Repair Systems

Automatic correction of type errors in functional programs

Arjen Langebaerd

Master's thesis
Thesis number: INF/SCR-05-79

Supervisors:
Jurriaan Hage
Bastiaan Heeren

July 2006

Abstract

Type errors in functional programs can often produce confusing error messages. Traditional compilers merely state the fact that two or more types are in conflict with each other, often including additional type variables in the reported types that were introduced during inferencing. In some compilers the application of heuristics that recognizes instances of certain errors can generate a more useful error message. This thesis aims to devise a method that attempts to correct type errors in a more generic way, by defining a set of program transformations and implementing a strategy to apply them.

Firstly, we will investigate which transformations will be useful and examine their properties. Secondly, an algorithm to search for solutions to type incorrect input expressions will be described. Methods of optimizing the search process will be investigated. Finally we will measure the effectiveness of the algorithm by applying it to a set of ill-typed programs that were produced by students during the functional programming course at the University of Utrecht.

Contents

Chapter 1

Introduction

Compilers for modern functional languages such as ML and Haskell often generate cryptic error messages when there is a conflict between two or more types of expressions, especially when the types are large. One of the reasons for this is that it can be a nontrivial affair to locate the real source of the error. While programmers that have been around for a while are usually able to decipher such messages effectively due to their experience, novices can be easily confused. Most of the recent work in this area has focused on developing ways to enhance the accuracy of which error location is reported, such as the constraint based inferencer that is part of the Helium compiler developed at the University of Utrecht. Some attempts have been made at letting the compiler suggest fixes to repair the original program. Usually this means detecting patterns in the incorrect program. The effectiveness of this method depends on the quality and quantity of the collection of patterns. This thesis aims to devise and implement a system to repair arbitrary ill typed expressions in a more generic way using the Helium platform to determine which expressions should be given this treatment.

Chapter 2

Preliminaries

In this chapter we will first introduce some of the notation and basic concepts that are required for the rest of this thesis. If you are familiar with type systems in functional languages you can safely skip this section.

2.1 Type systems and inferencing

This thesis deals with manipulating expressions in a purely functional language in order to attempt to automatically repair a certain class of errors, namely type errors. In a functional language every computation is done by the evaluation of expressions. The result of these evaluations are values. Every value has a type associated with it. There are different kinds of type.

- Type constants. These are the basic types such integers (Int), booleans ($Bool$), characters ($Char$) etc, as well as type constructors such as the list ($[\,]$) and tuple ($()$) constructors
- Type variables. Type variables are used to represent types that are not yet known. In this thesis type variables are used to denote polymorphic types in the target language, as well as fixed types in the context of our repair system where we have yet to establish the actual type.
- Type application. The application of types allows for the construction of type compositions, for example the application of the list constructor to a basic type represents a list of those basic types, or the application of the function constructor ($\rightarrow$) to a set of types represents a function type.

2.2 Type inferencing

Every language construct that combines a set of subexpressions into a new expression imposes limitations on the types that the subexpressions can assume. For example the **if** construct combines three expressions, the guard which has to be a boolean type and the **then** and **else** expressions that have to be the same type. These specifications of type restrictions are called inferencing rules and allow an inferencing algorithm to determine whether a certain expression is well typed or not. When an expression is not well typed this means that one or more of the inferencing rules broke down and the type of the expression as a whole could not be determined. The order in which the rules are applied is important, different orderings result in different locations where a potential type error is located.

2.3 Unification

The process of inferring the type of an expression implies that we have to be able to determine whether two different types are 'compatible' with each other. Because some or all parts of a type can be unknown (type variables are used as placeholders) at the time of unification it is not simply a matter of testing two types for equality. For example, the inference rule for the language construct of the conditional expression specifies that both **then** and **else** branches must be of the same type. Assume that the **then** branch has type $v_1 \rightarrow Int$ (in which v_1 is a type variable) and the **else** branch has type $Bool \rightarrow v_2$. Unification of these two types is possible by substituting $Bool$ for v_1 and Int for v_2. Thus if unification is successful a substitution is the result. If unification fails because the types are not compatible some kind of error must be returned.

Chapter 3

Literature

In order to provide a context in which to place this thesis, this section will provide an investigation of other research performed in the fields related to automated correction of type errors. We will provide a summary for a selection of works that will detail how it relates to the problem we have set out to solve.

3.1 Type inferencing

3.1.1 Damas and Milner

Milner and Damas [1], building on the work of Roger Hindley [4], have provided the basic theory behind type-inferencing of a polymorphic functional language, the mechanism used to statically determine the type of a given expression, if such a type exists. The algorithms used to determine the type of an expression are known as the Hindley-Milner or the Damas-Milner algorithm (also known as $\mathcal{W}$). In a joint paper [1] Milner and Damas proved that the algorithm will always find the most general type for an expression or declaration thereby implying that the problem of determining well-typedness of a program is decidable.

3.1.2 Lee and Yi

Lee and Yi [6] presented a similar algorithm to $\mathcal{W}$ known as the $\mathcal{M}$ algorithm and proved it correct with respect to the type inferencing rules. $\mathcal{M}$ is "context-aware" meaning it has the ability to access information about the context in which the current expression is placed. Soon they came to the conclusion that $\mathcal{M}$ reports different but not necessarily better locations compared to the $\mathcal{W}$ algorithm. To try and improve on this Lee and Yi devised the $\mathcal{G}$ algorithm which can be considered a hybrid of of the $\mathcal{W}$ and $\mathcal{M}$ algorithms.

3.2 Improving the quality of error messages

One of the problems with type-inferencing is that the location where type inconsistencies are detected, depends on the order in which the inferencing rules are applied. Two well-known inferencing algorithms: $\mathcal{W}$ which applies inferencing rules top-down and left-to-right, and $\mathcal{M}$, which pushes the expected type down and applies inferencing rules bottom-up will detect a type-inconsistency at different locations. With both algorithms there will be instances where the location of the reported error will not be the location where the programmer actually made the mistake and reporting of this errant location often gives rise to confusing error messages. This phenomenon has prompted researchers to try and find algorithms that will eliminate this fixed bias in an effort to give more precise error messages.

3.2.1 Wand

Wand [12] proposes to keep track of which inferencing rules have contributed to a particular type inconsistency. Reporting of all the elements of the program that contribute to the error gives a more complete picture, although it has the disadvantage that the size of the error message can grow beyond what is deemed desirable in many cases.

3.2.2 Yang

Yang [5],[13] solved the problem of eliminating bias in location of the error site by creating what is known as the $\mathcal{U}_{AE}$ algorithm. This algorithm computes an assumption set for a node in the abstract syntax tree by unifying the sets of its children. This bottom-up method of inferencing assures that when an inconsistency is detected this will happen at the deepest construct that contains the inconsistency. Apart from bottom-up computation of the assumption sets an additional environment that records information about predefined functions is passed along from the top of the expression to the leaves. Because the $\mathcal{U}_{AE}$ algorithm will prove useful in our own approach we will now present a more detailed examination.

The $\mathcal{U}_{AE}$ algorithm is based on the unification of assumption sets. Children of expressions are typed independently and checked for consistency at their root. If an inconsistency is detected, it means that there exists at least one entity in the assumption set of more than one of the children and that the typings of the entity in the different sets cannot be unified successfully. Because unification takes place at the root of the expression, both sites that contribute to the inconsistency are found at the same time, and can be treated equally thereby eliminating the inherent bias of the $\mathcal{W}$ and $\mathcal{M}$ algorithms. Yang defined the algorithm for a language based on λ-calculus including various basic types, conditionals and let-constructs that allow polymorphic types. A small fragment of the algorithm is shown in Figure 3.1. It shows how one particular construct, namely function application, is handled.

First, the algorithm is called recursively on both of its subexpressions, storing the types found in τ_1 and τ_2 and the type-environments in TE_1 and TE_2. According to the inferencing rules, the type of the function e_1 (τ_1) should be equal to the type of a function that takes e_2 as its argument and returns an as of yet unknown type (β). The EnvConstraints function looks up all variables that are shared in both environments and UnifyConstraintSet unifies them. The end result is the type of the function return value (β), possibly refined by substituting information found in S, and the combination of the environments of each subexpression.

Figure 3.1 Inferring the type of a function application in Yang's $\mathcal{U}_{AE}$ algorithm

```
$e_1\ e_2 \longrightarrow$
  **let**
    $(\tau_1, TE_1) = \mathcal{U}_{AE}\ (TE, e_1)$
    $(\tau_2, TE_2) = \mathcal{U}_{AE}\ (TE, e_2)$
    $\beta$ be a fresh type variable
    $\Delta = \{\tau_1 \doteq \tau_2 \rightarrow \beta\} \cup$ EnvConstraints $(TE_1, TE_2)$
    $S =$UnifyConstraintSet $(\Delta)$
  **in**
    $(S\ (\beta), S\ (TE_1 \oplus TE_2))$
```

3.2.3 Johnson and Walz

Johnson and Walz [11] realized that in a lot of practical scenarios the number of correct uses will be greater than the number of incorrect uses. Using a maximum flow technique to determine the most likely constraints to be in error the information is presented to the programmer in an editor using a highlighting scheme.

3.2.4 Rittri

Rittri [8] describes a way of determining the location of the inconsistency by creating an interactive system that will, upon detecting an inconsistency, formulate a series of questions to establish the types that the programmer expects certain expressions to have and comparing those with the types that have been inferred by the system.

3.2.5 Simon, Huch and Chitil

Another interactive method, suggested by Simon, Huch and Chitil [9] allows the programmer to select bits of code and have the system provide typing information, supported visually by the use of colors. If a particular selected subexpression cannot be typed, the system will suggest a number of possible types so the programmer can choose the one thought to be correct. The method has been partially implemented and is known as TypeView. TypeView operates on a small functional language including lambda, let and case constructs. Upon querying the type of a certain expression, the tool will display a list of types that the expression should be compatible with, while coloring the locations in the program that gave rise to these typings. When querying expressions in ill-typed programs some of the types in the list will be incompatible. By allowing the user to disregard the types that are deemed correct, the list of possible locations where the inconsistency is caused can be narrowed down. For some programs however the list of types provided for an expression may be large and work is ongoing to provide the tool with a mechanism that enables it to be more selective about the possible typings it reports.

3.2.6 Stuckey, Sulzmann and Wazny

Stucky, Sulzman and Wazy [10] also devised a method to aid the programmer to locate the source of type errors interactively. Their tool, known as the Chameleon Type Debugger will upon selection of a particular expression that cannot be typed, highlight all of the other pieces of code that contribute to the type inconsistency. The implementation supports among other things inferencing of types for arbitrary locations, error explanations outlining all locations involved and explanation of suspicious-looking types.

3.2.7 Heeren and Hage

Heeren and Hage [2] came up with a different strategy based on type inferencing using constraints. In this approach the inferencing process is divided into three distinct phases: collection, ordering and solving. The reason for this is that now the ordering of the constraints can be made independent of the way in which they are generated. In order to be able to do a more global analysis of the program a typegraph is created. The typegraph is a special data structure for storing information with the property that it is possible to represent inconsistent programs. Using a typegraph makes it possible to apply heuristics that capture expert programmer knowledge to maximize the chance of reporting the correct location of the error or even suggest a fix. An implementation of this approach has also been provided in the form of TOP, a type inferencing framework. This framework has been used successfully in a Haskell compiler known as Helium [3].

3.3 Automated repair systems

Finding the right location of a type-error is critical in presenting the user with an accurate error message. In some cases, however, this will not be enough and a better understanding would be facilitated by letting the system provide a possible solution to repair the error. While there can be no guarantee that the system will provide the correct solution, it is possible that presenting the programmer with a modified type-correct version of his/her own erroneous code will be an effective alternative to an error message that states only incompatible types and is often clouded further by the inclusion of intermediate type variables that have been introduced by the inferencer. Interactive systems may be more accurate in finding the right location of a type error, but by requiring an additional number of steps to perform by the user, are also prone to bogging down workflow. For this reason we are interested in a repair system that requires a minimum of user interaction and is able to give constructive feedback in the form of an error message.

3.3.1 McAdam

McAdam [7] proposes a way of using linear isomorphisms to rewrite expressions in order to repair a type inconsistency so that the result can be presented as a suggestion to the user. When regular unification fails, the clashing expressions are unified modulo linear isomorphism recording every morphism used and injecting those into the original expression. Partial evaluation of the resulting expression will yield a type-consistent expression that can be suggested as a fix. For example consider the following expression: $\mathit{filter}\ (\mathit{intList}, \mathit{even})$ in which the function filter is a predefined function of the type: $(a \rightarrow \mathit{Bool}) \rightarrow [a] \rightarrow [a]$, $\mathit{intList}$ is simply a list of integers ($[\mathit{Int}]$) and even is a function having type $a \rightarrow \mathit{Bool}$. Obviously this expression is in error. The author has made two mistakes: the arguments of the function filter should be given separately, not paired and the order of the arguments is also incorrect. McAdam's reparation system is able to correct the error and will give a helpful error message. Because of the two mistakes, two morphisms (which are ordinary lambda terms) are required to correct the type-inconsistency: the argument has to be curried ($\lambda f\ (x, y) \to f\ x\ y$) and then subsequently the order has to be reversed ($\lambda f\ x\ y \to f\ y\ x$). The resulting morphism ($\lambda f\ (x, y) \to f\ y\ x$) can be applied to the original expression giving a type correct expression, although it looks not particularly helpful yet in its current state: $((\lambda f\ (x, y) \to f\ y\ x)\ \mathit{filter}\ (\mathit{intList}, \mathit{even}))$. Partial evaluation of this expression results in the constructive error message displayed in Figure 3.2.

However, it is not always clear whether the expression can be partially evaluated to such a level that the visual evidence of the morphisms that were applied is eliminated completely. Additionally, allowing only linear isomorphism to be used ensures certain desirable properties (e.g.. reversibility) but can limit the number of program errors that can be successfully handled by the system. Linear isomorphisms have the property that every type variable that occurs on the lefthand side of the transformation will also occur exactly once in the righthand side. While this property has some advantages (eg. it limits the number of possible unique transformations) it is also restrictive and will only be able to successfully unify two inconsistent types in a small number of scenarios.

Figure 3.2 McAdam's helpful error message

```
Try changing
   filter (intList, even)
To
   filter even intList
```

Chapter 4

Transformations

Before we can discuss the various transformations used in our repair algorithm we first need to establish formally what the target language will be. In this paper we will focus on repairing a subset of the Haskell functional language that includes all of the constructs represented by our grammar shown in Figure 4.3. For simplicity we will not allow transformations to **let** and lambda constructs themselves but we will allow transformations in their children. By enforcing this rule we can avoid problems that could arise as a result of changing scope. For example examine the code in Figure 4.1.

Figure 4.1 Ill-typed expression with nested **let**

```
let
  f = head
in
  let
    f x = ¬ x
  in
    f 1
```

If we would allow changes to the inner **let** construct itself, this could result in the change of scope for the use of the identifier f. If the inner **let** expression were to be selected, a transformation could possibly replace it by only the body of the **let** and result in the solution shown in Figure 4.2 which would still be incorrectly typed.

Figure 4.2

```
let
  f = head
in
  f True
```

While a mechanism could be devised to keep track of things like scoping of identifiers, this will significantly increase complexity of the algorithm while it is unclear whether there is anything to be gained. Instead our approach in this instance will have to select as input for the repair algorithm either:

- the children of the inner **let** or
- the children of the outer **let** and treat the inner **let** as a block that cannot be altered internally

or possibly both. The same holds for declarations and lambda abstractions. For this reason the abstract syntax tree does not model these constructs explicitly: either a child of such an expression is selected for repair or it will be represented by a *Blk* construct when it is part of a selected expression, signifying its unalterable nature. The type-variable *info* is used to store various sorts of information for every node.

Figure 4.3 Grammar representing a subset of Haskell

```
data AExpr info =
      App info (AExpr info) [AExpr info]
    | If info (AExpr info) (AExpr info) (AExpr info)
    | Tup info [AExpr info]
    | Lst info [AExpr info]
    | Var info String
    | Blk info
```

In order to be able to perform repair operations we first need to define what it consists of. A repair operation is either a single transformation or a composition of transformations that transform an ill-typed expression into a well-typed expression. A single transformation is defined as a function that takes an expression as input and produces a set of expressions as output. Transformations that are applied to incompatible expressions will produce a singleton set containing only the original input expression. We will use *Transform* as a shorthand for this type. It is defined as is shown in Figure 4.4. In order to be able to

Figure 4.4 Type of a single transformation

```
type Transform = Expression → {Expression}
```

effectively use transformations in a repair algorithm, we will first need to establish some of their properties.

Definition 1 (Composition of transformations)**.** *Suppose t and t' are well-defined transformations and e is an expression. The composition of t and t' is defined as*

$$(t' \circ t)\ e = \{e'' \in t'(e') \mid e' \in t(e)\}$$

In other words, the composition $t \circ t'$ is the union of all of the sets that result from applying the second transformation t' to each of the elements of the result of the application of the first transformation t on the original expression e.

Definition 2 (Equality of transformations)**.** *Two transformations t and t' are equal if and only if*

$$\forall\ e\ .\ t\ (e) = t'\ (e)$$

Definition 3 (Commutativity of composed transformations)**.** *Two transformations g and h are said to be commutative if for every expression e, $(g\ \circ\ h)\ e = (h\ \circ\ g)\ e$.*

Whether two transformations are commutative obviously depends on the nature of the transformations themselves. In general we will find that this is not the case but for some instances of specific compositions of transformations this property will hold which indicates that the enumeration of all possible composition sequences will likely generate a plethora of sequences that will yield the same result. Identification of such properties will enable us to limit the number of sequences that have to be evaluated without changing the set of solutions.

Definition 4 (Idempotency of transformations)**.** *A transformation t is idempotent if for every expression e it holds that $(t\ \circ\ t)\ e = t\ e$*

The composition of two idempotent transformations is another example of a sequence that can be disregarded safely without negatively affecting the number of unique solutions. However, occurrences of these sequences may not always be straightforward to recognize. In this chapter we will investigate what properties we can identify for each of the transformations that we would like to use.

The isomorphic transformations will be explored first. Later in the section we will explore what kind of non-isomorphic transformations can be included in order to improve the error-repairing capabilities.

4.1 Isomorphic transformations

4.1.1 Permutation of arguments

This is a fairly straightforward transformation where the arguments of a function in a particular application are rearranged in a different order. Consider the following code

$$(\lambda x\ y \rightarrow \mathbf{if}\ x\ \mathbf{then}\ y\ \mathbf{else}\ y + 1)\ 1\ \mathit{True}$$

This is an example of an application that can be repaired by swapping the two arguments to the lambda. As McAdam [7] showed, this operation can be expressed as the following morphism

$$\lambda f\ \ a\ b \rightarrow f\ \ b\ \ a$$

Application of the morphism to the original expression results in

$$(\lambda f\ \ a\ b \rightarrow f\ \ b\ \ a)\ (\lambda x\ y \rightarrow \mathbf{if}\ x\ \mathbf{then}\ y\ \mathbf{else}\ y + 1)\ 1\ \mathit{True}$$

This expression is a well-typed expression. Here we can completely eliminate the morphism itself by using partial evaluation

$$(\lambda x\ y \rightarrow \textbf{if}\ x\ \textbf{then}\ y\ \textbf{else}\ y+1)\ \textit{True}\ 1$$

In this example there is only one (very obvious) solution to repair the type-inconsistency. However, in type-incorrect function applications where there are at least two arguments of the same type, there will be more than one solution. Consider the following expression

$$(\lambda x\ y\ z \rightarrow \textbf{if}\ z\ \textbf{then}\ x+1\ \textbf{else}\ y+1)\ 1\ \textit{True}\ 2$$

In this case the application of the following morphisms to the original expression will produce a new type-correct expression

$$(\lambda f\ a\ b\ c \rightarrow f\ a\ c\ b)$$
$$(\lambda f\ a\ b\ c \rightarrow f\ c\ a\ b)$$

In fact if we assume that there are n groups of arguments that have the same type and g_x represents the number of arguments in group x, then the number of solutions is equal to

$$\prod_{x=1}^{x=n} g_x!$$

In a function application that has a total of 6 arguments in which there are 2 groups of 3 arguments with the same type, which is not uncommon, the number of solutions would be $3! \cdot 3!$ resulting in a total of 36 solutions. Not all of these solutions will provide the same degree of meaningful information to a programmer if they were to be presented however since clearly a solution that makes a change in the order of arguments that is not required will have less chance of being the solution that the programmer intended then a solution that leaves a larger number of arguments in their original position. Even if we measure the quality of a solution by the number of arguments that have had their order changed and selecting only those solutions with maximum quality, it is still possible to be left with a multitude of solutions that have an equal number of order changes. How we can discern the quality of solutions will be examined later, For now we generate all of the possible permutations of arguments and filter the results on the condition that they are type-correct.

Function $\mathit{permute}$ (see Figure 4.5) will generate all of the possible permutations of arguments to a given function. Since a single $\mathit{permute}$ will generate all of the possible arrangements of arguments, composition of $\mathit{permute}$ on the same application node will not provide any additional candidates. In other words, function permute is idempotent.

Lemma 1 (Permute is idempotent)**.** *Composition of function permute with itself is idempotent:*

$$\mathit{permute}\ \circ\ \mathit{permute} = \mathit{permute}$$

Proof. Follows from definition 4 and the definition of function $\mathit{permute}$ (Figure 4.5) □

Figure 4.5 Generation of permutations

$$
\begin{array}{ll}
\mathit{permute} & :: \mathit{EmptyInfo\ info} \Rightarrow \mathit{Transform\ info} \\
\mathit{permute}\ (\mathit{App}\ \_\ f\ a) & = \{\mathit{App\ emptyInfo}\ f\ a' \mid a' \in \mathit{perms}\ (a)\} \\
\mathit{permute}\ \_ & = [\,]
\end{array}
$$

4.1.2 Currying

Another commonly found type-error is the invalid use of currying in function applications. Some functions require their arguments to be a pair while others require their arguments simply juxtaposed. For example consider the following expression

$$(\lambda(x, y) \rightarrow x + y)\ 1\ 2$$

In this case, the arguments should have been given as a pair. This transformation can be written as the morphism $\lambda f\ a\ b \rightarrow f\ (a, b)$. The opposite transformation takes an application that has its arguments paired and separates them as in the following example.

$$(\lambda x\ y \rightarrow x + y)\ (1, 2)$$

This type-error can be repaired by application of the morphism $\lambda f\ (a, b) \rightarrow f\ a\ b$. For applications that involve a greater number of arguments however the possible pairing-configurations increases, resulting in a greater number of possible morphisms to apply. Knowing which morphism to apply to fix the error involves finding out in what particular configuration the function that is being called expects its arguments. As a first approach to the problem we will generate all of the possible pairings and let the type-checker decide which one is valid. The transformation to curry applications is shown in Figure 4.7 and its companion to uncurry applications is shown in Figure 4.6. Instead of altering the function call it would also be possible to transform the pattern in the definition of the function that is being called. However in our algorithm we will refrain from doing so for the following reasons:

- Including transformations for both the definition and the application of a function does not lead to any additional unique solutions.
- Functions included from a library must be assumed correct anyway.
- Allowing transformation of the definition of a function could potentially break the program in other locations where the same function is called.

The last reason is a double edged sword since it ignores the scenario where we have a locally defined function and multiple uses of that function that all use their arguments in the same way but are inconsistent with the definition. In this scenario the most likely solution would be to correct the function definition so that all of its uses become consistent.

As with *permute*, the *curry* transformation generates all possible ways to eliminate tuples and its counterpart *uncurry* generates all possible ways to introduce them. Therefore, the composition of these two transformations with themselves will not provide any new candidate expressions.

Figure 4.6 Uncurrying of applications

```
uncurryTuple :: EmptyInfo info ⇒ Transform info
uncurryTuple (App info fun args) =
  let
    currySingle (Tup _ telems) = telems
    currySingle notup = [notup]
  in
    [App info fun args′ | args′ ← (optmap currySingle args [[]]),
      or (map isTuple args)]
uncurryTuple _ = [ ]
```

Figure 4.7 Currying of applications

```
curryTuple :: EmptyInfo info ⇒ Transform info
curryTuple (App info fun args) =
  let
    buildapp (l, (ain, aout)) =
      App info fun (l ++ [Tup emptyInfo ain] ++ aout)
    split2 = splitTwice 0 args
  in
    map buildapp split2
curryTuple _ = [ ]
```

4.1.3 Parenthesization

Erroneously omitting parentheses or writing them while none are required are both common mistakes made by novice programmers. In previous transformations we have seen that rewriting of a single construct was sufficient to correct the error. Parenthesizing a particular subexpression however requires a rewrite of a larger part of the tree. In order to understand this we'll examine the following expression:

$$head\ filter\ even\ [1\,..]$$

in which *head* is a function having type $[a] \to a$, *filter* is a function with type $(a \to Bool) \to [a] \to [a]$ and *even* is a function that has type $a \to Bool$. In this case the author forgot to parenthesize: the expression should have been $head\ (filter\ even\ [1\,..])$. While this appears to be a relatively straightforward fix, let us examine what the abstract syntax trees for both expressions look like. The erroneous expression will have a tree that looks like Figure 4.8. The fixed tree in shown in Figure 4.9. Examining the difference between the two syntax trees shows that the transformation required will have to perform changes on multiple nodes.

Figure 4.8 Incorrect expression without parenthesis

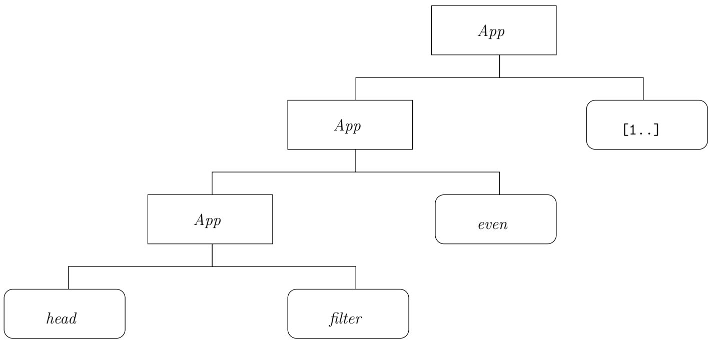
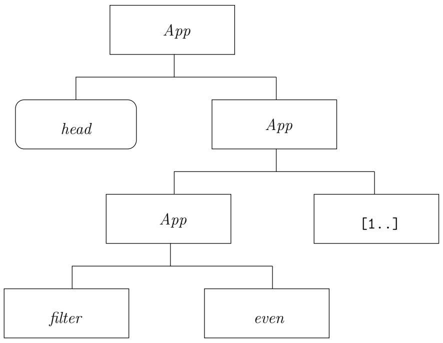


Figure 4.9 Correct expression with parenthesis



This transformation is in fact the composition of two transformations that change the order in which the applications are evaluated. The building block transformation, which we will call $swap$, can be written as a morphism in the following manner.

$$swap\ ((f\ \ a)\ b) = \{f\ \ (a\ \ b)\}$$

A visual representation of the transformation is show in Figure 4.10. This transformation has an inverse which we will call $iswap$ (Figure 4.11) that achieves the exact opposite effect.

$$iswap\ f\ \ (a\ \ b) = \{(f\ \ a)\ \ b\}$$

In the example mentioned earlier the application of two normal $swap$'s will repair the inconsistency. The original expression $head\ filter\ \ even\ [1\ldots]$ can be written as $(((head\ filter)\ \ even)\ [1\ldots])$. The first application of $swap$ will occur at the inner parenthesis-level and substitutes the values of f, a and b with $head$, $filter$ and $even$ respectively. The result of the first swap is:

$$(swap\ ((head\ filter)\ even))\ [1\,..] = (head\ (filter\ even))\ [1\,..]$$

The second application occurs at the outer level and substitutes the values of f, a and b with $head$, $(filter\ even)$ and $[1\,..]$ respectively. The end result looks like:

$$swap\ ((head\ (filter\ even))\ [1\,..]) = head\ (filter\ even\ [1\,..])$$

Figure 4.10 Visual representation of *swap*

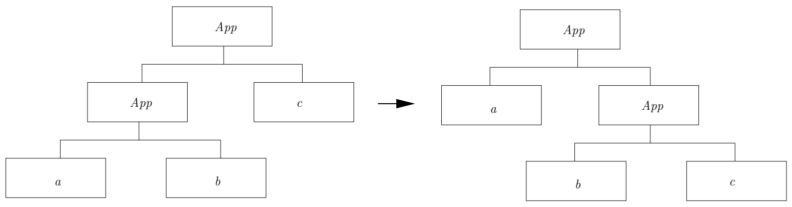
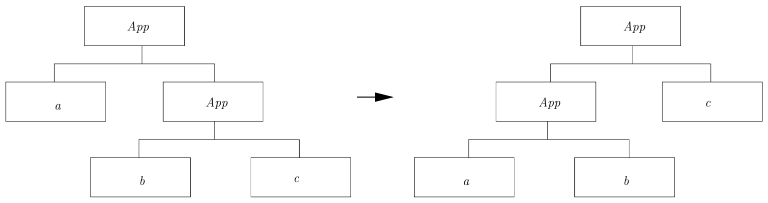


Figure 4.11 Visual representation of *iswap*



The implementation of the *swap* transformation is listed in Figure 4.12. The *swap* function determines all possible ways to split the arguments of a function application in three groups. The first and last groups contain arguments that will be left unaltered, the middle group contains arguments that will be used to build a new application that will be substituted in place of those arguments.

The implementation of the *iswap* function is shown in Figure 4.13. It builds new possible lists of arguments by examining the nature of the original arguments in a function application. Whenever an argument is found that is an application itself, two lists will be built: one will include this argument unaltered, the second will remove the application and substitute its function and arguments as arguments to the parent application.

Figure 4.12 Swapping an application

```
swap :: EmptyInfo info ⇒ Transform info
swap (App info fun args) =
  let
    splitTwice ml elems =
      let
        minlength = filter ((>ml) . length . fst . snd)
        splt as = zip (inits as) (tails as)
      in
    concatMap
      (minlength . (λ(l, r) → (map (λrs → (l, rs)) (splt r))))
      (splt elems)
    buildapp (l, (ain, aout)) =
      App emptyInfo
        fun
        (l ++ [App info (head ain) (tail ain)] ++ aout)
    split2 = splitTwice 1 args
  in
    map buildapp split2
swap _ = [ ]
```

Figure 4.13 Inverse of swapping an application

```
iswap :: EmptyInfo info ⇒ Transform info
iswap (App _ fun args) =
  let
    swapArgs (a : as) rlists =
      let newrlists =
        case a of
          app@(App _ fn arg) →
            (map (++[app]) rlists) ++
            (map (++[fn] ++ arg) rlists)
          noapp → map (++[noapp]) rlists
      in
        swapArgs as newrlists
    swapArgs [ ] rlists = rlists
  in
    [App emptyInfo fun args′ |
      args′ ← swapArgs args [[ ]], or (map isApp args)]
iswap _ = [ ]
```

4.2 Non-isomorphic transformations

4.2.1 Insertion and deletion

When traditional compilers try to infer function applications where one or more arguments have been forgotten, the error message can be especially confusing. For example consider the function application $f\ a\ b$ where we assume that the application is type-correct and that the type of a is not equal to the type of b. Now assume that the programmer has forgotten to include the first argument and has written $f\ b$ instead. The inferencer will now try to unify the type of the first argument that f expects with the totally unrelated type of b. A transformation that would be capable of introducing an argument with the right type for a would be valuable. For a more concrete example consider the following expression.

$$(\lambda x\ y\ z \rightarrow \mathbf{if}\ z\ \mathbf{then}\ x\ \mathbf{else}\ y)\ 1\ \mathit{True}$$

Intuition tells us that either the first or the second argument have been omitted, since the last argument has type Bool and is probably supposed to be used as the guard in the **if**-expression. Introduction of an additional argument would make this expression type-correct. The following morphisms, when applied to the original expression would both repair the inconsistency.

$$(\lambda f\ x\ y \rightarrow f\ a\ x\ y)$$
$$(\lambda f\ x\ y \rightarrow f\ x\ a\ y)$$

where a is a new variable having type Int. Both morphisms are equally valid: there is no reason to prefer the first over the second or vice-versa. The transformation to fix these kinds of type-inconsistencies will have to generate solutions that include all of the possible places where arguments can be inserted. In the case of this example that would include the generation of the solution where we assume that the first two arguments are correct and the last one has been forgotten (morphism $(\lambda f\ x\ y \rightarrow f\ x\ y\ a)$) but since that will result in another type-incorrect expression the type-checker will drop this variant. The actual transformation to insert missing arguments in every possible location is listed in Figure 4.14.

Figure 4.14 Insertion of arguments

```
insertArgument :: EmptyInfo info ⇒ Transform info
insertArgument (App info fun args) =
  let
    argsplit = zip (inits args) (tails args)
    newarg = Blk emptyInfo
    newarglists =
      case args of
        [] → [[newarg]];
        _ → map (λ(i, t) → (i ++ [newarg] ++ t)) argsplit
  in
    [App emptyInfo fun args' | args' ← newarglists]
insertArgument _ = []
```

To handle cases where a programmer supplies more arguments than are required, the opposite transformation, deletion of certain arguments, can also be convenient. This can be dangerous however, if we allow this transformation to be performed, deletion of all of the arguments of the toplevel application of a program for which a type-signature has not been provided results in a type-correct program, but will most likely not be what the programmer had in mind. Consider the following expression.

$$(\lambda x\ y \rightarrow x + y)\ 1\ 2\ 3\ .$$

Possible morphisms that result in type-correct expressions include the following

$$\begin{array}{l}(\lambda f\ \ a\ b\ c \rightarrow f\ \ a\ b) \\ (\lambda f\ \ a\ b\ c \rightarrow f\ \ a\ c) \\ (\lambda f\ \ a\ b\ c \rightarrow f\ \ b\ c) \\ (\lambda f\ \ a\ b\ c \rightarrow f\ \ a) \\ (\lambda f\ \ a\ b\ c \rightarrow f\ \ b) \\ (\lambda f\ \ a\ b\ c \rightarrow f\ \ c) \\ (\lambda f\ \ a\ b\ c \rightarrow f)\end{array}$$

The transformation is shown in Figure 4.15. If no context is provided, the number of possible solutions can grow enormously when allowing this transformation. While certain type-incorrect expressions cannot be repaired without it, allowing a large number of these types of transformations will however most likely not enhance the quality of the solution. In the next chapter we will outline a strategy for applying the transformations that we have seen that will take into account the potentially detrimental effects of allowing large numbers of these kinds of transformations.

Figure 4.15 Deletion of arguments

```
deleteArgument :: EmptyInfo info ⇒ Transform info
deleteArgument (App info fun args) =
    let
        delElem as i = (take (i − 1) as) ++ (drop i as)
        newarglists = map (delElem args) (enumFromTo 1 (length args))
    in
        [App info fun args' | args' ← newarglists]
deleteArgument _ = []
```

Chapter 5

Application of transformations

In the previous chapter we have seen that the application of a single transformation to a small expression can already result in a multitude of possible solutions. For many real world occurrences of type errors however a combination of transformations will be required to produce an adequate solution. Simply combining all of the possible transformations in every possible order will lead to an enormous explosion of the number of altered expressions that need to be evaluated. As with a single transformation like $permute$, we are most interested in solutions that require a minimum number of alterations to the original expression. But just minimization of the number of transformations is insufficient. For example compare the single use of a $permute$ transformation to the single use of a $deleteArgument$ transformation. The powerful $deleteArgument$ is capable of singlehandedly eliminating entire subexpressions, which is probably not what the programmer had in mind. This is especially dangerous when the expression in question is the toplevel expression of a program for which no type signature has been provided, essentially enabling the repair algorithm to produce solutions of any type as long as the expression itself is internally consistent. For example consider the following program.

$$map\ ([1,2],(+1))$$

Since no type signature has been provided the repair algorithm has no way of knowing what type the programmer expects the program to have. Therefore by applying a single instance of the $deleteArgument$ transformation, a solution is provided having type $(a \to b) \to [a] \to [b]$ (the type of map). However in this case what the programmer had in mind was probably $map\ (+1)\ [1,2]$. However to come to this expression we need the application of both the $curryTuple$ and $permute$ transformations. To restrict the power to make radical changes in expressions of transformations such as $deleteArgument$ we introduce the notion of a repair cost. Every transformation should be parametrizable with a cost that is proportional to the degree of modification that it is capable of. The use of powerful transformations like insertion and deletion can thus be avoided unless absolutely necessary by assigning a high cost value to them. By minimizing cost instead of the number of transformations we will be better able to distinguish the quality of a solution. In light of the example above, we will want to make sure that the combined repair cost of the solution $permute \circ curryTuple$ is lower than the cost of the solution $deleteArgument$.

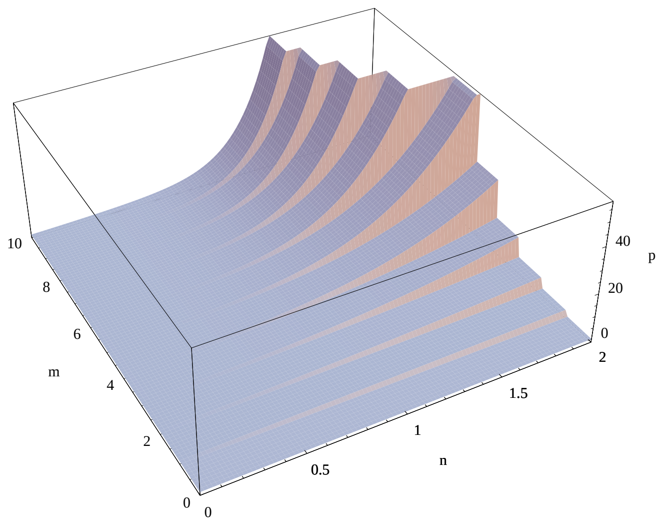
5.1 Solution space

The number of possible compositions of transformations grows with both the number of distinct transformations and the maximum allowed number of transformations that make up a composition. If we are using m unique transformations and a maximum of n transformations that make up the composition the total number of permutations (p) would be:

$$p(n, m) = \sum_{i=1}^{i=m} n^i$$

So even if we limit ourselves to the number of transformations discussed in the previous chapter ($m = 7$) with only a modest composition length (say $n = 4$) we already have 2800 possible compositions. How the amount of unique transformations and the maximum composition length of those transformations affect the total number of solutions is illustrated in Figure 5.1.

Figure 5.1 Effect of the maximum composition length (n) and number of unique transformations (m) on the size of the solution space (p)



The function that transforms a single expression into an (infinite) list of all possible transformed expressions uses the *transitiveClosure* function to construct the list of all possible compositions of transformation primitives and applies them to the expression in the list-comprehension shown in Figure 5.2.

If we allow each of these composed transformations to be applied to every node in the tree and we remember that some of the transformations (possibly all of them!) in these compositions produce a multitude of solutions themselves, we are faced with an enormous amount of possible tree-alterations to consider. So obviously a brute force strategy is unlikely to produce satisfying results, after all we would like to debug our program before the end of

Figure 5.2 Transformation of an expression

```
transform :: Transformations info → AExpr info → [AExpr info]
transform transList aexpr =
   [new | trans ← transitiveClosure transList, new ← trans aexpr]
```

the century if at all possible. In this chapter we will investigate what we can do to find a high quality solution while minimizing the number of solutions that are considered.

5.2 Bottom up aggregation

Just applying all of the possible transformation compositions to every possible node in the tree is of course a rather unsophisticated approach. It is likely that many subexpressions are in fact correctly typed themselves. A better approach is to try and leave parts of the program that are internally type-consistent unmodified as much as possible. To this end we will use a bottom-up strategy reminiscent of the one used by the $\mathcal{U}_{AE}$ algorithm described earlier. It will inspect subexpressions and leave them intact as much as possible, only transforming nodes whenever there is a clash between the types of subexpressions that are unified by their root node. When a particular subexpression has been typed successfully this information will be stored in the root of the that subexpression. When transformations are applied to some parent of this expression that require it to be type checked again, this information can be reused without recalculation of the type, in the entire subexpression.

5.2.1 Verifying type-correctness

Initially, when a new expression tree is used as input for the repair algorithm, only certain nodes are already supplied with type information. Since the expression is built up from terms for which the type is established already, every leaf in the syntax tree is already typed, while every other node does not have a type yet (and quite possibly none exists if there is an inconsistency). To determine the type of a node we determine the types of the children and then make sure the conditions that are associated with the node hold by applying the according inferencing rule. For example when we want to infer the the type of the function application *map even* $[1\,..]$ we know that function application requires the type of the arguments to match the function that is being applied. In this case that results in the condition: $v_3 = v_1 \rightarrow v_2 \rightarrow \beta$ where v_3 is the type of *map*, v_1 and v_2 are the types of the first and second arguments respectively and β is the return type. If unification of the types in the condition succeeds, it will result in a list of substitutions which are also stored in every node. To represent type information we will use the type *Typed* as defined below.

```
type Typed = (Tp, FiniteMapSubstitution)
```

To represent the fact that some nodes in the abstract syntax tree have no type information initially, the complete type to represent expressions becomes: *AExpr* (*Maybe Typed*). Any solutions resulting from running the repair algorithm over these input expressions will have to produce an expression for which all nodes can be typed, thus the type of solutions can be expressed as $AExpr\ (Typed)$. Because the inferred types are stored in every node they only have to be determined once. Whenever a transformation is applied, it will reset the type information to $Nothing$ so that the new expression can be evaluated again. To limit the amount of duplicate code we will separate the gathering of information about a certain type of node from the actual unification of the types in the conditions as much as possible. For applications the information gathering is implemented as is shown in Figure 5.3.

Figure 5.3 Typing function application

```
typeAExpr :: AExpr (Maybe Typed) → Maybe (AExpr Typed)
typeAExpr aexpr =
  case aexpr of
    App info fun args →
      do (funTyped, (tp1, s1)) ← recAExpr fun
         (argsTyped, (tps2, ss2)) ← recAExprs args
         let makeApp x = App x funTyped argsTyped
             β = freshTVar ()
         superUnify makeApp info β (s1 : ss2)
             [(tp1, foldr (.− > .) β tps2)]
    ...
```

First the types and substitutions of the children are collected by recursive calls. Additionally, a function is constructed that will rebuild the current node if it has been found to be consistent. Then the conditions (in the case of application, there is only one condition) are specified and the unification function is called with all of these results as arguments. The unification function itself is shown in Figure 5.4.

Figure 5.4 Unification of conditions

```
superUnify :: (Typed → AExpr Typed) → Maybe Typed → Tp →
  [FiniteMapSubstitution] → [(Tp, Tp)] → Maybe (AExpr Typed)
superUnify make maybeTyped β ss cs =
  case maybeTyped of
    Just t → return (make t)
    Nothing →
      do superSub ← unifySubstList synonyms ss
         let (ts1, ts2) = unzip (superSub |− > cs)
         s ← mguMaybe synonyms (tupleType ts1) (tupleType ts2)
         let final = s @@ superSub
         return $ make (final |− > β, final)
```

5.2.2 The repair algorithm

To avoid having to define the repair function for every possible type of node, we will define a function that will construct the following:

- a collection of all of the children of the current node
- a function that takes a collection of children and is able to rebuild the original node

The main function *repairExpr* (Figure 5.5) takes care of this and passes the items as arguments to the function that does the real work, *superRepair* which is shown in Figure 5.6. Since the implementation of *repairExpr* is similar for every type of node in the abstract syntax tree only the definition for function application is shown here.

Figure 5.5 Main repair function

```
repairExpr :: Expr → [Expr]
repairExpr App _ fun args =
  let
    children = fun : args
    buildApp = λ(fun : args) → App emptyInfo fun args
  in
    superRepair children buildApp
```

Function *superRepair* recursively calls *repairExpr* on each of the children in the list. This results in a list of possible alternatives for every child. These alternatives must be combined to produce a list of every possible combination of the children. This is achieved by simply making every possible combination of the available children in each list. These combinations are then verified for correctness by calling function *check*, where the node is rebuilt using the provided function-argument *makeNode*, and is then type-checked: if the expression is type-consistent it is returned unaltered, otherwise the transformations are applied.

Since every combination of alternate children is evaluated, we have to limit the list of allowed transformation compositions for the algorithm to terminate at all because in some cases the list of alternates will be infinite if we do not. Also, the current algorithm does not keep track of repair costs. This implies we are producing a list of solutions that is sorted by composition length, which we have concluded earlier to be inadequate for making sure the best solution comes out on top. In the next sections we will implement a way to only evaluate the most promising solutions by making use of lazy evaluation and introduce repair costs to the algorithm.

5.3 Lazy evaluation of solution space

5.3.1 The *Progress* data type

The main weakness with respect to the complexity of the algorithm lies in the fact that we do not keep track of the amount of work we are doing to produce a certain expression. For example consider we are currently evaluating the type-incorrect expression e. We find

Figure 5.6 Repair worker function

```
superRepair :: [Expr] → [Expr]
superRepair children makeNode =
  let
    check :: [Expr] → [Expr]
    check newChildren =
      let
        newNode = makeNode newChildren
        newExprs = transform newNode
      in
        case (typeCheck newNode) of
          Just typedExpr → [typedExpr]
          Nothing → catMaybes . (map typeCheck) . newExprs
  in
    concatmap check . combine . (map repairExpr children)
```

that by applying transformation t_1 (one of the first in the list of possible transformation compositions) we produce a type-consistent expression so it is added to the list of possible solutions. Now the algorithm will try to apply other, more complex transformation compositions to e, to see if there are any other solutions. Assume that it does not find any consistent alternatives until we arrive at the composition $t_n \circ .. \circ t_2 \circ t_1$ where n is rather large. This solution is added to the list as the second possible solution. To the algorithm there is little difference between the two solutions while in fact there is a huge difference: it took a very large amount of time to come up with the second solution compared to the first. So large in fact, that it is questionable whether we are interested in this particular solution at all. By keeping track of not only successful solutions but also failed attempts, we can restrict the number of solutions that are to be evaluated rather than restricting the number of transformation compositions. In other words, the amount of work required to come up with a particular solution is the deciding factor when determining whether we are interesting in it. Therefore we will encapsulate solutions in the data structure *Progress* where both successful and failed solutions are represented. The definition of the *Progress* data type can be seen in Figure 5.7.

Figure 5.7 The *Progress* data type

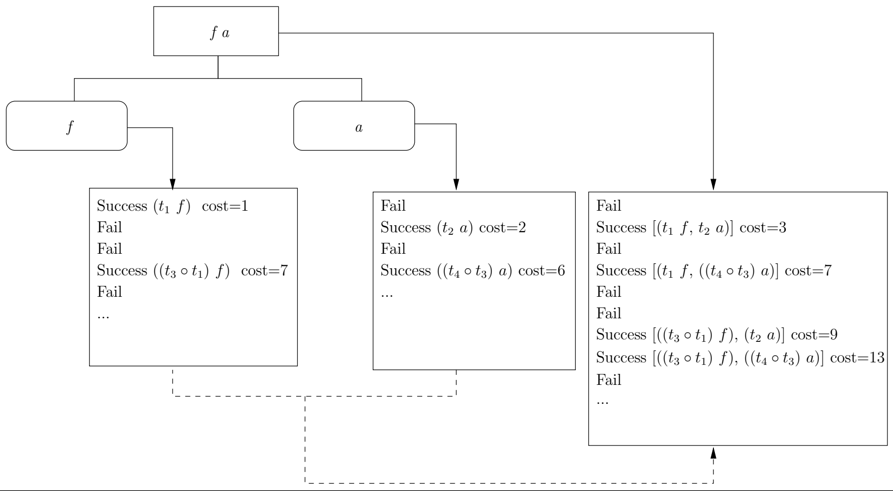
```
data Progress a = Success a | Fail
```

5.3.2 Combination and merging of solutions

Now we will redefine the function *combine* to adapt to the new data type. Every child expression now produces a list of *Progress a*, so all children together produce a list of a list of *Progress a*. We will combine this list of lists into a list of *Progress* $[a]$. The process is illustrated in Figure 5.8: in this example there is a list of alternate expressions for both the function f and its argument a. To be able to build all of the variations of the whole application $f\ a$ the lists are combined. The function to combine two lists of solutions is shown in Figure 5.9. For it to work correctly it requires the following:

- A function must be provided that maps instances of *Progress a* to an integer so that the order in which they should be appended to the result list can be determined.
- The two lists that are to be combined must be ordered themselves with respect to the ordering function for the algorithm to produce an ordered list of combinations as in the merge of merge sort.

Figure 5.8 Combination of solutions for subexpressions



$f\ a$
f
a
Success $(t_1\ f)$ cost=1
Fail
Fail
Success $((t_3 \circ t_1)\ f)$ cost=7
Fail
...
Fail
Success $(t_2\ a)$ cost=2
Fail
Success $((t_4 \circ t_3)\ a)$ cost=6
...
Fail
Success $[(t_1\ f,\ t_2\ a)]$ cost=3
Fail
Success $[(t_1\ f,\ ((t_4 \circ t_3)\ a)]$ cost=7
Fail
Fail
Success $[((t_3 \circ t_1)\ f),\ (t_2\ a)]$ cost=9
Success $[((t_3 \circ t_1)\ f),\ ((t_4 \circ t_3)\ a)]$ cost=13
Fail
...

Figure 5.9 Function to combine lists of solutions

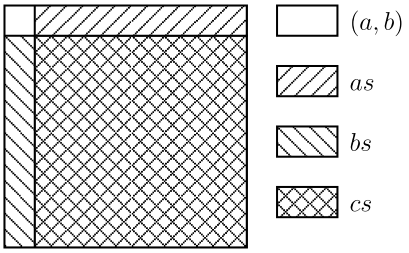
```
combine :: (a → Int) → [Progress a] → [Progress a] → [Progress (a, a)]
combine f = rec
where
  rec (Fail : xs) ys = Fail : (rec xs ys)
  rec xs (Fail : ys) = Fail : (rec xs ys)
  rec [] _ = []
  rec _ [] = []
  rec ((Success x) : xs) ((Success y) : ys) =
    let
      as = [Success (x, z) | Success z ← ys]
      bs = [Success (z, y) | Success z ← xs]
      cs = rec xs ys
      g (i, j) = f i + f j
    in
      (Success (x, y)) : (merge g as (merge g bs cs))
```

The algorithm starts at the head of the lists and examines one element of each at a time. When Fail elements are encountered they are moved to the result list until two Success entries (a and b) are discovered. The combination of these entries ((a, b)) is then added to the result list. Now we have to combine the following:

- $\mathit{Success}\ a$ with the remaining Success entries in the second list (as).
- The remaining Success entries in the first list with $\mathit{Success}\ b$ (bs).
- The remaining Success entries of both lists (cs).

If we represent values in the first list along the the x-axis and values in the second list along the y-axis, we can represent the combinations of these lists graphically as is shown in Figure 5.10.

Figure 5.10 Combination of solutions



To unite these three lists we use the function merge which is shown in Figure 5.11.

Figure 5.11 Merging lists of combinations

```
merge :: (a → Int) → [Progress a] → [Progress a] → [Progress a]
merge f = rec
  where
    rec [] ys = ys
    rec xs [] = xs
    rec (Fail : xs) ys = Fail : (rec xs ys)
    rec xs (Fail : ys) = Fail : (rec xs ys)
    rec ((Success x) : xs) ((Success y) : ys)
      | f x < f y = (Success x) : rec xs (Success y : ys)
      | otherwise = (Success y) : rec (Success x : xs) ys
    rec (x : xs) (y : ys) = x : y : (rec xs ys)
```

Merging of two lists is a little more straightforward: after processing all of the Fails until two Successes are discovered, the ordering function is used to determine which one should be placed up front and the rest of the list is determined by a recursive call to the remainder of the lists.

The incremental definition of the *combine* and *merge* functions (i.e. there is not a single part that requires manipulation of all of the elements of the lists at once) allows us to specify up to how many solutions we want to return (by using the prelude function *take*) without requiring the entire list to be evaluated.

5.4 Introducing repair costs

In the previous section we encountered the fact that a function is required to compare solutions for ordering purposes. In the introduction of this chapter we discussed the need for a method of keeping track of the cost of transformation compositions. In describing the *Progress* data-structure, we did not explicitly specify the type of a, only that some function having type $a \rightarrow Int$ must be provided. To realize a lazily evaluated list of solutions that are ordered by the total cost of the transformations used we will substitute $(AExpr, RepairAdmin)$ for a where RepairAdmin is the type shown in Figure 5.12.

Figure 5.12 Merging lists of combinations

```
newtype RepairAdmin = RepairAdmin ([String], Int)
```

The list of strings will serve to keep track of which transformations were used to arrive at a particular solution. This is useful during construction of the message that will inform the user about the nature of that solution. The integer will be used to accumulate cost-information. Every transformation will be initialized with a cost-value and the cost of compositions of transformations simply corresponds to the sum of their individual costs. The global value of *RepairAdmin* will have to be continually updated during calculations. By instantiating it as a member of the *Monoid* class we can avoid having to manage the data explicitly and thereby reduce code clutter. The instance definition is shown in Figure 5.13, it requires the definition of an identity element and an associative binary operator over its elements.

Figure 5.13 *RepairAdmin* is instantiated as a member of class *Monoid*

```
instance Monoid RepairAdmin where
  mempty = RepairAdmin ([], 0)
  mappend (RepairAdmin (xs1, i1)) (RepairAdmin (xs2, i2)) =
    RepairAdmin (xs1 ++ xs2, i1 + i2)
```

5.5 Integration of concepts

We are now ready to update the original 'naive' algorithm with the concepts of lazy combination of solutions and repair costs. The updated algorithm can be seen in Figure 5.14.

The call to *recAExprs* will now produce a list of solutions with type:

Figure 5.14 Updated repair algorithm

```
superRepair make =
  let
    op (Success pairs) rest =
      let
        (aexprs, admins) = unzip pairs
        app       = make (map (fmap Just) aexprs)
        newAdmin = mconcat admins
        newapps = (transform transList app)
        newExprs = concatMap (checkFail newAdmin) newapps
      in
      case typeAExpr synonyms app of
        Just typedExpr →
          Success (typedExpr, newAdmin) : rest
        Nothing → (Fail : (merge getCost newExprs rest))
    op _ rest = Fail : rest
  in foldr op [] . combiList getCost . recAExprs
```

$$Progress\ (AExpr\ \ Typed, RepairAdmin)$$

for every child. These lists are combined using the new combination function and the results are processed by *op* in the *foldr*. The original parent is reconstructed using function *make* and is checked for type-correctness. When the expression is found to be correct, it is certain that none of the possible solutions in the rest of the list will ever provide a cheaper solution: the current element of the solution list was cheaper than all of the remaining elements, and no additional transformations are necessary. When the expression is found to be incorrect, a Fail is added to the result list, and the transformations are applied to by a call to the *transform* function. Because we cannot be certain that the results of these transformations will produce solutions cheaper than other candidates in the rest of the list, the results have to be *merge*d with *rest*.

Chapter 6

Type propagation

6.1 Propagation rules for type information

Reparation of an abstract syntax tree that contains a type-inconsistency by traversing every node in a bottom-up method, building up type correct alternative subtrees along the way is a powerful way to come up with, in many cases, a large number of possible solutions. The problem is that the method is a bit too powerful and as we have seen the corresponding solution space can explode in relation to the number of unique transformations, the amount of transformations that we are allowed to apply and the number of constituent parts of the tree. While selecting solutions based on minimization of repair cost alone can provide a satisfactory solution in some cases, it will be totally off the mark in others. By incorporating type context into the picture we can finetune our selection process by quickly eliminating those solutions that are type-consistent internally but don't unify with the type that the rest of the program expects from this particular subexpression. Sometimes this information is not available, for example the selected expression is the root node of the tree and no type signature has been provided. In the many cases where type-context information is available however it would be preferable to make use of it while traversing the solution space. In order to accomplish this, the previous algorithm for the bottom-up generation of solutions will have to be expanded to enable type information flowing from top to bottom to be incorporated in the generation process. For this purpose we need to establish exactly in what way type information is propagated down into the tree. Once all of the available type-information has been distributed we can generate variants of subexpressions as before with the exception that now the context type can be taken into account. The complete algorithm will have three distinct stages:

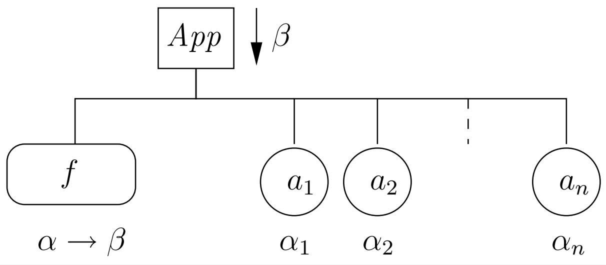
- Stage 1. Any available type information arising from the context will be propagated from top to bottom.
- Stage 2. Bottom up building up of subexpressions taking available type-information into account
- Stage 3. A conflict has been found. We can try to repair this conflict by applying transformations to the current node as before. Unlike before however we can now also build alternate solutions by initializing special instances of the first stage with more detailed and/or different type information that is to be pushed down.

Previously our algorithm was limited to only being able to apply transformations to the node where an inconsistency was detected. Allowing a greater number and more complex transformations could in theory compensate for this limitation somewhat, but will result in even worse running times. Propagating type information downward enables us to, upon detecting a type conflict in an expression, find out whether we can establish a type for the inconsistent node anyway by ignoring the type constraints posed by one or more of its children. To limit the potential for large modification we would like to ignore the constraints of as few of the children as possible. Unification of the conditions with respect to the sets of enabled children can produce a more specialized type for whatever children where not in the set. Re-evaluating these children by pushing down the more specialized types enables us to apply transformations on any part of the tree beneath the parent where the inconsistency was detected.

6.1.1 Function application

If the expected return type of the application of a function f is assumed to be β, then we can determine that the type of the function being applied is $\lambda\alpha_1 \to \lambda\alpha_2 \to ... \to \lambda\alpha_n \to \beta$ for some types $\alpha_1, \alpha_2, ..., \alpha_n$ where n is the number of arguments f is being applied to. Once

Figure 6.1 type propagation over function application



every alternative of each child has been validated and their types have been established, all of the combinations of the alternatives are inspected and there are a number of possible scenarios to consider:

- the arguments do not match the function type;
- there is a mismatch between the amount of arguments the function expects and the actual number of arguments supplied;
- the arguments match the function type, but the context type does not match the return type;
- the arguments match the function type and the context type matches the return type.

The first two scenarios signify an internal inconsistency and can be handled as before.
The second scenario is something we have not encountered before and recognition of this situation enables us to take action in order to produce a subexpression that will deliver the expected return type. As an example consider the simple code fragment in Figure 6.2:

Figure 6.2 Type incorrect program fragment containing function application

$g :: a \rightarrow a$
$g = \dots$
$f :: Int$
$f = g\ True$

The selected expression in this case will be $g\ True$ and while internally consistent the expected return type will be Int due to the presence of the type signature. The types that will be pushed down here are $\alpha \rightarrow Int$ for the function and α for the single argument. During evaluation of the function, unification of $\alpha \rightarrow Int$ with $\beta \rightarrow \beta$ will yield the substitution $\{\alpha := Int\}$ while evaluation of the argument leads to unification of α with $Bool$, yielding the substitution $\{\alpha$:=Bool$\}$ as a result. Clearly these substitutions are in conflict with each other. Apart from applying transformations to the application node we build a list of possibilities to initialize special instances of stage 1. In this example possible elements of this list would include:

- Keeping the function fixed and re-evaluating the argument pushing down Int for α
- Keeping the argument fixed and re-evaluating the function pushing down $Bool \rightarrow Int$

In the last scenario there are no problems and no further action is necessary

6.1.2 Conditionals

If the expected type of the expression is β as before, the type of the guard-expression must be $Bool$ while both the left-branch and the right branch must have type β. As an example consider the code in Figure 6.4.

Figure 6.3 type propagation over conditionals

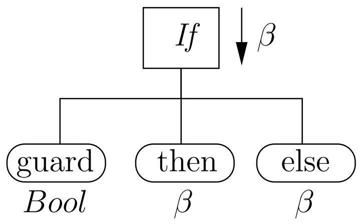


Figure 6.4 Type incorrect program fragment containing a conditional expression

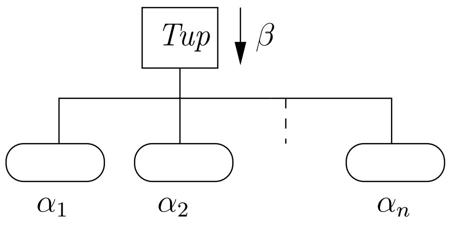
$f :: Bool \rightarrow Int$
$f\ x = \textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ False$

The conditional expression inside the declaration of f will be selected as the culprit: it contains an internal inconsistency. However at first sight it is not entirely clear whether the result of the expression must produce something of type Int or $Bool$. This is where context can help decide which of the two alternatives to favor, the type signature of function f requires the return type to be Int.

6.1.3 Tuples

For tuples, we must ensure that β is unifiable with $(\alpha_1, \alpha_2, .., \alpha_n)$ for some types α_1, α_2, .. , α_n.

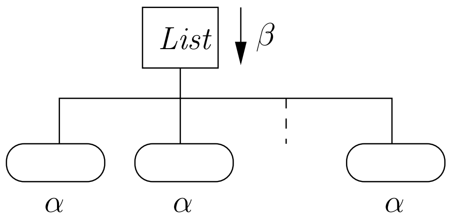
Figure 6.5 type propagation over tuple construct



6.1.4 Lists

For lists we must ensure that β is unifiable with $[\alpha]$ for some type α.

Figure 6.6 type propagation over list construct



6.2 Incorporating type context information

6.2.1 Narrowing down of the solution space

Now we will extend the original algorithm with the objective of limiting the number of possible solutions by using type context information. The function $typeExpr$ which previously took care of determining whether an expression was well-typed or not will be updated to take this information into account. Instead of using a fresh type variable for the type of the whole expression we will now substitute β to enforce compliance with the expected type. The updated function is called $typeAExprWithContext$ and the case for function application can be seen in figure 6.7.

Figure 6.7 Typing an application using context

```
typeAExprWithContext :: AExpr (Maybe Typed)
  → Tp → Maybe (AExpr Typed)
typeAExprWithContext aexpr β =
  case aexpr of
    App info fun args →
      do
        childTypes ← typePropagate synonyms aexpr β
        (funTyped, (tp1, s1)) ← recAExpr fun (head childTypes)
        (argsTyped, (tps2, ss2)) ← recAExprs args (tail childTypes)
        let makeApp x = App x funTyped argsTyped
        superUnify makeApp info β (s1 : ss2)
          [(tp1, foldr (.− > .) β tps2)]
```

The recursive calls to the children are now accompanied by their respective expected types which have been determined by the *typePropagate* function for which the case for applications is displayed in figure 6.8. Upon the first visitation of the application node no information about the types of the arguments is known and thus fresh type variables are instantiated to take their place. The expected type β is known however and this is reflected in the definition of the expected type of the function.

Figure 6.8 Propagating expected type of an application to its function and arguments

```
typePropagate :: AExpr info → Tp → Maybe Tps
typePropagate expr β =
  case expr of
    (App _ _ args) →
      let
        argTypes = freshVars (length args)
        funType = foldr (.− > .) β argTypes
      in
        Just (funType : argTypes)
```

6.2.2 Directed repair

As was mentioned before, apart from ruling out a host of incompatible solutions upon typing the current node, we can also try to generate more accurate solutions by re-evaluating specific branches in the expression-tree when more detailed type information has become available. Upon detection of a type conflict we can determine whether disregarding the constraints posed by one or more of the children of the current expression solves the type conflict. To realize this we implement the function *findConflict* shown in figure 6.9.

Figure 6.9 Determining sets of children that unify successfully with respect to the conditions associated with the current expression

```
findConflict :: [AExpr Typed] →
  (Tps → [(Tp, Tp)]) → [[(AExpr Typed, Bool)]]
findConflict exprs makeConditions = rec exprs 1
  where
  rec [] _ = []
  rec as omitAmount =
    let
      perms = (doPerms as omitAmount)
      typesAndConditions = evalTypesAndConditions makeConditions
      permsWithConditions =
        zip perms (map typesAndConditions perms)
      (unifiedPerms, _) =
        unzip (filter (doUnify syns) permsWithConditions)
    in
      if (length unifiedPerms) > 0
      then unifiedPerms
      else rec as (omitAmount + 1)
```

The arguments of the function consist of the list of children of the the current expression and a function that takes the list of types that have been determined for every child and produces the list of conditions associated with the current expression. The result is a list of lists of children coupled with a boolean that indicates whether this child is enabled or not. We would like to disable as few children as possible therefore we start disabling a single expression and increase the number until there is at least one permutation in the set for which unification succeeds.. All of the permutations for the current amount of disabled children are calculated after which they are paired with their types and conditions in the function *evalTypesAndConditions* (Figure 6.10).

Figure 6.10 Generating the conditions for sets of enabled children

```
evalTypesAndConditions :: ([Tp] → [(Tp, Tp)]) →
  [(AExpr Typed, Bool)] → ([Tp], [(Tp, Tp)])
evalTypesAndConditions makeC perms =
  let
    types = map getType perms
  in
    (types, makeC types)
      where
        getType (expr, True) = fst (getInfo expr)
        getType (_, False) = freshTVar ()
```

For an enabled child its previously determined type is used, while for a disabled child a fresh type variable is substituted. Subsequently the sets of enabled and disabled children are checked by the function *doUnify* (Figure 6.11) that tries to find out if they can be typed correctly.

Figure 6.11 Unifying a combination of enabled and disabled children

```
  doUnify :: ([(AExpr Typed, Bool)], ([Tp], [(Tp, Tp)])) → Bool
  doUnify (exprs, (_, cs)) =
    let
      enabledExprs = filter snd exprs
      substs = map (λe → snd . getInfo . fst $ e) enabledExprs
      unified = (unifySubstList syns substs)
    in
      case unified of
           Just superSub →
             let
               (ts1, ts2) = unzip (superSub |− > cs)
               s = (mguMaybe syns (tupleType ts1) (tupleType ts2))
             in
               case s of
                 Just _ → True
                 Nothing → False
           _ → False
```

Function *doUnify* collects the substitutions for every enabled expression and combines them using *unifySubstList*. When unification succeeds, the resulting substitution is applied to all the conditions and every lefthand side of a condition is unified with its righthand side at once by encapsulating both left and righthand sides within a tuple.

As an example consider the expression in Figure 6.4 once more. The *findConflict* function will be called once the inconsistency within the conditional expression has been detected. The arguments will be the list of correctly typed children (in this case $[(x : Bool), (1 : Int), (True : Bool)]$) and the function that given the types of the children produces the conditions that validate the type correctness of the **if** construct. In this case that function is $(\lambda(g : t : [e]) \rightarrow [(g, Bool), (t, \beta), (e, \beta)])$ where β is the expected type of the expression as a whole (in this case β is Int due to the presence of the type signature). Then all of the permutations of enabled and disabled children are calculated when a single one is to be disabled. In this case this will produce the following three possibilities.

$$[((x : Bool), False), ((1 : Int), True), ((True : Bool), True)]$$

$$[((x : Bool), True), ((1 : Int), False), ((True : Bool), True)]$$

$$[((x : Bool), True), ((1 : Int), True), ((True : Bool), False)]$$

For each of the possibilities the conditions are evaluated substituting a fresh type variable that we will label γ. For the first possibility this yields the set of conditions $[(\gamma, Bool), (Int, Int), (Bool, Int)$

Clearly we cannot unify integers and booleans so this possibility will be rejected. The same applies to the the second possibility which yields the set $[(\mathit{Bool}, \mathit{Bool}), (\gamma, \mathit{Int}), (\mathit{Bool}, \mathit{Int})]$. The third possibility however ($[(\mathit{Bool}, \mathit{Bool}), (\mathit{Int}, \mathit{Int}), (\gamma, \mathit{Int})]$) can be unified successfully and yields the substitution $\{\gamma := \mathit{Int}\}$.

6.2.3 Combination of solutions, revisited

Apart from solutions produced by the application of transformations to the current node, we now also produce a separate list of solutions that are a result of the re-evaluation of the children of the current node. Successfully combining these lists using the *merge*-function described earlier presents a problem however. Merging two lists of *Progress* values always prepends any *Fail* values found in either list before the first *Success* value to the result list. Now the possibility may arise that either of the two lists does not contain any solution. This implies an infinite list of *Fail* values in one list and thus an infinite number of *Fail* values will be prepended to the result list. For this reason we will modify the *Progress* data type to allow *Fail* values to also store solutions so that their cost can be determined by the *getCost* function. This allows comparison of the cost between a successful solution and a failed one and we can choose to allow a cheaper successful solution to precede a more expensive failed solution. The new definition for the *Progress* data type is shown in Figure 6.12.

Figure 6.12 The updated *Progress* data type

```
data Progress a = Success a | Fail a | Debug String deriving Show
```

This modification requires the merge and combine algorithms to be adjusted. For the combination of two solutions we must now also take into account the possibilities of combining a failed with a successful solution and two failed solutions. The updated combine algorithm is shown in Figure 6.13.

Merging now requires us to compare not only two successful solutions but also every other combination to maintain an ordered result list. The updated merge function can be seen in Figure 6.14.

6.2.4 Updating the repair function

Now we have all the ingredients required to update the main repair function and enable it to take advantage of the features that propagating type information make possible. The updated function is displayed in Figure 6.15.

As can be seen in the body of the *op*-function whenever an expression is found to be ill-typed, both the transformations are applied via the *transform* function and the expressions that are a result of re-evaluation of the children using function *getAltExprs* are combined using the *merge* function. Instead of the static *Fail* value that used to be prepended to the result list whenever typing of a subexpression failed a call to the *failedSolution* function

Figure 6.13 The updated combination function

```
    $doCombi :: Progress\ a \to Progress\ a \to Progress\ (a, a)$
    $doCombi\ ((Success\ a), (Success\ b)) = Success\ (a, b)$
    $doCombi\ ((Success\ a), (Fail\ b)) = Fail\ (a, b)$
    $doCombi\ ((Fail\ a), (Success\ b)) = Fail\ (a, b)$
    $doCombi\ ((Fail\ a), (Fail\ b)) = Fail\ (a, b)$
    $combi :: (a \to Int) \to [Progress\ a] \to [Progress\ a] \to [Progress\ (a, a)]$
    $combi\ f = rec$
      where
        $rec\ \_\ [\,] = [\,]$
        $rec\ [\,]\ \_ = [\,]$
        $rec\ (x : xs)\ (y : ys) =$
          let
            $as = map\ doCombi\ [(x, b) \mid b \leftarrow ys]$
            $bs = map\ doCombi\ [(a, y) \mid a \leftarrow xs]$
            $cs = (rec\ xs\ ys)$
            $g\ (a, b) = f\ a + f\ b$
          in
            $(doCombi\ (x, y)) : (merge\ g\ as\ (merge\ g\ bs\ cs))$
```

Figure 6.14 The updated merging function

```
    $merge :: (a \to Int) \to [Progress\ a] \to [Progress\ a] \to [Progress\ a]$
    $merge\ f = rec$
      where
        $rec\ [\,]\ bs = bs$
        $rec\ as\ [\,] = as$
        $rec\ ((Fail\ a) : as)\ ((Fail\ b) : bs)$
          $\mid f\ a < f\ b = (Fail\ a) : (rec\ as\ ((Fail\ b) : bs))$
          $\mid otherwise = (Fail\ b) : (rec\ ((Fail\ a) : as)\ bs)$
        $rec\ ((Success\ a) : as)\ ((Fail\ b) : bs)$
          $\mid f\ a < f\ b = (Success\ a) : (rec\ as\ ((Fail\ b) : bs))$
          $\mid otherwise = (Fail\ b) : (rec\ ((Success\ a) : as)\ bs)$
        $rec\ ((Fail\ a) : as)\ ((Success\ b) : bs)$
          $\mid f\ a < f\ b = (Fail\ a) : (rec\ as\ ((Success\ b) : bs))$
          $\mid otherwise = (Success\ b) : (rec\ ((Fail\ a) : as)\ bs)$
        $rec\ ((Success\ a) : as)\ ((Success\ b) : bs)$
          $\mid f\ a < f\ b = (Success\ a) : rec\ as\ (Success\ b : bs)$
          $\mid otherwise = (Success\ b) : rec\ (Success\ a : as)\ bs$
        $rec\ (a : as)\ (b : bs) = a : b : (rec\ as\ bs)$
```

Figure 6.15 The updated repair function

```
superRepair :: (∀ a . EmptyInfo a ⇒ [AExpr a] → AExpr a) →
               Tp → [Tp] → [AExpr (Maybe Tp)] →
               [Progress (AExpr Typed, RepairAdmin)]
superRepair make β childTypes =
  let
    op ::      Progress [(AExpr Typed, RepairAdmin)] →
               [Progress (AExpr Typed, RepairAdmin)] →
               [Progress (AExpr Typed, RepairAdmin)]
    op (Success pairs) rest =
      let
               (aexprs, admins) = unzip pairs
               node             = make (map (fmap (Just)) aexprs)
               transNodes       = transform transList node
               newAdmin         = mconcat admins
      in
               case typeAExprWithContext synonyms app β of
                 Just typedExpr →
                   Success (typedExpr, newAdmin) : rest
                 Nothing →
                   let
                        transExprs =
                          concatMap (checkFail newAdmin β) transNodes
                        altExprs   =
                          getAltExprs synonyms make transList node pairs β
                        bothExprs = merge getCost altExprs transExprs
                   in
                        (failedSolution (getRealCost newAdmin)) :
                          (merge getCost bothExprs rest)
    op (Fail pairs) rest =
      let (_, admins) = unzip pairs
               newAdmin = mconcat admins
      in
               (failedSolution (getRealCost newAdmin)) : rest
    op (Debug str) rest = (Debug str) : rest
  in foldr op [] . combiList getCost . (recAExprs childTypes)
```

is substituted that will create a solution with a dummy expression but with the actual administration data (cost and repair log) that have accumulated up to this point.

6.3 Additional enhancements

There are still a few notable scenarios where the repair system will fail to find a cheap solution while transformations exist that would be able to correct them. The problem is that in these scenarios the transformations are not applied to the correct node. The following scenarios demonstrate where the repair algorithm in it current state will fail to find a cheap solution while it is obvious one exists:

- Because we only allow correctly typed subexpressions when building up our solution list, a transformation that is capable of transforming itself as well as its children might potentially produce a well-typed expression even though one of the children by itself can not be typed correctly. As an example consider the following expression:

  $$f\ (x\ y)$$

  Assume that f has type $a \rightarrow a \rightarrow a$ and x and y have type Int. Clearly the application $x\ y$ is in error, but by allowing it into the list of possible solutions for this subexpression, we pave the way for the successful use of the $iswap$ transformation at the toplevel application:

  $$iswap\, f\ (x\ y) = (f\ x)\ y = f\ x\ y$$

- We rule out a number of solutions by restricting re-evaluation of subexpressions to those produced by directly by combining the different alternatives of the children. We do not currently allow re-evaluation and transformation of subexpressions to be combined at the same level in the abstract syntax tree. As an example take a look at the following expression:

  $$g\ x\ y$$

  Function g has type $Int \rightarrow [Bool] \rightarrow Bool$, x has type $Bool$ and y has type Int. The only cheap solutions the repair system in its current state will find are changing the type of g or x and y, deleting and inserting arguments and combinations of those. Allowing re-evaluation of the transformed expression $g\ y\ x$ which can be produced for little cost (a single $permute$), results in the additional solution $g\ y\ [x]$ in which the first argument was selected for re-evaluation and transformed by the $listify$ transformation.

To see what can be done about these limitations we will introduce some additional modifications to the repair system that will be described in the following sections.

6.3.1 Allowing incorrectly typed subexpressions

We modify the original repair function to include the original expression in the solution list even if typing has failed, in the hope that transformations applied to parent nodes will also fix the local inconsistency. Because including a solution into the result list requires the type of that solution to be specified we will use a special type named *errorType*. Additionally the repair log will be updated to indicate the status of this solution. Whenever the result of a transformation types successfully, this status indication will be removed.

6.3.2 Allowing re-evaluation of transformed expressions

This enhancement presents a problem. Since the list of transformed expressions is infinite, if we were to re-evaluate every transformed expression we would end up with an infinite amount of lists. While all of the lists of re-evaluated transformed expressions are ordered by cost, their concatenation is not. Therefore we would have to merge these lists, but because you cannot be certain of the position of an element in the result list until you have merged all of the input lists this would take infinite time. Because of this we have to pose a restriction on the number of transformed expressions that we will allow to be re-evaluated.

6.3.3 Integration of enhancements

The *repairAExpr* function will be modified in two places to realize the enhancements discussed in the earlier sections. Since both of the enhancements deal exclusively with the case where typing of an original expression has failed, only this fragment is displayed in Figure 6.16.

Figure 6.16 Enhanced handling of untypable expressions

```
let
  transExprs = (transform transList node)
  transExprsAlt =
    foldr (merge getCost) []
      (map
        (checkFail synonyms transList newAdmin τ True maxAltTrans)
        (take maxAltTrans transExprs))
  transExprsRest =
    concatMap
      (checkFail synonyms transList newAdmin τ False maxAltTrans)
      (drop maxAltTrans transExprs)
  allTrans = merge getCost transExprsAlt transExprsRest
  altExprs =
    getAltExprs synonyms make transList origNode pairs τ maxAltTrans
  allExprs = merge getCost allTrans altExprs
in
  (failedSolution (getRealCost newAdmin)) :
    (Success (errorNode,
      mappend newAdmin (RepairAdmin (["incorrect node"], 0)))) :
    (merge getCost allExprs rest)
```

The list of transformed expressions is split in two at the position determined by the value of *maxAltTrans* which has become an additional parameter of the *repairAExpr* function.

Both sides are still processed by the $checkFail$ function which has also received an additional parameter of type $Bool$ to indicate whether re-evaluation is allowed. The results of re-evaluated transformed expressions are merged while the others are simply concatenated. These two lists are then merged again. As can been seen in the return expression, the result list now contains a successful solution that is incorrectly typed to allow transformations to its parent nodes to correct it.

Chapter 7

Results

In this chapter we will investigate how the repair algorithm performs when it is applied to several selected input expressions to verify whether the behaviour of the repair system does not produce unexpected results. Additionally we will test the repair system on samples of a collection of ill-typed programs that were produced by students during the functional programming course at the University of Utrecht. We will compare the results of the repair system with the solutions that a human corrector would produce to give insight into the practical applicability of such systems. Finally we will analyze what the relations are between the size of the input expression, the parameters of the repair system and the efficiency of the system.

7.1 Application of the repair system to selected expressions

In this section we will analyze the results of executing the repair algorithm using a set of input expressions that have purposely been set up to contain a specific instance of a type-error that can be repaired by a relatively simple transformation. This allows us to verify that the repair system does not generate any incorrect solutions. Whether or not the results actually match the expected results will likely depend on the cost values that have been assigned to each of the transformations. By adjusting the cost values in such a way as to maximize the agreement between expected and actual results in the largest possible number of cases, we can build a template for transformation costs that will hopefully be equally successful when the repair system is applied to real world samples in the next section. The initial cost distribution of the transformations will be as follows:

- Isomorphic transformations: transformations that do not require the introduction of new subexpressions or elimination of existing subexpressions should be preferred over ones that do. Initial cost of isomorphic transformations will be 3.
- Non-isomorphic transformations: to discourage the widespread use of transformations that will alter the structure of the original expressions their cost has been set to 4.
- Changing the type of a variable or block should only be tried as a last ditch effort. Therefore the cost has been set to 6.

This distribution is not more than an educated guess and will likely need to be adjusted to perform optimally.

7.1.1 Permutation of arguments

Function f_perm shown below has been given the type $Int \rightarrow Char \rightarrow Float \rightarrow Bool$ and arguments a, b and c have been given types $Char$, Int and $Float$ respectively. The order of the first two arguments has been reversed. We would expect the repair system to be able to correct this in a single permutation transformation and produce the result $f\_perm\ b\ a\ c$.

$$f\_perm\ a\ b\ c$$

The output of the repair algorithm is shown in Figure 7.1. The results are ordered by repair cost, cheapest solutions first. Only successful solutions are displayed.

Figure 7.1 The result of the repair algorithm when applied to f_perm

```
results:
(Success (f_perm b a c)
    repairinfo: (["permute"],3)
(Success (f_perm b a c)
    repairinfo: (["permute","permute"],6))
(Success (f_perm a b c)
    repairinfo: (["Type of variable: f was changed"],6))
(Success (f_perm {Int} a c)
    repairinfo: (["insertArgument","deleteArgument"],8))
(Success (f_perm {Int} a c)
    repairinfo: (["deleteArgument","insertArgument"],8))
(Success (f_perm b a c)
    repairinfo: (["permute","permute","permute"],9))
```

We can see that the cheapest solution was indeed permuting the arguments once at cost = 3. Looking further in the list of results we can see that performing the same permutation twice also yielded the correct result as did inserting a new argument with type Int followed by deleting b. The same result could also be obtained doing the reverse, first deleting b followed by insertion of an argument. Finally we observe that the error could also be eliminated by changing the type of the function f_perm.

7.1.2 Currying

To test the ability of the repair system to successfully apply currying and uncurrying transformations we use as input the following expression:

$$(f\_curry1\ (a, b))$$

The variable f_curry1 has been assigned the type $Int \rightarrow Int \rightarrow Int$ and both a and b are of type Int. A single application of the $curryTuple$ transformation should eliminate this inconsistency which coincides what the repair system produces which is shown in Figure 7.2.

Figure 7.2 The result of the repair algorithm when applied to *f_curry1*

```
results:
(Success (f_curry1 a b) | cost=3 |
    repairlog: ["curryTuple"]
(Success (f_curry1 (a,b)) | cost=6 |
    repairlog: ["Type of variable: 'f_curry1' was changed"]
(Success (f_curry1 a b) | cost=6 |
    repairlog: ["curryTuple","curryTuple"]
(Success (f_curry1 b a) | cost=6 |
    repairlog: ["curryTuple","permute"]
(Success (f_curry1 a b) | cost=6 |
    repairlog: ["curryTuple","permute"]
```

To test the inverse transformation we use the following expression as input.

$$(f\_curry2\ a\ b)$$

The variable f_curry2 has type $(Int, Int) \rightarrow Int$ and both a and b have type Int. A single application of the transformation *uncurryTuple* should eliminate the type inconsistency. Output of the repair system which is shown in Figure 7.3 produces the expected result.

Figure 7.3 The result of the repair algorithm when applied to *f_curry2*

```
results:
(Success (f_curry2 (a,b))    | cost=3 |
    repairlog: ["uncurryTuple"]
(Success (f_curry2 a b)      | cost=6 |
    repairlog: ["Type of variable: 'f_curry1' was changed"]
(Success (f_curry2 (a,b))    | cost=6 |
    repairlog: ["uncurryTuple","curryTuple"]
(Success (f_curry2 (a,b))    | cost=6 |
    repairlog: ["uncurryTuple","permute"]
(Success (f_curry2 (b,a))    | cost=6 |
    repairlog: ["permute","uncurryTuple"]
```

7.1.3 Parenthesis

To verify whether the repair system will correct expressions with misplaced parenthesis, we run the algorithm on the following input expression.

$$(f\_swap\ a\ b)$$

Here, f_swap and a have type $Int \rightarrow Int$ and b has type Int. The $swap$ transformation is capable of repairing this expression (Figure 7.4.
For the reverse transformation we use the following input expression.

Figure 7.4 The result of the repair algorithm when applied to f_swap

```
results:
(Success (f_swap (a b)) | cost=3 |
    repairlog: ["swap"]
(Success (f_swap b) | cost=4 |
    repairlog: ["deleteArgument"]
(Success (f_swap a b) | cost=6 |
    repairlog: ["Type of variable: 'f_swap' was changed"]
(Success (f_swap (a b)) | cost=6 |
    repairlog: ["swap","iswap"]
(Success (f_swap (a b)) | cost=6 |
    repairlog: ["swap","permute"]
```

$$(f\_iswap\ (a\ b))$$

The types of the variables are the following. The type of f_iswap is $Int \rightarrow Int \rightarrow Int$ and the types of a and b are both Int. The results show a single application of $iswap$ is the most inexpensive solution in Figure 7.5.

Figure 7.5 The result of the repair algorithm when applied to f_swap

```
(Success (f_iswap a b) | cost=3 |
    repairlog: ["iswap"]
(Success (f_iswap a b) | cost=6 |
    repairlog: ["iswap","iswap"]
(Success (f_iswap b a) | cost=6 |
    repairlog: ["iswap","permute"]
(Success (f_iswap a b) | cost=6 |
    repairlog: ["iswap","permute"]
(Success (f_iswap a b) | cost=6 |
    repairlog: ["permute","iswap"]
```

7.1.4 Combination of transformations

To test the ability of the repair system to produce solutions composed of multiple transformations we will use the following expression.

$$f\ x\ y$$

The type of function f is $[Int] \to Bool \to Bool$ while the types of x and y are $Bool$ and Int respectively. The solution which we hope to find for this expression should reverse the order of the arguments of the function and re-evaluate y in order for the the *listify* transformation to be applied. The output of the system is shown in Figure 7.6.

Figure 7.6 The result of the repair algorithm when applied to f_reval

```
(Success (f x y) | cost=7 |
        repairlog: ["re-evaluation","Type of identifier: 'f' was changed"]
(Success (f [y] x) | cost=7 |
        repairlog: ["permute","re-evaluation","listify"]
(Success (f {v10000306} x) | cost=8 |
        repairlog: ["deleteArgument","insertArgument"]
(Success (f {v10000288} x) | cost=8 |
        repairlog: ["insertArgument","deleteArgument"]
(Success (f ({v10002056 -> v10002054} y) x) | cost=10 |
        repairlog: ["permute","insertArgument","flattenApp"]
```

While the solution that we expected was found, simply changing the type of identifier 'f' proved to be a solution of equal cost, indicating that in this case the chosen parametrization is not optimal.

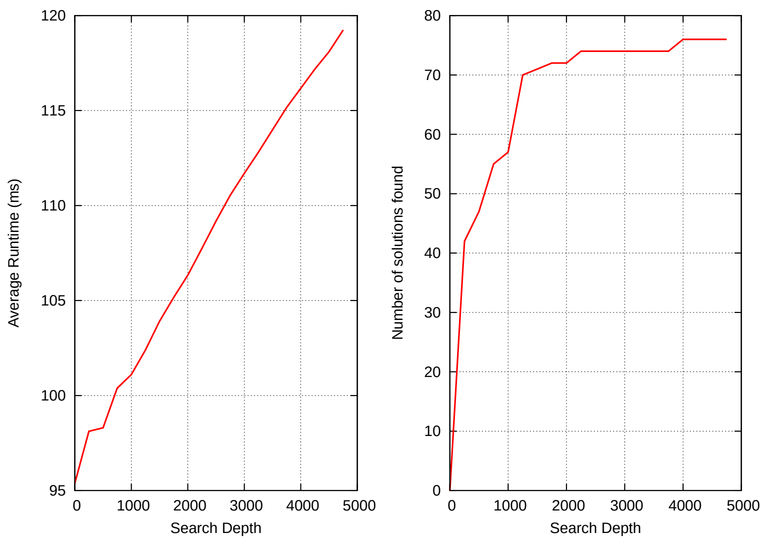
7.2 Application to real world programs

In this section we will apply the repair system to a number of ill-typed programs that were produced by students during the course of their functional programming class at the University of Utrecht. The repair system will be run with different parameters in order to judge the effectiveness of different parts of the algorithm. First we will investigate the ability of the system to produce a solution at various search lengths, taking into account the running time. Secondly the system will be run with and without re-evaluation enabled to measure the effectiveness of this feature. After this, in a selected number of cases where the repair system failed to produce a solution, we will analyze the reason for this failure. Finally in cases where the system did produce a solution we will try to compare this to the correction a human would produce (this is subjective, and not always possible).

7.2.1 Number of solutions

To determine the optimal parameters the repair algorithm will be executed a large number of times on a diverse set of input expressions. The relation between search length and the number of solutions is most important. It can be expected that at some point trying more and more transformations will not produce a significant amount of additional solutions unless the amount of nodes in the expression is large. For every run, the running time is recorded. For every series of runs at a specific search length the number of solutions is recorded. Running times for a series of runs at a particular length are averaged to produce the average running time in the leftmost graphic of Figure 7.7. The rightmost graphic displays the number of solutions that were found out of a maximum of 100 solutions.

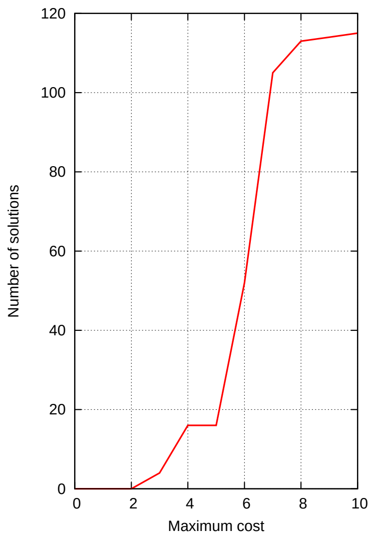
Figure 7.7 Effects of search length on running time (left) and number of solutions found (right)



Running time increases roughly linear with search length. The number of solutions increases rapidly at first, then quickly stabilizes around lengths of 2000. The expressions that were repaired at lengths greater than 2000 were for the most part those with a large number of nodes. This is hardly surprising, since a large amount of nodes in the input expression means a greater chance of multiple inconsistencies. Multiple inconsistencies cause a decrease in the number of intermediate solutions per node and this in turn causes a decrease in the maximum complexity of a partial solution that repairs a single inconsistency. In other words, at equal search lengths, the chance of finding a solution is partly determined by the number of inconsistencies in the input expression.

Figure 7.8 shows the relation between maximum cost and the number of solutions found out of a maximum of 150 solutions. The sharp drop-off in number of solutions found at around a cost of 8 suggests that, regardless of search length, increasing the maximum complexity of solutions at some point does not equal increasing the likelihood of finding additional solutions.

Figure 7.8 Relation between maximum cost and the number of solutions



7.2.2 Effectiveness of re-evaluation

To analyze the effects of re-evaluation on the number and cost of solutions we execute the algorithm both with and without this feature and compare the results. Table 7.1 shows the differences.

	without re-evaluations	with re-evaluations
Number of input expressions	150	150
Number of solutions	119	114
Average cost of solutions	6.4	6.0

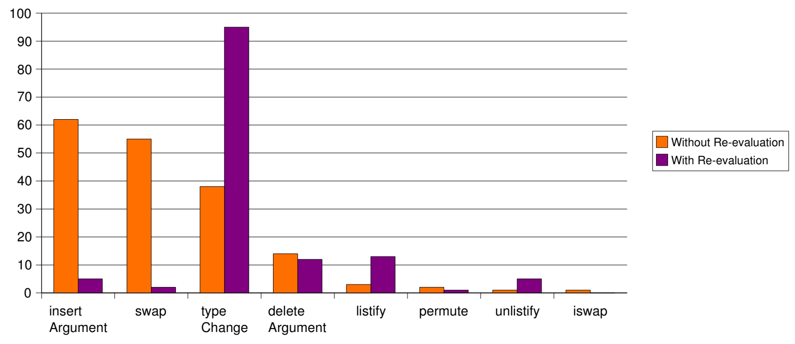
Table 7.1: Effects of re-evaluations on the number and cost of solutions

We can see that, even though the number of solutions is less than without re-evaluation enabled (a decrease of about 4.2%), the average cost of the solutions is about 6% lower. In 15% of the cases where both methods came up with a solution re-evaluation produced a cheaper one. Re-evaluation means that the same amount of transformation combinations leads to more intermediate solutions, therefore if the total number of solutions is kept fixed, re-evaluation causes a decrease in the overall number of transformation combinations that are evaluated. Thus to maintain the same chance of finding a solution for a particular expression with re-evaluation enabled, the search length must be increased.

It can be seen in Figure 7.9 that enabling re-evaluation also has a drastic effect on which transformations are used to produce a solution.

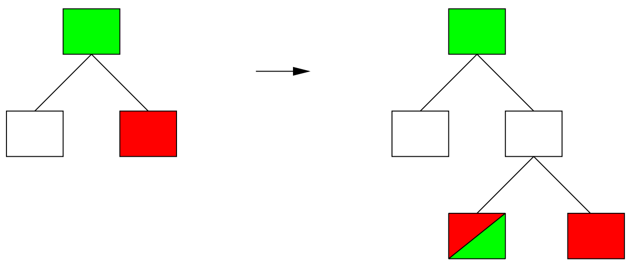
The most visible aspect of this change is the rather large decrease in the usage of both the $insertArgument$ and $swap$ transformations and the increase in the number of type changes to identifiers. Closer inspection of this discrepancy reveals that the combination

Figure 7.9 Transformations used, with and without re-evaluation enabled



of the *insertArgument* and *swap* transformations is providing a sort of catch-all solution to a large amount of inconsistencies. The type of the argument that is inserted can be of any type. The *swap* transformation turns this argument into a function that consequently can also have any type. Figure 7.10 illustrates the effect. The combination can repair any inconsistency (signified by the difference in color between the nodes) in any argument of an application, by inserting an application with a function that converts between the type of the argument and the type that the application expects this argument to have.

Figure 7.10 The *insertArgument* and *swap* transformation combination



When re-evaluation is enabled, this combination is replaced by simply changing the type of the function since with our chosen parametrization of the costs this results in a cheaper solution.

7.3 Analysis of the quality of solutions

In this section we will, for expressions where the repair system failed to produce a solution, examine the reasons, and for expressions where the system did produce a solution, compare this solution with a correction a human would make.

7.3.1 Failed repair

In this section we will analyze in detail some of the incorrectly typed programs from the collection that the system failed to repair in order to better understand the limitations of the system. The program fragment in Figure 7.11 was a submitted by a student during the course of an assignment.

Figure 7.11 Type incorrect submission from an assignment

$$\begin{array}{l} \mathit{discriminant} :: \mathit{Float} \to \mathit{Float} \to \mathit{Float} \to \mathit{Float} \\ \mathit{discriminant}\ a\ b\ c = b \ast .\ b -.\ 4.0 \ast .\ a \ast .\ c \\ \mathit{aantalOpl} :: \mathit{Float} \to \mathit{Float} \to \mathit{Float} \to \mathit{Int} \\ \mathit{aantalOpl}\ a\ b\ c = (d\ \mathit{div}\ \mathit{abs}\ (\mathit{negate}\ (d))) + 1.0 \\ \quad \textbf{where}\ d = \mathit{discriminant}\ a\ b\ c \end{array}$$

The objective was to write a function that calculates the number of real solutions to a quadratic equation of the form $ax^2 + bx + c = 0$. The compiler determines that the expression with the highest probability of being incorrect is: $(d\ \mathit{div}\ \mathit{abs}\ (\mathit{negate}\ (d))) + 1.0$ in which d has type Float and div, abs and negate are variables representing prelude functions having been instantiated with the types $\mathit{Int} \to \mathit{Int} \to \mathit{Int}$, $\mathit{Int} \to \mathit{Int}$ and $\mathit{Int} \to \mathit{Int}$ respectively. The expected return type of the expression is Int as specified by the type signature. There are a number of things wrong here. The expression d does not have a function type yet it is being used as one with two arguments. It appears the quote-characters around the div expression were forgotten. Furthermore div is a function that only accepts integers yet its intended arguments are floats.

Apart from the fact that the intended function definition is not quite correct (division by zero is possible) it would seem that transforming it into a type-correct expression would not be all that difficult. However even using exceptionally high values for the search length the system still fails to find a solution. The problem is related to the explosion of possibilities described in chapter 5. An inexpensive (although drastic) way to eliminate the type inconsistency with the transformations at our disposal would be to delete the entire subexpression $(d\ \mathit{div}\ \mathit{abs}\ (\mathit{negate}\ (d)))$, and change the type of the $+$-function. The cost of this would be 4 for the deletion, 1 for the re-evaluation of the function type of the toplevel application and 6 to change it to $\mathit{Float} \to \mathit{Int}$ totalling a cost of 11. It is the same problem as we have seen in the previous section in our analysis of the re-evaluation enhancements. The increase of the number of lists of re-evaluated transformed expressions limits the complexity of transformation compositions for fixed search lengths. In this program type inconsistencies exist at multiple nodes in the syntax tree, the algorithm will start generating lists of solutions for every one of those nodes likewise limiting the number of compositions that can be examined. In this example a cost of 11 implies a number of combinations approaching the hundreds of thousands.

7.3.2 Successful repair

In this assignment the task was to write a function to flatten a list of lists of integers into a single list. Figure 7.12 shows one attempt at solving the problem.

Figure 7.12 Flattening a list, incorrectly

$$\begin{aligned}
&\mathit{slaPlat} :: [[\mathit{Int}]] \to [\mathit{Int}]\\
&\mathit{slaPlat}\ [\,] = [\,]\\
&\mathit{slaPlat}\ [[a]] \qquad = [a]\\
&\mathit{slaPlat}\ (a : \mathit{rest}) = [a] +\!\!+\ \mathit{slaPlat}\ (\mathit{rest})
\end{aligned}$$

The mistake has been to write $[a]$ instead of a in the third case thereby producing an expression of type $[[\mathit{Int}]]$ which obviously cannot be concatenated to a list of type $[\mathit{Int}]$. The subexpression selected by the compiler is thus $[a] +\!\!+\ \mathit{slaPlat}\ (\mathit{rest})$. The transformation to fix this problem would be to remove the brackets around a. Figure 7.13 shows the output of the repair system.

Figure 7.13 Solutions for the ill typed flattening expression

```
results:
(Success (++ a (slaPlat rest)) | cost=4 |
        repairlog: ["re-evaluation","unlistify"]
(Success (++ [a] (slaPlat rest)) | cost=7 |
        repairlog: ["re-evaluation",
                "Type of variable: '++' was changed"]
(Success (++ [a] (slaPlat rest)) | cost=7 |
        repairlog: ["re-evaluation",
                "Type of variable: 'a' was changed"]
(Success (++ ({v10022240 -> v10022238} [a]) (slaPlat rest)) |
        cost=7 | repairlog: ["insertArgument","swap"]
(Success (++ (slaPlat rest) {v10022438}) | cost=8 |
        repairlog: ["deleteArgument","insertArgument"]
```

In this case the most inexpensive solution is in accordance with our own solution. The example in Figure 7.14 demonstrates that this is not always the case.

Figure 7.14 Converting a table to a string

$$\begin{aligned}
&\mathit{writeTable} :: \mathit{Table} \to \mathit{String}\\
&\mathit{writeTable}\ \mathit{table} = \mathit{unlines}\ .\ \mathit{head}\ \mathit{table}
\end{aligned}$$

Here the student has forgotten to include parenthesis around the function composition $\mathit{unlines}\ .\ \mathit{head}$. However the pattern for this mistake doesn't quite match the one defined in the swap and iswap functions. The solution produced instead removes the table argument altogether and inserts a new block of suitable type.

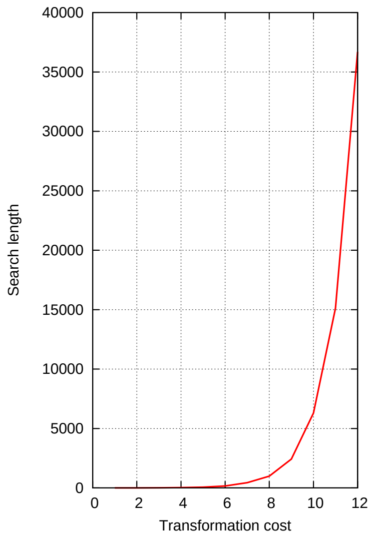
Another problem is that there is no measure of the degree to which the structure of the input expression is altered. Large subexpressions can be deleted at relatively low cost. Especially if the return type of an expression is not explicitly specified by the context, deleting arguments turns out to be a popular way of getting rid of typing problems. To discourage this behaviour we could make the cost of transformations that can potentially make large changes in the structure of an expression dependent upon the size of the subexpressions that they work on.

A related problem is that even though the cost of changing the types of variables has been initialized with a relatively high value, its power to singlehandedly eliminate type problems is compensating. For example when a type conflict has been detected in an application with several arguments, a single change in the type of the function variable can have a large impact, it can change the expected type of multiple arguments at once. To remedy this the cost of changing the type of a variable or block would also have to be made dependent on their impact. This is hard to realize however. Apart from prelude functions which should be considered unalterable, a seperate analysis would have to be made to analyze the repercussions of changing a certain variable type with respect to the rest of the program.

7.4 Complexity analysis

The algorithm has been designed to evaluate a fixed number of solutions. While the time spent on evaluating a particular solution can vary itself depending on size of the expression, number and properties of the transformations that are being applied and number and nature of the inconsistencies in the expression, the overall time consumed is still governed by the amount of solutions that have to be evaluated. Therefore the most interesting question is, how many solutions do we need to calculate? This in turn depends on the complexity (cost) of the best solution to a particular inconsistent expression. We have already witnessed in the previous chapter that for search lengths greater then 2000, there is little chance of finding additional solutions. This is equal to a cost of about 7. In Figure 7.15 the relation between transformation cost and search length is displayed for the chosen set of transformations and their cost parametrization. It confirms that finding solutions which require a cost larger then 10 simply becomes unfeasible.

Figure 7.15 Relation between search length and transformation cost



Chapter 8

Conclusions

Using the top-down type-propagation and bottom-up generation techniques for building up a solution space is an flexible alternative to checking for the existence of fixed patterns in erroneous code. While static pattern matching does provide a higher degree of certainty when it comes to finding the appropriate fix, it will always be limited by the number of patterns that are available to it. While the same could be said about the limited number of transformations that the repair system has at its disposal, its strength lies in the flexible way in which they can be combined.

Even when the repair system has found a single most inexpensive solution to a particular fragment of ill typed code, we have very little certainty that this solution coincides with the fix that would correct the program in the way the programmer intended it. Extending the system to include more heuristic knowledge about often made errors can help but it will grow more difficult to conclude anything definite as the input becomes more complex.

The explosion of the search space is very difficult to control. While type-propagation limits the number of possible intermediate solutions by a large amount, the actual number of computations performed is still very much dependent on the various parameters. The search length plays the largest role but the number of transformations, the amount of transformed expressions that are to be considered for re-evaluation, and number of inconsistencies in the input expression are also important factors.

Chapter 9

Future work

Better heuristics may be a means of both increasing the likelihood that the 'right' solution will be found as well as narrow down the solution space further, perhaps by ruling out certain combinations of transformations, or encouraging the use of specific combinations of transformations that have been found to repair a problem in a large number of instances. To realize this it will be required to study many type errors that are produced in real world settings in order to find the patterns that make such heuristics possible. It remains to be seen how effective such a method will be.

Research into how to best display possible solutions might be of value. Merely outputting the fixed code might encourage students to rely on the system too much and copy and paste the code without actually thinking about the source of the error themselves. This is contrary to what a compiler that is focused on educating its users is supposed to be achieving.

Bibliography

[1] Luis Damas and Robin Milner. Principal type-schemes for functional programs. *Ninth Annual Symposium on Principles of Programming Languages, Association of Computing Machinery*, pages 207–212, 1982.

[2] Bastiaan Heeren and Jurriaan Hage. Parametric type inferencing for helium. Technical Report UU-CS-2002-035, Institute of Information and Computing Science, University Utrecht, Netherlands, August 2002. Technical Report.

[3] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York, 2003. ACM Press.

[4] J. R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, pages 146:29–60, 1969.

[5] 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.

[6] O. Lee and K. Yi. Ordering type constraints: A structured approach. Technical report, Korea Advanced Institute of Science and Technology, 1997. Technical Report.

[7] B.J. McAdam. Repairing type errors in functional programs. Technical report, PhD Thesis, University of Edinburgh, Laboratory for Foundations of Computer Science, Division of Informatics, August 2001.

[8] Mikael Rittri. Finding the source of type errors interactively. *Proc. El Wintermote, Department of Computer Science, Chalmers University*, 1993.

[9] Axel Simon, Olaf Chitil, and Frank Huch. Typeview: A Tool for Understanding Type Errors. In Markus Mohnen and Pieter Koopman, editors, *Draft Proceedings of the 12th International Workshop on Implementation of Functional Languages*, pages 63–69, Aachen, Germany, September 2000. Aachener Informatik-Bericht 00-7, RWTH Aachen.

[10] Peter J Stuckey, Martin Sulzmann, and Jeremy Wazny. The chameleon type debugger (tool demonstration), 2003.

[11] 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.

[12] Mitchell Wand. Finding the source of type errors. *13th Symposium Principles of Programming Languages, SIGPLAN,ACM Press*, pages 38–43, 1986.

[13] Jun Yang. *Improving Polymorphic Type Explanations*. PhD thesis, Heriot-Watt University, Edinburgh, 2001.

Appendix A

Code

A.1 Data type definitions and their operations

```
module Top.Repair.AExpr where

import Top.Types
import Top.Repair.Progress
import Data.List (intersperse)
import Data.Monoid

newtype RepairAdmin = RepairAdmin ([String], Int) deriving (Show, Eq)

instance Monoid RepairAdmin where
    mempty = RepairAdmin ([ ], 0)
    mappend (RepairAdmin (xs1, i1)) (RepairAdmin (xs2, i2)) =
      RepairAdmin (xs1 ++ xs2, i1 + i2)

instance Ord RepairAdmin where
    compare (RepairAdmin (xs1, i1)) (RepairAdmin (xs2, i2)) =
      compare (i1, length xs1) (i2, length xs2)

class EmptyInfo info where
  emptyInfo :: info

class EmptyInfo info ⇒ RepairInfo info where
  getType :: info → (Maybe Tp)
  makeInfo :: Tp → info

instance EmptyInfo (Maybe a) where
  emptyInfo = Nothing

instance EmptyInfo (Typed) where
  emptyInfo = (errorType, emptySubst)

type TransformProgress info = AExpr info → [Progress (AExpr info)]
type Transformation info    = (TransformProgress info, RepairAdmin)
type Transformations info   = [Transformation info]

data AExpr info =
     App info (AExpr info) [AExpr info]
   | If info (AExpr info) (AExpr info) (AExpr info)
   | Tup info [AExpr info]
   | Lst info [AExpr info]
   | Var info String
   | Blk info
  deriving (Eq, Show)

getInfo :: AExpr info → info
getInfo (App info _ _) = info
getInfo (If info _ _ _) = info
getInfo (Lst info _) = info
getInfo (Tup info _) = info
getInfo (Var info _) = info
getInfo (Blk info) = info

getInfos :: AExpr info → [info]
getInfos aexpr =
    getInfo aexpr : concatMap getInfos (subexpressions aexpr)

editInfo :: AExpr info → (info → info) → AExpr info
editInfo aexpr f =
      case aexpr of
        App a fun args → App (f a) fun args
        If a e1 e2 e3  → If (f a) e1 e2 e3
        Tup a elts     → Tup (f a) elts
        Lst a elts     → Lst (f a) elts
        Var a s        → Var (f a) s
        Blk a          → Blk (f a)

instance Functor AExpr where
    fmap f aexpr =
      case aexpr of
        App a fun args → App (f a) (fmap f fun) (map (fmap f) args)
        If a e1 e2 e3  → If (f a) (fmap f e1) (fmap f e2) (fmap f e3)
```

```
          Tup a elts      → Tup (f a) (map (fmap f) elts)
          Lst a elts      → Lst (f a) (map (fmap f) elts)
          Var a s         → Var (f a) s
          Blk a           → Blk (f a)

mapAExpr :: (a → b) → (a → b) → AExpr a → AExpr b
mapAExpr nodeF leafF = rec
  where
    rec aexpr =
      case aexpr of
        App a fun args → App (nodeF a) (rec fun) (map rec args)
        If a e1 e2 e3  → If (nodeF a) (rec e1) (rec e2) (rec e3)
        Tup a elts     → Tup (nodeF a) (map rec elts)
        Lst a elts     → Lst (nodeF a) (map rec elts)
        Var a s        → Var (leafF a) s
        Blk a          → Blk (leafF a)

mapAExprM :: Monad m ⇒ (a → m b) → AExpr a → m (AExpr b)
mapAExprM f aexpr =
    case aexpr of
      App a fun args →
        do b    ← f a
           fun'  ← mapAExprM f fun
           args' ← mapM (mapAExprM f) args
           return (App b fun' args')
      If a e1 e2 e3 →
        do b ← f a
           e1' ← mapAExprM f e1
           e2' ← mapAExprM f e2
           e3' ← mapAExprM f e3
           return (If b e1' e2' e3')
      Lst a elts →
        do b    ← f a
           elts' ← mapM (mapAExprM f) elts
           return (Lst b elts')
      Tup a elts →
        do b    ← f a
           elts' ← mapM (mapAExprM f) elts
           return (Tup b elts')
      Var a s →
        do b ← f a
           return (Var b s)
      Blk a →
        do b ← f a
           return (Blk b)

subexpressions :: AExpr a → [AExpr a]
subexpressions aexpr =
    case aexpr of
      App _ fun args → fun : args
      If _ e1 e2 e3    → [e1, e2, e3]
      Lst _ elts       → elts
      Tup _ elts       → elts
      Var _ _          → []
      Blk _            → []

isBlock :: AExpr a → Bool
isBlock (Blk _) = True
isBlock _       = False

printType :: Tp → String
printType = show

isTuple :: AExpr info → Bool
isTuple (Tup _ _) = True
isTuple _ = False

isApp :: AExpr info → Bool
isApp (App _ _ _) = True
isApp _ = False

type Typed = (Tp, MapSubstitution)
type TypedList = (Tps, [MapSubstitution])

failedSolution :: Int → Progress (AExpr Typed, RepairAdmin)
failedSolution cost =
  addAdmin (admin cost "failed") (Fail (Var (boolType, emptySubst) "fail"))

failedSol :: EmptyInfo info ⇒ AExpr info
failedSol = (Var emptyInfo "fail")

admin :: Int → String → RepairAdmin
admin i s = RepairAdmin ([s], i)

addAdmin :: RepairAdmin → Progress (AExpr info) → Progress (AExpr info, RepairAdmin)
addAdmin admin (Success s) = Success (s, admin)
addAdmin admin (Fail s) = Fail (s, admin)
```

A.2 Implementation of the transformations

```
module Top.Repair.Transformations where

import List (inits, tails)
import Top.Types
import Top.Repair.AExpr

permute :: EmptyInfo info ⇒ Transform info
permute (App _ fun args) = [App emptyInfo fun args′ | args′ ← perms args]
permute _ = [ ]

listify :: EmptyInfo info ⇒ Transform info
listify aexp = [Lst emptyInfo [aexp]]

unlistify :: EmptyInfo info ⇒ Transform info
unlistify (Lst _ [lelem]) = [lelem]
unlistify _ = [ ]

insertArgument :: EmptyInfo info ⇒ Transform info
insertArgument (App _ fun args) =
  let
    argsplit = zip (inits args) (tails args)
    newarg = Blk emptyInfo
    newarglists = case args of [ ] → [[newarg]]; _ → map (λ(i, t) → (i ++ [newarg] ++ t)) argsplit
  in
    [App emptyInfo fun args′ | args′ ← newarglists]
insertArgument _ = [ ]

deleteArgument :: EmptyInfo info ⇒ Transform info
deleteArgument (App info fun args) =
  let
    delElem as i = (take (i − 1) as) ++ (drop i as)
    newarglists = map (delElem args) (enumFromTo 1 (length args))
  in
    [App info fun args′ | args′ ← newarglists]
deleteArgument _ = [ ]

flattenApp :: EmptyInfo info ⇒ Transform info
flattenApp (App info fun args) =
  let
    buildapp (l, (ain, aout)) = App emptyInfo fun (l ++ [App info (head ain) (tail ain)] ++ aout)
    split2 = splitTwice 1 args
  in
    map buildapp split2
flattenApp _ = [ ]

flattenAppRev :: EmptyInfo info ⇒ Transform info
flattenAppRev (App _ fun args) =
  let
    flattenRev (a : as) rlists =
            let newrlists =
              case a of
                app@(App _ fn arg) → (map (++[app]) rlists) ++ (map (++[fn] ++ arg) rlists)
                noapp → map (++[noapp]) rlists
            in
              flattenRev as newrlists
    flattenRev [ ] rlists = rlists
  in
    [App emptyInfo fun args′ | args′ ← flattenRev args [[ ]], or (map isApp args)]
flattenAppRev _ = [ ]

permuteTuple :: EmptyInfo info ⇒ Transform info
permuteTuple (Tup info telems) = [Tup info telems′ | telems′ ← perms telems]
permuteTuple _ = [ ]

curryTuple :: EmptyInfo info ⇒ Transform info
curryTuple (App info fun args) =
  let
    currySingle (Tup _ telems) = telems
    currySingle notup = [notup]
  in
    [App info fun args′ | args′ ← (optmap currySingle args [[ ]]), or (map isTuple args)]
curryTuple _ = [ ]

uncurryTuple :: EmptyInfo info ⇒ Transform info
uncurryTuple (App info fun args) =
  let
    buildapp (l, (ain, aout)) = App info fun (l ++ [Tup emptyInfo ain] ++ aout)
    split2 = splitTwice 0 args
  in
    map buildapp split2
uncurryTuple _ = [ ]

splitTwice :: Int → [a] → [([a], ([a], [a]))]
splitTwice ml elems =
  let
    minlength = filter ((>ml) . length . fst . snd)
    splt as = zip (inits as) (tails as)
  in
    concatMap (minlength . (λ(l, r) → (map (λrs → (l, rs)) (splt r)))) (splt elems)

optmap :: (a → [a]) → [a] → [[a]] → [[a]]
optmap f (e : es) fes = optmap f es ((map (++(f e)) fes) ++ (map (++[e]) fes))
optmap _ _ fes = fes

perms :: [a] → [[a]]
perms [ ] = [[ ]]
perms (a : x) = [z | y ← perms x, z ← insertions a y]

insertions :: a → [a] → [[a]]
insertions a [ ] = [[a]]
insertions a x@(b : y) = (a : x) : [b : z | z ← insertions a y]
```

A.3 Implementation of the combination algorithm

```
module Top.Repair.Progress where

data Progress a = Success a | Fail a | Debug String deriving Show

combiList :: (a → Int) → [[Progress a]] → [Progress [a]]
combiList f p = foldr op (map toList (last p)) (init p)
  where
    op as ass = conv (combi g (map toList as) ass)
    g = sum . map f

toList :: Progress a → Progress [a]
toList (Success s) = Success [s]
toList (Fail s) = Fail [s]
toList (Debug str) = Debug str

conv :: [Progress ([a], [a])] → [Progress [a]]
conv =
  let
    c p = case p of
      Success s → Success (uncurry (++) s)
      Fail s → Fail (uncurry (++) s)
      Debug str → Debug str
  in
    map c

combi :: (a → Int) → [Progress a] → [Progress a] → [Progress (a, a)]
combi f = rec
  where
      rec _ [ ] = [ ]
      rec [ ] _ = [ ]
      rec (x : xs) (y : ys) =
        let
          doCombi ((Success a), (Success b)) = Success (a, b)
          doCombi ((Success a), (Fail b)) = Fail (a, b)
          doCombi ((Fail a), (Success b)) = Fail (a, b)
          doCombi ((Fail a), (Fail b)) = Fail (a, b)

          as = map doCombi [(x, b) | b ← ys]

          bs = map doCombi [(a, y) | a ← xs]

          cs = (rec xs ys)

          g (a, b) = f a + f b
        in

          (doCombi (x, y)) : (merge g as (merge g bs cs))

merge :: (a → Int) → [Progress a] → [Progress a] → [Progress a]
merge f = rec
  where
    rec [ ] bs = bs
    rec as [ ] = as
    rec ((Fail a) : as) ((Fail b) : bs)
      | f a < f b = (Fail a) : (rec as ((Fail b) : bs))
      | otherwise = (Fail b) : (rec ((Fail a) : as) bs)
    rec ((Success a) : as) ((Fail b) : bs)
      | f a < f b = (Success a) : (rec as ((Fail b) : bs))
      | otherwise = (Fail b) : (rec ((Success a) : as) bs)
    rec ((Fail a) : as) ((Success b) : bs)
      | f a < f b = (Fail a) : (rec as ((Success b) : bs))
      | otherwise = (Success b) : (rec ((Fail a) : as) bs)
    rec ((Success a) : as) ((Success b) : bs)
      | f a < f b = (Success a) : rec as (Success b : bs)
      | otherwise = (Success b) : rec (Success a : as) bs
    rec (a : as) (b : bs) = a : b : (rec as bs)
```

A.4 Implementation of the type checking algorithm

```
module Top.Repair.TypedExpr where

import Top.Repair.AExpr
import Top.Types
import Data.IORef
import Data.List (intersect)
import System.IO.Unsafe (unsafePerformIO)
import Debug.Trace
import Data.Maybe (fromJust)

  -- produce i new fresh type variables
freshVars :: Int → [Tp]
freshVars 0 = [ ]
freshVars i = (freshTVar ()) : (freshVars (i − 1))

  -- how to propagate a type over the children of an expression
typePropagate :: OrderedTypeSynonyms → AExpr info → Tp → Maybe Tps
typePropagate syns expr τ =
  case expr of
    (App _ _ args) →
          let
            argTypes = freshVars (length args)
            funType = foldr (.− > .) τ argTypes
          in
            Just (funType : argTypes)
    (Tup _ elems) →
          do
            let
```

```
              tTypes = freshVars (length elems)
              tVars = map (λ(TVar i) → i) tTypes
            unified ← mguMaybe syns τ (tupleType tTypes)
            return (map (λi → lookupInt i unified) tVars)
  (Lst _ elems) →
          do
            let
              freshVar = freshTVar ()
              tVar = (λ(TVar i) → i) freshVar
            unified ← mguMaybe syns τ (listType freshVar)
            return (replicate (length elems) (lookupInt tVar unified))

typeAExprWithContext :: OrderedTypeSynonyms → AExpr (Maybe Typed) →
      Tp → Maybe (AExpr Typed)
typeAExprWithContext synonyms aexpr τ =
    case aexpr of
      App info fun args →
        do
          childTypes ← typePropagate synonyms aexpr τ
          (funTyped, (tp1, s1)) ← recAExpr fun (head childTypes)
          (argsTyped, (tps2, ss2)) ← recAExprs args (tail childTypes)
          let makeApp x = App x funTyped argsTyped
          (superUnify makeApp info τ (s1 : ss2) [(tp1, foldr (.− > .) τ tps2)])
      If info e1 e2 e3 →
        do (e1Typed, (tp1, s1)) ← recAExpr e1 boolType
          (e2Typed, (tp2, s2)) ← recAExpr e2 τ
          (e3Typed, (tp3, s3)) ← recAExpr e3 τ
          let makeIf x = If x e1Typed e2Typed e3Typed
          superUnify makeIf info tp2 [s1, s2, s3] [(tp1, boolType), (tp2, tp3), (tp2, τ)]
      Tup info es →
        do
          tupleChildrenTypes ← typePropagate synonyms aexpr τ
          (esTyped, (tps, ss)) ← recAExprs es tupleChildrenTypes
          let makeTup x = Tup x esTyped
          superUnify makeTup info τ ss [(τ, tupleType tps)]
      Lst info es →
        do
          lstChildrenTypes ← typePropagate synonyms aexpr τ
          (esTyped, (tps, ss)) ← recAExprs es lstChildrenTypes
          let makeLst x = Lst x esTyped
            beta2 = freshTVar ()
          (superUnify makeLst info τ ss ((τ, listType beta2) : zip tps (repeat beta2)))
      Var info string →
        let makeVar x = Var x string
        in (superUnify makeVar info τ [] [])

      Blk info →
        superUnify Blk info τ [] []
   where
    recAExpr :: AExpr (Maybe Typed) → Tp → Maybe (AExpr Typed, Typed)
    recAExpr aexpr τ =
      do new ← typeAExprWithContext synonyms aexpr τ
        return (new, getInfo new)

    recAExprs :: [AExpr (Maybe Typed)] → [Tp] → Maybe ([AExpr Typed], TypedList)
    recAExprs aexprs types =
      do xs ← mapM (λ(expr, tp) → recAExpr expr tp) (zip aexprs types)
        let (as, typedList) = unzip xs
        return (as, unzip typedList)

    superUnify :: (Typed → AExpr Typed) → Maybe Typed → Tp → [MapSubstitution] →
            [(Tp, Tp)] → Maybe (AExpr Typed)
    superUnify make maybeTyped β ss cs =
      case maybeTyped of
        Just t →
          do
            let (tp, _) = t
            checkType ← (mguMaybe synonyms tp τ)
            (return (make (tp, checkType)))

        Nothing →
          do
            superSub ← (unifySubstList synonyms ss)
            let (ts1, ts2) = unzip (superSub |− > cs)
            s ← (mguMaybe synonyms (tupleType ts1) (tupleType ts2))
            let final = s @@ superSub
            (return $ make (final |− > β, final))

unifySubstList :: OrderedTypeSynonyms → [MapSubstitution] → Maybe MapSubstitution
unifySubstList _ [] = Just emptySubst
unifySubstList synonyms (s1 : ss) =
    do s2 ← unifySubstList synonyms ss
      unifySubst synonyms s1 s2

unifySubst :: OrderedTypeSynonyms → MapSubstitution → MapSubstitution →
      Maybe MapSubstitution
unifySubst synonyms s1 s2 =
    let is = dom s1 `intersect` dom s2
      s1′ = removeDom is s1
      s2′ = removeDom is s2
      t1  = tupleType [lookupInt i s1 | i ← is]
      t2  = tupleType [lookupInt i s2 | i ← is]
    in case mguMaybe synonyms t1 t2 of
        Nothing → Nothing
        Just s3
          | null (dom s3) → Just (s1′ @@@ s2′)
          | otherwise     → unifySubst synonyms (s1′ @@@ s2′) s3
```

```
mguMaybe :: OrderedTypeSynonyms → Tp → Tp → Maybe MapSubstitution
mguMaybe synonyms t1 t2 =
    case mguWithTypeSynonyms synonyms t1 t2 of
      Left _       → Nothing
      Right (_, s) → Just s
  -- ----------------------------------------------------
  -- A global IO reference for fresh type variables
  -- ----------------------------------------------------
freshTVarRef :: IORef Int
freshTVarRef = unsafePerformIO (newIORef 10000000)

freshTVar :: () → Tp
freshTVar () =
    unsafePerformIO $
    do i ← readIORef freshTVarRef
      writeIORef freshTVarRef (i + 1)
      return (TVar i)
```

A.5 Implementation of the repair algorithm

```
module Top.Repair.RepairExpr where

import Top.Repair.AExpr
import Top.Repair.TypedExpr
import Top.Repair.Progress
import Top.Repair.Transformations (perms)
import Top.Types

import Data.Maybe (fromJust)
import Data.List (nub)
import Data.Monoid
import Debug.Trace (trace)

getLog :: RepairAdmin → [String]
getLog (RepairAdmin (alog, _)) = alog

emptyTransformation :: Transformation info
emptyTransformation = ((λexpr → [Success expr]), mempty)

getCost :: (a, RepairAdmin) → Int
getCost (_, admin) = getRealCost admin

getRealCost :: RepairAdmin → Int
getRealCost (RepairAdmin (_, cost)) = cost

checkFail :: OrderedTypeSynonyms → Transformations (Maybe Typed) → RepairAdmin → Tp →
      Bool → Int → Progress (AExpr (Maybe Typed), RepairAdmin) →
    [Progress (AExpr Typed, RepairAdmin)]
checkFail syns transList newAdmin τ doAlts maxAltTrans (Success (expr, extraAdmin)) =
    case (typeAExprWithContext noOrderedTypeSynonyms expr τ) of
      Nothing →
         let
            transAlts :: [Progress (AExpr Typed, RepairAdmin)]
            transAlts =
                let
                  children = subexpressions expr
                in
                  if (length children ≡ 0)
                  then [ ]
                  else
                    let
                          typeChild :: AExpr (Maybe Typed) → Maybe (AExpr Typed)
                          typeChild expr = case (getInfo expr) of
                            Just _ → Just (fmap (fromJust) expr)
                            Nothing → Nothing
                          typeChildren :: [AExpr (Maybe Typed)] → [AExpr Typed] →
                            Maybe [AExpr Typed]
                          typeChildren [ ] fcs = Just (reverse fcs)
                          typeChildren (c : cs) fcs =
                            case (typeChild c) of
                              Just tc → typeChildren cs (tc : fcs)
                              Nothing → Nothing
                          transPairs = case (typeChildren children [ ]) of
                            Just children → map (λexpr → (expr, RepairAdmin ([ ], 0))) children
                            Nothing → [ ]
                    in
                          if (length transPairs ≡ 0)
                          then [ ]
                          else
                            map (addToAdmin (mappend (removeErrors extraAdmin)
                              (removeErrors newAdmin)))
                               (getAltExprs syns (getMake expr)
                                             transList expr transPairs τ maxAltTrans)
         in
            ((failedSolution (getRealCost (mappend extraAdmin newAdmin))) :
                (if doAlts then transAlts else [ ]))
      Just typed → [Success (typed, removeErrors (mappend extraAdmin newAdmin))]
checkFail _ _ newAdmin _ _ _ (Fail (_, extraAdmin)) =
    [(failedSolution (getRealCost (mappend extraAdmin newAdmin)))]
checkFail _ _ _ _ _ _ (Debug str) = [Debug str]

removeErrors :: RepairAdmin → RepairAdmin
removeErrors (RepairAdmin (alog, cost)) = RepairAdmin (rec alog, cost)
    where
      rec [ ] = [ ]
      rec (logEntry : logs) =
          if (logEntry ≡ "incorrect node")
          then (rec logs)
          else (logEntry : (rec logs))
resetError :: (Maybe Typed) → (Maybe Typed)
resetError info@(Just (tp, _)) =
    if (tp ≡ errorType)
    then Nothing
    else info
resetError noError = noError
getMake :: EmptyInfo info ⇒ AExpr (Maybe Typed) → ([AExpr info] → AExpr info)
getMake aexpr =
    case aexpr of
      App _ _ _ → (λ(fun : args) → App emptyInfo fun args)
      If _ _ _ _ →       (λ[guard, thenb, elseb] → If emptyInfo guard thenb elseb)
      Tup _ _ → Tup emptyInfo
      Lst _ _ → Lst emptyInfo
wipeTypeInfo :: AExpr (Maybe Typed) → AExpr (Maybe Typed)
wipeTypeInfo expr =
    let
      leafWipe leafInfo = Just (fst (fromJust leafInfo), emptySubst)
      nodeWipe _ = Nothing
    in
      mapAExpr nodeWipe leafWipe expr
repairAExpr :: OrderedTypeSynonyms → Tp → Transformations (Maybe Typed) →
    Int → AExpr (Maybe Tp) → [Progress (AExpr Typed, RepairAdmin)]
repairAExpr synonyms τ transList maxAltTrans aexpr = (
    case aexpr of
      App _ fun args →
        let
          makeApp (y : ys) = App emptyInfo y ys
          branchTypes = fromJust (typePropagate synonyms aexpr τ)
        in superRepair makeApp τ branchTypes (fun : args)
      If _ e1 e2 e3 →
        let makeIf [a, b, c] = If emptyInfo a b c
        in superRepair makeIf τ [boolType, τ, τ] [e1, e2, e3]
      Tup _ es →
        case (typePropagate synonyms aexpr τ) of
          Just tctps → superRepair (Tup emptyInfo) τ tctps es
          _ →
              let childTypes = freshVars (length es)
              in superRepair (Tup emptyInfo) τ childTypes es
      Lst _ es →
        case (typePropagate synonyms aexpr τ) of
          Just lctp →
              superRepair (Lst emptyInfo) τ
                (take (length es) (repeat (head lctp))) es
          _ →
              let childTypes = freshVars (length es)
              in superRepair (Lst emptyInfo) τ childTypes es
      Var _ str →
        let
          e = fmap (fmap (λtp → (tp, emptySubst))) aexpr
          transVars =
              concatMap
                (checkFail synonyms transList
                  (RepairAdmin ([], 0)) τ False maxAltTrans)
                (transform transList e)
        in case (typeAExprWithContext synonyms e τ) of
          Just okay → [Success (okay, RepairAdmin ([], 0))]
          Nothing → (
              (Success (fmap (λmtp → (fromJust mtp, emptySubst)) aexpr,
                RepairAdmin ([], 0))) :
              merge getCost
                [(Success ((Var (τ, emptySubst) str),
                  RepairAdmin
                        (["Type of identifier: '" ++ str ++ "' was changed"], 6)))]
                transVars)
      Blk _ →
        let
          e = fmap (fmap (λtp → (tp, emptySubst))) aexpr
          transBlks =
              concatMap
                (checkFail synonyms transList
                  (RepairAdmin ([], 0)) τ False maxAltTrans)
                (transform transList e)
        in case typeAExprWithContext synonyms e τ of
          Just okay → [Success (okay, RepairAdmin ([], 0))]
          Nothing →
              Success (fmap (λmtp → (fromJust mtp, emptySubst)) aexpr,
                  RepairAdmin ([], 0)) :
              merge getCost
                [Success ((Blk (τ, emptySubst)),
                  RepairAdmin (["Type of block was changed"], 6))]
                transBlks)
  where
    recAExprs :: [Tp] → [AExpr (Maybe Tp)] → [[Progress (AExpr Typed, RepairAdmin)]]
    recAExprs childTypes children =
      map (λ(childTp, child) →
          repairAExpr synonyms childTp transList maxAltTrans child) (zip childTypes children)
    superRepair :: (∀ a . EmptyInfo a ⇒ [AExpr a] → AExpr a) → Tp → [Tp] →
      [AExpr (Maybe Tp)] → [Progress (AExpr Typed, RepairAdmin)]
    superRepair make τ childTypes children =
    let
      op :: Progress [(AExpr Typed, RepairAdmin)] → [Progress (AExpr Typed, RepairAdmin)] →
          [Progress (AExpr Typed, RepairAdmin)]
      op (Success pairs) rest =
          let

              (aexprs, admins) = unzip pairs
              errorNode        = make aexprs
              app              =   (fmap resetError (make (map (fmap (Just)) aexprs)))
              origApp          = make (map
                                   (fmap (λc → case c of
                                      Just tp → Just (tp, emptySubst);
                                      _ → Nothing)) children)
              newAdmin         = mconcat admins
          in
              case (typeAExprWithContext synonyms app τ) of

                  -- In this arrangement, the expression is well-typed.
                  -- Don't use the transformations.
                Just typedExpr →
                  (Success (typedExpr, removeErrors newAdmin)) : rest

                  -- Incorrectly typed, try to repair
                Nothing →
                  let
                            -- use transformations on current node
                          transExprs = (transform transList (wipeTypeInfo app))

                            -- for these transformed expressions,
                            -- allow re-evaluation of children
                          transExprsAlt =
                            foldr (λx y → merge getCost x y) [ ]
                              (map
                                      (checkFail synonyms transList
                                        newAdmin τ True maxAltTrans)
                                      (take maxAltTrans transExprs))

                            -- for these transformed expressions,
                            -- do not allow re-evaluation of children
                          transExprsRest =
                            concatMap
                              (checkFail synonyms transList newAdmin τ
                                              False maxAltTrans)
                              (drop maxAltTrans transExprs)

                              allTrans = merge getCost transExprsAlt transExprsRest

                            -- leave current node intact,
                            -- push needed types for correction to children
                          altExprs = (getAltExprs synonyms make transList
                            origApp pairs τ maxAltTrans)

                          allExprs = merge getCost allTrans altExprs
                  in
                          (failedSolution (getRealCost newAdmin)) :
                            ((Success (errorNode,
                              mappend newAdmin (RepairAdmin (["incorrect node"], 0))))) :
                                (merge getCost allExprs rest)
      op (Fail pairs) rest =
          let
              (_, admins) = unzip pairs
              newAdmin = mconcat admins
          in
              ((failedSolution (getRealCost newAdmin)) : rest)
      op (Debug str) rest = (Debug str) : rest
    in (foldr op [ ] . combiList getCost . (recAExprs childTypes)) children
showP :: Progress (AExpr Typed, RepairAdmin) → String
showP (Success (s, r)) = "* " ++ showSimple s ++ " --> " ++ show (getLog r)
showP (Fail _) = "* " ++ "fail"

isBlk :: AExpr info → Bool
isBlk (Blk _) = True
isBlk _ = False

getAltExprs :: OrderedTypeSynonyms → (∀ a . EmptyInfo a ⇒ [AExpr a] → AExpr a) →
    Transformations (Maybe Typed) → AExpr (Maybe Typed) → [(AExpr Typed, RepairAdmin)] →
    Tp → Int → [Progress (AExpr Typed, RepairAdmin)]
getAltExprs syns make transList expr pairs τ maxAltTrans =
  let
    (exprs, admins) = (unzip pairs)
    makeConditions = (getTypeInfo expr τ)

    disabledCombis :: [[(AExpr Typed, Bool)]]
    disabledCombis =  (findConflict syns exprs makeConditions)

    combisWithTypesAndConditions :: [([(AExpr Typed, Bool)], ([Tp], [(Tp, Tp)]))]
    combisWithTypesAndConditions =
      (zip disabledCombis
        (map (evalTypesAndConditions makeConditions) disabledCombis))

    altSubsts = map (fromJust . findSubst syns) combisWithTypesAndConditions

    altChildren :: [[Progress [(AExpr Typed, RepairAdmin)]]]
    altChildren =
      (evalCombis syns admins combisWithTypesAndConditions
        altSubsts transList maxAltTrans)

    altCombis :: [[Progress (AExpr Typed, RepairAdmin)]]
    altCombis = map (λcombi → map checkNewExpr combi) altChildren

    newExprs :: [Progress (AExpr Typed, RepairAdmin)]
    newExprs = foldr (λcombiList rest → merge getCost combiList rest) [ ] altCombis
  in
```

```
        newExprs
      where
        checkNewExpr :: Progress [(AExpr Typed, RepairAdmin)] →
            Progress (AExpr Typed, RepairAdmin)
        checkNewExpr (Success pairs) =
            let
                (exprs, admins) = unzip pairs
                app = make (map (fmap Just) exprs)
                newAdmin = mconcat admins
            in
                case typeAExprWithContext syns app τ of
                    Just typedExpr → (Success (typedExpr, newAdmin))
                    Nothing → (failedSolution (getRealCost newAdmin))
        checkNewExpr (Fail pairs) =
            let
                (_, admins) = unzip pairs
                newAdmin = mconcat admins
            in (failedSolution (getRealCost newAdmin))
        checkNewExpr (Debug str) = (Debug str)
evalTypesAndConditions :: ([Tp] → [(Tp, Tp)]) → [(AExpr Typed, Bool)] → ([Tp], [(Tp, Tp)])
evalTypesAndConditions makeC perms =
    let
        types = map getType perms
    in
        (types, makeC types)
            where
                getType (expr, True) = fst (getInfo expr)
                getType (_, False) = freshTVar ()
findConflict :: OrderedTypeSynonyms → [AExpr Typed] → ([Tp] → [(Tp, Tp)]) →
    [[(AExpr Typed, Bool)]]
findConflict syns exprs makeConditions = rec exprs 1
    where
    rec [] _ = []
    rec as omitAmount =
        let
            perms = (doPerms as omitAmount)
            permsWithConditions =
                zip perms (map (λp → evalTypesAndConditions makeConditions p) perms)
            (unifiedPerms, _) = unzip (filter (doUnify syns) permsWithConditions)
        in
            if (length unifiedPerms) > 0
            then unifiedPerms
            else rec as (omitAmount + 1)
getTypeInfo :: AExpr (Maybe Typed) → Tp → ([Tp] → [(Tp, Tp)])
getTypeInfo expr τ = case expr of
    (App _ _ _) →
        let
            makeAppConditions :: [Tp] → [(Tp, Tp)]
            makeAppConditions childTypes =
                let
                    funType = head childTypes
                    argTypes = tail childTypes
                  in
                    [(funType, foldr (.− > .) τ argTypes)]
        in makeAppConditions
    (If _ _ _ _) → (λ(g : t : [e]) → [(g, boolType), (t, τ), (e, τ)])
    (Lst _ _) → map (λtp → (tp, τ))
    (Tup _ _) → (λtps → [((tupleType tps), τ)])
doUnify :: OrderedTypeSynonyms → ([(AExpr Typed, Bool)], ([Tp], [(Tp, Tp)])) → Bool
doUnify syns (exprs, (_, cs)) =
    let
        enabledExprs = (filter snd exprs)
        substs = map (λe → snd . getInfo . fst $ e) enabledExprs
        unified = (unifySubstList syns substs)
    in
        case unified of
            Just superSub → (
                let
                    (ts1, ts2) = unzip (superSub |− > cs)
                    s = (mguMaybe syns (tupleType ts1) (tupleType ts2))
                in
                    case s of
                      Just _ → True
                      Nothing → False)
            _ → False
doPerms :: [a] → Int → [[(a, Bool)]]
doPerms as i =
    let
        enabled = take ((length as) − i) (repeat True)
        disabled = take i (repeat False)
        shuffles = nub (perms (enabled ++ disabled))
        doZip _ [] = []
        doZip (a : as) (s : ss) = (zip a s) : (doZip as ss)
    in
        doZip (repeat as) shuffles
findSubst :: OrderedTypeSynonyms → ([(AExpr Typed, Bool)], ([Tp], [(Tp, Tp)])) →
    Maybe MapSubstitution
findSubst syns (alt, (_, conditions)) =
    do
        let
```

```
          enabledBranches = filter snd alt
          substs = map (snd . getInfo . fst) enabledBranches
        -- determine the substitution of the enabled branches
      uSubst ← unifySubstList syns substs
      let
          (tp1, tp2) = unzip (uSubst |− > conditions)
      newSub ← (mguMaybe syns (tupleType tp1) (tupleType tp2))
      return (uSubst @@ newSub)
addToAdmin :: RepairAdmin → Progress (AExpr Typed, RepairAdmin) →
    Progress (AExpr Typed, RepairAdmin)
addToAdmin oldAdmin (Success (expr, newAdmin)) =
    (Success (expr, mappend oldAdmin newAdmin))
addToAdmin oldAdmin (Fail (expr, newAdmin)) =
    (Fail (expr, mappend oldAdmin newAdmin))
addToAdmin _ (Debug str) = (Debug str)
evalCombis :: OrderedTypeSynonyms → [RepairAdmin] →
    [([(AExpr Typed, Bool)], ([Tp], [(Tp, Tp)]))] →
    [MapSubstitution] → Transformations (Maybe Typed) → Int →
    [[Progress [(AExpr Typed, RepairAdmin)]]]
evalCombis _ _ [] _ _ _ = []
evalCombis syns admins ((a, (tps, _)) : as) (uSub : ss) transList maxAltTrans =
    let

      recBranches :: [(((AExpr Typed, Bool), RepairAdmin), Tp)] →
          [[Progress (AExpr Typed, RepairAdmin)]]
      recBranches [] = []
      recBranches ((((expr, enabled), admin), oldTau) : bs) =
          if enabled
          then
                -- enabled branch, return unmodified
            ([Success (expr, admin)] : recBranches bs)
          else

                -- disabled branch, re-evaluate with new tau
              let
                  newTau = (uSub |− > oldTau)
                  altAdmin = mappend (RepairAdmin (["re-evaluation"], 1)) admin
                  repairedBranch =
                    repairAExpr syns newTau transList maxAltTrans
                      (fmap (λ(tp, _) → Just (tp)) expr)
                  repairedBranchWithUpdatedAdmin = map (addToAdmin altAdmin) repairedBranch
              in
                (repairedBranchWithUpdatedAdmin) : (recBranches bs)

      branchesWithTypes = (zip (zip a admins) tps)
      newBranches = recBranches branchesWithTypes
      combiBranches = combiList getCost newBranches
    in
      (combiBranches : (evalCombis syns admins as ss transList maxAltTrans))
transform :: (EmptyInfo info) ⇒ Transformations info → AExpr info →
    [Progress (AExpr info, RepairAdmin)]
transform transList aexpr =
    [new | (trans, admin) ← transitiveClosure transList,
      new ← map (addAdmin admin) (trans aexpr)]
transitiveClosure :: EmptyInfo info ⇒ Transformations info → Transformations info
transitiveClosure xs = emptyTransformation : prod xs (transitiveClosure xs)
  where
    prod (x : xs) (y : ys) =
      (x ‘andThen‘ y) :
      (map (x‘andThen‘) ys . + +. map (‘andThen‘y) xs . + +. prod xs ys)
    prod _ _ = []
    (. + +.) = mergeBy (λ(_, admin1) (_, admin2) → compare admin1 admin2)
andThenList :: Transformations info → Transformation info
andThenList =
    foldr andThen emptyTransformation
andThen :: (a → [Progress a], RepairAdmin) → (a → [Progress a], RepairAdmin) →
    (a → [Progress a], RepairAdmin)
andThen (f1, a1) (f2, a2) =
    let
        checkf1 (Success s) = f2 s
        checkf1 other = [other]
        composed expr = concatMap checkf1 (f1 expr)
    in
      (composed, mappend a1 a2)
mergeBy :: (a → a → Ordering) → [a] → [a] → [a]
mergeBy f = rec
  where
    rec [] ys = ys
    rec xs [] = xs
    rec (x : xs) (y : ys) =
      case f x y of
        GT → y : rec (x : xs) ys
        _ → x : rec xs (y : ys)
```