \vdash_{Cons} (C \mid t)$ *and* $E, \Gamma_\lambda, e \vdash_{Def} P$ *such that* $C \longrightarrow^*_{P,P_E} D$, *all subsumption conditions* (f_a, f) *in* P *hold and* e *is let-realisable. Then,* $D, \Gamma \cup \Gamma_\lambda \vdash e : t$.

Proof: We only consider the type annotation case. We have:

$$\frac{E \cup \{f_a\}, e_2 \vdash_{Cons} (C \mid t)}{E, \Gamma_\lambda, \begin{array}{l}(f :: C' \Rightarrow t')_l \\ f = e_1\end{array} \text{ in } e_2 \vdash_{Cons} (C \mid t)}$$

$$\frac{\begin{array}{c}E \cup \{f_a\}, \Gamma_\lambda, e_1 \vdash_{Cons} (C'_1 \mid t'_1) \quad \Gamma_\lambda = \{x_1 : t_1, \ldots, x_n : t_n\} \quad t, x, r \text{ fresh} \\ E \cup \{f_a\}, \Gamma_\lambda, e_1 \vdash_{Def} P_1 \quad E \cup \{f_a\}, \Gamma_\lambda, e_2 \vdash_{Def} P_2 \\ P = P_1 \cup P_2 \cup \left\{ \begin{array}{l} f_a(t,x) \Longleftrightarrow t = t_l, (t_l = t')_l, (C')_l \\ f(t,x) \Longleftrightarrow f_a(t,x), C'_1, t = t'_1, x = \langle t_1, \ldots . t_n | r \rangle \end{array} \right\}\end{array}}{E, \Gamma_\lambda, \begin{array}{l}(f :: C' \Rightarrow t')_l \\ f = e_1\end{array} \text{ in } e_2 \vdash_{Def} P}$$

Let $\sigma = \forall fv(C', t').C' \Rightarrow t'$. Note that $P_E \cup \{f_a(t,x) \Longleftrightarrow t = t_l, (t_l = t')_l, (C')_l\}, E \cup \{f_a\} \sim \Gamma.f : \sigma, \Gamma_\lambda$. Application of the induction hypothesis to e_2 yields $D, \Gamma.f : \sigma \cup \Gamma_\lambda \vdash e_2 : t_2$ (1). By assumption $C \longrightarrow^*_{P_E,P} D$ and the expression is let-realisable. Hence, $C'_1 \longrightarrow^*_{P_E,P} D'$ for some D' (2). Application of the induction hypothesis to e_1 yields $D', \Gamma.f : \sigma \cup \Gamma_\lambda \vdash e_1 : t'_1$ (3). By assumption the subsumption condition for (f_a, f) holds. That is, $P_E, P \models f_a(t,x) \leftrightarrow f(t,x)$ (4). From (2) and the CHR Soundness Lemma 6 we conclude that $P_E, P \models f(t,x) \leftrightarrow \bar{\exists}_{t,x}.(D', t = t'_1, x = \langle t_1, \ldots . t_n | r \rangle)$. Note that $\models f_a(t,x) \leftrightarrow \bar{\exists}_{t,x}.(C', t = t')$. Hence, $\models C' \supset f_a(t', x)$. Hence, in combination with (4) we obtain $\models C' \supset \bar{\exists}_{t',x}.(D', t' = t'_1, x = \langle t_1, \ldots . t_n | r \rangle)$. Therefore $\models C' \supset \bar{\exists}_{t',\Gamma_\lambda}.(D', t' = t'_1)$. We conclude that $\models C' \supset \bar{\exists}_{t',\Gamma_\lambda}.\phi(D', t' = t'_1)$ (5) for some substitution ϕ not affecting $fv(t', \Gamma_\lambda)$. We apply ϕ to

(3) and obtain $\phi(D'), \Gamma.f : \sigma \cup \Gamma_\lambda \vdash e_1 : \phi(t'_1)$. Weakening (Lemma 5) yields $C', \Gamma.f : \sigma \cup \Gamma_\lambda \vdash e_1 : t'$. Hence, quantifier introduction and another weakening step yields $D, \Gamma.f : \sigma \cup \Gamma_\lambda \vdash e_1 : \sigma$ (6). Rule (Let) applied to (1) and (6) yields $D, \Gamma.f : \sigma \cup \Gamma_\lambda, \begin{array}{l}(f :: C' \Rightarrow t')_l \\ f = e_1\end{array}$ in $e_2 : t_2$ and we are done. □

Note that the soundness result holds for any type extension which can be expressed in terms of CHRs.

Next we address completeness. For convenience, we couple the rule for quantifier introduction and type annotations and make use of the following new rule instead:

$$\text{(LetA)}\frac{D'', \Gamma''.f : \sigma \vdash e_1 : t'' \quad D' \vdash (\forall fv(D'', t'') - fv(\Gamma'').D'' \Rightarrow t'') \preceq \sigma \quad D', \Gamma''.f : \sigma \vdash e_2 : t' \quad fv(\sigma) = \emptyset}{D', \Gamma'' \vdash \text{let } \begin{array}{l}(f :: \sigma)_l \\ f = e_1\end{array} \text{ in } e_2 : t'}$$

We show that any derivation with the original rules can be also be achieved with the new rule. Hence, it is sufficient to only consider the new (LetA) rule form now on.

Lemma 7 *Let $C, \Gamma \vdash e : \sigma$ derived with the original (LetA) rule. Then $C', \Gamma \vdash e : t'$ derived with the new (LetA) rule such that*

1. *$\sigma = t$: $\models C \supset \bar{\exists}_\Gamma.(C', t = t')$*
2. *otherwise: $C \vdash (\forall fv(C', t') - fv(\Gamma).C' \Rightarrow t') \preceq \sigma$*

Proof: The proof proceeds by induction over the derivation. We only show the interesting case. Assume we have that

$$\frac{C, \Gamma.f : \sigma \vdash e : \sigma \quad C, \Gamma.f : \sigma \vdash e' : t \quad fv(\sigma) = \emptyset}{C, \Gamma \vdash \text{let } \begin{array}{l}(f :: \sigma)_l \\ f = e\end{array} \text{ in } e' : t}$$

The induction hypothesis applied to the left premise yields that $D'', \Gamma.f : \sigma \vdash e : t''$ derived with the new rule such that $C \vdash (\forall fv(D'', t'') -$

$fv(\Gamma).D'' \Rightarrow t'') \preceq \sigma$. Induction applied to the right premise yields $D', \Gamma.f : \sigma \vdash e' : t'$ such that $\models C \supset \bar{\exists}_\Gamma.(D', t' = t)$ (note that $fv(\sigma) = \emptyset$). Lemma 5 allows us to conclude that $C, \Gamma.f : \sigma \vdash e' : t'$. Hence, all conditions for application of the new rule (LetA) are fulfilled. We find that $C, \Gamma \vdash \mathsf{let}\ \begin{matrix}(f :: \sigma)_l \\ f = e\end{matrix}\ \mathsf{in}\ e' : t'$ and we are done. □

Completeness in the presence of type annotations does not hold in general. See upcoming Example 87. We only show completeness for standard Hindley/Milner, i.e. equations are the only primitive constraints. This is due to the following observation. Consider the statement $D \vdash (\forall \bar{a}.C_1 \Rightarrow t_1) \preceq (\forall \bar{b}.C_2 \Rightarrow t_2)$ which is a variant of the side-condition in rule (LetA). Assume constraints contain only equations and $(\forall \bar{b}.C_2 \Rightarrow t_2)$ is closed. We normalise type schemes by applying m.g.u.'s. Hence, we find $D \vdash (\forall \bar{a}'.t'_1) \preceq (\forall \bar{b}'.t'_2)$ (1) for some appropriate $\bar{a}'$, t'_1, $\bar{b}'$ and t'_2. By definition, (1) is equivalent to $\models D \supset \exists \bar{a}'.t'_1 = t'_2$. A silent assumption is that all constraints appearing in intermediate judgements are satisfiable. Hence, we can argue that $\exists \bar{a}'.t'_1 = t'_2$ has a m.g.u. ϕ where none of the variables in t'_2 has been instantiated. Therefore $\vdash (\forall \bar{a}', fv(\pi(t'_2)).\pi(t'_2) = t'_1 \Rightarrow t'_1) \preceq (\forall \bar{b}'.t'_2)$ (2) holds where π is a renaming substitution on $fv(t'_2)$. Note that by definition, (2) is equivalent to $\models \exists \bar{a}', fv(\pi(t'_2)).(\pi(t'_2) = t'_1, t'_1 = t'_2)$ which is indeed true.

We can summarise our observation in the following lemma.

Lemma 8 *Let $D \vdash (\forall \bar{a}.C_1 \Rightarrow t_1) \preceq (\forall \bar{b}.C_2 \Rightarrow t_2)$ such that constraints are satisfiable and contain equations only and $(\forall \bar{b}.C_2 \Rightarrow t_2)$ is closed. Then,*
$\vdash (\forall \bar{a} fv(\pi(C_1), \pi(t_2)).\pi(C_2), C_1, \pi(t_2) = t_1 \Rightarrow t_1) \preceq (\forall \bar{b}.C_2 \Rightarrow t_2)$ where π is a renaming substitution on $fv(C_2, t_2)$.

Effectively, the lemma states that if a inferred type subsumes a closed annotated type, the subsumption condition must hold under the *True* assumption if we add a renamed copy of the annotation to the inferred type (which is what we do in our inference scheme). Here is an example highlighting why it is crucial to include the annotation when performing type inference for annotations.

Example 86 Consider

```
let f x = let g::Int->Int
              g y = x
```

```
          in g
in f
```

Inference would infer the type $\forall t_y.t_y \to t_x$ for g if we would not take into account the type annotation. Hence, the subsumption check would fail. □

Theorem 5 *Let $P_E, E \sim \Gamma, \Gamma_\lambda$, $fv(\Gamma) \subseteq fv(\Gamma_\lambda)$ and $D', \Gamma \cup \Gamma_\lambda \vdash e : t'$ such that constraints in primitive functions only mention equations. Then, $E, \Gamma_\lambda, e \vdash_{Cons} (C \mid t)$ and $E, \Gamma_\lambda, e \vdash_{Def} P$ for some constraint C, type t and set of CHRs P such that $P_E, P \models D' \supset \bar{\exists}_{fv(\Gamma_\lambda)}.(C, t = t')$ and all subsumption conditions (f_a, f) in P hold.*

Proof: We only consider type annotations.

Case (LetA): We have that

$$\frac{D'', \Gamma''.f : \sigma \vdash e_1 : t'' \quad D' \vdash (\forall fv(D'', t'') - fv(\Gamma'').D'' \Rightarrow t'') \preceq \sigma \qquad D', \Gamma''.f : \sigma \vdash e_2 : t' \quad fv(\sigma) = \emptyset}{D', \Gamma'' \vdash \mathsf{let} \begin{array}{l}(f :: \sigma)_l \\ f = e_1\end{array} \mathsf{in}\ e_2 : t'}$$

where Γ'' is a short-hand for $\Gamma \cup \Gamma_\lambda$ and $\sigma = \forall fv(C', t').C' \Rightarrow t'$.

Note that $P_E \cup \{f_a\}, E \cup \{f_a\} \sim \Gamma.f : \sigma, \Gamma_\lambda$ where $P_{E \cup \{f_a\}}$ is a short-hand for $P_E \cup \{f_a(t, x) \Longleftrightarrow (t' = t)_l, (C')_l\}$. Then, application of the induction hypothesis to the left premise yields $E \cup \{f_a\}, \Gamma_\lambda, e_1 \vdash_{Cons} (C_1 \mid t_1)$ and $E \cup \{f_a\}, \Gamma_\lambda, e_1 \vdash_{Def} P_1$ such that $P_{E \cup \{f_a\}}, P_1, \models D'' \supset \bar{\exists}_{fv(\Gamma_\lambda)}.(C_1, t_1 = t'')$. From that, we conclude that $\vdash (\forall \bar{b}.C_1' \Rightarrow t_1) \preceq (\forall fv(D'') - fv(\Gamma'').D'' \Rightarrow t'')$ (1) where $C_1 \longrightarrow^*_{P_{E \cup \{f_a\}}, P} C_1'$ and $\bar{b} = fv(C_1', t_1) - fv(\Gamma_\lambda)$.

We also find that $P_{E \cup \{f_a\}}, E \cup f_a \sim \Gamma.(\forall \bar{b}.C_1' \Rightarrow t_1), \Gamma_\lambda$ We are able to apply the induction hypothesis to the right premise and find that $E \cup \{f_a\}, \Gamma_\lambda, e_2 \vdash_{Cons} (C \mid t)$ and $E \cup \{f_a\}, \Gamma_\lambda, e_2 \vdash_{Def} P_2$ such that $P_{E \cup \{f_a\}}, P_2 \models D' \supset \bar{\exists}_{\Gamma_\lambda}.(C, t = t')$. Note that the final statement still holds if we add P_1. An easy observation shows that constraint and rule generation for (Let) applies. We are almost done.

We still need to verify that the subsumption condition for (f_a, f) holds. By assumption, we have $D' \vdash (\forall fv(D'', t'') - fv(\Gamma'').D'' \Rightarrow$

$t'') \preceq \sigma$. From that and (2) and Lemma 8 we can conclude that the subsumption condition must hold. Thus, we are done. □

Example 87 Here is a variation of an example mentioned in [21].

```
class Foo a b where foo :: a->b->Int
instance Foo Int b
instance Foo Bool b

test y = let f :: c->Int
             f x = foo y x
         in f y
```

Our inference scheme fails here (so do all other standard inference algorithms). Note that `test` may be given types $Int \rightarrow Int$ and $Bool \rightarrow Int$. The constraint $Foo\ t_y\ t_x$ arising out of the program text can be either satisfied by $t_y = Int$ or $t_y = Bool$. Hence, we conclude that there is no principal type which is the source for failure. □

In fact, there are other examples where "incompleteness" arises due to the order of inference rather than lack of principal types. Our inference scheme does not impose a fixed order, so we can support an improved form of inference for checking annotations. We refer to [80] for more details.

Appendix B

All-minimal-unsatisfiable-subsets proofs

Proofs of various statements from Chapter 7 can be found herein.

Proof(Lemma 1): Let us reason by contradiction, and thus assume that we can partition the nodes into two non-empty disjoint subsets $V_1 \cup C_1$ and $V_2 \cup C_2$ where V_i are variable nodes and C_i are constraint nodes such that there is no path from a node in $V_1 \cup C_1$ to a node in $V_2 \cup C_2$. Since we have also assumed that C is a minimal unsatisfiable set, C_1 (C_2) must be satisfiable. Thus, there exist θ_1 (θ_2) solutions of C_1 (C_2) on the variables V_1 (V_2). But then $\theta = \theta_1 \cup \theta_2$ must be a proper valuation (since the variables are disjoint) and thus a solution of C. Contradiction. □

Proof(Lemma 2): Let us first assume D is unsatisfiable. Then, the only minimal unsatisfiable constraint set $C \supseteq D$ must be $C = D$. Clearly the result holds since $C - D$ is empty.

Let us now assume D is satisfiable. Suppose to the contrary that we can partition C into non-empty disjoint subsets C_1 and C_2 which are not connected in $g(C)$ with variables W removed. Now, $D \cup C_1$ and $D \cup C_2$ must be satisfiable and hence have solutions θ_1 and θ_2, respectively. The only variables that C_1 and C_2 can share are W, otherwise they would be connected. But every solution of D gives these variables the same value. Hence, $\theta_1(v) = \theta_2(v), v \in W$. Thus

$\theta = \theta_1 \cup \theta_2$ is a correct valuation and a solution of C. Contradiction. □

Proof(Lemma 3): For a minimal unsatisfiable subset M we have that $\neg sat(M)$, and for each $M' \subset M$, $sat(M')$. Suppose $U = M - S_1 \neq M$, then $sat(U)$, but then also $sat(U \cup S_1)$ and $U \cup S_1 \supseteq M$ which is a contradiction. □

Proof(Lemma 4): Suppose to the contrary that minimal unsatisfiable subset $M \subseteq D \cup P$ such that $M \cap U \neq \emptyset$. Now $sat(M - U)$ since it is a strict subset of a minimal unsatisfiable subset M, but then $M - U \supseteq D$, and hence $M - U \Rightarrow D \Rightarrow U$, and thus $sat((M - U) \cup U)$ or equivalently $sat(M)$. Contradiction. □

Proof(Theorem 1): The invariant maintained by the algorithm is that for any call $\mathsf{all\_min\_unsat8}(D_0, \_, P_0, T_0, \_, \mathbf{A})$, D_0 is satisfiable, T_0 is satisfiable, $D_0 \subseteq T_0 \subseteq D_0 \cup P_0$, and $\mathbf{A}$ contains any minimal unsatisfiable subsets of $D_0 \cup P_0$ which are not supersets of D_0. The call returns all minimal unsatisfiable subsets M of $D_0 \cup P_0$.

Clearly the invariant holds for the initial call.

At the end of the while loop. In the first case $T = D_0 \cup P_0$ is satisfiable in which case there are no unsatisfiable subsets of $D_0 \cup P_0$, and the call returns correctly. In the second case $T - \{c\}$ is satisfiable. The recursive call $\mathsf{all\_min\_unsat8}(D_0, \_, P_0 - \{c\}, T - \{c\}, \_, \mathbf{A})$ satisfies the invariants since D_0 is satisfiable, $T - \{c\}$ is satisfiable, and $\mathbf{A}$ contains the minimal unsatisfiable subsets of $D_0 \cup P_0 \supseteq D_0 \cup P_0 - \{c\}$ that are not supersets of D_0.

After the return from the recursive call we have, by the invariant, that $\mathbf{A}$ holds all minimal unsatisfiable subsets of $D_0 \cup P_0 - \{c\}$.

We add c to D_0 and check satisfiability. If the resulting set is unsatisfiable we add it to $\mathbf{A}$ if there is no smaller subset already in $\mathbf{A}$. This is correct since any minimal unsatisfiable subset M of $D_0 \cup \{c\}$ must be in $\mathbf{A}$ already by the calling invariant, since it will not be a superset of D_0. If none exists then $D_0 \cup \{c\}$ is a minimal unsatisfiable subset. Now $\mathbf{A}$ already contains all minimal unsatisfiable subsets of $D_0 \cup P_0$ which (a) are not supersets of D_0 or (b) which are subsets of

$D_0 \cup P_0 - \{c\}$. Hence, with the possible addition $D_0 \cup \{c\}$, we have determined all minimal unsatisfiable subsets of $D_0 \cup P_0$, and, upon return, correctness is maintained.

Otherwise, $D_0 \cup \{c\}$ is satisfiable. For the subsequent recursive call, $\mathsf{all\_min\_unsat8}(D_0 \cup \{c\}, \_, P_0 - \{c\}, D_0 \cup \{c\}, \_, \mathbf{A})$, the satisfiability conditions clearly hold. $\mathbf{A}$ already contained all the minimal unsatisfiable subsets of $D_0 \cup P_0 = (D_0 \cup \{c\}) \cup (P_0 - \{c\})$ not supersets of D_0. We need to ensure it now holds all the subsets of $D_0 \cup P_0$ not supersets of $(D_0 \cup \{c\})$. These are minimal unsatisfiable subsets of $D_0 \cup P_0 - \{c\}$ which are already in $\mathbf{A}$.

On returning from the recursive call we have that $\mathbf{A}$ contains all minimal unsatisfiable subsets of $(D_0 \cup \{c\}) \cup (P_0 - \{c\}) = D_0 \cup P_0$, and hence the return is correct. □

Appendix C

Summary of Chameleon type debugger commands

The Chameleon type debugger supports the following commands.

`:type <expression>` Infers and displays the type of the given expression. In the event of an error, highlights contributing locations.

`:explain (<expression>) (<type>|<typescheme>)` Explains why the given expression has a type that is an instance of the given type. Contributing locations are highlighted.

`:declare (<reference>) (<type>|<typescheme>)` Declares that the definition referred to has the nominated type. This declaration is internal to the debugger, and does not affect the program source.

`:print` Displays the loaded source program.

`:set <flag>` Configures the debugger by setting internal options.
Available flags include:

`global` Selects global type/error explanations.

`local` Selects local type/error explanations.

`solver ?` Selects the version of the internal constraint solver to use. The available values are 0, 1 and 2. Higher numbers lead to more thorough explanations, but involve slower solvers.

`:help` Prints command help.

`:quit` Quits the debugger.

Index