Improving type-error messages in functional languages

Bastiaan Heeren
`bastiaan@cs.uu.nl`

November 15, 2000

Abstract

Type systems are indispensable in modern higher-order, polymorphic languages. An important explanation for Haskell's and ML's popularity is their advanced type system, which helps a programmer in finding program errors before the program is run. Although the type system in a polymorphic language is important, the reported error messages are often poor. The goal of this research is to improve the quality of error messages for ill-typed expressions.

In a unification-based system type conflicts are often detected far from the source of the conflict. To indicate the actual source of a type conflict an analysis of the complete program is necessary. For example, if there are three occurrences where `x::Int` and only one where `x::Bool`, we expect that there is something wrong with the occurrence of `x::Bool`. The order in which subexpressions occur should not influence the reported error. Unfortunately, this is not the case for unification-based systems.

This article presents another approach to inferring the type of an expression. A set of typing rules is given together with a type assignment algorithm. From the rules and the program at hand we construct a set of constraints on types. This set replaces the unification of types in a unification-based system. If the set is inconsistent, some constraints are removed from the set and error messages are constructed. Several heuristics are used to determine which constraint is to be removed. With this approach we increase the chance that the actual source of a type conflict is reported. As a result we are able to produce more precise error messages.

Contents

Chapter 1

Introduction

Polymorphic type systems are indispensable in modern programming languages. An important explanation for Haskell's and ML's popularity is their advanced type system combined with an inference system, which helps a programmer in finding program errors before the program is run. We believe that programmer productivity is substantially increased by a clear and advanced type system. However, programmer productivity is hampered by the poor type error messages of the compilers and interpreters for such languages. Consider for example the following definition:

```
f = \x -> case x of
             0 -> False
             1 -> "one"
             2 -> "two"
             3 -> "three"
```

The error message produced by Hugs (an interpreter for Haskell) for this program is:

```
ERROR "example.hs" (line 1): Type error in case expression
*** Term           : "one"
*** Type           : String
*** Does not match : Bool
```

In a case-expression all the right-hand sides must have the same type. If we compare the types at the right-hand sides we see that there are three expressions of type `String` and only one of type `Bool`. It is therefore reasonable to assume that the expression of type `Bool` is the one that is incorrect. The error message that is produced however strongly depends on the order in which the alternatives for the case expression are given. If we swap the first two alternatives the error message changes into:

```
ERROR "example.hs" (line 1): Type error in case expression
*** Term           : False
*** Type           : Bool
*** Does not match : [Char]
```

Because the construction of an error message for an expression depends on the order of subexpression, type errors can be confusing.

Type inference algorithms in current compilers are all based on the type assignment algorithm in [Mil78] and [DM82]. A number of articles explain the working of this algorithm. In [Jon99] a specification of a Haskell type-checker is given. Because this specification is written in Haskell, it is very comprehensible, especially for Haskell users. Although a lot of research has been done in the field of type inferencing, the type inferencing algorithms in current compilers suffer from the following problems:

- Every expression is assumed to have a correct type until an inconsistency is found. This is a very *local* approach.
- An error message is often a very brief explanation why the expression is ill-typed. It can be difficult to trace why a type is assigned to an expression.
- Only one type error message, the first inconsistency, is reported. Most compilers for imperative languages can report several errors. We would like to have the same behaviour in our compilers.
- No suggestions for improvements are given.

In this paper we present a different approach to type checking in order to improve the generated error messages. Instead of applying substitutions in a unification algorithm, we collect constraints. While we are solving the constraints we can tell if the set of constraints is consistent. An inconsistency in the set implies that there is a type error in the expression. With a minimal correction in the set of constraints we make the set consistent again. The advantages of this approach are:

- The decision which subexpression has an incorrect type is made by taking the whole program into account. We try to find as much evidence as possible to determine the type of an expression. In fact we will do *global* (instead of *local*) typing.
- The type inference program reports all inconsistencies in the constraint set. Therefore we can have multiple error messages for an expression.
- The type error message tells us exactly where something is wrong and why. We are interested in the source and an explanation of the type conflict.

Of course there is a trade off between speed and the quality of an error message. In order to type check a program on a global level we have to keep track of a lot of information. Hence, the new algorithm will be more time consuming. The goal of this research is to get good error messages, speed is not a main issue.

The idea to construct a set of constraints on types while we are inferring the type of an expression is not new. *BANE* (the Berkeley Analysis Engine) is intended to construct constraint-based program analyses. [AH95], [AFFS98] and [Aik99] explain how a set of constraints can be used to analyse a program. A type inferencer for a lambda calculus is expressed in *BANE*, but no suggestions are given how the constructed set of constraints can contribute to improving error messages.

In section 5 we introduce *equality graphs* to keep equalities between assigned type variables and type constants. Although in other articles graphs are used to detect a type conflict, equality graphs have not been introduced so far. In [GVS96] a set of constraints is constructed to suggest a correction of an ill-typed expression. The constraints are translated into a graph, containing vertices that arise due to the structure of a term and vertices that represent control information. The structure of graphs constructed in [McA99] depends on the type of the program at hand instead of the syntax tree. Vertices of the graph represent the types of fragments of the code.

Some research has been done to give a clear explanation why a type is assigned to an expression. In [BS93] an approach is presented for explaining why a unification failure for two types occurred. This approach can be used in an interactive system to explain why a type is assigned to an expression and it helps the programmer to acquire a better understanding about the working of the type system. A similar facility is desirable in our approach to type checking.

[McA98] discusses the unification of substitutions instead of the unification of types. The problem of the example above, which shows that the order of the alternatives of a case expression influences the error message, is referred to as a *left-to-right bias* and is removed if we unify substitutions.

[WJ86] points out that in a unification-based system conflicts are typically detected fairly far from the site of error. To avoid reporting misleading error messages, a set of type equations is collected that can be inconsistent. Multiple contradictory hypotheses about the type of a variable imply that the variable *predominantly* has the type that satisfies most of the hypotheses. Unfortunately which conflict is reported depends on the order in which rewrite rules are applied. This approach resembles the approach presented in this paper since both try to correct an inconsistency with a minimal correction. [Wan86] makes a similar observation and presents an approach for finding the true source of a type error. The algorithm keeps track of the *reasons* for the type checker to make deductions about the types of variables.

In [Jun00] an inference algorithm is given that reports conflicting sites rather than the site where a conflict is detected. Besides reporting conflicting sites, our approach is able to give a good prediction which site is erroneous.

This paper is organised as follows. Section 2 introduces an expression language together with a type system. Six type inference rules are presented to show which set of constraints is constructed for an expression. In section 3 we discuss how the set of constraints is solved. In section 4 we prove that after the set of constraints is solved, we have found the correct type for an expression. We prove that our type inference rules are equivalent to the rules presented in [DM82], which implies that our rules are sound and complete. In section 5 we discuss what to do if we have an inconsistent set of constraints. To restore the consistency in a set of constraints we remove constraints from the set, using some *cost* or *profit* function. The removal of a constraint results in the construction of an error message. Section 6 suggests some directions for future research. Section 7 contains a summary of the results and some final remarks.

Chapter 2

Collecting constraints

Functional programming languages use a type assignment algorithm to check if an expression is well typed. Often such an algorithm is based on unification. A unification algorithm fails and produces an error message if the expression is ill-typed. This error message indicates that there is a possible type error at the position where the unification failed. Unfortunately it often happens that the actual error is somewhere else. With a different approach, which is more precise than a unification algorithm, we try to report error messages for an ill-typed expression that are more precise.

Instead of applying substitutions to types, a set of constraints is generated from an expression. A constraint represents a delayed substitution. An expression is well typed if and only if the set of constraints is consistent. If the set is inconsistent we have a lot of information to determine where the error is.

In this chapter we will focus on collecting the constraints for an expression. In sections 2.1 and 2.2 we define a language for expressions and for types. In section 2.3 we discuss how a constraint is represented. A set of type inference rules is given in section 2.4. In section 2.5 the strategy behind the inference rules is explained. Finally section 2.6 shows how constraints are collected for an example expression.

2.1 Expression language

We start with the introduction of an expression language. We assume that there is no syntactic-sugar in the expression. Although syntactic sugar is convenient for programming languages, it does not introduce anything new when assigning a type to an expression. The expression language is defined as:

```
data Expr         = Variable     String
                  | Literal      String
                  | Apply        Expr Expr
                  | Lambda       String Expr
                  | Case         Expr Alternatives
                  | Let          Declarations Expr

data Alternatives = Empty
                  | Alternative  Expr Expr Alternatives

data Declarations = Empty
                  | Declaration  String Expr Declarations
```

An expression can be a variable, a literal, an application, a lambda expression, a case expression or a let expression. A variable is represented by a string and should be bound. Literals are expressions with a constant type: for instance they can be integers, characters or booleans. Function application is left associative, which means that $f\ x\ y$ is interpreted as $(f\ x)\ y$. Notice that an expression of the form $(\backslash x\ y \rightarrow expr)$ is just a shorthand for $(\backslash x\ \rightarrow (\backslash y \rightarrow expr))$.

Case expressions

A case expression can be seen as a selector function for data types. It has an expression on which we are going to pattern match, and a number of alternatives. For each case expression there is at least one alternative. An alternative is a pattern together with an expression which has to be evaluated if the expression matches the pattern. Some functional languages only allow a single constructor in a pattern but this restriction is not necessary for our language. Because our goal is to assign a type to an expression we are only interested in the type of the pattern.

Let expressions

A let expression has a number of declarations together with a body. A declaration contains a variable together with an expression. We do not allow patterns in the left-hand-side of the declaration. This does not restrict the language since a pattern is only syntactic sugar. All the variables in a declaration should be in the same binding group. For instance: the expression

```
let id = \x -> x;
    y  = id 3;
in y
```

should be replaced by:

```
let id = \x -> x;
in let y = id 3;
   in y
```

because *id* and *y* are not in the same binding group.

Data types

Without a lot of effort our language has data types. A data type has one or more constructors. Furthermore it can have a number of type variables. In Haskell we can define a data type for a binary tree that has elements of an arbitrary type *a* in its leaves:

```
data Tree a = Bin (Tree a) (Tree a)
            | Leaf a
```

In this example the constructors `Bin` and `Leaf` are functions that create a value of type `Tree`. Each constructor of a data type implies the existence of a function with a given type. The types of the constructors of our binary tree are:

```
Bin  :: Tree a -> Tree a -> Tree a
Leaf :: a                -> Tree a
```

It is easy to obtain the type of a constructor function from its data definition. The constructor is a function that takes its arguments as parameters and returns a value of the type for which it is defined. Because we can see a constructor function as a normal function we treat it as if it were a defined variable. To be able to recognise the difference between a constructor and a variable, we use identifiers starting with a capital letter for constructors and identifiers starting with a lower case letter for variables.

2.2 Type language

In section 2.1 we introduced an expression language. Not every expression that is part of this language is meaningful. With a type checker we can reject expressions that are ill-typed and construct an error message that explains why the expression is rejected. A type is assigned to each well typed expression in the expression language. First we give a representation of a type: a type can be a variable, a constant or an application of two types.

```
data Type = Variable   Int
          | Constant   String
          | Apply      Type Type
```

To distinguish type variables we give them numbers. Two variables with the same number refer to the same type variable. For the representation of type constants we use identifiers starting with a capital, for instance `Constant "Int"` and `Constant "Bool"`. Application is left associative which means that `Pair Int Bool` is interpreted as `(Pair Int) Bool`. The type arrow ($\rightarrow$) is not represented by a separate alternative in the definition of a type. The type `Bool` $\rightarrow$ `Int` is represented as:

```
    Apply (Apply (Constant "->")
                 (Constant "Bool"))
          (Constant "Int")
```

The type constant `"->"` is a primitive type that represents a type arrow. Because the type arrow does not have a separate alternative, we have a more general definition and this helps us later. Often when we want to present a type, we map the type variables to a list of identifiers starting with a lowercase letter, for instance $[{}^{\prime\prime}a^{\prime\prime}, {}^{\prime\prime}b^{\prime\prime}, {}^{\prime\prime}c^{\prime\prime}, \ldots]$. The only reason for this is convenience.

2.3 Constraints

We want to transform the type information inside an expression into constraints. A constraint is a relation between two types. We need two kinds of constraints to capture all the type information.

The first type of constraints is the equality constraint. This constraint guarantees that after this constraint is solved, the two types are exactly equal. We use $(\equiv)$ to write down an equality, for instance $(a \equiv Int)$ and $(Int \rightarrow Int \equiv a \rightarrow a)$. The equality constraint $(Int \equiv Bool)$ can occur but is clearly inconsistent. An equality constraint has some nice properties: it is reflexive, commutative and transitive. These properties help us to reason about equalities.

Because let declarations are a part of our expression language, a type can be polymorphic. Hindley-Milner uses *type schemes* to deal with polymorphism. A type scheme is a type that can be preceded by a number of $\forall$ quantifiers. For instance, the type of the polymorphic identity function is $\forall a\ (a \rightarrow a)$. At this point we take a different approach. We do not allow $\forall$ quantifiers in our type language, but instead we implicitly quantify every variable in a type. If we want to make an instance of a possibly polymorphic type we use another kind of constraint. We write $a < b$ to express that type a is an instance of type b. Because there can be monomorphic variables in b, we keep a set M for each instance constraint, containing all the monomorphic variables in b. In the rest of the paper we write $a <_M b$.

2.4 Type inference rules

Now that we have defined the expression language, the type language and the constraints, we can give type inference rules to assign a type to an expression. The type inference rules are syntax directed: for each of the six alternatives of the expression language there is one type rule. The rule $(\vdash e : \tau,\ \mathcal{A})$ expresses that expression e has type τ in the environment $\mathcal{A}$. In the environment we store type information about all the free variables in an expression.

$$[VAR] \qquad \frac{\alpha \ is \ fresh}{\vdash \ x \ : \ \alpha, \ [x \mapsto \alpha]}$$

$$constraints: \ no$$

The inference rule for a variable is straightforward: a new type variable α is assigned as type. In the environment we store that type α has been assigned to the variable x. No constraints are generated for a variable.

$$[LIT] \qquad \frac{}{\vdash \ literal \ : \ \text{primitive type}, \ \emptyset}$$

$$constraints: \ no$$

The literals are the easiest expressions to type because they all have their own (constant) type. For instance, the expression 3 has the primitive type Int. For a literal the type environment is empty and no constraints are generated.

$$[APP] \qquad \frac{\begin{array}{c}\alpha \ is \ fresh \\ \vdash \ f \ : \ \tau_f, \ \mathcal{A}_f \\ \vdash \ e \ : \ \tau_e, \ \mathcal{A}_e\end{array}}{\vdash \ (f \ e) \ : \ \alpha, \ \mathcal{A}_f \cup \mathcal{A}_e}$$

$$constraint: \ \tau_f \equiv \tau_e \rightarrow \alpha$$

In general, the result of an application of a function of type $a \rightarrow b$ to an argument of type a is a value of type b. In the algorithm of Damas and Milner we unify the types τ_f and $\tau_e \rightarrow \alpha$, where τ_f is the type of the function, τ_e is the type of the argument and α is a fresh variable. Instead of unifying the two types we store their equality in a constraint. The result of the application has type α. The type environment for an application is the combination of the environments of the sub-expressions. Because of this union it is possible that a variable occurs twice (or more) in a type environment with bindings to different type variables.

$$[ABS] \qquad \frac{\begin{array}{c}\alpha \ is \ fresh \\ \vdash \ e \ : \ \tau_e, \ \mathcal{A}\end{array}}{\vdash \ (\backslash x \rightarrow e) \ : \ \alpha \rightarrow \tau_e, \ \mathcal{A}\backslash x}$$

$$constraints: \ \{\sigma \equiv \alpha \mid (x \mapsto \sigma) \in \mathcal{A}\}$$

A lambda expression binds variables in an expression. All the variables that are bound are present in the type environment, together with their assigned type variable. We introduce a new type α as the type for the variable x. For each variable that we bind we generate exactly one constraint: the type that we assigned to the variable must be equal to α. In the new type environment we remove all the variables that are bound by this lambda. We use the notation $\mathcal{A}\backslash x$ for this removal.

$$[CASE] \quad \frac{\begin{array}{ll} \alpha\ is\ fresh & \\ \beta\ is\ fresh & \\ \vdash\ p\ :\ \tau_p,\ \mathcal{A}_p & \\ \vdash\ p_i\ :\ \tau_{p_i},\ \mathcal{A}_{p_i} & (for\ 1 \leq i \leq n) \\ \vdash\ e_i\ :\ \tau_{e_i},\ \mathcal{A}_{e_i} & (for\ 1 \leq i \leq n) \end{array}}{\begin{array}{l} (case\ p\ of \\ \quad p_1 \rightarrow e_1 \\ \quad \cdots \\ \quad p_n \rightarrow e_n)\ :\ \beta,\ \mathcal{A}_p \bigcup_{1 \leq i \leq n} (\mathcal{A}_{e_i} - \mathcal{A}_{p_i}) \end{array}}$$

$$constraints \begin{cases} \alpha \equiv \tau_p & \\ \alpha \equiv \tau_{p_i} & (for\ 1 \leq i \leq n) \\ \beta \equiv \tau_{e_i} & (for\ 1 \leq i \leq n) \\ \{\sigma \equiv \tau\ \mid\ (x \mapsto \sigma) \in \mathcal{A}_{p_i}, (x \mapsto \tau) \in \mathcal{A}_{e_i}\} & (for\ 1 \leq i \leq n) \end{cases}$$

For a case expression two fresh variables are created, α and β. Type α represents the type of the patterns of a case expression. Because all the types of the patterns must be equal, we construct the constraints $\alpha \equiv \tau_{p_1},\ \alpha \equiv \tau_{p_2},\ \cdots,\ \alpha \equiv \tau_{p_n}$. The switch expression also has this type: $\alpha \equiv \tau_p$. We do something similar for the types of the expressions on the right-hand side of the alternatives. We construct the constraints $\beta \equiv \tau_{e_1},\ \beta \equiv \tau_{e_2},\ \cdots,\ \beta \equiv \tau_{e_n}$, where β represents the type of the right-hand side.

We also have to take into account that we can introduce variables in the patterns that we can use in the expression on the right-hand side. For example, consider the Haskell function that computes the number of leaves in a tree, as defined on page 8:

```
leaves = \x -> case x of
                   Bin  l r -> leaves l + leaves r
                   Leaf a   -> 1
```

In this example the variable l is introduced in the pattern of the *Bin* and is used in the expression on the right side. Variables that occur free in a pattern bind the variables that occur free in the expression on the right. For each binding we construct an equality constraint.

The type of a case expression is the type on the right-hand side of the alternatives, which is β. The new environment is the environment of the switch expression merged with the environments of the right-hand sides, where we leave out all the variables that are bound in a pattern. Assumptions about constructors that are in the environments of the patterns, should also be included in the environment.

$$[LET] \qquad \frac{\begin{array}{ll} \alpha_i \ is \ fresh & (for\ 1 \leq i \leq n) \\ \vdash \ e_i \ : \ \tau_{e_i}, \ \mathcal{A}_{e_i} & (for\ 1 \leq i \leq n) \\ \vdash \ b \ : \ \tau_b, \ \mathcal{A}_b & \end{array}}{\begin{array}{rlll} \vdash (& let & v_1 & = \ e_1; \\ & & \cdots & \\ & & v_n & = \ e_n; \\ & in & b) & : \ \tau_b, \ \mathcal{A}_1 \cup \mathcal{A}_2 \end{array}}$$

$$constraints \left\{ \begin{array}{lll} \{ \quad \alpha_i \equiv \tau_{e_i} & | & 1 \leq i \leq n \qquad \} \\ \{ \quad x \equiv \alpha_i & | & (v_i \mapsto x) \in (\mathcal{A}_{e_1} \cup \ldots \cup \mathcal{A}_{e_n}) \quad \} \\ \{ \quad x <_{\mathcal{A}_2} \alpha_i & | & (v_i \mapsto x) \in \mathcal{A}_b \qquad \} \end{array} \right.$$

$$where \left\{ \begin{array}{lllll} \mathcal{A}_1 & = & \mathcal{A}_b & \backslash & \{v_1 \ldots v_n\} \\ \mathcal{A}_2 & = & (\mathcal{A}_{e_1} \cup \ldots \cup \mathcal{A}_{e_n}) & \backslash & \{v_1 \ldots v_n\} \end{array} \right.$$

The fresh type variables $\alpha_1, \ \alpha_2, \ \ldots, \ \alpha_n$ represent the types of the declarations. The type of the expression in a declaration should be equal to the new type variable that was assigned to this declaration, in other words: α_i should be equal to τ_{e_i} for every $i \in [1 \ldots n]$. If one of the declared variables is used in another declaration, we generate an equality constraint. The α_i of the declared variable should be equal to the type variable that is stored in the environment of the expression where it is used. The reason that the types are equal is because all the declarations are part of the same binding group.

We generate an instance constraint if one of the declared variables is used in the body of the let expression. The variables that are free in the declarations of the let (the variables in the environment) are monomorphic variables in the instance constraint.

The type of a let expression is the type of the expression in the body. We combine the environment of the body and the environments of the declarations, but we leave out the assumptions about the declared variables.

Top-level constraints

With the six type rules we can generate a set of constraints for every possible expression. In functional languages we are used to refer to functions defined in other modules. In Haskell a set of *prelude functions* is imported to define the most elementary operations. We would like to be able to do something similar for our expression language. Suppose we have a set of types for the prelude functions (and for other functions that are imported). In this set we also include the types of the constructor functions that we use. An example of such a set is:

```
plus  :: Int -> Int -> Int
map   :: (a -> b) -> [a] -> [b]
even  :: Int -> Bool
...
Empty :: [a]
Cons  :: a -> [a] -> [a]
Bin   :: Tree a -> Tree a -> Tree a
Leaf  :: a -> Tree a
...
```

At top-level we generate an equality constraint for each variable that occurs in this set and in the environment. For instance, for the assumption set $[plus \mapsto \tau_0, Leaf \mapsto \tau_1]$ the following equality constraints are constructed:

$$\begin{array}{rcl} \tau_0 & \equiv & Int \rightarrow Int \rightarrow Int \\ \tau_1 & \equiv & \tau_2 \rightarrow Tree\ \tau_2 \end{array}$$

where τ_2 is a fresh type variable.

2.5 Strategy

An advantage of the inference rules is that the type, the environment and the set of constraints are collected using one bottom-up recursive walk over the expression tree. Because the inference rules are also syntax-directed, the algorithm corresponding to the rules is a straightforward translation of the rules. There are no special rules for type instantiation or type generalisation.

If you take a closer look at the inference rules you see that some of the constraints that are constructed appear to be superfluous. Indeed we could have constructed a smaller set of constraints which results in the same type for a well typed expression. However, the goal for this new approach is not to type well typed expressions, but to give a better error message in case an ill-typed expression occurs. This improved error message pin-points the most likely spot of the type error in the program we try to infer the type of.

Superfluous constraints are generated for lambda abstractions, case expressions and let expressions. In an example the rule for a case expression is discussed. Justification for the extra constraints for lambda abstractions and let expressions follows the same line of reasoning.

In the $[CASE]$ rule the fresh variables α and β are introduced. Respectively they correspond to the type of the left-hand side and the right-hand side of the alternatives. Also the switch expression must have a type equal to α. Instead of generating the constraints $\alpha \equiv \tau_p,\ \alpha \equiv \tau_{p_1},\ \ldots,\ \alpha \equiv \tau_{p_n}$ for n alternatives, we can also generate the constraints $\tau_p \equiv \tau_{p_1},\ \ldots,\ \tau_p \equiv \tau_{p_n}$. This trick reduces the number of constraints and the number of type variables by one. If we do the same for β we get a smaller constraint set. Consider the next ill-typed expression:

```
case True of
       0 -> ...;
       1 -> ...;
       2 -> ...;
```

The constraints for this expression are:

$$\begin{array}{rcl} \alpha & \equiv & Bool_{(True)} \\ \alpha & \equiv & Int_{(0)} \\ \alpha & \equiv & Int_{(1)} \\ \alpha & \equiv & Int_{(2)} \end{array}$$

Between parentheses behind the types are the expressions from which the types originate. This set of constraints is obviously inconsistent. However, only one constraint has to be removed to restore the consistency. If we decrease the number of constraints the following set of constraints is constructed:

$$\begin{array}{rcl} Bool_{(True)} & \equiv & Int_{(0)} \\ Bool_{(True)} & \equiv & Int_{(1)} \\ Bool_{(True)} & \equiv & Int_{(2)} \end{array}$$

There is still an inconsistency in the set of constraints, but this time we can not restore the consistency by the removal of one constraint. In the previous set of constraints we were still able to conclude that α (probably) has type Int and not type $Bool$. In the optimised set we lost this valuable information. We use an extra constraint to prevent that a type variable that can be incorrect, shows up in many different constraints.

2.6 Example

The next example makes the rules more understandable. We show how constraints are generated for the expression:

```
let y = 3;
    i = \x -> (x,y);
in (i True,i)
```

In this expression there are two occurrences of a tuple. A tuple is a data structure containing two values. The notation for a tuple is syntactic sugar and has to be replaced by a constructor. In our example we replace the tuple (a, b) by the expression $Tuple2\ a\ b$. The type of the constructor $Tuple2$ is $a \rightarrow b \rightarrow Tuple2\ a\ b$, which will be presented as the type $a \rightarrow b \rightarrow (a, b)$. The two let declarations do not belong to the same binding group. We move the declaration of y to a nested let expression. The adjusted expression is:

```
let y = 3;
in let i = \x -> Tuple2 x y;
   in Tuple2 (i True) i
```

The expression can be analysed as two nested let-expressions. First we will walk bottom-up through the two declarations collecting constraints, then we do the same for the remaining expression. The first declaration, $y = 3$, doesn't cause any difficulties. Because the expression is a literal, no constraints are generated.

$$\frac{}{\vdash 3 : Int,\ \emptyset}\ [LIT]$$

We assign the new type variable τ_0 to the declared variable y. If we later on refer to y, we use τ_0 as its type. Because the type of the declaration is Int, the $[LET]$ rule gives us the first constraint:

$$\#0:\quad \tau_0 \equiv Int$$

The second declaration, $i = \lambda x \rightarrow Tuple2\ x\ y$, is a little bit more difficult. Consider the first part of the derivation:

$$\frac{\dfrac{}{\vdash Tuple2 : \tau_1,\ [Tuple2 \mapsto \tau_1]}\ [VAR] \qquad \dfrac{}{\vdash x : \tau_2,\ [x \mapsto \tau_2]}\ [VAR]}{\vdash Tuple2\ x : \tau_3,\ [Tuple2 \mapsto \tau_1,\ x \mapsto \tau_2]}\ [APP]$$

Two new type variables are assigned to the variables $Tuple2$ and x. The variables are put into the assumption set together with their assigned type. The rule for application introduces the new type variable τ_3 and returns this as the type for the application. The relation between the types τ_1, τ_2 and τ_3 is stored in a constraint:

$$\#1:\quad \tau_1 \equiv \tau_2 \rightarrow \tau_3$$

The set of assumptions from the two subexpressions are merged together. We continue with the derivation:

$$\frac{\dfrac{\vdash Tuple2\ x : \tau_3,\ [Tuple2 \mapsto \tau_1,\ x \mapsto \tau_2] \qquad \dfrac{}{\vdash y : \tau_4,\ [y \mapsto \tau_4]}\ [VAR]}{\vdash Tuple2\ x\ y : \tau_5,\ [Tuple2 \mapsto \tau_1,\ x \mapsto \tau_2,\ y \mapsto \tau_4]}\ [APP]}{\vdash \lambda x \rightarrow Tuple2\ x\ y : \tau_6 \rightarrow \tau_5,\ [Tuple2 \mapsto \tau_1,\ y \mapsto \tau_4]}\ [ABS]$$

$$\begin{array}{ll} \#2: & \tau_3 \equiv \tau_4 \rightarrow \tau_5 \\ \#3: & \tau_6 \equiv \tau_2 \end{array}$$

We assign the type variable τ_4 to variable y. The next application is similar to the previous one: it introduces τ_5 and adds #2 to the constraints. The lambda abstraction introduces τ_6 as a new type variable. All the assumptions concerning variable x are removed from the assumption set. For the assumption $x \mapsto \tau_2$ constraint #3 is generated. The result type of the expression is $\tau_6 \rightarrow \tau_5$. Again, a new type variable (τ_7) is introduced in case we refer to the declared variable i. Constraint #4 makes sure that τ_7 and the derived type $\tau_6 \rightarrow \tau_5$ are equal.

$$\#4: \quad \tau_7 \equiv \tau_6 \rightarrow \tau_5$$

Now we need a derivation for the expression $Tuple2\ (i\ true)$. A new type variable is assigned to the variable $Tuple2$:

$$\frac{}{\vdash Tuple2 : \tau_8,\ [Tuple2 \mapsto \tau_8]}\ [VAR]$$

After this we make a derivation for $i\ true$. The variables τ_9 and τ_{10} are introduced and constraint #5 is generated:

$$\frac{\dfrac{}{\vdash i : \tau_9,\ [i \mapsto \tau_9]}\ [VAR] \qquad \dfrac{}{\vdash True : Bool,\ \emptyset}\ [LIT]}{\vdash i\ True : \tau_{10},\ [i \mapsto \tau_9]}\ [APP]$$

$$\#5: \quad \tau_9 \equiv Bool \rightarrow \tau_{10}$$

When we apply $Tuple2$ on $(i\ true)$ constraint #6 is generated:

$$\frac{\vdash Tuple2 : \tau_8,\ [Tuple2 \mapsto \tau_8] \quad \vdash i\ True : \tau_{10},\ [i \mapsto \tau_9]}{\vdash Tuple2\ (i\ True) : \tau_{11},\ [Tuple2 \mapsto \tau_8,\ i \mapsto \tau_9]}\ [APP]$$

$$\#6: \quad \tau_8 \equiv \tau_{10} \rightarrow \tau_{11}$$

We assign τ_{12} to the second occurrence of the variable i. When we apply $Tuple2\ (i\ true)$ to i, the result for this expression is the type variable τ_{13} together with constraint #7:

$$\frac{\begin{array}{l}\vdash i : \tau_{12},\ [i \mapsto \tau_{12}] \\ \vdash Tuple2\ (i\ True) : \tau_{11},\ [Tuple2 \mapsto \tau_8,\ i \mapsto \tau_9]\end{array}}{\vdash Tuple2\ (i\ True)\ i : \tau_{13},\ [Tuple2 \mapsto \tau_8,\ i \mapsto \tau_9,\ i \mapsto \tau_{12}]}\ [APP]$$

$$\#7: \quad \tau_{11} \equiv \tau_{12} \rightarrow \tau_{13}$$

Let's think about the goal again: we want to show how a set of constraints is generated for the expression:

```
let y = 3;
in let i = \x -> Tuple2 x y;
   in Tuple2 (i True) i
```

We collected eight constraints (#0 . . . #7) and constructed the following three derivations:

$$\begin{array}{l}\vdash 3 : Int,\ \emptyset \\ \vdash \lambda x \rightarrow Tuple2\ x\ y : \tau_6 \rightarrow \tau_5,\ [Tuple2 \mapsto \tau_1,\ y \mapsto \tau_4] \\ \vdash Tuple2\ (i\ True)\ i : \tau_{13},\ [Tuple2 \mapsto \tau_8,\ i \mapsto \tau_9,\ i \mapsto \tau_{12}]\end{array}$$

Furthermore we assigned τ_0 to the declared variable y and τ_7 to i. The let rule should be applied twice to collect the last constraints and to find the type for the complete expression.

$$\frac{\begin{array}{l}\vdash \lambda x \rightarrow Tuple2\ x\ y : \tau_6 \rightarrow \tau_5,\ [Tuple2 \mapsto \tau_1,\ y \mapsto \tau_4] \\ \vdash Tuple2\ (i\ True)\ i : \tau_{13},\ [Tuple2 \mapsto \tau_8,\ i \mapsto \tau_9,\ i \mapsto \tau_{12}]\end{array}}{\vdash let\ i = \ldots;\ in\ \ldots : \tau_{13}, [Tuple2 \mapsto \tau_1,\ Tuple2 \mapsto \tau_8,\ y \mapsto \tau_4]}\ [LET]$$

$$\begin{array}{lll}\tau_9 & <_{\{\tau_4\}} & \tau_7 \\ \tau_{12} & <_{\{\tau_4\}} & \tau_7\end{array}$$

In the assumption set of the body of the let we find two assumptions about i, and thus two instance constraints are generated for these two assumptions. Because the variable y, used in the declaration of i, is bound outside the let expression its type τ_4 is in the set of monomorphic variables. The new assumption set contains three assumptions.

$$\frac{\begin{array}{l}\vdash 3 : Int,\ \emptyset \\ \vdash let\ i = \ldots;\ in\ \ldots : \tau_{13}, [Tuple2 \mapsto \tau_1,\ Tuple2 \mapsto \tau_8,\ y \mapsto \tau_4]\end{array}}{\vdash let\ y = \ldots;\ in\ \ldots : \tau_{13}, [Tuple2 \mapsto \tau_1,\ Tuple2 \mapsto \tau_8]}\ [LET]$$

$$\tau_4 \quad <_{\emptyset} \quad \tau_0$$

We have found another instance constraint because of the occurrence of variable y in the assumption set. There are no monomorphic variables for this constraint. There are two assumptions left in the environment, both concerning variable $Tuple2$. The polymorphic type for this constructor function is $\forall \alpha, \beta : \alpha \rightarrow \beta \rightarrow (\alpha,\ \beta)$. Remember that this type is just a shorthand-notation. Two instances are introduced for this polymorphic function and they are directly translated into equality constraints. Because there are two polymorphic variables (α and β) in the type, we have to introduce two new type variables for each instance.

$$\begin{array}{ll}\#8 : & \tau_8 \equiv \tau_{14} \rightarrow \tau_{15} \rightarrow (\tau_{14},\ \tau_{15}) \\ \#9 : & \tau_1 \equiv \tau_{16} \rightarrow \tau_{17} \rightarrow (\tau_{16},\ \tau_{17})\end{array}$$

The result is a set of ten equality constraints, three instance constraints and the inferred type τ_{13}. In chapter 4 we define the function *Solve* to solve a set of constraints for a type. It is interesting to show that

$$Solve(C, \tau_{13}) =_{\alpha} ((Bool,\ Int), a \rightarrow (a, Int))$$

where C is the set of collected constraints. We'll leave this proof to the interested reader.

Chapter 3

Solving constraints

The previous section described a method for constructing a set of constraints from an expression. In this section we present an algorithm for solving the set of constraints and for assigning a type to the expression. We assume that the expression is well typed and that there is not an inconsistency in the set of constraints. In chapter 5 we show how we can extend the type checker such that it accepts ill typed expressions. We show how we can determine the source of the error and give a method to solve an inconsistency in the set of constraints.

In order to find the type of the expression and to detect possible type errors, we keep a state while we solve the constraints that have been collected. We store the following information in the state:

- **set of constraints:** both the set of equality constraints and the set of instance constraints are stored in the state. The constraints in the state are the constraints that have not been taken into account yet.
- **result type:** the type assigned to the expression while collecting the constraints should also be stored in the state. Because of the application of substitutions, this type may change while we are solving the constraints.
- **equality classes:** an equality class is a tuple (V, C), where V is a set of type variables and C is a set of type constants. All the type variables and the type constants in an equality class have the same type. Every type variable occurs exactly once in an equality class.
- **erroneous constraints:** we keep a set of constraints that cannot be solved. The constraints in this set are translated into an error message that is reported to the programmer.

In the initial state we store the constraints and the type of an expression. For each type variable v we construct an equality class $(v, \emptyset)$. The set of erroneous constraints is empty. Each time one constraint is chosen and solved. Solving a constraint means converting the information in a constraint to a modification of the set of equality classes. Once the set of constraints is empty all type information is inside the equality classes, and we present the type of the expression.

Solving a set of constraints consists of two parts: most constraints can be simplified (translated into other constraints) or solved (translated into a modification of the equality classes). This process is called *simplification* of the state and is explained in section 3.1. Some constraints require the *decomposition* of a type variable and this is presented in section 3.2. In section 3.3 we give an algorithm to solve the state. Finally, two examples in section 3.4 illustrate the working of the algorithm.

3.1 Simplification

Most constraints can be solved or simplified in a straightforward way. Simplifying a constraint means translating a constraint into one or more different constraints. A constraint is solved if it is removed from the state and translated into a modification of the equality classes. First the equality constraints are discussed, then the instance constraints.

Equality constraints

In section 2.2 we defined a type language: a type can be a variable, a constant or an application of two types. Because an equality constraint contains two types, we give a rule for each of the nine possible combinations:

- **Constant** $c_1 \equiv$ **Constant** c_2**:** this constraint can either be true or false. If the constants c_1 and c_2 are equal we remove this constraint from the set of constraints. On the other hand, if the constants are different we cannot satisfy this constraint. The constraint is transferred from the set of constraints to the set of erroneous constraints. Every erroneous constraint will eventually be reported to the programmer.

- **Constant** $c \equiv$ **Variable** v**:** we add constant c to the equality class containing variable v. It is possible that the equality class of v already contains a number of constants. We require that every type variable and every constant in an equality class has the same type. What happens if the set of constants, including c, contains different constants? Although the class is obviously inconsistent we continue with our simplification. Later on this inconsistency will be resolved.

- **Constant** $c \equiv$ **Apply** $s\ t$**:** this constraint cannot be solved. The constraint is transferred from the set of constraints to the set of erroneous constraints.

- **Variable** $v_1 \equiv$ **Variable** v_2**:** the type variables v_1 and v_2 have to be in the same equality class. We combine the class containing v_1 and the class containing v_2 into a new equality class. A combination of two equality classes is the union of the two sets of variables and the union of the two sets of constants. It is not a problem if the class contains different constants. If the type variables v_1 and v_2 are already in the same equality class, we don't have to modify their equality class.

- **Variable** $v \equiv$ **Apply** $s_1\ s_2$**:** solving this constraint is postponed. This constraint requires the decomposition of variable v, which is discussed in 3.2. We keep the constraint in the set and handle it later.
- **Apply** $s_1\ s_2 \equiv$ **Apply** $t_1\ t_2$**:** this constraint can be replaced by two new constraints: $s_1 \equiv t_1$ and $s_2 \equiv t_2$. If possible we simplify the two new constraints.

There are of course three more cases, but since $a \equiv b$ is equal to $b \equiv a$ one can change the order of types in an equality constraint.

Instance constraints

Instance constraints take care of the specialisation of polymorphic types. For instance, to specialise $\forall a, b : a \to b \to a$ we need a substitution for the two type variables that are bound by the $\forall$ quantifier. If we use the substitution $[a := Int, b := a \to a]$, the resulting type is $Int \to (a \to a) \to Int$. To solve an instance constraint we do something similar. To solve the instance constraint $a <_M b$ we have to wait until b is *fixed*.

Definition:

> A type is *fixed* if all the equality classes, containing a type variable from the type, are fixed. An equality class is *fixed* if none of its variables appears in an equality constraint or appears on the left side in an instance constraint. A type (or an equality class) that is fixed does not change during the rest of the computation.

As soon as b is fixed (in the constraint $a <_M b$) and there is no inconsistency in the state, we can instantiate this type. If there is an inconsistency in the state, it will eventually be resolved. Each polymorphic type variable is replaced by a new type variable for each instance. A type variable t is polymorphic if and only if:

- there are no constants in the equality class of t.
- none of the type variables in the equality class of t are in the monomorphic set of the instance constraint.

Polymorphic type variables in the same equality class are replaced by the same new type variable. After the polymorphic type variables on the right-hand side of the instance constraint are replaced, we change the constraint into an equality of two types.

Example

In this example we show how we simplify an instance constraint. Consider the following state:

State before simplification

Constraints	#0:	τ_1	$\equiv$	$\tau_6 \to \tau_7$
		τ_1	$<_{\{\tau_0\}}$	$(\tau_5 \to \tau_2) \to \tau_4 \to \tau_3$
Equality classes	$(\ \{\tau_0, \tau_5\}$	,	$\emptyset$	$)$
	$(\ \{\tau_1\}$	,	$\emptyset$	$)$
	$(\ \{\tau_2\}$	,	$\{\text{Int}\}$	$)$
	$(\ \{\tau_3, \tau_4\}$	,	$\emptyset$	$)$
	$(\ \{\tau_6, \tau_7\}$	,	$\emptyset$	$)$
Result type	$\tau_0 \to \tau_1$			

We can simplify the only instance constraint in this state because the type variables on the right-hand side (τ_2, τ_3, τ_4 and τ_5) don't occur in an equality constraint or in the left side of an instance constraint. We determine which type variables are polymorphic and which are not:

- τ_2 is in the same equality class as the type constant Int: it is not polymorphic.
- τ_5 is in the same equality class as τ_0. Because τ_0 is monomorphic, τ_5 is not a polymorphic type variable.
- τ_3 and τ_4 are polymorphic. Because they are in the same equality class, only one new type variable is introduced to replace τ_3 and τ_4.

We introduce the new type variable τ_8. The substitution $[\tau_3 := \tau_8, \tau_4 := \tau_8]$ replaces all the polymorphic type variables into new type variables. We apply the substitution to the right-hand side of the instance constraint and we introduce a new equality class for τ_8. The final result of the simplification of the instance constraint is the following state:

State after simplification

Constraints	#0:	τ_1	$\equiv$	$\tau_6 \rightarrow \tau_7$	
	#1:	τ_1	$\equiv$	$(\tau_5 \rightarrow \tau_2) \rightarrow \tau_8 \rightarrow \tau_8$	
Equality classes	(	$\{\tau_0, \tau_5\}$	,	$\emptyset$	)
	(	$\{\tau_1\}$	,	$\emptyset$	)
	(	$\{\tau_2\}$	,	{Int}	)
	(	$\{\tau_3, \tau_4\}$	,	$\emptyset$	)
	(	$\{\tau_6, \tau_7\}$	,	$\emptyset$	)
	(	$\{\tau_8\}$	,	$\emptyset$	)
Result type	$\tau_0 \rightarrow \tau_1$				

3.2 Decomposition

In the previous paragraph we discussed how a constraint can be simplified or solved. The only constraints left are those of the form (Variable $v \equiv$ Apply s t). The following table summarizes how we simplify or solve an equality constraints:

	Variable	Constant	Apply
Variable	variables in same class	variable and constant in same class	**decomposition**
Constant	$\times$	error if constants are different	error
Apply	$\times$	$\times$	split constraint into two new constraints

Evidently the constraint (Variable $v \equiv$ Apply s t) expresses that the type of variable v is an application of two types. In fact, all the type variables in the equality class containing v are an application of two types. Substituting a type variable into an application of two types is called decomposition. We can only decompose a variable if there is no constant in its equality class. Let $(\{v_1, v_2, \ldots, v_n\}, \emptyset)$ be the equality class of the variable v. First we construct the substitution $[v_1 := s_1\ t_1, v_2 := s_2\ t_2, \ldots, v_n := s_n\ t_n]$, where $\{s_1, s_2, \ldots, s_n\}$ and $\{t_1, t_2, \ldots, t_n\}$ are new type variables. Next we apply this substitution to the state. The equality class is replaced by two new equality classes: $(\{s_1, s_2, \ldots, s_n\}, \emptyset)$ and $(\{t_1, t_2, \ldots, t_n\}, \emptyset)$.

Example

Consider the following state:

State before decomposition

Constraints	#0:	τ_0	$\equiv$	$\tau_3\ \tau_4$
	#1:	τ_1	$\equiv$	List Int
Equality classes	(	$\{\tau_0, \tau_1, \tau_2\}$	,	$\emptyset$)
	(	$\{\tau_3\}$	,	$\emptyset$)
	(	$\{\tau_4\}$	,	{Int})
Result type	$\tau_2 \rightarrow \tau_4$			

Because of constraint #0, type variable τ_0 requires a decomposition. The other type variables in the equality class of τ_0 (τ_1 and τ_2) also require a decomposition. We introduce the type variables $\tau_5, \tau_6, \cdots, \tau_{10}$ to construct the substitution $[\tau_0 := \tau_5\ \tau_6, \tau_1 := \tau_7\ \tau_8, \tau_2 := \tau_9\ \tau_{10}]$. When we apply the substitution to the state (to the constraints and the result type), the type variables τ_0, τ_1 and τ_2 disappear from the state. Two new equality classes are constructed to replace the class of τ_0. The state after decomposition is:

State after decomposition

Constraints	#0:	$\tau_5\ \tau_6$	$\equiv$	$\tau_3\ \tau_4$
	#1:	$\tau_7\ \tau_8$	$\equiv$	List Int
Equality classes	(	$\{\tau_5, \tau_7, \tau_9\}$	,	$\emptyset$)
	(	$\{\tau_6, \tau_8, \tau_{10}\}$	,	$\emptyset$)
	(	$\{\tau_3\}$	,	$\emptyset$)
	(	$\{\tau_4\}$	,	{Int})
Result type	$(\tau_9\ \tau_{10}) \rightarrow \tau_4$			

3.3 Algorithm

In this section we provide an algorithm for solving a set of constraints. The order in which the constraints are solved should not influence the outcome of the algorithm. There are a lot of possible orders to solve the two sets. An important aspect of the algorithm is dealing with inconsistencies in the state. First we define when there is an inconsistency in the state:

Definition:

A state is inconsistent if and only if one of the following conditions is true:

- there is an equality class containing two different constants.
- there is a type variable v such that the equality class of v contains a constant and there is an equality constraint (Variable $v \equiv$ Apply s t), where s and t are two arbitrary types. Variable v is a constant and should be decomposed at the same time.

In section 5 we discuss how an inconsistency can be removed from a state. The following algorithm describes solving a set of constraints:

Algorithm

Step 1: Simplify constraints.

Step 2: If there is a variable avaiable for decomposition, we decompose this variable. After the decomposition we go back to step 1.

Step 3: If there is an inconsistency in the state, we remove constraints from the state until it is consistent. Afterwards we go back to step 1.

Step 4: The set of constraints is empty and the state is consistent. We present the result type and report the constructed error messages.

Remarks on the algorithm:

- In step 1 both equality constraints and instance constraints are simplified until there are no more constraints left to simplify. After step 1 the only equality constraints in the state are those of the form (Variable v $\equiv$ Apply s t). The order in which the constraints are simplified is not important.
- The result of the decomposition of a variable is a substitution. After this substitution the set of constraints is simplified again.
- We wait as long as possible to solve an inconsistency, because the longer we wait the more information is collected. When we solve the inconsistency

we need as much information as possible.

- Termination of this algorithm is guaranteed because we are always able to remove an inconsistency. At the end of the algorithm the set of constraints is empty and there are no inconsistencies in the state.

All type information is now stored in the equality classes. The last step is to extract information from the classes to present the type of the expression. We construct a substitution for each type variable. Each type variable in an equality class containing a constant, is replaced by this constant. For equality classes without a constant we choose one type variable in the class. All the variables in the class are replaced by this type variable.

Example

Consider the following state, where the set of constraint is empty.

Final state

Constraints	empty
Equality classes	($\{\tau_0, \tau_2\}$, $\emptyset$)
	($\{\tau_1, \tau_4, \tau_5\}$, $\emptyset$)
	($\{\tau_3\}$, {Int})
	($\{\tau_6\}$, {Bool})
Result type	$(\tau_0 \rightarrow \tau_1) \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau_4$

Before we report the result type we construct a substitution that corresponds with the equality classes. The following substitution is constructed:

$$[\tau_0 := \tau_0, \tau_2 := \tau_0, \tau_1 := \tau_1, \tau_4 := \tau_1, \tau_5 := \tau_1 \tau_3 := \text{Int}, \tau_6 := \text{Bool}]$$

When we apply this substitution to the result type, we get the type:

$$(\tau_0 \rightarrow \tau_1) \rightarrow \tau_0 \rightarrow \text{Int} \rightarrow \tau_1$$

To make the type more comprehensible, we map the type variables to the variables a, b, c, ... After this mapping the representation of a type is equal to the representation of a type in Hugs. The type that is presented is:

$$(a \rightarrow b) \rightarrow a \rightarrow \text{Int} \rightarrow b$$

3.4 Examples

In this section two examples are given to demonstrate how the type inference algorithm works. The first step is to collect the constraints for an expression. After this step we solve the constraints and finally we present the inferred type.

Example 1

The following expression is very straightforward to type:

```
\x -> case x of
        1 -> 'a';
        2 -> 'b';
        3 -> 'c';
```

To infer the type for this expression we construct the inference tree. The rules give us a set of constraints and a type.

$$\dfrac{\dfrac{\dfrac{}{\vdash x : \tau_0, [x \mapsto \tau_0]}\,[VAR] \\ \begin{array}{cc} \dfrac{}{\vdash 1 : Int, \emptyset}\,[LIT] & \dfrac{}{\vdash' a' : Char, \emptyset}\,[LIT] \\ \dfrac{}{\vdash 2 : Int, \emptyset}\,[LIT] & \dfrac{}{\vdash' b' : Char, \emptyset}\,[LIT] \\ \dfrac{}{\vdash 3 : Int, \emptyset}\,[LIT] & \dfrac{}{\vdash' c' : Char, \emptyset}\,[LIT] \end{array}}{\vdash case\ x\ of\{\ldots\} : \tau_2,\ [x \mapsto \tau_0]}\,[CASE]}{\vdash \lambda x \rightarrow case\ x\ of\{\ldots\} : \tau_3 \rightarrow \tau_2,\ \emptyset}\,[ABS]$$

The following constraints are collected:

$$\textbf{Constraints} \quad \begin{array}{llcl} \#0: & \tau_1 & \equiv & \tau_0 \\ \#1: & \tau_1 & \equiv & \text{Int} \\ \#2: & \tau_1 & \equiv & \text{Int} \\ \#3: & \tau_1 & \equiv & \text{Int} \\ \#4: & \tau_2 & \equiv & \text{Char} \\ \#5: & \tau_2 & \equiv & \text{Char} \\ \#6: & \tau_2 & \equiv & \text{Char} \\ \#7: & \tau_3 & \equiv & \tau_0 \end{array}$$

The first step in the algorithm is to simplify the constraints. Fortunately all the constraints can be simplified. We don't have to decompose a variable and there is no inconsistency to be solved. The computed equality classes are:

$$\textbf{Equality classes} \quad \begin{array}{lllll} (& \{\tau_0, \tau_1, \tau_3\} & , & \{\text{Int}\} &) \\ (& \{\tau_2\} & , & \{\text{Char}\} &) \end{array}$$

Finally we have to apply the substitution $[\tau_0 := Int, \tau_1 := Int, \tau_2 := Char, \tau_3 := Int]$ to the type $\tau_3 \rightarrow \tau_2$. The type $Int \rightarrow Char$ is indeed the type we expect for the expression.

Example 2

The next example is a little bit more complicated. In this example there are two instantiations of the polymorphic identity function. Because we use the same example as Damas and Milner use in [DM82], we can spot the differences between the two type inference algorithms.

```
let i = \x -> x ;
in i i
```

We use the typing rules to collect a set of constraints and to generate a type for this expression.

$$\dfrac{\dfrac{}{\vdash i : \tau_3,\ [i \mapsto \tau_3]}\,[VAR] \qquad \dfrac{}{\vdash i : \tau_4,\ [i \mapsto \tau_4]}\,[VAR]}{\vdash i\ i : \tau_5,\ [i \mapsto \tau_3, i \mapsto \tau_4]}\,[APP]$$

$$\dfrac{\dfrac{\dfrac{}{\vdash x : \tau_0,\ [x \mapsto \tau_0]}\,[VAR]}{\vdash \lambda x \rightarrow x : \tau_1 \rightarrow \tau_0, \emptyset}\,[ABS] \qquad \vdash i\ i : \ldots}{\vdash let\ i = \lambda x \rightarrow x;\ in\ i\ i : \tau_5, \emptyset}\,[LET]$$

Constraints	#0:	τ_1	$\equiv$	τ_0
	#1:	τ_2	$\equiv$	$\tau_1 \rightarrow \tau_0$
	#2:	τ_3	$\equiv$	$\tau_4 \rightarrow \tau_5$
		τ_3	$<_\emptyset$	τ_2
		τ_4	$<_\emptyset$	τ_2

As long as τ_2 occurs in an equality constraint we cannot simplify the two instance constraints. We start with simplifying constraint #0, which results in combining the equality classes of τ_0 and τ_1. After this simplification there are only constraints left that require the decomposition of a variable. In this example we assume that simplifying the constraint (Variable $v \equiv s \rightarrow t$) will produce the substitution $[v := v_1 \rightarrow v_2]$, where v_1 and v_2 are new type variables. Because the function arrow ($\rightarrow$) is not a primitive in our type language, the substitution actually is $[v := v_1\ v_2]$. After applying the substitution to the constraint we get $v_1\ v_2 \equiv (\rightarrow s)\ t$. This requires an extra decomposition for the variable v_1. We do something similar during the simplification step: $s_1 \rightarrow s_2 \equiv t_1 \rightarrow t_2$ is simplified into the equality constraints $s_1 \equiv t_1$ and $s_2 \equiv t_2$.

The decomposition for constraint #1 gives us the substitution $[\tau_2 := \tau_6 \rightarrow \tau_7]$. When the substitution is applied to the state we get:

State during computation

Constraints	#1:	$\tau_6 \rightarrow \tau_7$	$\equiv$	$\tau_1 \rightarrow \tau_0$
	#2:	τ_3	$\equiv$	$\tau_4 \rightarrow \tau_5$
		τ_3	$<_\emptyset$	$\tau_6 \rightarrow \tau_7$
		τ_4	$<_\emptyset$	$\tau_6 \rightarrow \tau_7$
Equality classes	($\{\tau_0, \tau_1\}$, $\emptyset$)			
	($\{\tau_3\}$, $\emptyset$)			
	($\{\tau_4\}$, $\emptyset$)			
	($\{\tau_5\}$, $\emptyset$)			
	($\{\tau_6\}$, $\emptyset$)			
	($\{\tau_7\}$, $\emptyset$)			
Result type	τ_5			

After the substitution the type variable τ_2 has completely disappeared from the state. We can translate constraint #1 into two new constraints: $\tau_6 \equiv \tau_1$ and $\tau_7 \equiv \tau_0$. The new constraints will result in an equality class containing τ_0, τ_1, τ_6 and τ_7. The current state is:

State during computation

Constraints	#2:	τ_3	$\equiv$	$\tau_4 \rightarrow \tau_5$
		τ_3	$<_\emptyset$	$\tau_6 \rightarrow \tau_7$
		τ_4	$<_\emptyset$	$\tau_6 \rightarrow \tau_7$
Equality classes	($\{\tau_0, \tau_1, \tau_6, \tau_7\}$, $\emptyset$)			
	($\{\tau_3\}$, $\emptyset$)			
	($\{\tau_4\}$, $\emptyset$)			
	($\{\tau_5\}$, $\emptyset$)			
Result type	τ_5			

Because the variables τ_6 and τ_7 don't appear in the set of constraint any longer, we can simplify the instance constraints into equality constraints. Because τ_6 and τ_7 are in the same class we have to make instances for the polymorphic type $\forall\alpha : (\alpha \rightarrow \alpha)$. We will introduce a fresh variable for the two instances (τ_8 and τ_9) to replace the α. After the instantiation the state is:

State during computation

Constraints	#2:	τ_3	$\equiv$	$\tau_4 \rightarrow \tau_5$
	#3:	τ_3	$\equiv$	$\tau_8 \rightarrow \tau_8$
	#4:	τ_4	$\equiv$	$\tau_9 \rightarrow \tau_9$
Equality classes	(	$\{\tau_0, \tau_1, \tau_6, \tau_7\}$	,	$\emptyset$)
	(	$\{\tau_3\}$	,	$\emptyset$)
	(	$\{\tau_4\}$	,	$\emptyset$)
	(	$\{\tau_5\}$	,	$\emptyset$)
	(	$\{\tau_8\}$	,	$\emptyset$)
	(	$\{\tau_9\}$	,	$\emptyset$)
Result type	τ_5			

The constraint #2 and #3 require the decomposition of τ_3. After the substitution $[\tau_3 := \tau_{10} \rightarrow \tau_{11}]$ and the simplification of #2 and #3 the type variables τ_4, τ_5, τ_8, τ_{10} and τ_{11} are in the same class. At this point the only constraint left is #4 and to solve this we have to decompose τ_4 into $\tau_{12} \rightarrow \tau_{13}$. All the variables in the equality class of τ_4 have to be decomposed too: $[\tau_4 := \tau_{12} \rightarrow \tau_{13}, \tau_5 := \tau_{14} \rightarrow \tau_{15}, \tau_8 := \tau_{16} \rightarrow \tau_{17}, \tau_{10} := \tau_{18} \rightarrow \tau_{19}, \tau_{11} := \tau_{20} \rightarrow \tau_{21}]$. The state after this substitution is:

Final state

Constraints	empty			
Equality classes	(	$\{\tau_0, \tau_1, \tau_6, \tau_7\}$	,	$\emptyset$)
	(	$\{\tau_9, \tau_{12}, \tau_{13} \ldots \tau_{21}\}$	,	$\emptyset$)
Result type	$\tau_{14} \rightarrow \tau_{15}$			

Because τ_{14} and τ_{15} are in the same class, the result is the type $a \rightarrow a$. This is indeed the type that we expect for this function.

Chapter 4

Correctness

In the previous chapters we presented an algorithm for inferring the type of an expression. The question is whether for each expression the result of the algorithm is equal to the result of other type inference algorithms. In this chapter we compare our inference rules with the inference rules given by Hindley-Milner and we show that the rules are equivalent. This implies that our rules are both sound and complete.

Before we can compare the typing rules from [DM82] and the typing rules from section 2.4, we have to restrict the expression language. The new expression language contains variables, applications, lambda expressions and let expressions with one declared variable. Because only one variable can be declared in a let expression, the typing rule for a let expression is simplified. In this proof we skip case expressions and literals. The two sets of rules are presented in table 1 and table 2.

This chapter is organised as follows. In section 4.1 we explain how we can solve a type under a set of constraints on types. In section 4.2 we discuss the differences between the assumption set in the Hindley-Milner type system and the assumption set in the constraint-based type system. In section 4.3 we proof that the rules of the two type systems are equivalent.

VAR:	$\mathcal{A}' \vdash_{\text{HM}} x : \sigma$	$(x \mapsto \sigma\ in\ \mathcal{A}')$
APP:	$\dfrac{\begin{array}{c}\mathcal{A}' \vdash_{\text{HM}} f : \tau_1 \to \tau_2 \\ \mathcal{A}' \vdash_{\text{HM}} e : \tau_1\end{array}}{\mathcal{A}' \vdash_{\text{HM}} (f\ e) : \tau_2}$	
ABS:	$\dfrac{\mathcal{A}' \backslash x \cup \{x \mapsto \tau_1\} \vdash_{\text{HM}} e : \tau_2}{\mathcal{A}' \vdash_{\text{HM}} (\lambda x \to e) : \tau_1 \to \tau_2}$	
LET:	$\dfrac{\begin{array}{c}\mathcal{A}' \vdash_{\text{HM}} e : \tau_1 \\ \mathcal{A}' \backslash x \cup \{x \mapsto \tau_1\} \vdash_{\text{HM}} b : \tau_2\end{array}}{\mathcal{A}' \vdash_{\text{HM}} (let\ x = e\ in\ b) : \tau_2}$	
INST:	$\dfrac{\mathcal{A}' \vdash_{\text{HM}} e : \sigma}{\mathcal{A}' \vdash_{\text{HM}} e : \sigma'}$	$(\sigma > \sigma')$
GEN:	$\dfrac{\mathcal{A}' \vdash_{\text{HM}} e : \sigma}{\mathcal{A}' \vdash_{\text{HM}} e : \forall \alpha \sigma}$	$(\alpha\ not\ free\ in\ \mathcal{A}')$

Table 1: Hindley-Milner type inference rules

VAR:	$\dfrac{\alpha\ is\ fresh}{\vdash x : \alpha,\ [x \mapsto \alpha]}$	
APP:	$\dfrac{\begin{array}{c}\alpha\ is\ fresh \\ \vdash f : \tau_f, \mathcal{A}_f \\ \vdash e : \tau_e, \mathcal{A}_e\end{array}}{\vdash (f\ e) : \alpha,\ \mathcal{A}_f \cup \mathcal{A}_e}$	$\tau_f \equiv \tau_e \to \alpha$
ABS:	$\dfrac{\begin{array}{c}\alpha\ is\ fresh \\ \vdash e : \tau_e,\ \mathcal{A}_e\end{array}}{\vdash (\backslash x \to e) : \alpha \to \tau_e,\ \mathcal{A}_e \backslash x}$	$\{\alpha \equiv \sigma \mid (x \mapsto \sigma) \in \mathcal{A}_e\}$
LET:	$\dfrac{\begin{array}{c}\vdash e : \tau_e,\ \mathcal{A}_e \\ \vdash b : \tau_b,\ \mathcal{A}_b\end{array}}{\begin{array}{c}\vdash (let\ x = e\ in\ b) : \tau_b, \\ \mathcal{A}_e \cup (\mathcal{A}_b \backslash x)\end{array}}$	$\{\sigma <_{\mathcal{A}_e} \tau_e \mid (x \mapsto \sigma) \in \mathcal{A}_b\}$

Table 2: Type inference rules with type constraints

4.1 Solve

In this section we define how we solve a set of constraints for a type. The function *Solve* is undefined for an inconsistent set of constraints. The type of the function is:

$$Solve :: (Constraints, Type) \rightarrow Type$$

Definition:

(1) $Solve\ (\emptyset, \tau) \quad = \quad \tau$

(2) $Solve\ (\{TypeVar\ a \equiv b\} \cup C, \tau) \quad = \quad Solve([a := b]C, [a := b]\tau)$

(3) $Solve\ (\{a\ b \equiv c\ d\} \cup C, \tau) \quad = \quad Solve(\{a \equiv c, b \equiv d\} \cup C, \tau)$

(4) $Solve\ (\{TypeCon\ a \equiv TypeCon\ a\} \cup C, \tau \quad = \quad Solve(C, \tau)$

(5) $Solve\ (\{a <_M b\} \cup C, \tau) \quad = \quad Solve(\{a \equiv [x_i := y_i]b\} \cup C, \tau)$
only if no type variables in b occur in an equality constraints in C or in the left-hand side of an instance constraint in C, where x_i are all the type variables in b but not in M and y_i are all new type variables.

Rule (1) defines the function *Solve* for an empty set of constraints. Because $(\equiv)$ is commutative and *Solve* is only defined for a consistent set of constraints, rules (2), (3) and (4) cover all consistent combinations in an equality constraints. Rule (5) translates an instance constraint into an equality constraint.

Theorem 1:

$$Solve(\{a \rightarrow b \equiv c \rightarrow d\} \cup C, \tau) = Solve(\{a \equiv c, b \equiv d\} \cup C, \tau)$$

Proof:

Use rules (3) and (4). $\diamond$

Theorem 2:

$$Solve(C, a \rightarrow b) = Solve(C, a) \rightarrow Solve(C, b)$$

Proof:

Solving a set of constraints is constructing and applying a substitution. For all substitutions S and types a and b: $S(a \rightarrow b) = S(a) \rightarrow S(b)$. Because we can distribute a substitution over $(\rightarrow)$, we can also distribute the function *Solve* over $(\rightarrow)$. $\diamond$

Theorem 3:

> *Solve* $(C \cup C', \tau) =$ *Solve* (C, τ) if there is no type variable in C' that is also in C or in τ

Proof:

> The constraints in C' construct substitutions about the type variables in C'. The application of this substitution does not influence C or τ. $\diamond$

Theorem 4:

> If *Solve* $(C, \tau_1) =_\alpha$ *Solve* (C, τ_2)
> then *Solve* $(\{\tau_1 \equiv \tau_2\} \cup C, \tau_3) =_\alpha$ *Solve* (C, τ_3)

Proof:

> The substitution S, that is constructed by the constraint set C, has the property $S(\tau_1) =_\alpha S(\tau_2)$. The substitution that is constructed for the equality constraint $\tau_1 \equiv \tau_2$ does not influence S and can be removed from the set of constraints. $\diamond$

4.2 Assumptions

Although we keep a set of assumptions in both sets of typing rules, we use the sets quite differently. In the Hindley-Milner rules a set of assumptions is *threaded* to subexpressions. A variable cannot occur more than once in the assumption set. On the contrary, the assumption set in the new rules is collected *bottom-up* and each occurrence of a free variable occurs exactly once in the assumption set. A variable can occur several times in an assumption set.

In this section we define the function $\triangle$, to give a relation between two assumption sets, with the type:

$$\triangle :: (\mathit{Assumptions}, \mathit{Assumptions}) \rightarrow \mathit{Constraints}$$

Definition:

$$(\tau \equiv \tau') \in \triangle(\mathcal{A}, \mathcal{A}') \qquad \Leftrightarrow \qquad (x \mapsto \tau) \in \mathcal{A} \wedge (x \mapsto \tau') \in \mathcal{A}'$$

We give some properties of the function $\triangle$. With the definition of $\triangle$ we can prove that the rules are correct.

$$
\begin{array}{lrcl}
(1) & \triangle(\mathcal{A}, \emptyset) & = & \emptyset \\
(2) & \triangle(\mathcal{A}, \mathcal{A}') & = & \triangle(\mathcal{A}', \mathcal{A}) \\
(3) & \triangle(\mathcal{A} \cup \{x \mapsto \tau\}, \mathcal{A}') & = & \triangle(\mathcal{A}, \mathcal{A}') \cup \{\tau \equiv \tau' \mid (x \mapsto \tau') \in \mathcal{A}'\} \\
(4) & \triangle(\mathcal{A}\backslash x, \mathcal{A}') & = & \triangle(\mathcal{A}\backslash x, \mathcal{A}'\backslash x) \\
(5) & \triangle(\mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{A}_3) & = & \triangle(\mathcal{A}_1, \mathcal{A}_3) \cup \triangle(\mathcal{A}_2, \mathcal{A}_3)
\end{array}
$$

The type language described in section 2.2 does not contain a $\forall$ quantifier. Because the type system of Hindley-Milner can contain a $\forall$ quantifier, we have to translate quantified types into types without a quantifier.

$$\mathit{Solve}\ (C \cup \{\forall x.\tau_1 \equiv \tau_2\}, \tau_3) = \mathit{Solve}\ (C \cup \{[x := y]\tau_1 \equiv \tau_2\}, \tau_3)$$
where y is a fresh type variable

4.3 Proof of correctness

In this section we show that the type systems presented in table 1 and table 2 are equivalent. We use the function *Solve* to solve the set of constraints that is collected. We use the function $\triangle$ to describe the relation of the assumption sets of the two type systems. The set of constraints that is constructed for an expression e is notated as C_e. The constraints for an expression are constructed *bottom-up*, just like the type and the assumption set for an expression. Because everything is constructed bottom-up, the sets of type variables in two sub expressions are always disjoint. This property is very important for the proof.

In the type system of Hindley-Milner there is a typing rule for generalising types and a rule for instantiating types. In the constraint-based type system the existence of polymorphism is handled completely different. An instance constraint combines both the generalisation and the specialisation of a type. The set of monomorphic variables, kept with an instance constraint, ensures that we do not generalise type variables assigned to variables bound outside the let expression.

Theorem 5 (Polymorphism):

In the Hindley-Milner type system, the polymorphic type of a let declaration is present in the set of assumptions in the body of a let expression. The definition of $\triangle$ justifies the equality constraint in the proof of theorem 6, replacing an instance constraint.

Proof:

In the Hindley-Milner type system the generalisation rule quantifies all the type variables that are not free in the assumption set. The application of this rule is only necessary for the generalisation of the type of the declaration. The result of instantiation of a polymorphic type is equal in the two type systems. $\diamond$

In theorem 6 we use the notation $\vdash e : \tau, \mathcal{A}, C$ to express that type τ is assigned to expression e and that assumption set $\mathcal{A}$ and constraint set C are collected. The set C is implicitly present in the typing rules in table 2.

Theorem 6 (Correctness):

$$\vdash e : \tau, \mathcal{A}, C \qquad \Leftrightarrow \qquad \mathcal{A}' \vdash_{\text{HM}} e : \tau'$$

such that $\mathit{Solve}\ (C \cup \triangle(\mathcal{A}, \mathcal{A}'), \tau) =_\alpha \tau'$

Proof:

To proof theorem 6, we use induction on the expression. We use the abbreviation i.h. for induction hypothesis.

variable

$$\begin{array}{ll}
\mathit{Solve}\ (\emptyset \cup \triangle([x \mapsto \alpha], \mathcal{A}'), \alpha) & (x \mapsto \sigma) \in \mathcal{A}' \\
\qquad = \triangle, \text{ rule (3)} & \\
\mathit{Solve}\ (\{\alpha \equiv \beta \mid (x \mapsto \beta) \in \mathcal{A}'\} \cup \triangle(\emptyset, \mathcal{A}'), \alpha) & (x \mapsto \sigma) \in \mathcal{A}' \\
\qquad = \triangle, \text{ rules (1) and (2)} & \\
\mathit{Solve}\ (\{\alpha \equiv \beta \mid (x \mapsto \beta) \in \mathcal{A}'\}, \alpha) & (x \mapsto \sigma) \in \mathcal{A}' \\
\qquad = & \\
\mathit{Solve}\ (\{\alpha \equiv \sigma\}, \alpha) & \\
\qquad = \text{Solve, rule (2)} & \\
\mathit{Solve}\ (\emptyset, \sigma) & \\
\qquad = \text{Solve, rule (1)} & \\
\sigma &
\end{array}$$

apply

i.h. : $\mathit{Solve}\ (C_f \cup \triangle(\mathcal{A}_f, \mathcal{A}'), \tau_f) =_\alpha \tau_1 \to \tau_2$
$\mathit{Solve}\ (C_e \cup \triangle(\mathcal{A}_e, \mathcal{A}'), \tau_e) =_\alpha \tau_1$

Assumption: τ_f is a function (when satisfying the constraints C_f and $\triangle(\mathcal{A}_f, \mathcal{A}')$), therefore $\tau_f = \tau_{f1} \to \tau_{f2}$. We strengthen the first induction hypothesis:

$$\begin{aligned}
&\mathit{Solve}\ (C_f \cup \triangle(\mathcal{A}_f, \mathcal{A}'), \tau_{f1} \to \tau_{f2}) =_\alpha \tau_1 \to \tau_2 \\
&\qquad = \text{theorem 2} \\
&\mathit{Solve}\ (\ldots, \tau_{f1}) \to \mathit{Solve}\ (\ldots, \tau_{f2}) =_\alpha \tau_1 \to \tau_2 \\
&\qquad =
\end{aligned}$$

i.h.* : $\begin{cases} \mathit{Solve}\ (C_f \cup \triangle(\mathcal{A}_f, \mathcal{A}'), \tau_{f1}) =_\alpha \tau_1 \\ \mathit{Solve}\ (C_f \cup \triangle(\mathcal{A}_f, \mathcal{A}'), \tau_{f2}) =_\alpha \tau_2 \end{cases}$

sub proof :
$$\begin{aligned}
&\mathit{Solve}\ (C_f \cup C_e \cup \triangle(\mathcal{A}_f \cup \mathcal{A}_e, \mathcal{A}'), \tau_{f1}) \\
&\qquad = \triangle, \text{ rule (5)} \\
&\mathit{Solve}\ (C_f \cup C_e \cup \triangle(\mathcal{A}_f, \mathcal{A}') \cup \triangle(\mathcal{A}_e, \mathcal{A}'), \tau_{f1}) \\
&\qquad = \text{theorem 3} \\
&\mathit{Solve}\ (C_f \cup \triangle(\mathcal{A}_f, \mathcal{A}'), \tau_{f1}) \\
&\qquad =_\alpha \text{i.h.*} \\
&\tau_1 \\
&\qquad =_\alpha \text{i.h.} \\
&\mathit{Solve}\ (C_e \cup \triangle(\mathcal{A}_e, \mathcal{A}'), \tau_e) \\
&\qquad = \text{theorem 3} \\
&\mathit{Solve}\ (C_f \cup C_e \cup \triangle(\mathcal{A}_f, \mathcal{A}') \cup \triangle(\mathcal{A}_e, \mathcal{A}'), \tau_e) \\
&\qquad = \triangle, \text{ rule (5)} \\
&\mathit{Solve}\ (C_f \cup C_e \cup \triangle(\mathcal{A}_f \cup \mathcal{A}_e, \mathcal{A}'), \tau_e)
\end{aligned}$$

$$\begin{aligned}
&\mathit{Solve}\ (C_f \cup C_e \cup \{\tau_f \equiv \tau_e \to \alpha\} \cup \triangle(\mathcal{A}_f \cup \mathcal{A}_e, \mathcal{A}'), \alpha) \\
&\qquad = \text{assumption} \\
&\mathit{Solve}\ (C_f \cup C_e \cup \{\tau_{f1} \to \tau_{f2} \equiv \tau_e \to \alpha\} \cup \triangle(\mathcal{A}_f \cup \mathcal{A}_e, \mathcal{A}'), \alpha) \\
&\qquad = \text{theorem 1} \\
&\mathit{Solve}\ (C_f \cup C_e \cup \{\tau_{f1} \equiv \tau_e, \tau_{f2} \equiv \alpha\} \cup \triangle(\mathcal{A}_f \cup \mathcal{A}_e, \mathcal{A}'), \alpha) \\
&\qquad = \text{Solve, rule (2)} \\
&\mathit{Solve}\ (C_f \cup C_e \cup \{\tau_{f1} \equiv \tau_e\} \cup \triangle(\mathcal{A}_f \cup \mathcal{A}_e, \mathcal{A}'), \tau_{f2}) \\
&\qquad = \text{theorem 4 and sub proof} \\
&\mathit{Solve}\ (C_f \cup C_e \cup \triangle(\mathcal{A}_f \cup \mathcal{A}_e, \mathcal{A}'), \tau_{f2}) \\
&\qquad = \triangle, \text{ rule (5)} \\
&\mathit{Solve}\ (C_f \cup C_e \cup \triangle(\mathcal{A}_f, \mathcal{A}') \cup \triangle(\mathcal{A}_e, \mathcal{A}'), \tau_{f2}) \\
&\qquad = \text{theorem 3} \\
&\mathit{Solve}\ (C_f \cup \triangle(\mathcal{A}_f, \mathcal{A}'), \tau_{f2}) \\
&\qquad = \text{i.h.*} \\
&\tau_2
\end{aligned}$$

lambda

i.h. : $\quad \mathit{Solve}\ (C_e \cup \triangle(\mathcal{A}_e, \mathcal{A}'\backslash x \cup \{x \mapsto \tau_1\}), \tau_e) =_\alpha \tau_2$

$$\begin{aligned}
&\mathit{Solve}\ (C_e \cup \{\alpha \equiv \sigma \mid (x \mapsto \sigma) \in \mathcal{A}_e\} \cup \triangle(\mathcal{A}_e\backslash x, \mathcal{A}'), \alpha \to \tau_e) \\
&\qquad = \triangle, \text{ rule (4)} \\
&\mathit{Solve}\ (C_e \cup \{\alpha \equiv \sigma \mid (x \mapsto \sigma) \in \mathcal{A}_e\} \cup \triangle(\mathcal{A}_e\backslash x, \mathcal{A}'\backslash x), \alpha \to \tau_e) \\
&\qquad = \triangle, \text{ rule (3)} \\
&\mathit{Solve}\ (C_e \cup \triangle(\mathcal{A}_e, \mathcal{A}'\backslash x \cup \{x \mapsto \alpha\}), \alpha \to \tau_e) \\
&\qquad =_\alpha \\
&\mathit{Solve}\ (C_e \cup \triangle(\mathcal{A}_e, \mathcal{A}'\backslash x \cup \{x \mapsto \tau_1\}), \tau_1 \to \tau_e) \\
&\qquad = \text{theorem 2} \\
&\mathit{Solve}\ (C_e \cup \triangle(\mathcal{A}_e, \mathcal{A}'\backslash x \cup \{x \mapsto \tau_1\}), \tau_1) \to \mathit{Solve}\ (\ldots, \tau_e) \\
&\qquad = \\
&\tau_1 \to \mathit{Solve}\ (C_e \cup \triangle(\mathcal{A}_e, \mathcal{A}'\backslash x \cup \{x \mapsto \tau_1\}), \tau_e) \\
&\qquad =_\alpha \text{ i.h.} \\
&\tau_1 \to \tau_2
\end{aligned}$$

let

i.h. : $\quad \mathit{Solve}\ (C_e \cup \triangle(\mathcal{A}_e, \mathcal{A}'), \tau_e) =_\alpha \tau_1$
$\quad \mathit{Solve}\ (C_b \cup \triangle(\mathcal{A}_b, \mathcal{A}'\backslash x \cup \{x \mapsto \tau_1\}), \tau_b) =_\alpha \tau_2$

$$\begin{aligned}
&\mathit{Solve}\ (C_e \cup C_b \cup \{\sigma <_{\mathcal{A}_e} \tau_e \mid (x \mapsto \sigma) \in \mathcal{A}_b\} \cup \\
&\qquad\qquad\qquad \triangle(\mathcal{A}_e \cup (\mathcal{A}_b\backslash x), \mathcal{A}'), \tau_b) \\
&\qquad = \text{theorem 5} \\
&\mathit{Solve}\ (C_e \cup C_b \cup \{\sigma \equiv \tau_e \mid (x \mapsto \sigma) \in \mathcal{A}_b\} \cup \\
&\qquad\qquad\qquad \triangle(\mathcal{A}_e \cup (\mathcal{A}_b\backslash x), \mathcal{A}'), \tau_b) \\
&\qquad = \triangle, \text{ rule (5)} \\
&\mathit{Solve}\ (C_e \cup C_b \cup \{\sigma \equiv \tau_e \mid (x \mapsto \sigma) \in \mathcal{A}_b\} \cup \\
&\qquad\qquad\qquad \triangle(\mathcal{A}_e, \mathcal{A}') \cup \triangle(\mathcal{A}_b\backslash x, \mathcal{A}'), \tau_b) \\
&\qquad = \text{i.h.} \\
&\mathit{Solve}\ (C_e \cup C_b \cup \{\sigma \equiv \tau_1 \mid (x \mapsto \sigma) \in \mathcal{A}_b\} \cup \\
&\qquad\qquad\qquad \triangle(\mathcal{A}_e, \mathcal{A}') \cup \triangle(\mathcal{A}_b\backslash x, \mathcal{A}'), \tau_b) \\
&\qquad = \text{theorem 3} \\
&\mathit{Solve}\ (C_b \cup \{\sigma \equiv \tau_1 \mid (x \mapsto \sigma) \in \mathcal{A}_b\} \cup \triangle(\mathcal{A}_b\backslash x, \mathcal{A}'), \tau_b) \\
&\qquad = \triangle, \text{ rule (4)} \\
&\mathit{Solve}\ (C_b \cup \{\sigma \equiv \tau_1 \mid (x \mapsto \sigma) \in \mathcal{A}_b\} \cup \triangle(\mathcal{A}_b\backslash x, \mathcal{A}'\backslash x), \tau_b) \\
&\qquad = \triangle, \text{ rule (3)} \\
&\mathit{Solve}\ (C_b \cup \triangle(\mathcal{A}_b, \mathcal{A}'\backslash x \cup \{x \mapsto \tau_1\}), \tau_b) \\
&\qquad =_\alpha \text{ i.h.} \\
&\tau_2
\end{aligned}$$

$\diamond$

Chapter 5

Solving inconsistencies

In chapter 3 an algorithm for computing the type of an expression from a set of constraints is given. In this chapter we discuss dealing with inconsistencies. In order to decide which constraint is most likely to contain an error, we have to keep some extra information. First we introduce a new data structure to store additional information in an equality class. In section 5.2 we define (again) what we mean with an inconsistency. In sections 5.3 and 5.4 two different strategies are explained to remove an inconsistency in a state. In both cases an example is given to clarify the definitions and the strategies. In section 5.5 we compare the two approaches and discuss the advantages and disadvantages of the two methods. Finally, in section 5.6 we explain how we generate an error message.

5.1 Equality graphs

Equality classes are continuously modified while we are solving the equality constraints in the state. Instead of only storing which variables and constants form an equality class, we also store why this is the case. From the algorithm we see that there are only two possible operations on an equality class: a constant is put in the same class as a variable or the equality classes of two variables are combined into one new equality class. We use an undirected graph to represent the equality classes. The graph has the following properties:

- a vertex corresponds to a type variable or a type constant in the state.
- an edge corresponds to the constraint from the initial state that is responsible for the two types in the vertices being equal.
- a vertex containing a type constant has exactly one edge to a vertex containing a type variable.
- each type variable in the state occurs exactly once in a vertex. A type constant can occur in many vertices.

- an equality class is represented by a *connected component* in the graph.

Consider the first example in 3.4 again and compare the equality class and the graph in figure 5.1. As you can see the graph consists of two connected components, each representing an equality class.

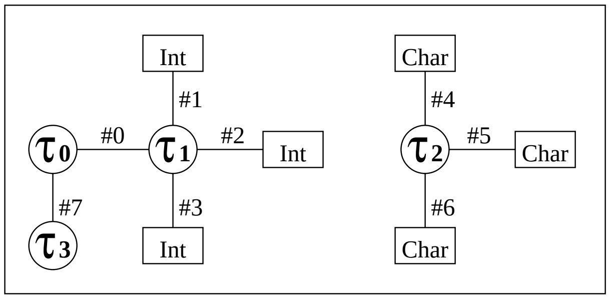

Figure 5.1: A graph containing two equivalence classes

Two types are in the same class if there is a path between the vertices containing those types. In [CLR90] a definition of a path is given:

Definition:

A *path* of length k from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $< v_0, v_1, v_2, \cdots, v_k >$ of vertices such that $u = v_0$, $u' = v_k$ and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \cdots, k$. [...] The path contains the vertices $v_0, v_1, \cdots, v_k$ and the edges $(v_0, v_1), (v_1, v_2), \cdots, (v_{k-1}, v_k)$.

Because cycles can occur in the graph it is possible that there is an infinite number of paths between two vertices. To avoid this we strengthen the definition with the condition that a path can contain each vertex only once.

5.2 Consistency

In section 3.3 we defined when there is an inconsistency in a state. Now that we are using graphs as a representation for an equality class we use the same ideas. A state is inconsistent if there is an inconsistent equality graph. There are two possibilities for a graph to be inconsistent:

- two different type constants are in the same connected component. In other words: there is a path between two vertices containing different type constants. This path is called an *error* path.
- a variable has to be decomposed, but it is also part of a connected component containing a type constant. A variable v has to be decomposed if and only if there is a constraint (Variable $v \equiv$ Apply s t) in the state,

where both s and t are arbitrary types. In other words: there is a path between such a vertex containing type variable v and a vertex containing a type constant. This path is called a *decompose* path.

To remove the inconsistency we should remove (at least) one edge for each *error* and *decompose* path.

5.3 Approach 1: lowest total cost

We will throw away a set of edges from a graph to make it consistent again. There is always such a set available: if we remove all edges the graph is certainly consistent. The problem is to throw away those edges for which the corresponding error message makes sense to the programmer.

Definition of the cost function

We construct a cost function which returns for each edge the removal cost.

$$cost :: edge \rightarrow \mathbf{N}$$

We choose this cost function such that the higher the cost the more likely it is that the equality, represented by the edge in the graph, is correct. On the other hand, the lower the value the more evidence there is that this edge (or this constraint) is the source of an inconsistency. Each constraint receives a trust value when it is constructed. This trust value represents the trust we have in a constraint *before* we start solving the constraints. For instance, a constraint that is created because we use a prelude function receives a high trust value. The cost function should ensure that:

- we destroy as few as possible *good* paths. A good path is a path between two vertices containing the same type constant.
- it is expensive to remove an edge that is associated with a constraint that has a high trust factor compared to the removal cost of an edge associated with a constraint with a low trust factor.

The cost associated with the deletion of edge e, constructed by constraint c, is defined as:

$$(1 + \text{occurrences of } e \text{ in a } \textit{good} \text{ path}) * \textit{trust}(c)$$

Definition:

A is a *minimal set* in graph $G = (V, E)$ if and only if:

- $A \subseteq E$
- graph $(V, E - A)$ is consistent
- $\forall_{a \in A}$: $(A \backslash a)$ is not a minimal set in G

The set A, containing the edges that we remove from the graph, must satisfy the following conditions:

- A is a minimal set
- for each minimal set B: $\sum\limits_{a \in A} cost(a) \leq \sum\limits_{b \in B} cost(b)$

Example

We want to solve the inconsistency in the following graph:

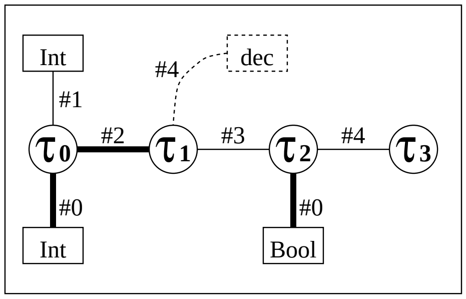

Figure 5.2: A graph with an inconsistency

Say that the trust values for the constraints are in the range from 1 to 5. The edges labelled with #0 and #2 are drawn thicker to indicate that these constraints have a high trust factor compared to the others. In the graph there are two edges labelled with #0 and two edges labelled with #4. It is possible that there are several edges labelled with the same constraint number; while we are solving a constraint it can be split into two new constraints. From τ_1 there is an edge to a vertex with *dec*. This is to suggest that equality constraint #4 requires the decomposition of τ_1. The line is dashed because the edge and the constant *dec* are not in the actual graph, although that would have been convenient for calculating the cost function. The motivation for the absence of the *decompose* constant in the graph is that we want to separate the set of constraints and the equality classes in the state. The constant *dec* is only necessary for the calculation of the cost function. First we have to create a cost function for this graph. There is only one *good* path present, that connects the two type constants *Int*. The edges for this path are labelled with #0 and #1. This results in the following cost function:

	good	trust	cost
#0	1	5	10
#1	1	1	2
#2		5	5
#3		1	1
#4		1	1

Before we calculate the minimal sets we need to find all the inconsistencies:

- there are two *wrong* paths: {#1, #2, #3, #0} and {#0, #2, #3, #0}.
- there are three *decompose* paths: {#4, #2, #1}, {#4, #2, #0} and {#4, #3, #0}.

We can summarise these inconsistencies in a matrix: each row corresponds with a *wrong* path or a *decompose* path, each row corresponds with a constraint. The number of crosses in a column indicate the number of occurrences of the constraint in paths that cause an inconsistency.

#0	#1	#2	#3	#4
×	×	×	×	
×		×	×	
	×	×		×
×		×		×
×			×	×

A minimal set contains at least one constraint from each erroneous path. For each set the total cost value is calculated:

minimal set	total cost
{#0, #1}	12
{#0, #2}	15
{#0, #4}	11
{#2, #3}	6
{#2, #4}	6
{#3, #4}	2

The edges in the minimal set with the lowest total cost value, {#3, #4}, are removed from the graph to solve the inconsistency. For each constraint that is removed we produce an error message. In 5.6 we discuss constructing an error message for an equality constraint that is removed from the set of constraints.

5.4 Approach 2: greedy-choice

A different approach is to consider the *profit* of removing an edge from a graph instead of its *cost*. From this point of view it is reasonable to remove the one edge with the highest profit. The profit function we define in this section is not the negation of the cost function because it also takes erroneous paths into account, and not just the number of occurrences in *good* paths.

The approach that we discuss in this section is an example of a *greedy algorithm*. In a greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblems arising after the choice is made. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems ([CLR90]). Unfortunately this problem doesn't meet the *greedy-choice property*: the global optimal solution cannot be found by constructing locally optimal solutions. Still we can make some interesting observations when we use this technique.

Definition of the profit function

First we define the profit function. All the edges in an *error* path or a *decompose* path should be penalised: we increase the profit function for these edges. The longer an *error* path is the more edges we can remove to get rid of the *error* path.

Definition:

Let P be the set of error paths in the equality graph, then P_e contains all the paths in P that contain edge e. For some heuristic value $\mathcal{H}_{error} > 0$ we define:

$$error(e) = \sum_{p \in Pe} \frac{\mathcal{H}_{error}}{\text{number of different edges in p}}$$

We introduce the heuristic value $\mathcal{H}_{error}$ to represent the total profit for removing one error path. Similar to the definition of $error(e)$ we can introduce a heuristic value $\mathcal{H}_{decompose} > 0$ for defining a function $decompose(e)$ to penalise the edges in a decompose path. To prevent that a good path is removed, the profit value for an edge is decreased if the edge occurs in a good path. The definition for the function $good(e)$ is similar to $error(e)$ and $decompose(e)$, with the only difference that $\mathcal{H}_{good} < 0$. Finally we use the trust value of the constraint to calculate the profit value. The profit for removal of an edge e, generated by constraint c, is:

$$\frac{error(e) + good(e) + decompose(e)}{trust(c)}$$

With this profit function we can determine which edge is removed:

Definition:

We remove edge $a \in E$ from graph $G = (V, E)$ if and only if:

- $\forall_{e \in E} : profit(a) \geq profit(e)$

Example

We will use the same example as used in 5.3. Take a look again at the graph in figure 5.2. First we need a set of heuristic values, let us assume that:

$$\begin{array}{lcr} \mathcal{H}_{error} & = & 24 \\ \mathcal{H}_{good} & = & -12 \\ \mathcal{H}_{decompose} & = & 24 \end{array}$$

Now we can calculate the profit value for each constraint. There are two error paths, one good path and three decompose paths. Again we assume that the constraints #0 and #2 have a high trust value and the other constraints have a normal trust value.

	error	good	decompose	sub total	trust	profit
#0	14	-6	16	24	5	4.8
#1	6	-6	8	8	1	8.0
#2	14		16	30	5	6.0
#3	14		8	22	1	22.0
#4			24	24	1	24.0

Using the greedy method we decide that constraint #4 is erroneous. We remove all the edges referring to this constraint in the graph. It is possible that an equivalence class is split into two new classes because of the removal of an edge. In this example variable τ_3 is separated from the variables τ_0, τ_1 and τ_2. Figure 5.3 shows the situation after constraint #4 has been removed.

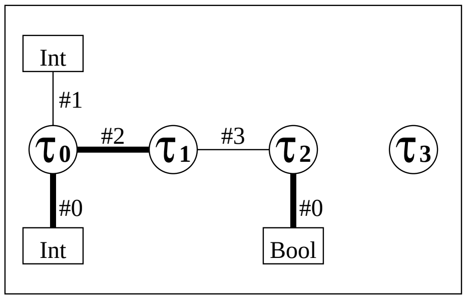

Figure 5.3: The graph after #4 has been removed

Unfortunately the inconsistency has not been removed by throwing away one constraint. In order to make another *greedy* step we have to recalculate the profit table

	error	good	decompose	sub total	trust	profit
#0	14	-6		8	5	1.6
#1	6	-6		0	1	0.0
#2	14			14	5	2.8
#3	14			14	1	14.0

The graph is consistent again after the removal of constraint #3. Figure 5.4 shows the final graph.

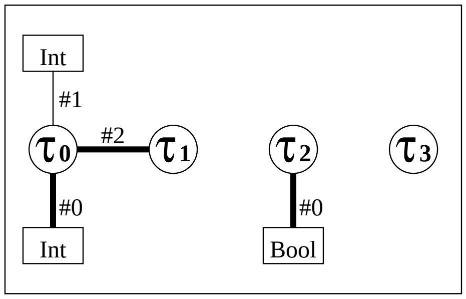

Figure 5.4: The graph after #3 has been removed

5.5 Comparison of the approaches

We presented two approaches to remove an inconsistency. The first approach uses a *cost* function and removes the minimal set with the lowest total cost from the state. The second approach uses a *profit* function and removes only the constraint with the highest profit. In this section we compare the two approaches.

The inconsistency in figure 5.2 was used as an example to illustrate both methods. The final result is in both cases the same: the constraints #3 and #4 are removed. This is not a coincidence because both approaches were designed to remove erroneous paths in the graph and to save (as many as possible) correct paths. Although the results are equal for this example, the two methods do not remove the same set of constraints for every inconsistency.

The advantage of searching for a minimal set of constraints is that the state is consistent if the constraints in this set are removed. A disadvantage is that it can be expensive to compute every minimal set. In the worst case, the total number of minimal sets for n constraints is exponential. Consider the graph with $n/2$ erroneous paths: $\{\#1, \#2\}$, $\{\#3, \#4\}$, ..., $\{\#(n-1), \#n\}$. This graph has $2^{n/2}$ different minimal sets. Fortunately, because the set of constraints is generated from an expression, it is not reasonable to assume that we encounter inconsistent graphs with such an extreme high number of minimal sets.

The disadvantage of the greedy approach is that after the constraint with the highest profit is removed, the state does not have to be consistent. Worst case, we have to make several greedy choices before we have a consistent state. For each greedy step we have to calculate a new profit function. Again it is not reasonable to assume that we have to make several greedy steps before we have a consistent state. We only expect a limited number of type errors in an expression. This approach performs best if only a small number of constraints is to be removed.

5.6 Error messages

This chapter explains how we remove constraints to restore consistency in a state. The price we have to pay for the removal of a constraint is the construction of an error message. For each constraint that is removed we construct exactly one error message. It is important to determine which typing rule generated the erroneous constraint. The information in the error message depends on which typing rule generated the constraint. For instance, if an erroneous equality constraint is constructed in the typing rule for an application, we indicate that this application is the source of a type error. If the function `plus` has type `Int` $\rightarrow$ `Int` $\rightarrow$ `Int`, the Hugs interpreter produces the following error message for the function `plus 3 True`:

```
ERROR "example.hs" (line 4): Type error in application
*** Expression     : plus 3 True
*** Term           : True
*** Type           : Bool
*** Does not match : Int
```

In this section we discuss how we can produce the same error message with the constraint-based type inferencer. The initial state that is constructed for the expression `plus 3 True` is:

Initial state

Constraints	#0:	τ_0	$\equiv$	Int $\rightarrow \tau_1$
	#1:	τ_1	$\equiv$	Bool $\rightarrow \tau_2$
	#2:	τ_0	$\equiv$	Int $\rightarrow$ Int $\rightarrow$ Int
Result type	τ_2			

Constraint #0 and #1 are constructed in the rule for applications. Constraint #2 is constructed for the variable `plus` that has type `Int` $\rightarrow$ `Int` $\rightarrow$ `Int`. We assume that #2 has a high trust value. When we solve the set of constraints we apply the substitution $[\tau_0 := \tau_3 \rightarrow \tau_4, \tau_4 := \tau_5 \rightarrow \tau_6, \tau_1 := \tau_7 \rightarrow \tau_8]$ to decompose variables, until we discover an inconsistency in the equality graph. The following figure shows the inconsistent graph at this point:

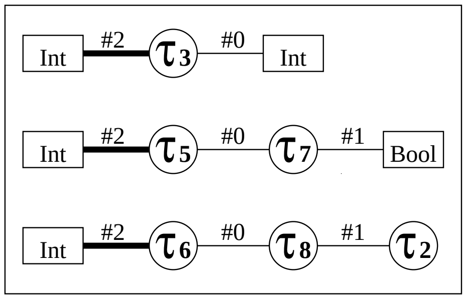

Figure 5.5: Equality graph for `plus 3 True`

After an analysis of the graph, constraint #1 is nominated to be removed: the graph is consistent again after the removal of constraint #1. In the initial state constraint #1 is $\tau_1 \equiv \text{Bool} \rightarrow \tau_2$. Before we can construct a nice error message we have to take a look at the part of the inference tree that constructed this constraint:

$$\dfrac{\dfrac{\cdots}{\vdash \mathit{plus}\ 3 : \tau_1, [\mathit{plus} \mapsto \tau_0]}\,[\mathit{APP}] \qquad \dfrac{}{\vdash \mathit{True} : \mathit{Bool}, \emptyset}\,[\mathit{LIT}]}{\vdash (\mathit{plus}\ 3)\ \mathit{True} : \tau_2, [\mathit{plus} \mapsto \tau_0]}\,[\mathit{APP}]$$

If we apply the substitution in the final state to the types in constraint #1, the result is the constraint (Int $\rightarrow$ Int $\equiv$ Bool $\rightarrow \tau_2$). This equality constraint explains the inconsistency. If we apply all the substitutions to the assigned type variables in the typing rules, we can construct a nice error message. For instance, the type variable assigned to the expression `plus 3`, type variable τ_1, is substituted into `Int` $\rightarrow$ `Int`.

```
ERROR: Type error in application
*** plus 3      :: Int -> Int
*** True        :: Bool
*** plus 3 True :: a
```

We can give a more specific error message because we see that the type of the argument (`Bool`) is different from the type the function expects (`Int`). A better error message for this example is:

```
    ERROR: Type error in argument of application
    *** Expression     : plus 3 True
    *** Term           : True
    *** Type           : Bool
    *** Does not match : Int
```

It is also possible that the type of the function turns out to have a constant type. If we try to assign a type to the expression `3 True` we want a message that is similar to the following error message:

```
    ERROR: Type error in function of application
    *** Expression     : 3 True
    *** Term           : 3
    *** Type           : Int
    *** Explanation    : a function type was expected
```

For every constraint that is removed from the constraint set we create an appropriate error message.

Chapter 6

Future work

The motivation for this article is the obscurity of some of the error messages generated by the Haskell interpreter. In this article we present a new approach to generating error messages for ill-typed expressions that are easier to understand for an inexperienced programmer. The trade off is that type inferencing takes more time, also for well typed expressions. There are still a lot of things to do before error messages are much easier to understand for an inexperienced programmer. Research in improving error messages at the University of Utrecht will continue. In this chapter some possibilities for future research are given.

Extending the expression language

The expression language that is used as input for the type inferencer is only a small subset of the expressions accepted by Haskell. Only if we can type check all the expressions that are accepted by Haskell, it can be useful to replace the current type checker by the type checker described in this paper. Besides the absence of syntactic sugar in our language, there are some features missing in the language that require an implementation.

It is interesting to assign a type to an expression that is *explicitly typed*, especially if the declared type does not match the type that was assigned to the expression by the type checker. The declared type is the type the programmer expects the expression to have. An explicit type gives us more type information and makes it easier to correctly point out where the type error occurs. Explicit typing is not only a tool to document a program, it also extends the expression language. Consider the next Haskell function:

```
f :: (forall a . (a -> a)) -> (Char,Bool)
f i = (i 'a',i True)
```

With a *forall* quantifier in an explicit type, lambda bound variables can have a polymorphic type. This function is well typed only if it is explicitly typed. We expect that explicit typing can increase the exactness of an error message considerably. Besides explicit typing there are some other features in Haskell that are interesting to investigate:

- Type classes in Haskell add a lot of convenience for the programmer. One of the benefits of type classes is *overloading*. We have not looked at extending the algorithm such that we can use type classes in our expression language.
- [GJ96] discusses a polymorphic type system for *Extensible Records*.

Soft typing

The presented constraint-based type system is static because it detects an ill-typed expression at compile-time. There are also type systems that detect ill-typed programs at run-time, they are called *soft* type systems ([CF91], [AWL94]). If it is possible to give a constraint-based type checker that detects errors dynamically, is still a question.

Type synonyms

Type synonyms introduce new names for existing types. They allow us to assign more comprehensive types to expressions. Consider the next part of a program:

```
type Counter = Int

zeroCounter :: Counter
zeroCounter = 0
```

The type of the function `zeroCounter` is very intuitive. Unfortunately, the Haskell type checker removes synonyms. For instance, the inferred type of the expression `(id zeroCounter)` is `Int`. For this example the synonym `Counter` is more appropriate. If we use type synonyms in the generated error messages, we produce error messages that are easier to understand.

Type tracing

Often it is not clear why a certain type is assigned to an expression. The possibility to ask for an explanation of the assigned type could be very helpful. In [BS93] an implementation for this facility is presented. Because we construct several equality classes, we can generate a similar explanation fairly easily: all information that is needed is available in the classes.

Advanced output

The presentation of error messages in current Haskell interpreters is restricted to *ASCII* output. A possible solution to avoid this restriction is to export the error message to an output file. For instance, the error message is translated into a XML document. When we present an document in XML we have the disposal of features (such as colour and style) that improve the presentation of an error. *Interaction* between the programmer and the error message is another improvement. The possibility to ask for the type of a variable in an expression, or to ask why a certain type is assigned to an expression, can increase the understanding of an error message.

Optimising data structures

Although speed is not a primary issue, it is important not to neglect efficiency issues. Our current implementation is very time consuming, especially the decomposition of variables costs a lot of time. For the definition of the function `foldr` there are 28 type variables in the initial state. At the end there are 174 different type variables: 144 variables are introduced because of decomposition. If we use a different data structure to represent equality classes (a structure that doesn't require decomposition of a variable), we get a more efficient implementation.

BANE

BANE is a toolkit to construct constraint-based program analyses. An example of a simple type inferencer for a lambda calculus, expressed in *BANE*, is presented in [AFFS98]. It is interesting to see if we can use the toolkit to replace the part in the implementation of our algorithm that is responsible for solving the set of constraints.

Other heuristics

If we investigate why certain type errors occur frequently, we can add new heuristics to our type system that construct clear error messages for common mistakes. For instance, it happens often that two arguments are passed in the wrong order to a function. It is easy to check if changing the order of arguments resolves the inconsistency in the set of constraints.

Chapter 7

Conclusion

The goal of this research is to improve the quality of error messages for ill-typed expressions. The approach presented in this paper combines ideas from several articles. Instead of unification on types a set of constraints is constructed, as suggested in [AFFS98]. Six syntax-directed typing rules are used for the construction of this set. The *left-to-right bias*, described in [McA98], is completely removed because the order in which the constraints are solved is irrelevant. The set of constraints generates an equality graph, containing each assigned type variable. The *decomposition* of type variables, while solving the constraints, explains the differences with graphs presented in [McA99] and [GVS96]. In case of an ill-typed expression, the equality graphs gives us the possibility to apply several heuristics to determine the most likely source of the type conflict. The heuristics described in this paper increase the exactness of the error message considerably. With the information available in the equality graph we can give an explanation of the type conflict that points out the conflicting sites. A constraint-based approach to inferring the type of an expression is implemented in Haskell.

Acknowledgements

Part of the research for this paper has been done while I visited the *Universidad Mayor de San Simón* in Cochabamba, Bolivia. I thank Pablo Azero Alcocer for his contribution to the typing rules and the type assignment algorithm. I also thank Johan Jeuring and Doaitse Swierstra for their comments and suggestions.

Appendix A

Collection of constraints

We use an attribute grammar to collect the constraints from an expression in the implementation of the type inferencer. The grammar is a translation of the type inference rules that were discussed in section 2.4.

The *UU_AG* system, developed by Doaitse Swierstra, compiles this file into a correct Haskell program. More information about the *UU_AG* system can be found at:

`http://www.cs.uu.nl/groups/ST/Software/UU_AG/index.html`

```
import Literal ( Literal, typeLiteral                     )
import Type    ( Type, TypeVariable, TypeConstant, (.->.) )
import List    ( partition                                )

\BT
data Constraint  = Equality Type Type
                 | Instance Monos Type Type deriving Show

type Assumption  = (Identifier,Type)
type Assumptions = [Assumption]
type Constraints = [Constraint]
type Identifier  = String
type Mono        = Type
type Monos       = [Mono]
```

```
bind :: (Type -> Type -> Constraint) -> Assumptions
        -> Assumptions -> (Constraints,Assumptions)
bind combine table = foldr op ([],[])
  where op a@(v,t) (cs,as)
          = let x = filter ((v==).fst) table
            in if null x then (cs,a:as)
                         else (map (combine t.snd) x++cs,as)

isConstructor :: String -> Bool
isConstructor []    = False
isConstructor (x:_) = isUpper x

\ET

\BC

DATA Expr
   | Variable  variable : Identifier
   | Literal   literal  : Literal
   | Apply     fun      : Expr        arg  : Expr
   | Lambda    variable : Identifier  expr : Expr
   | Case      expr     : Expr        alts : Alternatives
   | Let       decls    : Decls       expr : Expr

DATA Alternatives
   | Empty
   | Alternative  patt:Expr expr:Expr alts:Alternatives

DATA Decls
   | Empty
   | Decl  variable:Identifier expr:Expr decls:Decls

SEM Expr [
         | unique : Int            -- type variable counter
         | type   : Type           -- type
           ass    : Assumptions    -- assumptions
           cons   : Constraints    -- collected constraints
         ]

  | Variable LHS.type    = "alpha"
               .ass     = "[(variable,alpha)]"
               .cons    = "[]"
               .unique  = "lhs_unique+1"
            LOC.alpha   = "TypeVariable lhs_unique"

  | Literal  LHS.type    = "TypeConstant (typeLiteral literal)"
               .ass     = "[]"
               .cons    = "[]"
  | Apply    LHS.type    = "alpha"
               .ass     = "fun_ass++arg_ass"
               .cons    = "new:fun_cons++arg_cons"
               .unique  = "arg_unique+1"
            LOC.alpha   = "TypeVariable arg_unique"
               .new     = "Equality fun_type (arg_type .->. alpha)"

  | Lambda   LHS.type    = "alpha .->. expr_type"
               .ass     = "bs"
               .cons    = "new++expr_cons"
               .unique  = "expr_unique+1"
            LOC.alpha   = "TypeVariable expr_unique"
               .new     = "map (Equality alpha.snd) as"
               .(as,bs) = "partition ((==variable).fst) expr_ass"

  | Case     LHS.type    = "beta"
               .ass     = "expr_ass++alts_ass"
               .cons    = "new:expr_cons++alts_cons"
               .unique  = "alts_unique+2"
            LOC.alpha   = "TypeVariable alts_unique"
               .beta    = "TypeVariable (alts_unique+1)"
               .new     = "Equality alpha expr_type"

  | Let      LHS.type    = "expr_type"
               .ass     = "bs++ds"
               .cons    = "as++cs++decls_cons++expr_cons"
            LOC.monos   = "map snd ds"
               .(as,bs) = "bind (Instance monos) decls_env expr_ass"
               .(cs,ds) = "bind (Equality      ) decls_env decls_ass"


SEM Alternatives [ alpha  : Type           -- type of lhs
                   beta   : Type           -- type of rhs
                 | unique : Int            -- type variable counter
                 | ass    : Assumptions    -- assumptions
                   cons   : Constraints    -- collected constraints
                 ]

  | Empty       LHS.ass     = "[]"
                   .cons    = "[]"

  | Alternative LHS.ass     = "bs++cs++alts_ass"
                   .cons    = "new++as++patt_cons++expr_cons++alts_cons"
               LOC.new     = "[ Equality lhs_alpha patt_type"
                             ", Equality lhs_beta  expr_type ]"
                   .(as,bs) = "bind Equality ds expr_ass"
                   .(cs,ds) = "partition (isConstructor.fst) patt_ass"
```

```
SEM Decls [
          | unique : Int            -- type variable counter
          | ass    : Assumptions    -- assumptions
            cons   : Constraints    -- collected constraints
            env    : Assumptions    -- alpha's for the declarations
          ]

  | Empty LHS.ass    = "[]"
             .cons   = "[]"
             .env    = "[]"

  | Decl  LHS.ass    = "expr_ass++decls_ass"
             .cons   = "new:expr_cons++decls_cons"
             .env    = "(variable,alpha):decls_env"
             .unique = "decls_unique+1"
          LOC.alpha  = "TypeVariable decls_unique"
             .new    = "Equality alpha expr_type"

\EC
```

Bibliography

[AFFS98] A. Aiken, M. Fahndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Proceedings of the second International Workshop on Types in Compilation*, Kyoto, Japan, March 1998.

[AH95] A. Aiken and N. Heintze. Constraint-based program analysis. In *POPL'95 Tutorial*, January 1995.

[Aik99] A. Aiken. Introduction to set constraint-based program analysis. In *Science of Computer Programming, 35(1)*, pages 79–111, 1999.

[AWL94] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 163–173, Portland, Oregon, January 1994.

[BS93] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. In *ACM Letters on Programming Languages*, volume 2, pages 17–30, December 1993.

[CF91] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI '91)*, pages 278–292, 1991.

[CLR90] T. Corman, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[DB96] D. Duggan and F. Bent. Explaining type inference. In *Science of Computer Programming 27*, pages 37–83, 1996.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Principles of Programming Languages (POPL '82)*, pages 207–212, 1982.

[GJ96] B. Gaster and M. Jones. *A Polymporphic Type System for Extensible Records and Variants.* University of Nottingham, Nottingham NG7 2RD, UK, March 1996. Technical Report NOTTCS-TR-96-3.

[GVS96] M. Gandhe, G. Venkatesh, and A. Sanyal. Correcting errors in the curry system. In *Chandrum V. and Vinay, V. (Eds.): Proc. of 16th Conf. on Foundations of Software Technology and Theoretical Computer Science, LNCS vol. 1180, Springer-Verlag*, pages 347–358, 1996.

[Jon99] M. Jones. Typing haskell in haskell. In *Haskell Workshop*, September 1999.

[Jun00] Yang Jun. Explaining type errors by finding the sources of type conflicts. In Greg Michaelson, Phil Trindler, and Hans-Wolfgang Loidl, editors, *Trends in Functional Programming*, pages 58–66. Intellect Books, 2000.

[McA98] Bruce J. McAdam. On the Unification of Substitutions in Type Inference. In Kevin Hammond, Anthony J.T. Davie, and Chris Clack, editors, *Implementation of Functional Languages (IFL '98), London, UK*, volume 1595 of *LNCS*, pages 139–154. Springer-Verlag, September 1998.

[McA99] Bruce J. McAdam. Generalising techniques for type explanation. In *Scottish Functional Programming Workshop*, pages 243–252, 1999. Heriot-Watt Department of Computing and Electrical Engineering Technical Report RM/99/9.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[PT97] Benjamin C. Pierce and David N. Turner. *Local Type Inference*. Indiana University, November 1997. CSCI Technical Report 493.

[Wan86] M. Wand. Finding the source of type errors. In *13th Annual ACM Symp. on Principles of Prog. Languages*, pages 38–43, January 1986.

[WJ86] 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, St. Petersburg, FL, January 1986.