Type Inference for a Declarative Intermediate Language

Jonas Oberschweiber

Bachelor-Thesis
eingereicht im September 2012

Christian-Albrechts-Universität zu Kiel
Programmiersprachen und Übersetzerkonstruktion

Betreut durch: Prof. Dr. Michael Hanus und M.Sc. Björn Peemöller

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Contents

1 Introduction **1**

2 Curry and FlatCurry **3**
2.1 Curry . . . 3
2.1.1 Modules . . . 3
2.1.2 Built-in Data Types . . . 4
2.1.3 Datatypes . . . 4
2.1.4 Functional Programming . . . 5
2.1.5 Logic Programming . . . 6
2.2 FlatCurry . . . 7
2.2.1 Prog . . . 8
2.2.2 TypeExpr . . . 8
2.2.3 TypeDecl . . . 9
2.2.4 Expr . . . 10
2.2.5 FuncDecl . . . 12
2.2.6 OpDecl . . . 14

3 Type Inference **15**
3.1 Generating Equations . . . 15
3.1.1 A more involved example . . . 16
3.2 Substitutions . . . 17
3.3 Unification . . . 18

4 Implementation **21**
4.1 AnnotatedFlatCurry . . . 21
4.2 Unification . . . 22
4.3 Inference . . . 27
4.3.1 Creating a Type Environment . . . 27
4.3.2 Converting and Annotating FlatCurry Programs . . . 28
4.3.3 Rules for Generating Equations . . . 30
4.3.4 Implementing the Rules in Curry . . . 34
4.3.5 Interfacing with the Unifier . . . 36
4.3.6 Applying a Substitution to an Expression . . . 37
4.3.7 Generating a Function Type and Normalizing Type Variables . . . 37
4.3.8 Handling Errors . . . 39
4.4 Testing the Inference Process . . . 40
4.5 Performance . . . 42
4.5.1 The `GUI` Module . . . 43
4.5.2 The `Char` Module . . . 44

5 Conclusions **45**

A Interface Description **47**
A.1 AnnotatedFlatCurry 47
A.2 Unification 49
A.3 Inference 51

B Raw Performance Data **53**

C BankersQueue in FlatCurry **55**

Bibliography **57**

Chapter 1

Introduction

Modern compilers perform numerous transformations during the compilation process to speed up execution, reduce memory usage or optimize for more exotic criteria such as energy usage (e.g. [KVI02]). The compilers for the functional logic language *Curry*, in particular the *KiCS2* compiler ([Han+12a]), are no exception. There are many aspects to the correctness of such transformations, the most obvious one being whether the transformed programs perform computations equivalent to the original ones.

For strongly typed languages such as Curry, another interesting aspect is whether these transformations leave a program type correct. In many Curry compilers, including *KiCS2*, transformations are usually performed on a representation of the program in an intermediate language called *FlatCurry*. The FlatCurry representation of a Curry program is generated by the compiler's front end. During this translation from Curry to FlatCurry, the front end performs its own type checking and calculates (*infers*) all types in the original Curry program. The resulting FlatCurry program can thus be assumed to be type correct.

The *KiCS2* compiler does not, however, have any facilities for explicitly checking the type correctness of FlatCurry programs. Thus, there is currently no way for the compiler to automatically verify that the various transformations it performs on FlatCurry programs during the compilation process leave those programs type correct.

The goal of this bachelor thesis is the development of a program that can infer and check the types of FlatCurry programs. This type inferrer should have a compact and easy-to-use interface; since the *KiCS2* compiler is implemented in Curry and the type inferrer's main purpose is to eventually be used by *KiCS2*, it should itself be a Curry program.

In the next chapter, we give a cursory introduction to Curry and describe the FlatCurry language in more detail. Chapter 3 gives a general introduction to type inference, while Chapter 4 discusses the specifics of our implementation. We present conclusions and possible improvements in Chapter 5.

Chapter 2

Curry and FlatCurry

This chapter will give a short introduction to Curry and a more detailed description of FlatCurry and how the two relate to each other.

2.1 Curry

Curry is an integrated functional logic language, i.e. it combines the functional and logic programming paradigms into one language. A complete description of the language can be found in [Han+06]. Syntactically Curry is similar to the functional programming language Haskell ([Pey+03]).

2.1.1 Modules

Curry programs and libraries are organized into *modules*. A module can be declared at the beginning of a Curry source file and must contain at least a module name, for instance `Example`:

```
module Example where
```

The module's function and datatype definitions follow the `where` keyword. If a Curry source file does not contain a module definition, it is implicitly assumed to define a module of the same name as the source file. As it is not possible to define more than one Curry module inside a Curry source file, each Curry file contains exactly one module.

Curry modules can import other Curry modules using the `import` syntax:

```
module Example where

import List
import Char
```

It is also possible to selectively import functions or datatypes from other modules. In the following example, only the `findIndex` function is imported from the `List` module:

```
module Example where

import List (findIndex)
```

There are a few other ways of importing modules, which are explained in detail in Chapter 6 of [Han+06].

Using the minimal module declaration above, every function and datatype inside the module is *exported* and can be imported and used by other modules. It is also possible to explicitly list all functions and datatypes that should be exported. All other functions and datatypes will then remain internal to the module. The following definition, for example, would declare

that the `classifyAnimals` function, the `Animal` datatype and the `Classification` datatype should be exported from the module. The `(..)` after the `Animal` datatype denotes that all of `Animal`'s constructors should be exported as well, whereas none of `Classification`'s constructors will be exported. This facility of exporting dataypes without exporting their constructors is useful when defining abstract data types that should be usable for other modules without exposing their internals.

```
module Example (classifyAnimals, Animal(..), Classification) where
```

2.1.2 Built-in Data Types

Before we introduce custom data types we will look at the most important built-in ones: integers, characters, lists, tuples and strings. All of these are defined in the `Prelude` module, which is the module that contains the language's basic operations and data types and is implicitly imported into every Curry module. There is support for literal values of each of these types in Curry.

- `Int` is the data type for integral numbers. Its literal values look like one would expect: `4`, `-42` and `10389` are some examples.
- `Char` is the character data type. Character literals must be enclosed in single quotes, like this: `'a'`.
- The basic list data type in Curry is a singly linked list. There is support for prepending an item to the front of a list (via the `:` constructor), for getting the first element and the rest of a list (via the `head` and `tail` functions) and for appending one list to another (via the `++` operator). All elements of a list must be of the same type.

  Literal list values are expressions enclosed in brackets (`[]`) and separated by commas: `[1, 2, 3]` or `['a', 'b', (someFunction 1)]` are examples.
- *Tuples* are like their mathematical namesakes: ordered collections of a fixed number of elements. Contrary to a list, the elements of a tuple can be of differing types. A tuple's elements are usually accessed via pattern matching or the `fst` and `snd` functions, which return the first and second components of pairs, respectively. Literal tuple values are denoted by expressions enclosed in parentheses (`()`) and separated by commas, for instance: `(1, 2)` or `([1, 2], 3)`.
- The `String` data type is just a type synonym for a list of characters and therefore behaves like a regular list: One can prepend characters to a string with the `:` constructor, append strings to one another with the `++` operator and so on. To make working with strings nicer, there is a literal syntax for them in Curry: A sequence of characters enclosed in double quotes is considered a string literal, as in

  ```
  "hello" ++ " world"
  ```

2.1.3 Datatypes

Curry is a strongly typed language featuring a polymorphic type system. Datatypes are defined by a name, a list of polymorphic type parameters and constructor definitions separated by vertical bars. The `data` keyword introduces a datatype definition. For instance,

```
data Maybe a = Nothing | Just a
```

defines the datatype `Maybe`, which has one polymorphic type parameter, `a`, and two constructors. The first constructor, `Nothing`, does not take any arguments, it is *nullary*. The second constructor, `Just`, takes one argument of the polymorphic type `a`.

Besides simple polymorphic type parameters, it is also possible to define constructors that take more complex, constructed types as parameters. Constructors can also be self-referential, taking arguments of the datatype that is being defined. For example,

```
data JValue = JString String
            | JNumber Float
            | JBool Bool
            | JNull
            | JObject [(String, JValue)]
            | JArray [JValue]
```

from Chapter 5 of [OSG08] defines a datatype for storing *JSON* data (JavaScript Object Notation, see [Cro06]). The `JString`, `JNumber` and `JBool` constructors each take a `String`, `Float` and `Bool` argument respectively, the `JNull` constructor is another example for a *nullary* constructor. `JObject` and `JArray` are examples of constructors that take more complex, constructed types as arguments. Both are also self-referential.

Type synonyms

In addition to defining *new* datatypes, Curry also makes it possible to define *synonyms* for existing types. A definition of a type synonym is made up of a name, a list of polymorphic type parameters and a type expression Type synonyms are introduced by the `type` keyword. For instance,

```
type StringTuple a = (String, a)
```

defines a synonym for tuples that have a `String` fixed as their first component but are polymorphic in their second component. The type of a tuple of `String` and `Int` would then be

```
StringTuple Int
```

2.1.4 Functional Programming

Curry functions are defined by sets of *equations* or *rules*. Equations come in two shapes: simple equations and conditional equations.

Simple equations consist of the name of the function followed by a list of *patterns* for the function's arguments on their left-hand sides and an expression to evaluate on their right-hand sides, like this:

```
square x = x * x
```

In this example, the function `square` has only one rule that takes a single parameter which is bound to the name `x`. Its expression is `x * x`. We can also use more complex patterns:

```
squareFst (x,_) = x * x
```

Here, the parameter is *deconstructed* using the tuple constructor. `x` is bound to the tuple's first component, while its second component is ignored. The expression is the same as in the previous example.

Conditional equations have one or more *guards* on their left-hand sides, introduced by a vertical bar character after the function name and its rule patterns, which is then followed by an expression, called the *condition*.

If there are multiple guards, their conditions must all be *boolean* expressions. A single guard can be either a *boolean* expression or a *constraint* (see 2.1.5). The right-hand side of the first guard clause whose condition evaluates to `True` will be evaluated. To make programming with multiple guards more convenient, a boolean function called `otherwise` is predefined, which always evaluates to `True`. An example of a rule with multiple guard clauses is the following function that calculates a person's body mass index (BMI) given his or her height in meters and weight in kilograms and then classifies the resulting value (this example was adapted from Chapter 4 of [Lip11]):

```
bmiTell :: Float -> Float -> String
bmiTell width height
    | bmi <= 18.5 = "Underweight"
    | bmi <= 25.0 = "Normal"
    | bmi <= 30.0 = "Overweight"
    | otherwise   = "Very overweight"
    where bmi = weight / (height * height)
```

The example also shows the use of the `where` keyword to define a variable local to the conditions and right-hand sides of the rule and the optional type annotations for functions.

2.1.5 Logic Programming

The two main concepts of logic programming as implemented in Curry are *free variables* and *constraints*. Constraints can be specified as part of the rule of a function (if there is only one guard that guard can be a constraint, see 2.1.4). If a constraint is specified, the rule is only applied if that constraint can be *satisfied*. Curry's basic constraint is the *equational constraint* `=:=` (the following example is taken from [Han+06]):

```
[x] =:= [0]
```

This equational constraint can be satisfied if `x` is bound to `0`. Multiple constraints can be combined to form a conjunction using the `&` symbol:

```
[x] =:= [0] & x =:= f h & h x =:= g x
```

Conjunctions are interpreted concurrently, see Section 2.6 of [Han+06] for details.

When using a constraint such as `[x] =:= [0]` from above, the variable `x` is *free*, meaning it is not bound and not instantiated. It will be instantiated and bound to `0` in the process of satisfying the constraint. In Curry expressions, free variables must be declared inside `let` or `where` expressions using the syntax `x free`. For instance, the following is a convoluted way of defining a 0-ary function returning the number 0:

```
zero | [x] =:= [0] = x where x free
```

The free variable `x` is instantiated and bound such that it satisfies the constraint `[x] =:= [0]` and its value is returned. If there are multiple bindings satisfying a constraint, all possibilities

are computed. For instance, the function `allMammals` in the following example would return `Elephant` and `Dingo`:

```
data Class = Mammal | Bird
data Animal = Elephant | Eagle | Hawk | Dingo

class Elephant = Mammal
class Eagle = Bird
class Hawk = Bird
class Dingo = Mammal

allMammals | class x =:= Mammal = x where x free
```

This is an example of a non-deterministic program, as the function `allMammals` has two results. Another way to introduce non-determinism into Curry programs are overlapping rules: Functions in Curry can have rules with overlapping left-hand sides and such functions are evaluated non-deterministically. For instance, the function

```
addOne :: (Int, Int) -> Int
addOne (x, _) = x + 1
addOne (_, x) = x + 1
```

has overlapping left-hand sides: any tuple of `Int`s will match the patterns in both rules, as any tuple of `Int`s has a first and a second component. Thus applying the function to the tuple `(0, 10)` will yield both 1 and 11.

2.2 FlatCurry

Curry programs are translated into the intermediate language FlatCurry by the compiler's front end. FlatCurry programs are usually represented by the Curry datatypes found in the `FlatCurry` module that ships with *KiCS2*. We will give an overview of the language and the datatypes in the `FlatCurry` module by translating parts of a Curry program to FlatCurry.

The program that we will translate is a simple implementation of a queue, based on the *banker's queue* from [Oka98] and the source code of the `Dequeue` module that ships with *KiCS2*.

```
module BankersQueue (Queue, empty, isEmpty,
                     append, head, tail) where

data Queue a = Q [a] Int [a] Int

empty :: Queue a
empty = Q [] 0 [] 0

isEmpty :: Queue a -> Bool
isEmpty (Q _ l _ _) = l == 0

queue :: [a] -> Int -> [a] -> Int -> Queue a
queue f lf r lr | lr <= lf  = Q f lf r lr
                | otherwise = Q (f ++ (reverse r)) (lf + lr) [] 0
```

```
append :: Queue a -> a -> Queue a
append (Q f lf r lr) x = queue f lf (x:r) (lr + 1)

head :: Queue a -> a
head (Q []    _ _ _) = error "E"
head (Q (x:_) _ _ _) = x

tail :: Queue a -> Queue a
tail (Q []    _  _ _)  = error "E"
tail (Q (_:f) lf r lr) = queue f (lf - 1) r lr
```

2.2.1 Prog

The top-level datatype of a FlatCurry program is `Prog`:

```
data Prog = Prog String [String] [TypeDecl] [FuncDecl] [OpDecl]
```

Its constructor takes arguments corresponding to the main parts of a Curry program: The name of the module it defines, a list of imported modules, a list of type declarations, a list of function declarations and a list of operator fixity declarations.

For now, our `BankersQueue` module looks like this (remember that the `Prelude` module is automatically imported into every module):

```
Prog "BankersQueue" ["Prelude"] [] [] []
```

2.2.2 TypeExpr

Before moving on to declaring datatypes, we have to first take a look at `TypeExpr`, which is used to represent FlatCurry types:

```
data TypeExpr = TVar TVarIndex
              | FuncType TypeExpr TypeExpr
              | TCons QName [TypeExpr]
type TVarIndex = Int
data QName = (String, String)
```

As can be seen in the `TypeExpr` datatype's definition, a type can be one of three things:

(1) a polymorphic type variable (`TVar`)

(2) a function type (`FuncType`)

(3) a constructed type (`TCons`)

A polymorphic type variable is identified by an index, which is why the first and only argument to `TVar` is a `TVarIndex` (which is just a synonym for `Int`).

Function types are represented by `FuncType`, which takes two types (`TypeExpr`) as its arguments. In case a function takes more than one argument, the second argument of the top-level `FuncType` is another `FuncType`. A Curry function with a type signature of `a -> b -> c -> a` would thus have the following type in FlatCurry (modulo the renaming of type variables):

```
FuncType (TVar 0) (FuncType (TVar 1) (FuncType (TVar 2) (TVar 0)))
```

This nesting of `FuncType` represents the right-associativity of functions in Curry and FlatCurry and makes function currying (partial application) easy to implement.

`TCons` is used to represent constructed types. It takes the *qualified name* (`QName`, a tuple of module name and type name) of the datatype and a list of types for its type parameters. The type of an `Int`, for instance, would be:

```
TCons ("Prelude", "Int") []
```

whereas the type of a *list* of `Int`s would be:

```
TCons ("Prelude", "[]") [TCons ("Prelude", "Int") []]
```

2.2.3 TypeDecl

Types are defined by the `TypeDecl` type, which has two constructors:

```
data TypeDecl = Type QName Visibility [TVarIndex] [ConsDecl]
              | TypeSyn QName Visibility [TVarIndex] TypeExpr
data Visibility = Public | Private
```

`Type` is used to define actual datatypes, while `TypeSyn` is used to define type synonyms (Curry's `type` construct).

A type synonym is defined by its qualified name, its visibility (`Public` or `Private`, depending on whether it is exported from the module or not), a list of the indices of all of the synonym's polymorphic type variables and the type it is a synonym for. If we were, for example, to define the synonym `IntList` in the module `Example` for a list of `Int`s and export that type synonym from the module, the resulting `TypeSyn` could look like this:

```
TypeSyn ("Example", "IntList") Public []
    (TCons ("Prelude", "[]") [TCons ("Prelude", "Int") []])
```

To define a datatype using the `Type` constructor, we also need its qualified name, its visibility and a list of polymorphic type variable indices. In addition, `Type` expects a list of constructor declarations (`ConsDecl`):

```
data ConsDecl = Cons QName Int Visibility [TypeExpr]
```

A constructor declaration is made up of a qualified name for the constructor, the number of parameters it expects (its *arity*), its visibility and a list of the types of its expected parameters. In a well-formed FlatCurry program the arity is always equal to the length of the list of parameter types and is only provided for convenience.

One could define the `Queue` type with its `Q` constructor from our `BankersQueue` module like this:

```
Type ("BankersQueue", "Queue") Public [0] [
  Cons ("BankersQueue", "Q") 4 Private
    [TCons ("Prelude", "[]") [TVar 0],
     TCons ("Prelude", "Int") [],
     TCons ("Prelude", "[]") [TVar 0],
     TCons ("Prelude", "Int") []]]
```

As the `Queue` type is the only datatype in the `BankersQueue` module, we are now ready to extend our program definition:

```
Prog "BankersQueue" ["Prelude"] [
  Type ("BankersQueue", "Queue") Public [0] [
    Cons ("BankersQueue", "Q") 4 Private
      [TCons ("Prelude", "[]") [TVar 0],
       TCons ("Prelude", "Int") [],
       TCons ("Prelude", "[]") [TVar 0],
       TCons ("Prelude", "Int") []]]]
  [] []
```

2.2.4 Expr

Before looking at how functions are declared, we will take a look at FlatCurry expressions. In FlatCurry, there are seven different types of expressions, represented by the `Expr` datatype:

```
data Expr = Comb CombType QName [Expr]
          | Var VarIndex
          | Lit Literal
          | Let [(VarIndex, Expr)] Expr
          | Free [VarIndex] Expr
          | Or Expr Expr
          | Case CaseType Expr [BranchExpr]
```

We will take a look at these one by one.

Comb

Combinations, which can be function or constructor calls, are represented by `Comb`. Its first argument, the `CombType`, determines whether it is a call to a function or a constructor and whether all parameters are present:

- ▷ `ConsCall` - a complete call to a constructor
- ▷ `FuncCall` - a complete call to a function
- ▷ `ConsPartCall Int` - a partial call to a constructor, where the argument is the number of parameters missing
- ▷ `FuncPartCall Int` - a partial call to a function, where the argument is the number of parameters missing

Secondly, `Comb` expects the qualified name of the function or constructor to call, and, lastly, a list of expressions to pass to the function/constructor as parameters.

A sample call to construct an empty list could look like this:

```
Comb ConsCall ("Prelude", "[]") []
```

Var

`Var` evaluates to the value of a variable. Variables in FlatCurry, like polymorphic type variables, are numbered. `Var` takes as its parameter the number of the variable whose value it should evaluate to.

While `VarIndex` and `TVarIndex` are both type synonyms for `Int`, it is important to distinguish between *variable indices* used in `Var` and *type variable indices* used in `TVar`.

Assuming that both hold values of type `Int`, adding the values of the variables with indices 2 and 3 could look like this:

```
Comb FuncCall ("Prelude", "+") [Var 2, Var 3]
```

Lit

Literal values are represented by `Lit` expressions. Its only parameter is a `Literal`:

```
data Literal = Intc Int | Floatc Float | Charc Char
```

Adding the literal values 2 and 3 could look like this:

```
Comb FuncCall ("Prelude", "+") [Lit (Intc 2), Lit (Intc 3)]
```

Let

`Let` is used to bind variables to values in the context of a single expression (the binding has no effect outside of that expression). It is similar to Curry's `let` and `where` constructs, although these two constructs are not always translated to FlatCurry `Let`s by the compiler. If the bound expressions are simple enough, not dependent on each other and not used more than once in the inner expression, they are usually directly embedded into that inner expression.

`Let` expects an *association list*, a list of tuples of variable indices and expressions, as its first parameter. Each tuple in that list represents the binding of a variable to an expression, where the variable index is the tuple's first component and the expression is the tuple's second component. When `Let` is evaluated, it in turn evaluates the expression in its second parameter with the bindings from the association list in effect.

We can use the `Let` construct to make our above example of adding the numbers 2 and 3 a little less straight-forward:

```
Let [(1, Lit (Intc 2)), (2, Lit (Intc 3))]
    (Comb FuncCall ("Prelude", "+") [Var 1, Var 2])
```

Free

`Free` introduces local free/logic variables (see Section 2.1.5). It takes the indices of the free variables as its first argument and the expression to introduce these variables in as its second argument.

Or

`Or` non-deterministically evaluates both expressions passed to it as arguments. It is used to translate overlapping rules in Curry functions.

Case

A `Case` is a construct that takes an expression, the *subject*, and evaluates another expression, one of the *branches*, based on the form of the subject expression. If none of the branches match the subject, the case expression fails to evaluate. The `Case` constructor takes the subject as the second argument and a list of branch expressions as the third argument. It also takes a type (`CaseType`) as its first argument, which is relevant with regard to free variables, but irrelevant for type inference. For details, refer to Appendix D of [Han+06].

Each branch of the `Case` is represented by a `BranchExpr`. A `BranchExpr` consists of a `Pattern`, used to describe the structure a value must have for the branches' expression to be evaluated, and an `Expr`.

```
data BranchExpr = Branch Pattern Expr
data Pattern = Pattern QName [VarIndex]
             | LPattern Literal
```

There are two kinds of patterns: literal patterns, `LPattern`, and constructor patterns, `Pattern`. Literal patterns only match the literal value specified in their argument; a branch with pattern `LPattern (Intc 0)` would be chosen only if the case's subject expression evaluates to the integer 0.

Constructor patterns take a constructor's qualified name as their first argument. We will call this constructor the *match-constructor*. A constructor pattern will match any value constructed by its match-constructor. Additionally, the matched value will be *deconstructed*: The variables specified in `Pattern`'s second argument are bound to the values passed to the match-constructor when the matched value was constructed (in the order of the match-constructor's arguments). For example, the following branch/pattern combination will match all pairs and the matched pair's first component will be bound to the variable with index 1, while its second component will be bound to the variable with index 2 inside the branch's expression. So, if chosen the whole branch would evaluate to the pair's first component:

```
Branch (Pattern ("Prelude", "(,)") [1, 2]) (AVar 1)
```

To give a complete example of a case expression, the following would evaluate to one if the list in variable 1 were not empty and to zero if it were:

```
Case Flex (Var 1) [Branch (Pattern ("Prelude", ":") [2, 3]) (Lit (Intc 1)),
                   Branch (Pattern ("Prelude", "[]") []) (Lit (Intc 0))]
```

2.2.5 FuncDecl

Functions are represented by the `FuncDecl` datatype, which is defined as:

```
data FuncDecl = Func QName Int Visibility TypeExpr Rule
```

The constructor's parameters are the function's name (`QName`), its number of parameters (also called its arity), whether it is exported from the module (`Visibility`), its type (`TypeExpr`) and a `Rule`. All of these types have been covered in the section on `TypeDecls` (2.2.3), except for `Rule`. `Rule` is defined as follows:

```
data Rule = Rule [VarIndex] Expr
          | External String
```

The `External` constructor is used for functions that are defined externally. We will not cover those here. Instead, we will focus on the `Rule` constructor, which takes as its arguments a list of variable indices for its parameters (`VarIndex` is just a synonym for `Int`) and an `Expr`. Note that the length of the `VarIndex` list is equal to the arity of the function.

The `Expr` is the function's body; the variable indices listed in `Rule`'s first argument are used to refer to the function's formal parameters inside the body.

We now have all definitions we need to translate the `BankersQueue` module's functions. As an example, we will translate the `tail` function. `tail` takes a `Queue` as its argument and distinguishes two cases: the first argument of the `Q` constructor may be an empty list or a non-empty list. Thus, we need a way to deconstruct the function's first argument and reference the `Q` constructor's individual arguments. We do this using a `Case` expression with a single branch (assuming that the variable with index 1 refers to `tail`'s first parameter):

```
Case Flex (Var 1) [
  Branch (Pattern ("BankersQueue", "Q") [2,3,4,5])
  ...]
```

We can now add another, inner `Case` expression with branches for empty and non-empty lists. Note that we refer to the variable with index 2, which has been bound to `Q`'s first argument in the outer `Case`:

```
Case Flex (Var 1) [
  Branch (Pattern ("BankersQueue", "Q") [2,3,4,5])
         (Case Flex (Var 2) [
           Branch (Pattern ("Prelude", "[]") []) ...,
           Branch (Pattern ("Prelude", ":") [6,7]) ...])]
```

Finally, we can fill in the calls to `error` and `queue` completing the expression for our function:

```
Case Flex (Var 1) [
  Branch (Pattern ("BankersQueue", "Q") [2,3,4,5])
         (Case Flex (Var 2) [
           Branch (Pattern ("Prelude", "[]") [])
                  (Comb FuncCall ("Prelude", "error") [
                    Comb ConsCall ("Prelude", ":") [
                      Lit (Charc 'E'),
                      Comb ConsCall ("Prelude", "[]") []]]),
           Branch (Pattern ("Prelude", ":") [6,7])
                  (Comb FuncCall ("BankersQueue", "queue") [
                    Var 7,
                    Comb FuncCall ("Prelude", "-") [
                      Var 3,
                      Lit (Intc 1)],
                    Var 4,
                    Var 5])])]
```

Now all that is left to do is to add the actual `FuncDecl` to our `Prog` (some of the expressions have been abbreviated to reduce clutter):

```
Prog "BankersQueue" ["Prelude"]
  [Type ("BankersQueue", "Queue") Public [0] [
     Cons ("BankersQueue", "Q") 4 Private
       [TCons ("Prelude", "[]") [TVar 0],
        TCons ("Prelude", "Int") [],
        TCons ("Prelude", "[]") [TVar 0],
        TCons ("Prelude", "Int") []]]]
  [Func ("BankersQueue", "tail") 1 Public
        (FuncType (TCons ("BankersQueue", "Queue") [TVar 0])
                  (TCons ("BankersQueue", "Queue") [TVar 0]))
        (Rule [1]
          Case Flex (Var 1) [
            Branch (Pattern ("BankersQueue", "Q") [2,3,4,5])
                   (Case Flex (Var 2) [
                     Branch (Pattern ("Prelude", "[]") []) ...,
                     Branch (Pattern ("Prelude", ":") [6,7]) ...])])]
  []
```

You can find a complete translation of the `BankersQueue` module in Appendix C.

2.2.6 OpDecl

The `OpDecl` type is used to represent declarations of operator fixity:

```
data OpDecl = Op QName Fixity Int

data Fixity = InfixOp | InfixlOp | InfixrOp
```

These definitions are a straight-forward translation of Curry's `infix`, `infixl` and `infixr` constructs. As operator fixities are not relevant to type inference in FlatCurry and are not used anywhere else in this thesis, please refer to appendix C.4 of [Han+06] for more details.

Chapter 3

Type Inference

In this chapter we will briefly introduce the theory of type inference. The goal of type inference is to find the most general types of a given expression and all its subexpressions that leave the expression type correct (Section 1.5.2, [Han10]). To achieve this, we first insert *type variables* (as placeholders for the types we seek) into the expression and its subexpressions and then *generate type equations* (see Section 3.1) based on the languages' semantics. These equations are then solved to obtain a *type substitution* (Section 3.2), a process called *unification* (Section 3.3). This substitution can then be applied to the original expression to replace the type variables by the inferred types.

To be able to reason about the types of expressions, we introduce type variable annotations into Curry expressions, denoted by superscript. In the expression

`(`**`head`**$^{\tau_1}$ `x`$^{\tau_2}$`, 0`$^{\tau_3}$`)`$^{\tau_4}$

the overall tuple expression has type τ_4, while `head` has type τ_1, `x` has type τ_2 and `0` has type τ_3. Instead of type variables like τ_3, we may also sometimes insert concrete types, like `Int`:

`(`**`head`**$^{\tau_1}$ `x`$^{\tau_2}$`, 0`$^{\texttt{Int}}$`)`$^{\tau_4}$

3.1 Generating Equations

To infer the type of an expression, we first annotate that expression with *fresh* type variables to represent the unknown types we wish to infer. A *fresh* type variable is one that does not yet occur in the expression. Once we have inserted type variables, we can capture in *equations* what we know about how the types of the expression (and thus the type variables we just inserted) relate to each other based on the semantics of the programming language. For example, the type of the first argument expression of a function call must be equal to the type of that function's first formal parameter and the type of a branch must be equal to the overall type of the case expression.

As a concrete example, we introduce type variables into the expression `0 + 1`:

`(0`$^{\tau_1}$ `+`$^{\tau_4}$ `1`$^{\tau_2}$`)`$^{\tau_3}$

We know that the type of the first argument expression (the literal `0`) must be equal to the type of the `+` function's first formal parameter. Likewise, the type of the second argument expression (the literal `1`) must be equal to the type of `+`'s second formal parameter. Lastly, the type of the whole function application expression must be equal to the return type of the `+` function.

Every FlatCurry `FuncDecl` (see Section 2.2.5) contains the type of the function it defines; additionally, a FlatCurry program contains the types of all constructors it defines. Since we can load any Curry module's FlatCurry representation via the `readFlatCurry` function from the

`FlatCurry` module, we can look up the type of any function or constructor we might encounter in a FlatCurry program (see Section 4.3.1 for details).

Looking up the type of the + function, we find:

```
+ :: Int -> Int -> Int
```

Thus, $\tau_4 \doteq$ `Int -> Int -> Int`. Since, as explained above, we can assume knowledge about the types of all functions and constructors, we omit type variables and equations for functions and constructors from the examples below. The type of + also tells us that the type of the function call's first argument must be `Int`. The same is true for the call's second argument and the type of the call itself:

$$\begin{aligned} \tau_1 &\doteq \texttt{Int} \\ \tau_2 &\doteq \texttt{Int} \\ \tau_3 &\doteq \texttt{Int} \end{aligned}$$

Additionally, we know that the types of the two literals `0` and `1` must be `Int`:

$$\begin{aligned} \tau_1 &\doteq \texttt{Int} \\ \tau_2 &\doteq \texttt{Int} \end{aligned}$$

These two equations might seem redundant, but we can see that they are important when we consider the following (type-incorrect) expression:

`(0`$^{\tau_1}$ `+ True`$^{\tau_2}$`)`$^{\tau_3}$

If we just equated the literals `0` and `True` to the argument types expected by +, we would, as before, end up with the following equations:

$$\begin{aligned} \tau_1 &\doteq \texttt{Int} \\ \tau_2 &\doteq \texttt{Int} \\ \tau_3 &\doteq \texttt{Int} \end{aligned}$$

Only by additionally generating equations based on what we know about the types of the literals themselves can we detect the type conflict:

$$\begin{aligned} \tau_1 &\doteq \texttt{Int} \\ \tau_2 &\doteq \texttt{Bool} \end{aligned}$$

Our equations tell us that τ_2 is supposed to be equal to both `Int` and `Bool`, which means that there is a type conflict and the expression is not type correct.

3.1.1 A more involved example

For a more involved example that includes polymorphism, we will look at the following expression:

```
(head x, 0)
```

First, we need to introduce fresh type variables:

```
((head x^τ1)^τ2, 0^τ3)^τ4
```

Without looking up any types, we can see that τ_3 is equal to `Int`, as the expression is an integer literal.

Looking up the types of the tuple constructor and `head`, we find

```
(,) :: a -> b -> (a, b)
head :: [a] -> a
```

We replace the polymorphic type variables `a` and `b` by new ones according to our τ-notation:

```
(,) :: τ1 -> τ2 -> (τ1, τ2)
head :: [τ1] -> τ1
```

Before we can use these types to generate equations, we have to *rename* their polymorphic type variables so they do not clash with those in our original expression or each other; otherwise the same type variable number might end up referring to different types and the inference process will either fail or generate incorrect types. To do this, we just replace them by fresh ones. We need three fresh type variables and our original expression has type variable numbers going up to 4, so we can safely use the numbers 5, 6 and 7:

```
(,) :: τ5 -> τ6 -> (τ5, τ6)
head :: [τ7] -> τ7
```

With the types renamed, we can continue forming our equations. τ_3 must be equal to τ_6, as the `Int` literal is the second argument to the tuple constructor. Furthermore, the result of the call to `head`, τ_2, is the first argument to the tuple constructor, so τ_2 is equal to τ_5. τ_2 is also equal to τ_7.

We also know that `head`'s argument is a list of τ_7, which means that τ_1 must be equal to $[\tau_7]$. Finally, the type of the overall call to the tuple constructor is equal to the constructed tuple's type, so τ_4 is equal to (τ_5, τ_6). Summing up:

$$
\begin{aligned}
\tau_3 &\doteq \texttt{Int} \\
\tau_3 &\doteq \tau_6 \\
\tau_2 &\doteq \tau_5 \\
\tau_2 &\doteq \tau_7 \\
\tau_1 &\doteq [\tau_7] \\
\tau_4 &\doteq (\tau_5, \tau_6)
\end{aligned}
$$

The complete rules for generating equations based on FlatCurry's semantics will be given in the implementation section (4.3.3).

3.2 Substitutions

In general, let there be sets of constant symbols (C) and variables (V) and sets A_i of functions of arity i. Then a *term* is defined as follows ([MM82]):

(1) constant symbols and variables are terms

(2) if $t_1, ..., t_n$ ($n \geq 1$) are terms and $f \in A_n$, then $f(t_1, ..., t_n)$ is a term

A *substitution* σ is a mapping from variables to terms ([MM82]). We will use the notation from [Han10]:

$$\sigma = \{v_1 \mapsto t_1, v_2 \mapsto t_2, ...\}$$

A *substitution* σ can be applied to a *term* t by simultaneously substituting all occurrences of every *variable* v_i in t by $\sigma(v_i)$. For instance, given the *term* $t = f(g(v_1), v_2, h(v_3))$ and the substitution $\sigma = \{v_1 \mapsto j(v_3), v_2 \mapsto v_4, v_3 \mapsto v_3\}$, σ applied to t would yield the term

$$t' = f(g(j(v_3)), v_4, h(v_3))$$

We can apply these general definitions to FlatCurry when looking at type expressions (`TypeExpr`) as *terms*. A *variable* as defined above is a type variable (`TVar`), while the *functions* mentioned in (2) are type constructors (`TCons`) and function types (which can be interpreted as binary type constructors with a special name). Type constructors that take no arguments (such as `Int`'s constructor) can be interpreted as constant symbols.

3.3 Unification

Given a set of k equations

$$t_j \doteq s_j, j = 1, ..., k$$

any substitution that makes all $t_j, s_j, j \in \{1, ..., k\}$ equal to each other is called a *unifier* ([MM82]). A unifier σ of a set of equations E is called a *most general unifier* if all other unifiers are special cases of σ, specifically if for all unifiers ϕ of E there exists a substitution ψ such that $\phi(\tau) = \psi(\sigma(\tau))$ for all terms τ ([Han10]). The process of finding a most general unifier is called *unification*.

The following algorithm for unification is given by *Martelli* and *Montanari* in [MM82]. Given a set of equations E, keep applying the following transformations until none is applicable anymore, then stop with success:

(1) Select any equation of the form $t = x$ where x is a variable and t is not and reinsert it into the set as $x = t$ (swap x and t).

(2) Select any equation of the form $x = x$ where x is a variable and erase it from E.

(3) Select any equation of the form $t' = t''$, where t' and t'' are not variables. Let $g \in A_m, h \in A_k$ be functions, $x_1, ..., x_m, y_1, ..., y_k$ be terms and $t' = g(x_1, ...x_m), t'' = h(y_1, ...y_k)$. If $h \neq g$ or $m \neq k$, stop with failure (the two terms *clash*). Otherwise we insert $x_1 = y_1, ..., x_m = y_m$ into E and remove $t' = t''$ from the set.

(4) Select any equation of the form $x = t$ where x is a variable, $t \neq x$ and x occurs in some other equation in E. If x occurs in t, then stop with failure (we cannot further process a variable that occurs in the term it maps to). Otherwise, apply the substitution $\{x \mapsto t\}$ to every equation in E except $x = t$.

When done, translate every equation in E to a mapping in a substitution by taking the left-hand side (which due to (1) is always a variable) and mapping it to the right-hand side.

As an example, we will apply it to the equations generated in the previous section (3.1.1):

$$E = \{\tau_3 \doteq \texttt{Int}, \tau_3 \doteq \tau_6, \tau_2 \doteq \tau_5, \tau_2 \doteq \tau_7, \tau_1 \doteq [\tau_7], \tau_4 \doteq (\tau_5, \tau_6)\}$$

In the first round of applications, we apply (4) to $\tau_3 \doteq \texttt{Int}$ and $\tau_2 \doteq \tau_5$. We end up with:

$$E = \{\tau_3 \doteq \texttt{Int}, \texttt{Int} \doteq \tau_6, \tau_2 \doteq \tau_5, \tau_5 \doteq \tau_7, \tau_1 \doteq [\tau_7], \tau_4 \doteq (\tau_5, \tau_6)\}$$

Next, we apply (1) to $\texttt{Int} \doteq \tau_6$ and (4) to $\tau_5 \doteq \tau_7$:

$$E = \{\tau_3 \doteq \texttt{Int}, \tau_6 \doteq \texttt{Int}, \tau_2 \doteq \tau_7, \tau_5 \doteq \tau_7, \tau_1 \doteq [\tau_7], \tau_4 \doteq (\tau_7, \tau_6)\}$$

We can now apply (4) to $\tau_6 \doteq \texttt{Int}$:

$$E = \{\tau_3 \doteq \texttt{Int}, \tau_6 \doteq \texttt{Int}, \tau_2 \doteq \tau_7, \tau_5 \doteq \tau_7, \tau_1 \doteq [\tau_7], \tau_4 \doteq (\tau_7, \texttt{Int})\}$$

No more rules are applicable, which means we are done. We can now translate E to a substitution σ:

$$\sigma = \{\tau_3 \mapsto \texttt{Int}, \tau_6 \mapsto \texttt{Int}, \tau_2 \mapsto \tau_7, \tau_5 \mapsto \tau_7, \tau_1 \mapsto [\tau_7], \tau_4 \mapsto (\tau_7, \texttt{Int})\}$$

Although we have only defined what it means to apply a substitution to a term, extending that definition to our type-annotated Curry expressions defined above can be done intuitively: Apply the substitution to each type annotation. Applying σ to our example, we get its type-inferred version:

$((\textbf{head}\ \texttt{x}^{[\tau_7]})^{\tau_7},\ \texttt{0}^{\texttt{Int}})^{(\tau_7,\texttt{Int})}$

Chapter 4

Implementation

The implementation of the type inferrer is split into three Curry modules: *AnnotatedFlatCurry*, containing a version of FlatCurry datatypes that can be annotated with arbitrary data (type information in our case); *Unification*, a module implementing general data structures and functions for unifying equations; and *Inference*, the actual type inferrer containing functions for generating equations, renaming type variables, extracting type information from imports, interfacing with the unifier and so on. We will look at each of these modules, starting with AnnotatedFlatCurry.

4.1 AnnotatedFlatCurry

The *AnnotatedFlatCurry* module contains a version of FlatCurry's core datatypes that take an additional polymorphic type parameter to be used for arbitrary annotations.

```
data AProg a = AProg String [String] [TypeDecl] [AFuncDecl a] [OpDecl]

data AFuncDecl a = AFunc QName Int Visibility TypeExpr (ARule a)

data ARule a = ARule [VarIndex] (AExpr a)
             | External String

data AExpr a = AVar a VarIndex
             | ALit a Literal
             | AComb a CombType QName [AExpr a]
             | ALet a [(VarIndex, AExpr a)] (AExpr a)
             | AFree a [VarIndex] (AExpr a)
             | AOr a (AExpr a) (AExpr a)
             | ACase a CaseType (AExpr a) [ABranchExpr a]

data ABranchExpr a = ABranch (APattern a) (AExpr a)

data APattern a = APattern a QName [VarIndex]
                | ALPattern a Literal
```

The datatype's names are similar to the ones in *FlatCurry*, except for a leading `A` character (standing for *A*nnotated). The polymorphic type parameter for the annotation data is just passed down the type hierarchy to `AExpr`, which is the first time it is actually used in any of the constructors. When the type parameter is used in a constructor, it is always the constructor's first argument.

In the following sections we will be dealing with `TypeExpr` as the annotation type, as typed expressions are what our inferrer is interested in.

4.2 Unification

The *Unification* module offers datatypes for terms and substitutions (see Section 3.2) and functions to deal with these in addition to the core unification functionality. Specifically, the module has functionality for creating new substitutions, extending substitutions, combining substitutions, applying substitutions to terms and equations and, finally, unifying equations. For a complete description of all functions, see the interface description in appendix A.2.

Terms are represented by the following datatype, which is mostly a direct translation of the definition from Section 3.2, except that constant symbols are represented by constructors with empty argument lists:

```
type VarIdx = Int

data Term = TermVar VarIdx | TermCons String [Term]
```

To unify sets of equations, the module uses a Curry implementation of the ML implementation given in Section 4.7 of [BN98]. This algorithm is a straight-forward translation of the abstract algorithm description given in Section 3.3:

Start with the list of equations to be unified, E, and an empty substitution, σ. Keep applying the following transformations until E is empty:

(a) If the head of E equates a variable term x to a constructor term t, check whether x occurs in t. Stop with failure if it does. Otherwise, apply the substitution $\{x \mapsto t\}$ to the rest of E and to the right-hand sides of all elements of σ. Afterwards, add $\{x \mapsto t\}$ to σ and recursively continue unifying with the rest of E and the new σ. This combines (1) and (4) of the original algorithm description.

(b) If the head of E equates two variables, x and y, recursively continue unifying with the rest of E and the unchanged σ if $x = y$. Otherwise, apply the substitution $\{x \mapsto y\}$ to the rest of E and to the right-hand sides of all elements of σ, as in (a). Also add $\{x \mapsto y\}$ to σ and recursively continue unifying with the rest of E and the new σ. This combines (2) and (4) of the original algorithm description.

(c) If the head of E equates two term constructors, $f(t_1, ..., t_n)$ and $g(t'_1, ..., t'_m)$, and $f \neq g$ or $n \neq m$, stop with failure. If $f = g$, continue unifying with the equations $t_1 = t'_1, ..., t_m = t'_m$ prepended to E and the unchanged σ. This corresponds to (3) of the original algorithm description.

The core of this algorithm is implemented in the `unify'` and `elim` functions (in Listing 4.1) from the `Unification` module. Note that the `unify'` function is an internal function. It deals with two lists of equations instead of one list of equations and one substitution, because the `FiniteMap`-based substitutions in the `Unification` module, while very efficient for finding the term a variable maps to, are not well-suited for updating all terms they contain. `unify'` transforms the second list of equations into *solved form*, which means that the left-hand sides of all equations consist exclusively of variables. Thus, turning this list into a substitution is a very straight-forward task, taken care of by a user-facing wrapper function around `unify'` simply called `unify`. To avoid confusion, we will call the second list, the one that will be transformed into solved form and converted into a substitution, the *result list*.

The `elim` function substitutes a term for a variable inside all equations in the yet-to-unify list and the right-hand sides of all members of the result list. It also adds an equation between said

Listing 4.1. Core unification algorithm

```
data TermEq = (Term, Term)
data TermEqs = [TermEq]
data UnificationError = Clash TermEq | OccurCheck TermEq | Unexpected TermEq

unify' :: TermEqs -> TermEqs -> Either UnificationError TermEqs
unify' []                                    s = Right s
unify' (((TermVar i),    b@(TermCons _ _)):e) s = elim i b e s -- (a)
unify' ((a@(TermCons _ _), (TermVar i)):e)    s = elim i a e s -- (a)
unify' ((TermVar i,      b@(TermVar i')):e)   s | i == i' = unify' e s -- (b)
                                                | otherwise = elim i b e s -- (b)
unify' ((a@(TermCons f fts), b@(TermCons g gts)):e) s = if f == g
    then unify' ((zip fts gts) ++ e) s -- (c)
    else Left (Clash (a, b)) -- (c)

elim :: VarIdx -> Term -> TermEqs -> TermEqs -> Either UnificationError TermEqs
elim i t e s | (TermVar i) `occursIn` t = Left (OccurCheck (TermVar i, t))
             | otherwise = unify' (substitute i t e) ((TermVar i, t):s')
    where s' = map (\(x, y) -> (x, termSubstitute' i t y)) s
```

variable and said term to the result list. See alternatives (a) and (b) of the algorithm description above.

`occursIn` checks whether its first argument term appears in its second argument term and `substitute'` and `termSubstitute'` expect a variable and a term as arguments and substitute the term for the variable inside a list of equations or a single term, respectively.

While this version of the unification algorithm works in the sense that it leads to correctly unified equations and thus correctly inferred programs, it is rather slow. Inferring the `GUI` module from Curry's standard library takes roughly 38 seconds, which, even for one of the largest modules in the standard library, is too much for practical applications. The optimization presented below reduces the time needed to infer the `GUI` module to about 3.5 seconds, an improvement of one order of magnitude.

To improve performance, we could try to implement one of the more efficient unification algorithms available, for instance the near-linear time algorithm presented in [MM82]. However, in Section 7 of [DB95] the authors suggest that the initial overhead of many such asymptotically better algorithms leads to worse real-world performance on the sort of unification problems usually encountered in type inference. A possible improvement would be a variant of the algorithm implemented above operating on a graph-based representation of terms, as explained in Section 4.8 of [BN98]. We take a somewhat similar approach to speed up our algorithm. Parts of the implementation below are based on the *ML* implementation given in the appendix of [DB95].

In particular, we want to improve the following call inside `elim`:

```
unify' (substitute' i t e) ((TermVar i, t):s')
    where s' = map (\(x,y) -> (x, termSubstitute' i t y)) s
```

The problem with this call is that we have to iterate through both the list of equations that have yet to be unified (`e`) and the entire current result list (via the map on `s`) and then recursively walk through every term inside these. As the inferrer will generate quite a few equations even

for short functions, this incurs a significant cost. What we would like to be able to do is apply a substitution to every equation and every right-hand side inside the result list *without* having to look at each of them.

To this end, we introduce new internal datatypes for terms and equations:

```
data RTerm = RTermCons String VarIdx
           | RTermVar VarIdx
           | Ref VarIdx
type REq   = (RTerm, RTerm)
type REqs  = [REq]
```

`RTermCons` and `RTermVar` are exactly the same as `TermCons` and `TermVar`. `Ref` is a type used to create a *reference* to a type variable, the value of which is stored inside a `RefTable` (implemented using a `FiniteMap`, a datatype in Curry's standard library providing an efficient way to map keys to values). The `deref` function can be used to retrieve the value of a `Ref`; chained references are handled transparently:

```
type RefTable = FM Int RTerm
deref :: RefTable -> RTerm -> RTerm
deref t (Ref i) = case lookupFM t i of
    Just a -> case a of
        (RTermVar _)    -> a
        (RTermCons _ _) -> a
        (Ref _)         -> deref t a
    Nothing -> error "Deref failed!"
```

When a `Term` is converted to an `RTerm`, all `TermVars` are automatically changed into `Refs`:

```
termToRTerm :: RefTable -> Term -> (RefTable, RTerm)
termToRTerm r (TermVar i)    = (addToFM r i (RTermVar i), Ref i)
termToRTerm r (TermCons n l) = (r', RTermCons n l')
    where (r', l') = mapAccumL termToRTerm r l
```

We change the `unify'` function to work on `RTerms` instead of `Terms` and add a few rules to handle `Refs`. Those rules simply dereference the reference and recursively call `unify'` again. Additionally, `unify'` now expects and returns a `RefTable`:

```
unify' :: RefTable -> REqs -> REqs -> Either UnificationError (RefTable, REqs)
unify' r []                                    s = Right (r, s)
unify' r (((RTermVar i), b@(RTermCons _ _)):e) s = elim r i b e s
unify' r ((a@(RTermCons _ _), (RTermVar i)):e) s = elim r i a e s
unify' r ((RTermVar i, b@(RTermVar i')):e)     s | i == i' = unify' r e s
                                                 | otherwise = elim r i b e s
unify' r ((a@(RTermCons f fts), b@(RTermCons g gts)):e) s = if f == g
    then unify' r ((zip fts gts) ++ e) s
    else Left (Clash (rTermToTerm r a, rTermToTerm r b))
unify' r ((a@(Ref _), b@(RTermVar _)):e)       s = unify' r ((deref r a, b):e) s
unify' r ((a@(Ref _), b@(RTermCons _ _)):e)    s = unify' r ((deref r a, b):e) s
unify' r ((a@(Ref _), b@(Ref _)):e)            s = unify' r ((deref r a,
                                                              deref r b):e) s
unify' r ((a@(RTermVar _), b@(Ref _)):e)       s = unify' ((a, deref r b):e) s
unify' r ((a@(RTermCons _ _), b@(Ref _)):e)    s = unify' ((a, deref r b):e) s
```

While the changes to `unify'` are as expected and do not yield any improvements (if anything, the extra step of having to dereference should slightly decrease performance), the new version of `elim` in Listing 4.2 results in a large performance gain:

Listing 4.2. Improved elim function

```
elim :: RefTable -> VarIdx -> RTerm -> REqs -> REqs
        -> Either UnificationError (RefTable, REqs)
elim r i t e s | occursIn r (RTermVar i) t = Left (OccurCheck (TermVar i,
                                                               rTermToTerm r t))
               | otherwise = case t of
                    RTermVar i'   -> unify' (addToFM r i (Ref i')) e
                                            ((RTermVar i, (Ref i')):s)
                    RTermCons _ _ -> unify' (addToFM r i t) e
                                            ((RTermVar i, t):s)
```

The iteration through all equations and the right-hand sides of all elements of the result list has been replaced by changing the corresponding entry in the `RefTable` to the new element, with the special case that instead of changing an entry to an `RTermVar`, we change it to a reference to that term variable's entry in the `RefTable`.

To illustrate how the enhanced algorithm works, we will unify the following list of equations:

```
[(TermVar 1, TermVar 12),
 (TermVar 1, TermCons "Int" []),
 (TermVar 3, TermCons "List" [TermVar 1])]
```

The steps of the unification algorithm are given in Table 4.1, beginning with the conversion of the above list of equations to `RTerms`. Dereferencing everything inside the resulting list and converting it back to `Term` yields:

```
[(TermVar 3, TermCons "List" [TermCons "Int" []]),
 (TermVar 12, TermCons "Int" []),
 (TermVar 1, TermCons "Int" [])]
```

Table 4.1. Sample application of unification algorithm

Step No.	Equations	Substitution	RefTable
Step:	Initialization		

1	`[(Ref 1, Ref 12),`
	`(Ref 1, RTermCons "Int" []),`
	`(Ref 3, RTermCons "List" [Ref 1])]`
		`[]`	`[(1, RTermVar 1),`
			`(12, RTermVar 12),`
			`(3, RTermVar 3)]`

Step:	Deref inside `unify'`		

2	`[(RTermVar 1, RTermVar 12),`
	`(Ref 1, RTermCons "Int" []),`
	`(Ref 3, RTermCons "List" [Ref 1])]`
		`[]`	`[(1, RTermVar 1),`
			`(12, RTermVar 12),`
			`(3, RTermVar 3)]`

Step:	Update `RefTable` and add to substitution		

3	`[(Ref 1, RTermCons "Int" []),`
	`(Ref 3, RTermCons "List" [Ref 1])]`
		`[(RTermVar 1, Ref 12)]`	`[(1, Ref 12),`
			`(12, RTermVar 12),`
			`(3, RTermVar 3)]`

Step:	Deref inside `unify'`		

4	`[(Ref 12, RTermCons "Int" []),`
	`(Ref 3, RTermCons "List" [Ref 1])]`
		`[(RTermVar 1, Ref 12)]`	`[(1, Ref 12),`
			`(12, RTermVar 12),`
			`(3, RTermVar 3)]`

Step:	Deref inside `unify'`		

5	`[(RTermVar 12, RTermCons "Int" []),`
	`(Ref 3, RTermCons "List" [Ref 1])]`
		`[(RTermVar 1, Ref 12)]`	`[(1, Ref 12),`
			`(12, RTermVar 12),`
			`(3, RTermVar 3)]`

Step:	Update `RefTable` and add to substitution		

6	`[(Ref 3, RTermCons "List" [Ref 1])]`	`[(RTermVar 12, RTermCons "Int" []),`
		`(RTermVar 1, Ref 12)]`
			`[(1, Ref 12),`
			`(12, RTermCons "Int" []),`
			`(3, RTermVar 3)]`

Step:	Deref inside `unify'`		

7	`[(RTermVar 3, RTermCons "List" [Ref 1])]`	`[(RTermVar 12, RTermCons "Int" []),`
		`(RTermVar 1, Ref 12)]`
			`[(1, Ref 12),`
			`(12, RTermCons "Int" []),`
			`(3, RTermVar 3)]`

Step:	Update `RefTable` and add to substitution		

8	`[]`	`[(RTermVar 3, RTermCons "List" [Ref 1]),`
		`(RTermVar 12, RTermCons "Int" []),`
		`(RTermVar 1, Ref 12)]`
			`[(1, Ref 12),`
			`(12, RTermCons "Int" []),`
			`(3, RTermCons "List" [Ref 1])]`

4.3 Inference

The *Inference* module contains functionality for reading FlatCurry programs, generating fresh type variables, extracting type information from imported modules and interacting with the unifier. When asked to infer the types of a function or program, it takes the following steps to arrive at an inferred version of that function or program:

(1) Create a type environment by extracting type information from the program's imported modules.

(2) Convert the program to AnnotatedFlatCurry and annotate it with fresh type variables in the process.

(3) Generate equations for each function's body.

(4) Translate the equations into a form understood by the unifier.

(5) Unify the equations.

(6) Translate the terms inside the substitution back into FlatCurry types.

(7) Apply the substitution to the original expression.

(8) Regenerate the function type and normalize type variable numbers.

Steps (3) through (8) happen on a per-function basis even if the caller asks for an inferred version of the whole program. We will look at each step in order, starting with the creation of a type environment. Afterwards, we will discuss the handling of errors during the inference process.

A complete description of `Inference`'s interface can be found in appendix A.3.

4.3.1 Creating a Type Environment

A type environment is a mapping from qualified names to types. It is used to look up the types of known functions and constructors when generating equations. When a program is first loaded and the inferrer is not given a type environment to use, it creates a new one. To do this, it reads the *interface files* of all imported modules of the given program, which will contain all types needed to infer that program's types.

An *interface file* is simply a FlatCurry program with the bodies of all functions omitted to make it more compact and efficient to parse. Thus, it only contains function and datatype/constructor definitions (and operator fixities, but those do not interest us since they carry no type information). The type of a function is contained in its definition, so extracting it is simply a matter of looking at the `FuncDecl`. A little more work needs to be done for `TypeDecls`: If the type is a synonym, we can just insert a mapping from its qualified name to its target type. In case of a datatype, we create a function type for each of its constructors, where the constructor's parameter types are the function's parameter types and the datatype's type is the function's return type.

To illustrate how types are extracted from `TypeDecls`, we will look at the `Prelude`'s `Either` datatype as an example:

```
data Either a b = Left a | Right b
```

The corresponding TypeDecl in FlatCurry looks like this:

```
Type ("Prelude", "Either") Public [0, 1] [
    Cons ("Prelude", "Left") 1 Public [TVar 0],
    Cons ("Prelude", "Right") 1 Public [TVar 1]]
```

According to the process outlined above, we extract function types for the constructors `Left` and `Right`. The `Left` constructor's only argument is of type `TVar 0`. A call to the constructor results in the constructed type `TCons ("Prelude", "Either") [TVar 0, TVar 1]`. So the resulting function type looks like this:

```
FuncType (TVar 0) (TCons ("Prelude", "Either") [TVar 0, TVar 1])
```

Similarly, the `Right` constructor's only argument is of type `TVar 1` and it results in the same constructed type. So its function type is:

```
FuncType (TVar 1) (TCons ("Prelude", "Either") [TVar 0, TVar 1])
```

Regardless of whether a type environment is given to the inferrer as a parameter or newly created by the inferrer itself, it is always extended with the types from the module that is to be inferred. This step is needed to be able to infer functions that refer to constructors or other functions inside the same module (or themselves in case of recursive functions).

Type variable numbers in the types of functions and constructors are usually normalized by the front end when they are generated, so the type variables of most polymorphic functions start with the number 0. Just using these type variable numbers when looking up the types of functions and constructors in the type environment will lead to clashes, meaning that different types will end up being associated with the same type variable number. This will likely lead to either a failure to infer any types at all or a type-inferred expression with wrong types. As a consequence, we need to replace type variables in any looked-up type (of functions or constructors) with fresh ones to prevent these clashes.

This replacement is done by the `renameTVars` function. The process of replacing the type variables is simple: Replace every `TVar` with a fresh one, recursively refresh the type expressions of a `FuncType`, and recursively refresh the argument type expressions of a `TCons`. The only complexity is that we have to keep track of what `TVar` has been replaced by what fresh `TVar` and replace any other occurrence of the original `TVar` by the same fresh one.

Otherwise, if we did not keep track of already-replaced variables and just replaced every occurrence of any type variable by a fresh one, the tuple type

```
TCons ("Prelude", "(,)") [TVar 0, TVar 0]
```

would be renamed like this, assuming the next fresh type variable is 5:

```
TCons ("Prelude", "(,)") [TVar 5, TVar 6]
```

We have now lost valuable information: Whereas the former tuple type clearly specified that both components were of the same type, the new one has no such restrictions.

4.3.2 Converting and Annotating FlatCurry Programs

When a FlatCurry program is loaded, the inferrer needs to convert it to AnnotatedFlatCurry and insert fresh type variables as annotations to represent the unknown types that we wish to infer (see Section 3.1). Since the AnnotatedFlatCurry datatypes are similar to the FlatCurry datatypes except for the additional polymorphic type parameter at the beginning of each expression

(a `TypeExpr` in our case), this conversion is straightforward: Convert a function's top-level expression to AnnotatedFlatCurry, insert fresh type variables as the annotation and recursively convert all of its subexpressions.

Converting a function or program datatype is even easier, since we do not have to insert any type variables. For a program, we convert all functions, keep the other parameters of the datatype, and construct an annotated program using the converted functions and the original parameters. For a function, we can keep all of the original parameters when constructing its annotated version except for its rule. There are two alternatives when constructing a rule: external rules only take a string parameter which we can leave untouched; regular rules take an expression, which needs to be converted, and a list of variable indices, which we can use as-is when constructing the rule's annotated version.

The `annotate*TVar` functions are responsible for this conversion, starting with the top-level `annotateTVar`. Since we want to annotate the converted expressions with fresh type variables, we need a way to come up with ones. The easiest way and the one we use is to start with type variable 0 and keep counting. Whenever we need a fresh type variable number, we use the current count and increment it afterwards.

As Curry is a purely function programming language, there is no global location to store our count of variable numbers (like a static class member in *Java*). Instead, functions that need the count take the current count as a parameter and return a tuple of the new count and their actual return value. For example, the type signature of the `annotateExprTVar` function looks like this:

```
annotateExprTVar :: Int -> Expr -> IntState (AExpr TypeExpr)
```

Where `IntState a` is a type synonym for `(Int, a)`. The function's implementation is straightforward in simple cases such as the following:

```
annotateExprTVar n (Var i) = (n + 1, AVar (TVar n) i)
```

The type variable with number `n`, the current count, is used for the type variable inserted into the `Var` expression, so `n + 1` becomes the new count. However, this becomes more complex when we need to recursively translate subexpressions, which can be nested arbitrarily deeply themselves and thus have a need for an unknown number of fresh type variables. To translate the `Or` expression, for example, we need to first translate its first alternative expression and then, using the new count returned from that translation, translate the second alternative expression. We can then use the count returned from this second translation to generate the fresh type variable needed to annotate the `Or` itself. So something like this would not work:

```
annotateExprTVar n (Or a b) = (n + 1, AOr (TVar n)
                                          (annotateExprTVar n a)
                                          (annotateExprTVar n b))
```

To thread the variable number count through the various function calls, we can use Curry's `where` construct:

```
annotateExprTVar n (Or a b) = (n'' + 1, AOr (TVar n'') a' b')
    where (n', a')  = annotateExprTVar n a
          (n'', b') = annotateExprTVar n' b
```

This kind of implementation works, but becomes tedious to write and hard to read when we need to thread the variable count through more than a couple of function calls. Instead, we introduce a very simple operator to make threading more pleasant to write and more visually apparent:

```
(>-=) :: (b, a) -> (b -> a -> (b, c)) -> (b, c)
(>-=) (s, a) f = f s a
```

Using this and a few anonymous functions, the above example becomes:

```
annotateExprTVar n (Or a b) = annotateExprTVar n a >-= \n' a' ->
                              annotateExprTVar n' b >-= \n'' b' ->
                              (n'' + 1, AOr (TVar n'') a' b')
```

This is already enough for most use cases, but an important case that is not covered is mapping over a list while threading the count through all function calls. We can define a `mapState` function to do this for us:

```
mapState :: (s -> b -> (s, c)) -> s -> [b] -> (s, [c])
mapState _ s [] = (s, [])
mapState f s (x:xs) = (s'', x':xs')
    where (s', x')   = f s x
          (s'', xs') = mapState f s' xs
```

Using this function, we can easily implement the translation of combinations, where we need to translate each parameter expression:

```
annotateExprTVar n (Comb t q es) = mapState annotateExprTVar n es >-= \n' es' ->
                                   (n' + 1, AComb (TVar n') t q es')
```

This functionality is enough to implement all conversion functions from FlatCurry to AnnotatedFlatCurry. The rest of the rules are very similar to the examples presented.

4.3.3 Rules for Generating Equations

We will look at the rules for generating equations abstractly before discussing their Curry implementation. In the examples below, we define the `typeof` function to be the type of its argument expression as specified in its constructor. So `typeof(AVar (TVar 0) 1)` would be `TVar 0`. Please note that `typeof` is the type of the argument expression as specified in its *constructor*, not the type that will eventually be inferred for the expression.

AVar

```
AVar TypeExpr VarIndex
```

Variables are easy: since we only have the information about the variable's type given to us in `AVar`'s first argument, we cannot generate any equations that would encapsulate new information.

ALit

```
ALit TypeExpr Literal
```

The type of a literal expression is the type of the literal it represents. In FlatCurry, this can be either an `Int`, a `Char` or a `Float`. Thus, the type of the `ALit` expression must be equal to the type of its `Literal`. For instance, the expressions

```
ALit (TVar 0) (Intc 0)
ALit (TVar 0) (Charc 'a')
ALit (TVar 0) (Floatc 0.1)
```

Would yield the following equations, respectively:

```
▷ TVar 0 ≐ TCons ("Prelude", "Int") []
```

```
▷ TVar 0 ≐ TCons ("Prelude", "Char") []
```

```
▷ TVar 0 ≐ TCons ("Prelude", "Float") []
```

AOr

```
AOr TypeExpr (AExpr TypeExpr) (AExpr TypeExpr)
```

An `AOr` expression introduces non-determinism, meaning it can evaluate to none, one or both of its argument expressions. As an expression is of exactly one type, both of `AOr`'s alternative expressions must be of the same type. That type must also be the type of the `AOr` itself. This yields three equations: One that equates the type of the `AOr` to the type of its first alternative expression, one that equates the type of the `AOr` to its second alternative expression, and one that equates the types of both alternative expressions to one another. Additionally, we recursively generate equations for each alternative expression. For example:

```
AOr (TVar 0) e1 e2
```

would yield the following equations, in addition to the equations recursively generated for `e1` and `e2`:

$$\begin{aligned} \texttt{TVar 0} &\doteq \texttt{typeof(e1)} \\ \texttt{TVar 0} &\doteq \texttt{typeof(e2)} \\ \texttt{typeof(e1)} &\doteq \texttt{typeof(e2)} \end{aligned}$$

AComb

```
AComb TypeExpr CombType QName [AExpr TypeExpr]
```

A combination is a call to a function or a constructor, partially (some of the callee's formal parameters missing) or fully applied. In either case the type of the callee can be represented by a FlatCurry `FuncType`, as constructors can be interpreted as functions that take the constructor's parameters as their formal parameters and return a value of the datatype the constructor belongs to.

We can use this function type to find the types of the expressions in the `AComb`'s parameter list: The type of the i-th expression is the type of the i-th formal parameter. Or, when expressed in terms of `FuncTypes`: The type of the i-th expression is the type of the first parameter to the `FuncType` at the i-th level. Furthermore, if the list of parameter expressions has length k, then the type of the call's return value, and thus the type of the overall `AComb` expression, is the type of the second argument of the `FuncType` at the k-th level. Because of the nesting nature of `FuncTypes`, this also takes care of partial calls because in that case the second argument of the `FuncType` at the k-th level would simply be another `FuncType`. For an illustration, consider the following expression:

```
AComb (TVar 0) FuncCall ("Example", "fourArgs") [e1, e2, e3]
```

Assuming that the type of `fourArgs` is

```
FuncType (TVar 1) -- first level
    (FuncType (TVar 2) -- second level
        (FuncType (TVar 3) -- third level
            (FuncType (TVar 4) (TVar 5)))) -- fourth level
```

we can use the above method to conclude that the type of `e1` must be the type of the first argument to the first level `FuncType`, namely `TVar 1`. By the same rule, the type of `e2` must be the type of the first argument to the second level `FuncType`, namely `TVar 2`. The same rule applies to `e3`, so its type must be `TVar 3`. As we only have three arguments, the type of the overall `AComb` must be the type of the second argument to the third level `FuncType`, which is `FuncType (TVar 4) (TVar 5)`. This yields the following equations in addition to any equations generated recursively for `e1`, `e2` and `e3`:

```
typeof(e1)  ≐  TVar 1
typeof(e2)  ≐  TVar 2
typeof(e3)  ≐  TVar 3
    TVar 0  ≐  FuncType (TVar 4) (TVar 5)
```

ALet

```
ALet TypeExpr [(VarIndex, AExpr TypeExpr)] (AExpr TypeExpr)
```

`ALet` introduces a set of variable-to-expression bindings local to another expression, its *inner expression*. Inside this inner expression, any `AVar` referencing a variable present in `ALet`'s bindings evaluates to the expression that variable is bound to. Thus, the types of those `AVar`s must be the same as the types of the expressions their respective variables are bound to by the `ALet`. If variable *k* is bound to the expression *ek*, then we can generate the equation `TVar 0` ≐ `typeof(ek)` for any `AVar (TVar 0) k` inside the inner expression.

As the `ALet` itself evaluates to its inner expression, its type must be the same as that of the inner expression. Furthermore, we can recursively generate equations for each expression in `ALet`'s bindings and its inner expression.

So, for the following expression

```
ALet (TVar 0) [(1, e1), (2, e2)]
    (AOr (TVar 1) (AVar (TVar 2) 1) (AVar (TVar 3) 2))
```

we can generate the following equations, in addition to any equations generated recursively for `e1`, `e2` and the inner `AOr` expression:

```
TVar 0  ≐  TVar 1
TVar 2  ≐  typeof(e1)
TVar 3  ≐  typeof(e2)
```

ABranch and APattern

```
ABranch (APattern TypeExpr) (AExpr TypeExpr)
data APattern TypeExpr = APattern TypeExpr QName [VarIndex]
                       | ALPattern TypeExpr Literal
```

We have to distinguish between branches with literal patterns and branches with regular constructor patterns. In case of an `ALPattern`, a literal pattern, we can only generate one equation in addition to the equations recursively generated for the branch's expression: The type of the pattern itself must be equal to the type of its `Literal`, similarly to an `ALit` expression.

Constructor patterns, `APatterns`, as explained in Section 2.2.4, match any value that was constructed by the constructor identified by the qualified name passed to the pattern. Additionally, the matched value is deconstructed: The variable with the index at the i-th position in the list of variable indices passed to the pattern is bound to the i-th parameter originally passed to the constructor of the matched value. As an example, given the list `[1, 2, 3]` as the variable indices, a constructor pattern with constructor `(,,)` (the 3-tuple constructor) would match the value `(a, b, c)` and bind the variable with index 1 to expression a, the variable with index 2 to expression b, and the variable with index 3 to expression c. This binding is in effect when evaluating the parent branch's expression. Similarly to `ALet`, the type of every variable reference inside the branch's expression that references one of the variables bound by the deconstruction process must be equal to the type of the expression that variable is bound to.

Since variables are bound to parameters originally passed to the pattern's constructor, we can derive their types from the type of the constructor using the rule described in the above section on `AComb`: A constructor type is represented by a function type (`FuncType`). The type of the variable with the index at the i-th position of the pattern's variable index list is the same as the type passed as the first parameter to the i-th level `FuncType` in the constructor's function type.

Similarly to combinations, the type of the overall pattern is the same as the type of the second parameter of the k-th level `FuncType` in the constructor's function type, where k is the number of variable indices given in the pattern's variable index list. In a well-formed FlatCurry program, this should always be the constructor's return type, which is the datatype it constructs.

Thus, we can generate the following equations for branches with constructor patterns, in addition to any equations generated recursively for the branch's expression:

- ▷ The type of every reference to a variable whose number appears at index i of the pattern's variable index list must be equal to the type computed for the i-th variable by the process outlined above (for variable references inside the branch's expression).
- ▷ The type of the overall pattern must be equal to the type that remains after all bound variables have been matched to their types by the process outlined above.

For instance,

```
ABranch (APattern (TVar 0) ("Prelude", ":") [1,2])
        (AVar (TVar 1) 1)
```

would yield the following equations, assuming that we know that the type of `:` is a function type from `TVar 3` to a list of `TVar 3` to a list of `TVar 3`:

```
TVar 0  ≐  TCons ("Prelude", "[]") [TVar 3]
TVar 1  ≐  TVar 3
```

ACase

```
ACase TypeExpr CaseType (AExpr TypeExpr) [ABranchExpr TypeExpr]
```

We can equate the type of the overall `ACase` to each of the types of the branches' expressions. We also equate the type of the subject expression to the types of each of the branches' patterns. Furthermore, we recursively generate equations for the subject expression and each of the branches.

For example, the following `ACase` with two branches,

```
ACase (TVar 0) subj [
    ABranch (APattern (TVar 2) ("Prelude", "Nothing") []) e1,
    ABranch (APattern (TVar 3) ("Prelude", "Just") [2]) (AVar (TVar 4) 2)]
```

could yield these equations in addition to any equations recursively generated for the subject, the expression `e1` and the inner expression of the second branch:

```
      TVar 0  ≐  typeof(e1)
      TVar 0  ≐  TVar 4
typeof(subj)  ≐  TVar 2
typeof(subj)  ≐  TVar 3
      TVar 2  ≐  TCons ("Prelude", "Maybe") [TVar 6] -- (see section on ABranch)
      TVar 3  ≐  TCons ("Prelude", "Maybe") [TVar 6] -- (see section on ABranch)
      TVar 4  ≐  TVar 6 -- (see section on ABranch)
```

AFree

```
AFree TypeExpr [VarIndex] (AExpr TypeExpr)
```

An `AFree` introduces free variables into an expression, but does not offer any insight into the types of those variables. Thus, the only thing we know with certainty is that the type of the `AFree` itself must be equal to the type of its inner expression. For example,

```
AFree (TVar 0) [1] (AVar (TVar 1) 1)
```

yields the equation `TVar 0` ≐ `TVar 1`.

4.3.4 Implementing the Rules in Curry

Most of the rules in the previous section are implemented inside the `genPairs` function (see Listing 4.3). As with all function definitions in this section, error and state handling have been omitted to keep the examples short and uncluttered.

There are a few functions that `genPairs` depends on. Some of these are easily explained:

- `lookupType` looks up a type inside a type environment and replaces all type variables with fresh ones (see Section 4.3.2 for details on how this is done)
- `literalType` returns the type of a literal; either `Int`, `Float` or `Char`
- `typeExpr` extracts the `TypeExpr` from an `AExpr TypeExpr`
- `matchCombType` does the matching from parameter types to a function type explained in Section 4.3.3

Listing 4.3. Function for generating equations

```
type TypeEnv = FM QName TypeExpr
type Equations = [(TypeExpr, TypeExpr)]
genPairs :: TypeEnv -> AExpr TypeExpr -> Equations
genPairs env (AComb ex _ f ps)    = map (genPairs env) ps ++
                                    matchCombType ex (lookupType env f) ps
genPairs env (ACase ce _ subj bs) = map (genBranchPairs env ce subj) bs ++
                                    genPairs env subj
genPairs env (AVar _ _)           = []
genPairs env (ALit t l)           = [(t, literalType l)]
genPairs env (AOr t a b)          = genPairs env a ++
                                    genPairs env b ++
                                    [(typeExpr a, typeExpr b), (typeExpr a, t),
                                     (typeExpr b, t)]
genPairs env (ALet t b e)         = genVarPairs env
                                                (map (\(a, b) -> (a, typeExpr b)) m)
                                                e ++
                                    map (genPairs env . snd) b ++
                                    genPairs env e ++
                                    [(t, typeExpr e)]
genPairs env (AFree t _ e)        = genPairs env e ++ [(t, typeExpr e)]
```

We are left with `genBranchPairs` and `genVarPairs`. As the former depends on the latter, we will start with `genVarPairs`:

```
genVarPairs :: TypeEnv -> [(VarIndex, TypeExpr)] -> AExpr TypeExpr -> Equations
genVarPairs env vs (AComb _ _ _ ps) = map (genVarPairs env vs) ps
genVarPairs env vs (ACase _ _ s bs) = map (genBranchVarPairs env vs) bs ++
                                      genVarPairs env vs s
genVarPairs _   vs (AVar e k)       = case (lookup k vs) of
    Just a  -> [(e, a)]
    Nothing -> []
genVarPairs _   _  (ALit _ _)       = []
genVarPairs env vs (AOr _ a b)      = genVarPairs env vs a ++ genVarPairs env vs b
genVarPairs env vs (ALet _ m e)     = map (genVarPairs env vs . snd) m ++
                                      genVarPairs env vs e
genVarPairs env vs (AFree _ _ e)    = genVarPairs env vs e
```

This function recursively searches an expression for variables occurring in the binding from variables to types given in its second argument - as can be seen in the rule matching `AVar` expressions. When it finds such a variable, the type of that `AVar` expression is equated to the type in the binding. This functionality is used in generating equations for `ALet` expressions and pattern deconstruction.

The `genBranchVarPairs` function used in the `ACase` rule above just evaluates `genVarPairs` for the branch's expression. That leaves us with `genBranchPairs`:

```
genBranchPairs :: TypeEnv -> TypeExpr -> AExpr TypeExpr
                  -> ABranchExpr TypeExpr -> Equations
genBranchPairs env t subj (ABranch pat@(APattern pt f _) e) = genPairs env e ++
    genVarPairs env (patternVarTypes ct pat) e ++
    [(typeExpr subj, patternType ct pat), (t, typeExpr e),
     (pt, patternType ct pat)]
    where ct = lookupType env f
genBranchPairs env t (ABranch (ALPattern pt l) e) = genPairs env e ++
    [(typeExpr subj, literalType l), (t, typeExpr e), (pt, literalType l)]
```

This function generates equations for a branch. If that branch has a literal pattern, it recursively generates equations for the branch's expression and equates the parent case's subject type to the type of the literal, the parent case's overall type to the type of the branch's expression and the type of the pattern to the type of the literal.

If the branch has a regular pattern, the function also recursively generates equations for its expression. It uses the functions `patternVarTypes` and `patternType` to generate bindings from the variables used in deconstruction to their types and return the type of the pattern constructor, respectively (see Section 4.3.3 for details). The bindings are used to generate equations for all occurrences of the variables used in deconstruction inside the branch's expression via `genVarPairs`. It equates the pattern's constructor result type to the type of the pattern itself and to the parent case expression's subject type. Finally, it equates the overall type of the parent case expression to the branch expression's type.

In short, `genPairs` is responsible for generating equations and calls out to `genVarPairs` to handle variable bindings, `genBranchPairs` to handle branches and `matchCombType` to handle function calls.

4.3.5 Interfacing with the Unifier

The unification module works with `Terms` and not `TypeExprs`, which means that we have to find a way to convert a type expression into a term and vice versa. Luckily, the definitions of terms and type expressions are structurally similar:

```
data Term = TermVar VarIdx
          | TermCons String [Term]

data TypeExpr = TVar TVarIndex
              | TCons QName [TypeExpr]
              | FuncType TypeExpr TypeExpr
```

As `VarIdx` and `TVarIndex` are both type synonyms for `Int`, we can easily map `TVar` to `TermVar`. To map `TCons` to `TermCons`, we have to convert `QName` to a regular string and back. Joining the two parts of a `QName` by a semicolon (`;`) character does the trick - according to appendix C.1 of [Han+06] the semicolon is not a valid character for either identifiers or operator names in Curry.

This leaves us with `FuncType`, which we can translate to a `TermCons` with name `->`. Even if a function of this name were to occur in a FlatCurry program, it could only appear inside a `TypeExpr` as a qualified name, implying that it would always be prefixed by a module name; or at the very least the semicolon character used to separate the two parts of a qualified name when translating `TCons` to `TermCons`.

Once the conversion to `Term` is done, we can hand the equations to the unifier. The terms inside the resulting substitution are converted back into type expressions when the substitution is applied to the inferred expression, as describe in the next section.

4.3.6 Applying a Substitution to an Expression

To apply a substitution to an expression, we recursively look at every type variable occurrence. If it exists in the substitution, we translate the looked-up `Term` back to a `TypeExpr` and substitute that for the type variable. If it does not exist in the substitution, we leave the type variable alone.

To translate a `Term` back into a `TypeExpr`, we use the following rules:

- ▷ If the term is a `TermVar`, we simply translate it into a `TVar` with the same number.
- ▷ If the term is a `TermCons` with name `->`, we make sure that it has exactly two parameters and translate these into `TypeExpr`. We then use the translated parameters as the arguments to the resulting `FuncType`.
- ▷ If the term is a `TermCons` with any name other than `->`, we split its name at the first occurrence of the semicolon character and use the resulting two parts to create a qualified name (the first part is used as the module name). We recursively translate the parameters to `TypeExpr` and construct a `TCons` with the generated qualified name and the converted parameters as arguments.

In the implementation, we define the function `lookupConvertDefault` to look up a type variable in the substitution and convert it if it is found or just return the original type variable if it is not. Constructed types and function types are returned as-is. `termToTypeExpr` is simply a straight-forward implementation of the rules presented above to convert a `Term` to a `TypeExpr`.

```
lookupConvertDefault :: TypeExpr -> Substitution -> TypeExpr
lookupConvertDefault e@(TVar n)       sub = case lookupSubstitution sub n of
    Just v  -> termToTypeExpr v
    Nothing -> e
lookupConvertDefault o@(TCons _ _)    _   = o
lookupConvertDefault f@(FuncType _ _) _   = f
```

This makes the implementation of `substTypes`, the top-level function responsible for applying a substitution to an expression, very straightforward to both read and write. For example, the rule for `AComb` looks like this:

```
substTypes :: Substitution -> AExpr TypeExpr -> AExpr TypeExpr
substTypes sub (AComb e t f ps) = AComb (lookupConvertDefault e sub) t f
                                        (map (substTypes sub) ps)
```

4.3.7 Generating a Function Type and Normalizing Type Variables

After the equations have been generated and unified and the inferred types have been substituted into the original program or function, we renumber the type variables starting from 0. This process is called *normalization* and is done to make the inferred program easier to read. We also regenerate the function type inside each function definition to make sure that any type variables inside it line up with their counterparts in the function's body. The `extractFuncType` function

is responsible for generating the function type, while `normalizeFunc` and `normalizeExpr` handle type variable normalization.

Type variable normalization is a straightforward process: recursively walk through the expression and for each type variable encountered either assign to it a new number starting from 0 if we have not seen it before, or assign to it whatever number we assigned at a previous occurrence. This means that we have to keep track of the next free type variable number and a mapping from type variables to type variables for the ones we have already encountered. When normalizing a complete function, we first normalize the type given in its `FuncDecl` to make sure that the type variables used inside that are always assigned the lowest numbers.

To extract a function's type given its body, we use the variables declared as its formal parameters inside the `ARule` definition. We look up the types of those variables inside the function's body and make them the types of the function's parameters. If a parameter is not used, the corresponding variable will not be found inside the function's body. In this case, we generate a fresh type variable and use that as the type of that particular function parameter. The type of the top-level expression becomes the function's return type.

```
-- Curry equivalent: a x _ = x + 5
AFunc ("Example", "a") 2 Public (TVar 42) (ARule [1, 2]
    (AComb (TCons ("Prelude", "Int") []) FuncCall ("Prelude", "+") [
        AVar (TCons ("Prelude", "Int") []) 1,
        ALit (TCons ("Prelude", "Int") []) (Intc 0)]))
```

In the above function definition, for instance, the first formal parameter's variable number is referenced in an `AVar` whose type has been inferred to be `TCons ("Prelude", "Int") []`. Thus, we know that the type of the function's first formal parameter must also be `TCons ("Prelude" , "Int") []`. Likewise, the top-level expression has type `TCons ("Prelude", "Int") []`, which tells us that this is also the function's return type. The variable with number 2, which is the variable number assigned to the function's second formal parameter in the `ARule`, however, does not appear anywhere inside the function's body. We assign it a fresh type variable, 0, to make it a polymorphic parameter, since we do not have any information about its type. The generated function type becomes:

```
FuncType (TCons ("Prelude", "Int") [])
         (FuncType (TVar 0) (TCons ("Prelude", "Int") []))
```

In practice, a freshly generated function type can be more general than the type specified in the original FlatCurry program. For instance, the *PAKCS* ([Han+12b]) compiler's front end sometimes adds helper functions that return parts of a constructed value when translating from Curry to FlatCurry. One such function could simply return the first component of a pair:

```
helper (a, _) = a
```

The compiler's front end uses the type information available to it when generating the function to speed up type inference. If the helper functions is generated in the context of a integer-tuple, it will have the following type in its FlatCurry representation:

```
helper :: (Int, Int) -> Int
helper (a, _) = a
```

The type inferrer, however, will generate the following, more general type:

```
helper :: (a, b) -> a
helper (a, _) = a
```

4.3.8 Handling Errors

During equation generation and unification, we might run into errors like clashes during unification or missing type information in the type environment. When such an error occurs, we need a way to inform our caller that it occurred and what kind of error it is.

The most straight-forward way to do this is using an `Either` return type:

```
data Either a b = Left a | Right b
```

We use the `Left` constructor to signal an error and the `Right` constructor to wrap the actual return value in case of success. All errors returned from functions in the Inference module are `Strings`, so we define a type synonym to make type signatures more compact:

```
type StrErr a = Either String a
```

The `genPairs` function calls functions that look up types in a type environment, an operation that can fail, so we need to incorporate `StrErr` into its type signature. As type variables in looked-up function types are replaced with fresh ones, `genPairs` also makes use of `IntState` to keep track of the type variable count (see Section 4.3.2).

```
genPairs :: TypeEnv -> Int -> AExpr TypeExpr -> IntState (StrErr Equations)
```

In the simple case, including support for error handling inside the rules is trivial:

```
genPairs _ n (AVar _ _) = (n, Right [])
```

As discussed above, a lone variable does not give us any valuable type information, so we do not need to generate a rule. A variable also does not have any subexpressions, so no recursive calls to `genPairs` that could generate errors are necessary. All we need to do is wrap the result inside the `Right` constructor. As soon as we call `genPairs` recursively though, we have to check every such call for a possible error return value. `AOr` makes for a good example:

```
genPairs tab n (AOr e a b) = case (genPairs tab n a) of
    (Left err) -> Left err
    (Right (n', a')) -> case (genPairs tab n' b) of
        (Left err) -> Left err
        (Right (n'', b')) -> Right (a' ++ b' ++ [(typeExpr a, typeExpr b),
            (typeExpr a, e), (typeExpr b, e)])
```

Clearly, this nesting of `cases` will lead to complicated and unreadable code sooner or later. We can define a simple operator to make error handling implicit (and handle threading the type variable count at the same time):

```
(>+=) :: (b, StrErr a) -> (b -> a -> (b, StrErr c)) -> (b, StrErr c)
(>+=) (s, (Left err)) _ = (s, Left err)
(>+=) (s, (Right a)) f = f s a
```

Using this operator, the above example becomes:

```
genPairs tab n (AOr e a b) = genPairs tab n a >+= \n' a' ->
                             genPairs tab n' b >+= \n'' b' ->
                             (n'', a' ++ b' ++ [(typeExpr a, typeExpr b),
                                                (typeExpr a, e),
                                                (typeExpr b, e)])
```

We can go one step further and also make appending the lists of resulting equations into one overall result list implicit using the following operator:

```
(++=) :: (b, StrErr [a]) -> (b -> (b, StrErr [a])) -> (b, StrErr [a])
(++=) (s, Left err) _ = (s, Left err)
(++=) (s, Right xs) = let (s', r) = f s in case r of
    (Left err) -> (s', Left err)
    (Right xs') -> (s', Right (xs ++ xs'))
```

With this implicit error-handling, count-threading and list-appending operator, the above rule of `genPairs` can be written like this:

```
genPairs tab n (AOr e a b) = genPairs tab n a ++= \n' ->
                             genPairs tab n' b ++= \n'' ->
                             (n'', [(typeExpr a, typeExpr b),
                                    (typeExpr a, e),
                                    (typeExpr b, e)])
```

There are a few other helper functions to help with handling errors, but they are all based on the concepts presented here.

4.4 Testing the Inference Process

The `InferTester` module is supplied in the source distribution of the inferrer as a means of automatically testing the `Inference` module and the unifier for correctness. To this end, the `InferTester` module contains functionality for inferring the types of a Curry module and comparing the result to a manually type-inferred version of the same program.

These manually type-inferred programs are stored alongside the original Curry files inside the `test` subdirectory of the inferrer's source distribution in files ending in `.tfcy` (for *t*yped *F*latCurr*y*). `InferTester`'s `main` function automatically tests each `.tfcy` file in the `test` directory and reports whether its contents match the inferrer's result when applied to the corresponding Curry program.

The programs supplied in the source distribution's test folder are taken from the *PAKCS* ([Han+12b]) examples and make use of many of Curry's constructs.

To compare the manually inferred FlatCurry programs to the automatically inferred ones, we have to define what it means for two FlatCurry programs to be equal with regard to their types. We start with the equality of two expressions with regard to types:

(1) They share the same constructor (`AVar`, `ALet`, `AOr`, ...), and

(2) their types are equal, and

(3) the rules in Table 4.2 hold based on the expression's constructor

To fully understand this definition, we need to define what it means for two FlatCurry types to be equal to each other in our context. Two function types are equal to each other if their argument type expressions are equal. Two constructed types are equal to each other if their qualified names are equal and equal types appear in equal positions inside their lists of argument types (and both have an equal number of argument types).

Table 4.2. Rules for comparing expressions

Kind	Rules
`AVar`	the variable numbers are equal
`ALit`	the constructor and value of the literal are equal
`AComb`	the combination type and qualified name are equal; both have the same number of argument expressions and equal argument expressions occur in the same order
`ALet`	the bindings are equal, meaning that exactly the same variable numbers occur in exactly the same order and they are bound to equal expressions; the inner expressions are equal
`AFree`	the lists of free variable numbers are exactly the same, meaning the same numbers occur in the same order
`AOr`	the first inner expressions are equal and the second inner expressions are equal
`ACase`	the case types and subject expressions are equal; both have the same number of branches and equal branches occur in the same order
`ABranchExpr`	the patterns are equal and the inner expressions are equal
`APattern`	the qualified names are equal and the lists of variable numbers are exactly the same, meaning the same numbers occur in the same order
`ALPattern`	the kind and value of the literal are equal

When are two type variables equal to each other, for example `TVar 0` and `TVar 1`? To answer this question, we look at two examples. The following two constructed types are equal as it does not matter whether the type variable number of their parameter is 0 or 1.

```
TCons ("Prelude", "[]") [TVar 0]
TCons ("Prelude", "[]") [TVar 1]
```

The following two function types, however, are not. The first is a function that takes parameters of one polymorphic type and another polymorphic type and returns a value of the first polymorphic type, while the second is a function that takes parameters of one polymorphic type and another polymorphic type and returns values of yet another, different polymorphic type:

```
FuncType (TVar 0) (FuncType (TVar 2) (TVar 0))
FuncType (TVar 2) (FuncType (TVar 3) (TVar 1))
```

The examples show that whether two type variables are equal depends on whether one or both of them have been seen before in the current scope. In the examples the scope is just the type expression, when comparing whole programs it is the current function. So in the first example we can conclude that `TVar 0` is indeed equal to `TVar 1`. In the second example we already know that the type represented by `TVar 0` in the first function type is the same as the type represented by `TVar 2` in the second function type. We can therefore conclude that the two function types are not equal when we encounter `TVar 0` as the return type of the first and `TVar 1` as the return type of the second. After first encountering the types of `TVar 0` and `TVar 2` as being equal, a `TVar 0` in the first type expression can never be equal to anything but a `TVar 2` in the second one and vice versa.

So when deciding whether two type variables with numbers a (in the first expression) and b (in the second expression) are equal to each other, we have to follow these rules:

(1) If a has been encountered in the first expression and b has not been encountered in the second expression, then they are not equal

(2) If b has been encountered in the second expression and a has not been encountered in the first expression, then they are not equal

(3) If a has not been encountered in the first expression and b has not been encountered in the second expression, then they are equal

(4) If a has been encountered in the first expression and b has been encountered in the second expression and the previous occurrence of a was equal to b, then they are equal

(5) If a has been encountered in the first expression and b has been encountered in the second expression and the previous occurrence of a was not equal to b, then they are not equal

After this somewhat complicated definition of what it means for two types to be equal, we have to define what it means for two functions to be equal:

(1) Both functions have the same name, and

(2) the functions have equal types, and

(3) the functions have equal bodies

Now all that is left to define is what it means for two programs to be equal:

(1) Both programs have the same number of functions, and

(2) the functions, when compared in order, are equal to each other (so the first function of the first program is equal to the first function of the second program and so on)

4.5 Performance

The inferrer's original purpose is to check optimized programs for type-correctness, making it part of the compilation process at the very least during development and testing of said optimizations. As such it is worthwhile to take a look at how much of a burden it places on compilation speed. We will measure the performance of the inference process for a large module from the Curry standard library, both for the whole module and a single function. Additionally, we will measure the time needed to compile each of these modules using the *KiCS2* compiler.

As the time required to print out the result of the computation can be significant, leading to measurements that are not meaningful when compared to each other, we define the `forceAndDiscard` helper function.

```
forceAndDiscard :: a -> IO Int
forceAndDiscard d = do
    r <- return $!! d
    return 0
```

`forceAndDiscard` forces the evaluation of its argument to normal form using the $!! operator, circumventing Curry's lazy evaluation, and returns the integer value 0, which takes only little time to print.

Measurements using `forceAndDiscard` will be more suited to comparison with each other than measurements taken without it, but evaluating the result of a computation to normal form will still take more time for a more complex result. We have to be careful when comparing the time needed for computations with results of different complexities.

The following measurements were made using version 0.2.1 of the *KiCS2* compiler ([Han+12a]). For each measurement we started *KiCS2*, loaded the `Inference` module, set the `+time` option telling *KiCS2* to measure the time taken to evaluate an expression and then evaluated the expression in question ten times.

We measured the time needed to compile a module by copying that module's Curry source file from the *KiCS2* library directory to another directory and running the following command ten times, removing the `.curry` subdirectory before each iteration to get rid of any cached results from the compilation process:

```
time kics2 :load modname :quit
```

The raw data of all measurements taken below can be found in Appendix B.

4.5.1 The `GUI` Module

The `GUI` module is one of the largest modules in the Curry standard library. We measured the time taken to infer the whole module, to create a type environment, to simply read its FlatCurry version using `readFlatCurry` from the `FlatCurry` module and to infer a single one of the functions it defines. The single function we chose was `menu2tcl`, since it is of about average complexity compared to the other functions in the module.

First we inferred the whole module using the following expression:

```
readFlatCurry "GUI" >>= progWithTypes >>= forceAndDiscard
```

This took an average of 2.928 seconds with a standard deviation of 0.016 seconds. Just creating the type environment using

```
readFlatCurry "GUI" >>= extractTypeEnv >>= forceAndDiscard
```

took 1.896 seconds on average, with a standard deviation of 0.023 seconds. As discussed above, we have to take the complexity of the computation's results into account when comparing these two figures. Measured simply in the bytes consumed by their `String` representations, the inferred program is about an order of magnitude larger than the type environment (1,247,440 bytes versus 115,760 bytes). So we can safely conclude that evaluating the inferred program to normal form will, on average, take at least as much time as evaluating the type environment to normal form. This means that the actual process of generating equations and unifying them for all of `GUI`'s functions took at most 1.032 seconds, the difference between the average time measured for the creation of the type environment and the average time taken for the inference of the `GUI` module.

We measured how much of the time taken for the creation of the type environment was being used on reading the FlatCurry program using the following expression:

```
readFlatCurry "GUI" >>= forceAndDiscard
```

This yielded an average of 1.806 seconds with a standard deviation of 0.026 seconds. To accurately compare this figure to the time taken to create the type environment, we again have to take into account the complexities of the computation's results. In terms of the size of their `String` representations, the FlatCurry program is about four times larger than the type

environment (480,461 bytes versus 115,760 bytes). This is not all that much of a size difference, which means it is probably safe to conclude that generating the type environment took about 0.1 seconds.

The same problem plagues our measurements of the time taken to infer the `menu2tcl` function. The `String` representation of the inferred function weighs in at 6,357 bytes, more than an order of magnitude less than the 115,760 bytes taken up by the type environments `String` representation. This explains why evaluating the expression

```
readFlatCurry "GUI" >>= functionWithTypes ("GUI", "menu2tcl") >>= forceAndDiscard
```

took an average of only 1.881 seconds with a standard deviation of 0.016 seconds, 0.015 seconds *less* than it took to create the type environment. The only thing we can somewhat safely conclude from this measurement is that the time needed to infer `menu2tcl` should be somewhere in the sub-0.1 second range.

Compiling the `GUI` module using *KiCS2* took an average of 5.8254 seconds with a standard deviation of 0.026 seconds.

4.5.2 The `Char` Module

The `Char` module is a rather small module in the Curry standard library. As with `GUI`, we measured the time to infer the whole module, to create a type environment, to read its FlatCurry representation and to infer a single function. The single function we chose for `Char` is `intToDigit`.

Inferring the whole module using

```
readFlatCurry "Char" >>= progWithTypes >>= forceAndDiscard
```

took an average of 0.155 seconds (standard deviation 0.007 seconds). Only creating the type environment using

```
readFlatCurry "Char" >>= extractTypeEnv >>= forceAndDiscard
```

took 0.140 seconds on average (standard deviation 0.004 seconds). Since the results of both operations are about the same size (27,152 bytes for the inferred program versus 29,738 bytes for the type environment) and there is only a 0.015 second difference between both times, we can conclude that the time actually spent on inferring types is very small.

Only reading the `Char` module using `readFlatCurry`,

```
readFlatCurry "Char" >>= forceAndDiscard
```

took an average of 0.041 seconds (standard deviation 0.003 seconds). There is about a 0.1 second difference between the time needed to read the FlatCurry program and the time needed to read the FlatCurry program and create a type environment. Even though the type environment is more complex in terms of data structure size than the FlatCurry program, it is safe to say that for the `Char` module most of the time is not spent on inference or reading the program, but on creating the type environment.

Inferring only the `intToDigit` function took 0.149 seconds on average, with a standard deviation of 0.007 seconds, roughly as long as inferring the complete module and thus reinforcing the observation that for such a small module the time spent on the actual type inference is rather small compared to the time spent on generating the type environment.

Compiling the module using *KiCS2* took an average of 0.5748 seconds with a standard deviation of 0.007 seconds.

Chapter 5

Conclusions

The goal of this thesis was the development of a type inferrer for FlatCurry programs. As outlined in the general description of type inference in Chapter 3, this posed the following main problems:

(1) Finding a way to associate type information with FlatCurry expressions.

(2) Generating equations based on FlatCurry's semantics.

(3) Solving the resulting equations using unification.

We developed a version of the datatypes used to represent FlatCurry programs that supports arbitrarily annotated expressions. This allows us to associate type information with FlatCurry expressions by directly storing the type information as an annotation, which makes for a very simple programming interface: given a FlatCurry program, the inferrer will return a version of the same FlatCurry program that has been annotated with the calculated types or a string with an error message if the program contains type errors.

To solve the second problem, generating equations based on our knowledge about the types in an expression, we first devised rules on how to generate equations for the different kinds of FlatCurry expressions and then implemented functions that use these rules to generate the equations (see Sections 4.3.3 and 4.3.4).

We solved the third problem, unification of the resulting equations, by developing a general-purpose unification module operating on a datatype for terms. After making some performance measurements, we modified the algorithm to make use of a datatype for references, which improved performance by roughly an order of magnitude.

The inferrer was automatically tested by inferring sample programs and comparing the results to versions that had been manually annotated with type information. Additionally, we measured the performance of the finished program. In its current form, the inferrer should be fast enough to be integrated into *KiCS2* for the testing of program transformations during development. Especially for large modules, however, it might prove too slow to be permanently integrated into the compilation process. There is some potential for performance improvements:

- ▷ Type environments for commonly used modules could be cached, especially for the `Prelude`, as it is rather large and will not change frequently.
- ▷ The process of generating equations could be improved by eliminating some redundancies in the way that bindings are handled for `Let` and `Pattern`.
- ▷ Although the authors of [DB95] state that the initial overhead of unification algorithms with near-liner time complexity leads to worse real-time performance for the problems typically encountered in type inference, the article is rather old. It might be worth investigating whether an implementation of a near-linear unification algorithm leads to better performance.

Appendix A

Interface Description

A.1 AnnotatedFlatCurry

The `AnnotatedFlatCurry` module defines datatypes for an annotated version of the FlatCurry intermediate language.

Exported types

`data AProg a`

Datatype for a FlatCurry module.

▷ `AProg :: String -> [String] -> [TypeDecl] -> [AFuncDecl a] -> [OpDecl] -> AProg a`
`AProg mod imps types funcs ops`
Defines the module named `mod`, which imports modules `imps` and defines types `types`, functions `funcs` and operator fixities `ops`.

`data AFuncDecl a`

Datatype for FlatCurry function definitions.

▷ `AFunc :: QName -> Int -> Visibility -> TypeExpr -> ARule a -> AFuncDecl a`
`AFunc name arity visi type rule`
Defines the function named `name` with arity `arity`, visibility `visi`, type `type` and rule `rule`

`data ARule a`

Datatype for FlatCurry function rules.

▷ `ARule :: [VarIndex] -> AExpr a -> ARule a`
`ARule parms expr`
Defines the a rule with body expression `expr`. The variables specified in `vars` are bound to the corresponding function's parameters inside the body.

▷ `External :: String -> ARule a`
`External name`
An externally defined rule with name `name`.

`data AExpr a`

Datatype for FlatCurry expressions.

▷ `AVar :: a -> VarIndex -> AExpr a`
`AVar a i`
Defines an expression that evaluates to the value of the variable with index `i`, carrying annotation `a`.

▷ `ALit :: a -> Literal -> AExpr a`
`ALit a l`
A literal expression that evaluates to the value given in `l`. Carries the annotation `a`.

▷ `AComb :: a -> CombType -> QName -> [AExpr a] -> AExpr a`
`AComb a type name args`
A call to the function or constructor (which of the two is specified in `type`) named `name` with arguments `args`. Carries the annotation `a`.

▷ `ALet :: a -> [(VarIndex, AExpr a)] -> AExpr a -> AExpr a`
`ALet a bs expr`
Let expression; evaluates `expr` with the bindings `bs` in effect. Carries the annotation `a`.

▷ `AFree :: a -> [VarIndex] -> AExpr a -> AExpr a`
`AFree a vs expr`
Evaluates `expr` with the variables in `vs` declared as free variables. Carries the annotation `a`.

▷ `AOr :: a -> AExpr a -> AExpr a -> AExpr a`
`AOr a ex1 ex2`
Evaluates the expressions `ex1` and `ex2` non-deterministically. Carries the annotation `a`.

▷ `ACase :: a -> CaseType -> AExpr a -> [ABranchExpr a] -> AExpr a`
`ACase a type subj bs`
Rigid or flex (depends on `type`) case expression that chooses one of the branches in `bs` based on the form of `subj`.

`data ABranchExpr a`

Datatype for branches in case expressions.

▷ `ABranch :: APattern a -> AExpr a -> ABranchExpr a`
`ABranch pat expr`
A branch that is chosen if `pat` matches the parent case's subject expression; evaluates to `expr` if chosen.

`data APattern a`

Datatype for patterns.

▷ `APattern :: a -> QName -> [VarIndex] -> APattern a`
`APattern a name vs`
A pattern that matches any value constructed by the constructor with name `name`. The original arguments to the constructor are bound to the variables specified in `vs` when the parent branch's expression is evaluated. Carries the annotation `a`.

▷ `ALPattern :: a -> Literal -> APattern a`
`ALPattern a l`
A pattern that matches exactly the value specified in the literal `l`. Carries the annotation `a`.

A.2 Unification

The `Unification` module defines datatypes for terms and substitutions and provides functionality for unifying equations between these terms.

Exported types

`type VarIdx = Int`

The type used for variable term numbers.

`data Term`

Datatype for representing terms.

▷ `TermVar :: VarIdx -> Term`
`TermVar i`
A variable term with number `i`.

▷ `TermCons :: String -> [Term] -> Term`
`TermCons n as`
A constructor term with name `n` and arguments `as`.

`type TermEq = (Term, Term)`

The type used for equations between terms.

`type TermEqs = [TermEq]`

The type used for multiple equations.

`data UnificationError`

Datatype for the different kinds of errors that can occur during unification.

▷ `Clash :: TermEq -> UnificationError`
`Clash (a, b)`
The two constructor terms `a` and `b` with different constructors are supposed to be equal.

▷ `OccurCheck :: TermEq -> UnificationError`
`OccurCheck (a, b)`
Term `a` occurs in term `b` but the two are supposed to be equal.

▷ `Unexpected :: TermEq -> UnificationError`
`Unexpected (a, b)`
An unexpected error occurred when unifying the two terms `a` and `b`.

`type Substitution`

The abstract datatype for substitutions.

Exported functions

`emptySubstitution :: Substitution`

The empty substitution.

`extendSubstitution :: (VarIdx, Term) -> Substitution -> Substitution`

Extends a substitution by a mapping from a variable to a term.

`makeSubstitution :: VarIdx -> Term -> Substitution`

Creates a substitution with the single mapping provided.

`combineSubstitutions :: Substitution -> Substitution -> Substitution`

Combines two substitutions.

`lookupSubstitution :: Substitution -> VarIdx -> Maybe Term`

Searches the substitution for a mapping from the given variable index to a term.

`substitute :: Substitution -> TermEqs -> TermEqs`

Applies a substitution to a list of equations.

`substituteSingle :: Substitution -> TermEq -> TermEq`

Applies a substitution to an equation.

```
applySubstitution :: Substitution -> Term -> Term
```

Applies a substitution to a term.

```
unify :: TermEqs -> Either UnificationError Substitution
```

Unifies a list of equations.

A.3 Inference

The `Inference` module contains functionality for inferring FlatCurry programs.

Exported types

```
type TypeEnv
```

A datatype for type environments.

```
type StringError a = Either String a
```

Datatype used for return values of functions that can fail.

Exported functions

```
extractTypeEnv :: Prog -> IO TypeEnv
```

Generates a type environment for a FlatCurry program.

```
progWithTypes :: Prog -> IO (StringError (AProg TypeExpr))
```

Infers the types of all expressions of all functions of a FlatCurry program.

```
functionWithTypes :: QName -> Prog -> IO (StringError (AFuncDecl TypeExpr))
```

Infers the types of all expressions of one function of a FlatCurry program.

```
progWithTypesUsingTypeEnv :: TypeEnv -> Prog -> StringError (AProg TypeExpr)
```

Infers the types of all expressions of all functions of a FlatCurry program. Uses the passed type environment instead of generating one on its own.

```
functionWithTypesUsingTypeEnv :: TypeEnv -> QName -> Prog -> StringError (AFuncDecl TypeExpr)
```

Infers the types of all expressions of one function of a FlatCurry program. Uses the passed type environment instead of generating one on its own.

Appendix B

Raw Performance Data

Table B.1. Performance figures for the `GUI` module

whole module	type environment	`readFlatCurry`	`menu2tcl`	Compilation using *KiCS2*
2.90*s*	1.90*s*	1.79*s*	1.87*s*	5.871*s*
2.87*s*	1.90*s*	1.80*s*	1.86*s*	5.825*s*
2.94*s*	1.92*s*	1.84*s*	1.89*s*	5.777*s*
2.89*s*	1.88*s*	1.80*s*	1.89*s*	5.848*s*
2.98*s*	1.87*s*	1.77*s*	1.88*s*	5.823*s*
2.94*s*	1.94*s*	1.84*s*	1.87*s*	5.839*s*
2.98*s*	1.90*s*	1.83*s*	1.90*s*	5.805*s*
2.92*s*	1.91*s*	1.82*s*	1.88*s*	5.828*s*
2.96*s*	1.88*s*	1.81*s*	1.91*s*	5.796*s*
2.90*s*	1.86*s*	1.76*s*	1.86*s*	5.842*s*
$\bar{x}$ = 2.928*s*	$\bar{x}$ = 1.896*s*	$\bar{x}$ = 1.806*s*	$\bar{x}$ = 1.881*s*	$\bar{x}$ = 5.8254*s*
σ = 0.036*s*	σ = 0.023*s*	σ = 0.026*s*	σ = 0.016*s*	σ = 0.026*s*

Table B.2. Performance figures for the `Char` module

whole module	type environment	`readFlatCurry`	`intToDigit`	Compilation using *KiCS2*
0.16*s*	0.14*s*	0.04*s*	0.15*s*	0.584*s*
0.16*s*	0.14*s*	0.04*s*	0.14*s*	0.565*s*
0.16*s*	0.14*s*	0.04*s*	0.14*s*	0.584*s*
0.16*s*	0.14*s*	0.04*s*	0.15*s*	0.564*s*
0.15*s*	0.14*s*	0.04*s*	0.15*s*	0.570*s*
0.16*s*	0.15*s*	0.04*s*	0.14*s*	0.574*s*
0.15*s*	0.14*s*	0.04*s*	0.16*s*	0.571*s*
0.16*s*	0.13*s*	0.04*s*	0.15*s*	0.575*s*
0.14*s*	0.14*s*	0.04*s*	0.16*s*	0.579*s*
0.15*s*	0.14*s*	0.05*s*	0.15*s*	0.582*s*
$\bar{x}$ = 0.155*s*	$\bar{x}$ = 0.14*s*	$\bar{x}$ = 0.041*s*	$\bar{x}$ = 0.149*s*	$\bar{x}$ = 0.5748*s*
σ = 0.007*s*	σ = 0.004*s*	σ = 0.003*s*	σ = 0.007*s*	σ = 0.007*s*

Appendix C

BankersQueue in FlatCurry

```
Prog "BankersQueue" ["Prelude"] [
  Type ("BankersQueue", "Queue") Public [0] [
    Cons ("BankersQueue", "Q") 4 Public [
      TCons ("Prelude", "[]") [TVar 0],
      TCons ("Prelude", "Int") [],
      TCons ("Prelude", "[]") [TVar 0],
      TCons ("Prelude", "Int") []]]] [
  Func ("BankersQueue", "empty") 0 Public
       (TCons ("BankersQueue", "Queue") [TVar 0])
       (Rule [] (
    Comb ConsCall ("BankersQueue", "Q") [
      Comb ConsCall ("Prelude", "[]") [],
      Lit (Intc 0),
      Comb ConsCall ("Prelude", "[]") [],
      Lit (Intc 0)])),
  Func ("BankersQueue", "isEmpty") 1 Public
       (FuncType (TCons ("BankersQueue", "Queue") [TVar 0])
                 (TCons ("Prelude", "Bool") []))
       (Rule [1] (
    Case Flex (Var 1) [
      Branch (Pattern ("BankersQueue", "Q") [2, 3, 4, 5]) (
        Comb FuncCall ("Prelude", "==") [
          Var 3,
          Lit (Intc 0)])])),
  Func ("BankersQueue", "queue") 4 Public
       (FuncType (TCons ("Prelude", "[]") [TVar 0])
                 (FuncType (TCons ("Prelude", "Int") [])
                           (FuncType (TCons ("Prelude", "[]") [TVar 0])
                                     (FuncType (TCons ("Prelude", "Int") [])
                                               (TCons ("BankersQueue", "Queue") [TVar 0])))))
       (Rule [1, 2, 3, 4] (
    Case Rigid (Comb FuncCall ("Prelude", "<=") [Var 4, Var 2]) [
      Branch (Pattern ("Prelude", "True") []) (
        Comb ConsCall ("BankersQueue", "Q") [Var 1, Var 2, Var 3, Var 4]),
      Branch (Pattern ("Prelude", "False") []) (
        Case Rigid (Comb FuncCall ("Prelude", "otherwise") []) [
          Branch (Pattern ("Prelude", "True") []) (
            Comb ConsCall ("BankersQueue", "Q") [
              Comb FuncCall ("Prelude", "++") [
                Var 1,
                Comb FuncCall ("Prelude", "apply") [
```

```
                Comb FuncCall ("Prelude", "reverse") [],
                Var 3]],
            Comb FuncCall ("Prelude", "+") [Var 2, Var 4],
            Comb ConsCall ("Prelude", "[]") [],
            Lit (Intc 0)]),
        Branch (Pattern ("Prelude", "False") []) (
          Comb FuncCall ("Prelude", "failed") [])])])),
Func ("BankersQueue", "append") 2 Public
     (FuncType (TCons ("BankersQueue", "Queue") [TVar 0])
               (FuncType (TVar 0) (TCons ("BankersQueue", "Queue") [TVar 0])))
     (Rule [1, 2] (
  Case Flex (Var 1) [
    Branch (Pattern ("BankersQueue", "Q") [3, 4, 5, 6]) (
      Comb FuncCall ("BankersQueue", "queue") [
        Var 3,
        Var 4,
        Comb ConsCall ("Prelude", ":") [Var 2, Var 5],
        Comb FuncCall ("Prelude", "+") [Var 6, Lit (Intc 1)]])])),
Func ("BankersQueue", "head") 1 Public
     (FuncType (TCons ("BankersQueue", "Queue") [TVar 0]) (TVar 0))
     (Rule [1] (
  Case Flex (Var 1) [
    Branch (Pattern ("BankersQueue", "Q") [2, 3, 4, 5]) (
      Case Flex (Var 2) [
        Branch (Pattern ("Prelude", "[]") []) (
          Comb FuncCall ("Prelude", "error") [
            Comb ConsCall ("Prelude", ":") [
              Lit (Charc 'E'),
              Comb ConsCall ("Prelude", "[]") []]]),
        Branch (Pattern ("Prelude", ":") [6, 7]) (Var 6)])])),
Func ("BankersQueue", "tail") 1 Public
     (FuncType (TCons ("BankersQueue", "Queue") [TVar 0])
               (TCons ("BankersQueue", "Queue") [TVar 0]))
     (Rule [1] (
  Case Flex (Var 1) [
    Branch (Pattern ("BankersQueue", "Q") [2, 3, 4, 5]) (
      Case Flex (Var 2) [
        Branch (Pattern ("Prelude", "[]") []) (
          Comb FuncCall ("Prelude", "error") [
            Comb ConsCall ("Prelude", ":") [
              Lit (Charc 'E'),
              Comb ConsCall ("Prelude", "[]") []]]),
        Branch (Pattern ("Prelude", ":") [6, 7]) (
          Comb FuncCall ("BankersQueue", "queue") [
            Var 7,
            Comb FuncCall ("Prelude", "-") [Var 3, Lit (Intc 1)],
            Var 4,
            Var 5])])])] []
```

Bibliography

[BN98] Franz Baader and Tobias Nipkow. Term rewriting and all that. Cambridge University Press, 1998. ISBN: 978-0-521-45520-6.

[Cro06] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational). Internet Engineering Task Force, July 2006. URL: http://www.ietf.org/rfc/rfc4627.txt.

[DB95] Dominic Duggand and Frederick Bent. "Explaining type inference". In: *Science of Computer Programming*. 1995, pp. 37–83.

[Han+06] M. Hanus et al. Curry, An Integrated Functional Logic Language, Version 0.8.2. 2006.

[Han+12a] M. Hanus, B. Braßel, B. Peemöller, and F. Reck. KiCS2. 2012. URL: http://www-ps.informatik.uni-kiel.de/kics2/.

[Han+12b] M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS. 2012. URL: http://www-ps.informatik.uni-kiel.de/~pakcs/.

[Han10] M. Hanus. Deklarative Programmiersprachen, Lecture Notes CAU Kiel. 2010.

[KVI02] Mahmut Kandemir, N. Vijaykrishnan, and Mary Jane Irwin. "'Power aware computing'". In: ed. by Robert Graybill and Rami Melhem. Norwell, MA, USA: Kluwer Academic Publishers, 2002. Chap. Compiler optimizations for low power systems, pp. 191–210. ISBN: 0-306-46786-0. URL: http://dl.acm.org/citation.cfm?id=783060.783071.

[Lip11] Miran Lipovaca. Learn You a Haskell for Great Good!: A Beginner's Guide. 1st. San Francisco, CA, USA: No Starch Press, 2011. ISBN: 1593272839, 9781593272838.

[MM82] Alberto Martelli and Ugo Montanari. "An efficient unification algorithm". In: *ACM Trans. Program. Lang. Syst.* 4.2 (1982), pp. 258–282. ISSN: 0164-0925.

[Oka98] Chris Okasaki. Purely Functional Data Structures. New York, NY, USA: Cambridge University Press, 1998.

[OSG08] Bryan O'Sullivan, Don Stewart, and John Goerzen. Real World Haskell. O'Reilly Media, 2008.

[Pey+03] Simon Peyton Jones et al. "The Haskell 98 Language and Libraries: The Revised Report". In: *Journal of Functional Programming* 13.1 (2003), pp. 0–255.