Input/Output in Functional Languages

Using Algebraic Union Types

Name	R.J. Rorije
Address	Beneluxlaan 28
	7543 XT Enschede
E-mail	r.j.rorije@student.utwente.nl
Study	Computer Science
Track	Software Engineering
Chair	Formal Methods and Tools

Input/Output in Functional Languages Using Algebraic Union Types

By

R.J. Rorije

Under the supervision of:

dr. ir. Jan Kuper
ir. Philip Hölzenspies
dr. ir. Arend Rensink

University of Twente

Enschede, The Netherlands

June, 2006

Contents

Abstract

Functional programming languages are not directly suitable to program I/O. In this thesis different models of I/O in functional languages are discussed with their advantages and disadvantages. Comparing these advantages and disadvantages results in the observation that the stream I/O-model is - at least from an educational point of view - the most appropriate model, because it does not create any new constructs to the language and the I/O-model is easy to understand. One problem in the stream I/O model is that event types in the form of algebraic types in the language are fixed.

In this thesis I will show that it is possible to add a new type construction that can combine algebraic data types in untagged unions. The benefit of these unions is, that from the perspective of the programmer the union behaves as if it were defined as the list of all constructors of both types. Now the union can be used as a normal algebraic type. In order to add this union construct to a functional programming language, new typing rules must be introduced. Obviously subtyping is closely related to unions, since values of the constituents of a union are also values of the (composite) union. therefore subtyping rules must be added to the type system. It will be shown that using these type rules the intuitive notion of unions being commutative, associative and idempotent are preserved.

To implement this union construct in a functional language, the language must be extended to handle union types. Therefore two rewriting schemes are given in this thesis. A naïve implementation for the language Amanda and a more extensive one using some features that are particular to Haskell.

Lastly the use of unions in the stream I/O model is shown with two examples, one using the naïve rewriting scheme and one using the more extensive rewriting scheme. In particular is shown that events can be defined separately in libraries and can the events from this libraries can be combined in a union to use them in a program.

Acknowledgements

I would like to acknowledge the help of many people during my study and my thesis. First I would like to thank my supervisors, Jan Kuper, Philip Hölzenspies and Arend Rensink. Jan for his enthusiasm and his knowledge about all kinds of mathematical subjects and the willingness to educate on that at almost any time. Furthermore I am very grateful for his input about how this thesis could be written best. Philip for all his new ideas and knowledge to combine different ideas like the use of transformational patterns which I would not have linked to this subject by myself. Also for all his knowledge about the English language, which helped me a lot. I would like to thank Arend for his support and relevant remarks on my thesis and for creating the opportunity to do this assignment in the first place. My thanks also goes to all members of the FMT chair that inspired me with new thoughts during the presentation I gave during their free lunch break. Especially Theo Ruijs who was willing to spend an entire afternoon talking about this subject and directing it more to type classes.

Last but not least I want to thank Nick Tinnemeier for his comments and his knowledge about functional languages and the (written) English language.

1 Introduction

In computing, a lot of work is being done to create computers that can interact with humans in a fluent way. Operating systems, for instance, are programs that react on input given by the programmer. It is obvious that the mere existence of most programs is based on run-time interaction with its users. This users can be human, but can also be other programs. When programs behave in an autistic way, i.e. there is no interaction with the "outside" world whatsoever, the program most likely is not very useful.

From a software engineering perspective programming is not only about having all necessary constructions in the language to create programs that have all necessary functionality, but also about how easy is it to create, extend and maintain programs. The focus in this thesis therefore is not to create new functionality in the language, but to increase the extendability of programs in the language by means of libraries.

Input and output is difficult to achieve in lazy functional programming languages, due to referential transparency and the unknown evaluation order. In this thesis I will compare different models to handle input and output in lazy functional programming languages. Not only the abstract model, but also an implementation in a functional programming language is discussed.

Focus will be on one of these models, the stream I/O model. One of the problems with this approach is that it is not possible to extend the event types, modeled by an algebraic type. Therefore a new construction will be added to the language which allows the combination of algebraic types in a union. This union can be used to create libraries of events that can be developed independently of each other.

In the next chapters two rewriting schemes are given to rewrite a language with unions to a language without unions. These rewriting schemes are a systematic way of rewriting the language which can be automated easily. An example preprocessor that implements these rewriting rules is implemented and shows how these rewriting rules can be implemented.

In the example that is given it is shown that the use of unions in stream based I/O make it possible to create more abstract events that can be used just like any other event. The programming effort to extend the event types is reduced to a minimum as will be shown.

Models to handle I/O in functional languages, along with their implementations will be discussed in Chapter 2. To extend the use of the stream I/O model in Chapter 3 a new construction will be introduced to extend a functional programming language. Chapter 4 describes a way to implement this in a functional language. An example of the use of this construction in combination with the stream I/O model is given in Chapter 6. In Chapter 5 another way to implement this construction in a functional programming language. Lastly future research will be discussed in Chapter 7. An actual implementation in the language Amanda is given in Appendix D.

2

Input/Output in Functional Languages

2.1 Programming Languages in General

There are several paradigms on which programming languages are founded, the two most important paradigms are the imperative and declarative programming paradigm. The difference between imperative and declarative languages is the way they address a problem. Imperative languages focus on *how* a problem must be solved, whereas declarative languages focus on *what* the solution is like. SQL is a declarative language for instance, because queries are not given in terms of *how* to find data, but instead give criteria for the desired data. An outline of the classification of programming languages is given in Figure 2.1. Most programming languages fit nicely into this categorization, however there are some exotic languages, that try to combine the best of both worlds and therefore do not fit in one box. Examples of such languages are Curry (Hanus, Kuchen, & Moreno-Navarro, 1995), which is a hybrid language of the functional and logic paradigm; OCaml (Leroy, 2004), which combines functional, logic and object oriented features; and Oz (Müller, Müller, & Roy, 1995), which uses multiple paradigms: functional, logic, imperative, OO and others. Yet, in this thesis I will focus on non-hybrid functional programming languages.

2.2 Functional Programming Languages

Functional programming languages (FPLs) are less known than imperative or object-oriented languages, therefore I will discuss some features that are particular to FPLs and do not have an equivalent in the better known imperative programming languages. One feature of FPLs is that functions in general do not have side-effects. The evaluation of a function can not have any other ef-

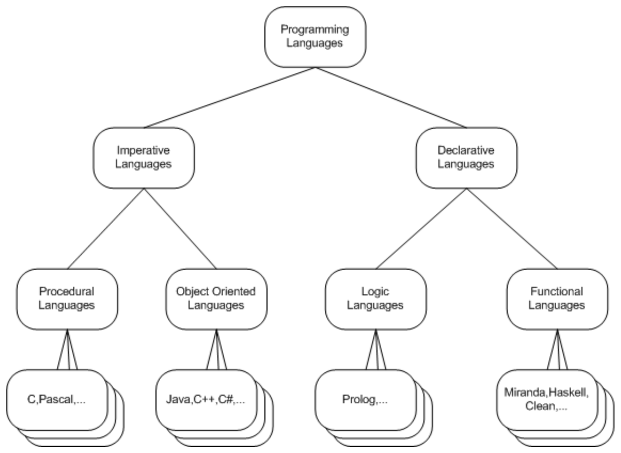


Figure 2.1: (Not extensive) Overview of Programming Languages

fect than to compute its result. This eliminates a major source of bugs and makes the evaluation order in this respect irrelevant (Hughes, 1989). Therefore, proofs about the correctness of programs are easier. On the other hand, this creates some problems also, as will be discussed below. Furthermore the use of functions as first-class citizens and the use of lazy evaluation enhances the applications of functional programming languages. Several functional programming language are lazy, i.e. expressions are only evaluated when the result is needed. This evaluation strategy is based on the call-by-name evaluation strategy, which substitutes function arguments directly into the function body. Call-by-name is used almost only in theory, because arguments are re-evaluated each time they are used. The programming languages I will address in this chapter all use a memoized form of the call-by-name evaluation strategy, named call-by-need. This means that functions are evaluated once instead of multiple times, when possible. Not all functional programming languages behave in a lazy way, there are languages that behave in a strict way, i.e. expressions are evaluated when they are found (eager). Examples of the latter are the functional programming languages ML and Erlang.

Most Code examples throughout this thesis are written in Amanda. Yet in some cases, when language specific examples are given, this will be in Haskell(98) or occasionally Clean.

Another important feature of functional programming languages is that functions can be passed as first class citizens to other functions. In Code 2.1 this is shown, the function `double` is passed as a first class citizen, as an argument, to the function **map**. The function **map** will evaluate the function `double` for each element of the list.

```
doubleList :: [num] -> [num]
doubleList xs = map double xs

double :: num -> num
double = (*2)
```

Code 2.1: Function passed as a first class-citizen

Furthermore the lazy evaluation of functions makes it possible to define an infinite list of 2's as shown in Code 2.2. This function can be used in other functions because the function creating this list is only evaluated to create the part of the list that is necessary to compute the result of the function `fortytwo`. This function therefore evaluates to 42 instead of evaluating the function `twos` ad infinitum.

```
twos :: [num]
twos = 2:twos

fortytwo :: num
fortytwo = sum (take 21 twos)
```

Code 2.2: Lazy evaluation

With the absence of assignments, it is not possible to have a mutable global variable. This can create problems in some cases. An example of functionality that can not be created very elegantly in a functional language is when a program needs to create a unique identifier. To do this the current value of the unique identifier needs to be stored. In functional programming languages it is not possible to store the value in a variable that can be updated during the execution. Therefore the value must be passed to all functions or needs to be stored in a file. Yet this creates a lot of passing of arguments to and from functions that do not need this argument, or it creates a lot of overhead by accessing files. This can be relieved somewhat by wrapping these values in a state that can be passed between functions, but still the state must be passed.

Referential transparency, the ability to substitute equals for equals, is closely related to the fact that functions can not have side effects. Equals can only be substituted when no other (side) effects occur during evaluation.

I highlighted briefly some features of functional programming languages. As is already known all paradigms have elegant parts and have parts that are less desirable. The choice of a programming language is therefore greatly influenced by personal preference.

Input and output (I/O) is one part that is difficult to achieve in a (pure) functional programming language. Yet, the ultimate purpose of running a program is invariably to cause some side effect: a changed file, some new pixels on the screen, a message sent or something else. A program that has no observable effect whatsoever (no input, no output) would not be very useful.

There are two closely related features in pure lazy FPL that do not cooperate well with I/O, (a) the unknown evaluation order and (b) referential transparency. Because the evaluation order is not known beforehand and is something the programmer should not be bothered with, it is not trivial to use functions that have side effects. These side effects usually need to occur in a certain order, therefore the programmer needs to know or should be able to control the evaluation order.

2.2.1 Evaluation Order

As discussed above the evaluation order of functions does not influence the result of functions in non-strict pure FPLs provided that the functions terminate. Yet in some cases the evaluation order needs to be known, especially when handling I/O. Consider for instance the functions `foo` and `bar` in Code 2.3 (which has some functions that are not referentially transparent). The types of the file accessing functions can be seen in Code 2.4.

```
foo :: [char]
foo = (fread "temp.txt") ++ bar

bar :: [char]
bar = "b"                               , if fwrite "temp.txt" "b"
    = error "Problem writing to file"   , otherwise
```

Code 2.3: Evaluation order influencing results

```
||          path      contents
fread  :: [char] -> [char]
||          path      contents   isSucceeded?
fwrite :: [char] -> [char] -> bool
```

Code 2.4: File accessing functions

Assume that the content of the file `temp.txt` in Code 2.3 is "`a`". When `foo` is evaluated before `bar` the result of function `foo` will be "`ab`". Otherwise when `bar` is evaluated first the value will be "`bb`", provided that the writing to the file succeeds. This illustrates both that the evaluation order is relevant and that referentially opaque functions can be dangerous.

In Amanda there are some constructs which allow the user to influence the evaluation order, like `strict` and `seq`. Although useful, these functions pollute the functional program with control flow behaviour undermining the above-mentioned great benefit of lazy FPLs.

An even more intuitive example of problems with lazy languages and side effecting functions can be illustrated when using the function `xs` in Code 2.5. In this function different side-effecting functions are "stored" in a list. In a lazy programming language it is not known when these characters are written to the screen. When the only use of `xs` in a lazy language is in the execution of `length xs` the characters will not be written at all, because the values of the list are not needed to determine the length of the list. The characters will be written to the screen however, when the list is used as done in Code 2.6. Provided that the type of `printChar` is `printChar` :: **char** −> **bool** and the function **and** is defined as the logical conjunction between all elements in a list. It is even more difficult to predict what will be printed, because **and** itself is implemented in a lazy way, i.e. it will not continue when a value **False** is found.

```
xs = [printChar 'a', printChar 'b']
```

Code 2.5: Side effecting functions in a list

```
print = and xs
```

Code 2.6: Evaluated side effecting functions

2.2.2 Referential Transparency

In FPLs functions are said to be referentially transparent. This means that an expression will evaluate to the same value every time. Another common explanation for referential transparency is that a function is referentially transparent when equals can be replaced by equals. In Code 2.7 the two functions (`f` and `g`) give the same result in any case.

```
f :: num -> num
f x = (x^2) + (x^2)

g :: num -> num
g x = y + y
      where
       y = x^2
```

Code 2.7: Referentially transparent functions

This shows that the function ˆ is referentially transparent, i.e. 4ˆ2 is always equal to 16. This sounds very straightforward, but this is not always the case. Code 2.8 shows that `y` is not always equivalent to `random`. The only values function `g` can have are $\forall x{\in}\mathbb{R}\ \exists i{\in}\{1\ldots5\}\ \bigl(g(x) = 2i^2\bigr)$ whereas `f` can have values of $\forall x{\in}\mathbb{R}\ \exists i{\in}\{1\ldots5\}\ \exists j{\in}\{1\ldots5\}\ \bigl(f(x) = i^2 + j^2\bigr)$ and these are clearly not equivalent.

```
f :: num -> num
f x = (random^2) + (random^2)

g :: num -> num
g x = y + y
      where
       y = random^2

|| in this case random returns an integer
|| value between 1 and 5
random :: num
```

Code 2.8: Referentially opaque functions

2.3 What is Input and Output

In computing, Input/Output, commonly abbreviated I/O, is the collection of information sent from or to a functional unit, i.e. a device or program, etcetera. Input and output are dual concepts. That is, input for one functional unit is output of another functional unit. A mouse is an output device from the perspective of a human, but an input device from the perspective of a computer (program). Likewise a modem can be seen either as an input device or an output device, depending on the way it is used.

I will focus on input and output that occurs during the execution time. Input occurring before execution, and the resulting output given when the program terminates can be considered I/O, yet it is handled by the programming environment in most cases and therefore no part of the I/O I would like to consider in this thesis.

In order to create some overview about the types of I/O that are possible Perry categorized I/O in (Perry, 1991) in the following types:

- *Terminal I/O*
  is I/O considering the input read from the keyboard and written to the screen. Graphical information and other input devices like a mouse can be seen as an extension to simple terminal I/O. Crucial with this kind of I/O is relative ordering between input and output. E.g. user input must be read after the request for it has been displayed on the screen.

- *Device I/O*
  handles all I/O to communication devices, like file-systems, tapes, CD's etc. The requirements are similar to terminal I/O, however synchronization is less (but not un-)important.

- *Process creation*
  covers the ability for a process to start separate processes, not necessarily processes in the programming language environment itself. But may also be processes that are instantiated from within the programming environment, but are not part of the programming language.

- *Inter process communication*
  handles the communication between different processes. Once being established it can be handled just as device I/O.

- *Signal handling*
  is I/O considering *signals* like `<ctrl>-C` (DOS) and `SIGKILL` (POSIX), which are sent to or from the program and must be handled to create a robust program.

- *Environment interaction*
  is I/O aimed at the communication between the programming language and the operating system. The programming language should be able to obtain information from and provide information to the operating system. An example is obtaining the current date and time.

2.4 Current Approaches to I/O in Functional Languages

In this Section I will describe several solutions to handle input and output in functional programming languages. I show how these solutions cope with the unknown evaluation order and the referential transparency. For each solution I will give its advantages and disadvantages. Then I will show how these solutions are implemented in a functional language and lastly draw a conclusion from the advantages and disadvantages of the I/O solutions.

2.4.1 Side-effecting I/O

The most simple solution for handling I/O is to ignore the problems occurring from the use of input and output when defining the programming language. This can be done by just adding constructs to the language that are not referentially transparent, thereby shifting the problem from the programming language to the programmer. The programmer has to ensure the correctness of his program when using referentially opaque functions. languages that offer this solution mostly have some (limited) way to control the evaluation order of the program, to be able to ensure correctness. An example of side-effecting I/O can be seen in Code 2.4.

Advantages

The advantage of side-effecting I/O from the perspective of the programming language and to a lesser extent also to the programmer is its simplicity. In the programming language it is easy to add functions or constructs because it is not necessary to reason about the evaluation order and referential transparency. When a programmer knows exactly what he is doing, side-effecting I/O can be simpler because the programmer does not have to reason about additional constructs defining the evaluation order, like monads, streams, etc.

Disadvantages

There are two problems using side-effecting I/O (Peyton Jones & Wadler, 1993), equational reasoning (the substitution of equals for equals) is not possible when using side-effecting functions. Therefore the prove that functional programs that use side-effecting I/O becomes far more difficult. Therefore the programmer needs to know the evaluation order and must verify that the program as a whole functions as required.

2.4.2 Continuation Passing Style

The Continuation Passing Style (CPS) model, which is a direct predecessor of monads (Wadler, 1997), (see Section 2.4.5), uses a set of transactions to model flow of control. In this style a function is not just defined to return a value, but is given another argument, the continuation argument, which is meant to receive the result of the computation. The continuation passing style is a style that can be used for all functions, for instance the function `fac` can be defined in a "normal" style and in a continuation passing style using λ-abstraction, as done in Code 2.9 and Code 2.10 respectively.

The flow of control is explicitly modeled which solves the problem of the evaluation order. Now input and output can be modeled using an algebraic type named `CPS` in Code 2.11.

```
fac 0 = 1
fac n = n * (fac (n-1))

show (fac 6)
```
Code 2.9: Normal Function

```
fac 0 k = k 1
fac n k = fac (n-1) (res -> k (n*res))

fac 6 show
```
Code 2.10: Continuation Function

```
CPS ::= Done
      | Input   ([char] -> CPS)
      | Output  [char] CPS
```
Code 2.11: Continuation Passing Style (CPS) functions

The internal language's evaluation mechanism need not be changed, however when a root function (`main`) has type `main :: CPS`, the algebraic constructors can be interpreted or executed in the following way (Gordon, 1994):

`Input k`	Read characters `v` from the keyboard and execute `k v`.
`Output v q`	Output characters `v` to the printer, and then execute `q`.
`Done`	Terminate.

The implementation of these constructors closely resembles the monadic style. In the Section about monads a more extensive notion of the use of this kind of operators will be given.

An example function using this I/O model can be seen in Code 2.12, where a line is read from the terminal and the line is printed two times to the screen again. The variable `x` is the value that is read from the terminal, is passed to `double` and is printed by the `Output` constructor afterwards.

```
printDouble :: CPS
printDouble = Input (x -> Output (double x) Done)

double :: [char] -> [char]
double x = x ++ "\n" ++ x
```
Code 2.12: Continuation Passing Style Function

Another example, the same as given in Code 2.5, is given in a continuation passing style in Code 2.13.

```
xs = [x-> Output "a" x, x->Output "b" x]

main :: [cps -> cps] -> [char]
main xs  = print (foldl1 (.) xs Done)
```

Code 2.13: Evaluating I/O functions in a list

Advantages

The advantage of the continuation passing style is that it directly supports error jumps and other changes in the flow of control (Wadler, 1997). A function might notice that a division by 0 would occur and return an appropriate error message instead of continuing the flow of control. The flow of control can be altered because of the way the functions are constructed. Therefore the flow of control can be specified in a natural way, Whereas in side effecting I/O extra special functions need to be specified in the language that support change in control flow.

Disadvantages

Just like in the stream model (Section 2.4.4) all algebraic constructors used to address I/O must be pre-defined in the language. Extending these algebraic types is not trivial and must be done inside the programming language. Furthermore, this way of programming I/O enforces the need of an extra argument in the function, cluttering the program code. Also the functions can become more complex, as is illustrated in the function `fac` in Code 2.9.

2.4.3 Uniqueness Typing

The I/O model using uniqueness typing is called the systems model. The systems model uses a series of transformations on an initial state, that captures the state of the operating system. This state is passed through the functions, thereby specifying the evaluation order (Hudak & Sundaresh, 1989). To ensure a single flow of control the state must not be copied, i.e. the state is unique. This is done using uniqueness (linear) typing. The overall structure of these functions is shown in Code 2.14.

```
program ::
  (input, initial_environment) -> (output, new_environment)
```

Code 2.14: General outline of a I/O function using Uniqueness typing

The functional programming language Clean (see Section 2.5.1) can be seen as an implementation of this I/O-style. A small example of a function adhering to this style is `fappendLine`. The type of this function is given in Code 2.15[1]. This function can be used as done in Code 2.16.

```
fappendLine :: String *File -> *File
```

Code 2.15: System-style file appending function

```
appendFile :: *File -> *File
appendFile f = fappendLine "test" f
```

Code 2.16: Function evaluation `fappendLine`

The uniqueness typing is indicated here by the $*$, which indicates that the value of this type can not be copied. Adherence to this constraint is enforced by the type system. Suppose a function `f` has a unique argument denoted by a $*$ in the type definition of `f`. It is guaranteed that `f` has private (unique) access to this particular argument (Barendsen & Smetsers, 1993).

The line "`test`" will be written in file `f`. The result of the function is a new file with the line added. If file `f` could be used again in the same scope, this file would not contain "`test`", because of referential transparency, i.e. the value of `f` is not altered by the function `fappendline`. Note that the fact that files can be used as variables creates the need for uniqueness typing to ensure file-variables are not used multiple times. The `File`-argument in this example can be seen as the system being explicitly passed through the functions.

Some research has been done to rewrite the Clean system-approach to a monadic approach, for more information on this subject see (Jones, 1995).

Advantages

An advantages of this uniqueness typing approach is that when you are able to split the system (or world) into subsystems, it is possible to use different subsystems in parallel when they do not interact. For instance, when the state of the screen can be separated from the state of the file system, these two can be handled in parallel, or at least in arbitrary order, instead of fixing the evaluation order by the whole system (Wadler, 1997). Furthermore the uniqueness typing system can also be used to facilitate destructive updates (Smetsers, Barendsen, Eekelen, & Plasmeijer, 1993).

[1]The Clean syntax is used here. In Clean no arrows are used between the arguments, only between the arguments and the result

Disadvantages

The type system of the functional language must be extended to handle uniqueness types. This extension is not trivial. Furthermore, the use of an explicit system is inclined to result in functions with extra arguments that need to be passed around. For instance, in the `fappendline` the file must be given to and retrieved from the function. In a function reading information from a file, the result of the function, the updated file (a different file pointer) and the text read, needs to be combined in a tuple. According to (Perry, 1991) uniqueness typing can only handle terminal and device I/O easily.

2.4.4 Streams

Landin introduced streams in 1965 and they are used in several FPLs since. In these stream-based languages predefined identifiers are bound to I/O channels. For instance, `kb` would be bound to the stream of all characters read from the keyboard. In what is called *Landin Stream I/O* a functional program maps a possibly infinite list of input events to a list of output events. The lazy evaluation ensures that output may be interleaved with input (Noble & Runciman, 1994).

A very simple example of a program using streams is Code 2.17. Provided that the stream of characters of the keyboard is bound to the identifier `kb` and the list of output events implicitly is bound to the screen, the program takes a list of characters and writes the characters in uppercase to the screen.

```
main :: [char]
main = map upper kb
```

Code 2.17: A simple stream program

The streams solution can be divided into two kinds. The simple character streams, which send and receive lists of characters, and the token-streams, which can be a single input and a single output list in which different values are distinguished by an algebraic constructor.

Advantages

The advantages of streams are both practical and theoretical. (Wadler, 1997) states that streams are useful as a theoretical tool to use as a denotational semantics for, for instance, the monad model. (Thompson, 1992) states that streams are useful because they create explicit notations for all values on input and output. There is no need for a construct to ensure referential transparency in combination with I/O, like the monadic **IO** or the system-like state.

Disadvantages

Much has been written about all kinds of streams like Landin, lazy, synchronous streams. In most papers where streams are compared to monads, several disadvantages of streams surface. (Hudak & Sundaresh, 1989) describes that streams are not completely general since typically the I/O-devices are pre-determined in the language. Furthermore they state that the possibility of error is generally not accounted for.

A tricky possibility with lazy *synchronized* streams is the extraction of the wrong element out of the list, which results in a deadlock (Peyton Jones & Wadler, 1993). An example of a function which could block because multiple input events need to be evaluated before output can be generated is shown in Code 2.19. The grossly simplified types `Request` and `Response` are shown in Code 2.18, as can be seen in this simplification the evaluation of I/O requests will always succeed. Therefore, when the request `PutC` is evaluated with a character, the character will be written on the terminal and the corresponding response will be `OK`. In the same way the request `GetC` corresponds with the response `OKCh` with the character read from the keyboard. If the number of elements dropped had been 1 instead of 2, the pattern matching would result in an error, because the head of the response list will be `OK` instead of `OKCh`. When the number of dropped elements would be 3, the function could result in a deadlock, because the value of the third element of the list is not yet evaluated, and never will be, because the corresponding request will never occur.

```
Request ::= PutC Char
          | GetC

Response ::= Ok
           | OkCh Char
```

Code 2.18: The type of Requests and Responses

Perry states that with streams it is hard to handle signals.

2.4.5 Monads

The monad model offers a possibility to define the order in which functions are evaluated, thereby solving the problem of the fixed evaluation order of I/O

```
echo :: [Response] -> [Request]
echo resps = GetC :
              if a == EOF
              then []
              else PutC a :
                echo (drop 2 resps)
            where
              OKCh a = resps!!0
```

Code 2.19: Blocking event

actions. Monad types are types that have two functions: `return` and >>=. The types of these functions are given in Code 2.20, where `a` and `b` are type variables representing an arbitrary type and `m` represents the monad type.

```
return :: a -> m a
>>=    :: m a -> (a -> m b) -> m b
```

Code 2.20: Monad functions

Intuitively, **return** creates (wraps) the value in a monad, whereas >>= (pronounced 'bind') sequences functions using monads.

These functions that are defined on a monad must obey certain properties to define a real monad. These properties are defined in Code 2.21.

```
x :: a
f :: a -> m b
g :: a -> m b
n :: m a

(return x) >>= f   == f x
(n >>= f) >>= g    == n >>= (\x -> f x >>= g)
n >>= return       == n
```

Code 2.21: Monad properties

An example of a monad that is easy to understand is the **Maybe** monad. The implementation of the **Maybe** monad can be seen in Code 2.22.

The use of monads can be shown by an example using the **Maybe** monad. This **Maybe** monad can be used to create functions which result is either a value or nothing. Consider the following case, taken from (Newbern, 2004), representing sheep that are cloned, i.e. they *may* have a father. This can be modeled using the **Maybe** monad with the type as defined in Code 2.23.

```
Maybe a = Nothing
        | Just a

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(Just a) >>= f = f a
Nothing  >>= _ = Nothing

return :: a -> m a
return n = Just n
```
Code 2.22: Maybe monad

```
type Sheep = ...

father :: Sheep -> Maybe Sheep
father = ...
```
Code 2.23: Functionality using Maybe

The function >>= can be used to compose functions out of other functions. The grandfather (of the fathers side) can be composed with the function `father`, as done in Code 2.24. Intuitively the bind function reverses the function and argument, `x` >>= `father` can be read as: if `x` is a sheep evaluate `father x` otherwise the function will evaluate to **Nothing**.

```
grandfather :: Sheep -> Maybe Sheep
grandfather s = (Just s) >>= father >>= father
```
Code 2.24: Grandfather function

Monads are more thoroughly explained in (Wadler, 1990), which is based on the work of (Moggi, 1989).

Advantages

By using monads we have the intuitive sequential nature of imperative Input/Output which can be used in the same way as imperative programming, without losing referential transparency and the extensive type system of functional languages (Russell, 1997). The use of syntactic sugar, like the **do**-notation in Haskell (2.5.3), can create a very low-level imperative construct, which can be used by functional programmers without having to reason about monads. The notation hides a lot of lambda-abstractions and bind functions, therefore making the code much more readable. Because of the implicit passing of the **IO**-monad the construction is very composable. The program structure between monads and the continuation passing style is almost identical, monads though, tend to be slightly more abstract. All functionality using CPS can be modeled using monads (Wadler, 1997, page 20). Comparing monads and CPS shows that monads are more abstract and can therefore be used in more areas.

Disadvantages

The downside of I/O using monads is the need for a monad that can not be unwrapped. So, when using monadic I/O there is no way to get rid of the I/O monad. Furthermore, it is not as intuitive as one would like it to be. A prerequisite to good software design is a thorough understanding of the structures and glues of the implementation language. Monads are an interesting addition to both of these sets, and therefore a design implication is that the designer needs to be aware of the potential benefits of using monads (Russell, 1997). Yet the understanding of monads is not trivial. The extensive amount of tutorials and questions on the Internet strengthen this thought. Another disadvantage is that monads tend to be an all-or-nothing solution (Wadler, 1997). The transition from no interaction at all to a single form of interaction is not very smooth. Parts of the program need to be rewritten. In general monads are not easily composable, although the use of monad transformers lighten this somewhat.

2.5 Current Implementations of I/O Models

2.5.1 Clean

Clean is a lazy, pure, higher order functional programming language with explicit graph rewriting semantics (Plasmeijer & Eekelen, 2001). The way Clean handles I/O is quite different from other approaches and is therefore interesting to examine. In Clean the environment (a uniqueness type `world`) is explicitly passed as an argument to all I/O-functions.

In Clean there is a way to overcome the problem of not knowing in which order functions are evaluated. This is done by the use of explicit environment passing (Achten, Groningen, & Plasmeijer, 1993). Explicit environment passing means the environment is accompanied as an argument in the function and will be part of the result. Clean is an example of the system approach as described in Section 2.4.3. The function `freadline` in Code 2.25 can be used as done in Code 2.26.

`freadline` takes a file (`fileA`) as its argument and gives a String (`text`) and an updated file (`fileB`) as result. The difference between `fileA` and `fileB` is the

```
freadline :: *File -> (String,*File)
```

Code 2.25: freadline-function

```
(text,fileB) = freadline fileA
```

Code 2.26: freadline use

position of the pointer in the file. This updated file (`fileB`) can subsequently be used in the rest of the program, thereby ensuring that this function is evaluated before the rest of the program, which uses this file, is executed.

A problem when using the `world` record is, for instance, the function `save` that saves a score in a file as shown in Code 2.27. The `pSt` is a record containing information about the system, like current open files, and program specific information added by the programmer. The local state `ls` contains the current score, `io` contains information about the I/O, like screen and files.

```
save :: *(PSt StoreTp) -> *PSt StoreTp
save pSt = saveToFile ls.score pSt
           where
             {ls=ls,io=io} = pSt
```

Code 2.27: Uniqueness typing - incorrect

However, the code in Code 2.27 is not correct, because type `PSt` is a uniqueness type and can not be used twice on the right hand side, because this may result in a copy of `PSt`.

The formal definition of a uniqueness type is more complicated. According to (Achten et al., 1993) in the graph rewriting of a program a node n in graph G is unique with respect to node m in G, if n is only reachable from the root of G via m and there exists exactly one path from m to n. A property of a unique node is the fact that it has a reference count - the number of incoming paths - (in-grade) of one. A reference count of one is however not sufficient for uniqueness, the whole path from m to n must have reference count one, as shown in Figure 2.2.

A uniqueness type adds an extra constraint to the use of variables of this function to ensure that variables with unique types will be used only once on the right hand side of the function, thereby ensuring that no duplication of this unique variable occurs.

Although `pSt` is not used and updated, it is still not possible to pass it as an argument. A workaround is creating the record again as done in Code 2.28.

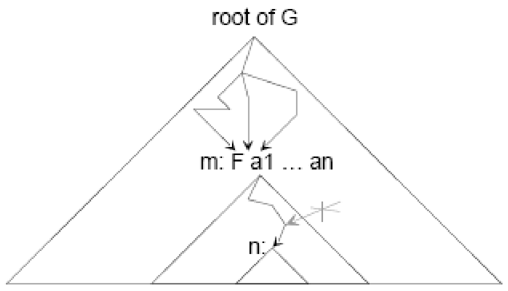


Figure 2.2: Uniqueness typing Graph

```
save :: *(PSt StoreTp) —> *PSt StoreTp
save pSt = saveToFile ls.score {ls=ls,io=io}
          where
            {ls=ls,io=io} = pSt
```

Code 2.28: Uniqueness typing - correct

In this way no identifier with a unique type is used more than once on the right side of a function.

Graphical elements in Clean are based on abstract devices. An abstract device is a representation of the actual object. Each graphical element must implement functions from the abstract device. Portability to different operating systems is easier this way than when all functions need to be implemented. Furthermore it is easier to extend the graphical user interface toolbox with new objects by implementing these functions for a user defined graphical object.

The authors of a case study in which Clean is used to create a spreadsheet application came to the following conclusion about the modifiability of I/O in Clean with respect to imperative languages:

> *The advantage of Clean I/O is its relatively direct way of interfacing to system calls. In particular for the relatively I/O intensive parts like scrolling (in the sheet or in the editor), this was important in order to achieve a proper efficiency of interaction. It is our impression that using Clean I/O it is easier to modify and read I/O programs than using an imperative language.*

(Hoon, Rutten, & Eekelen, 1995)

In Clean the I/O solution has some influence on the rest of the program. When a variable is used which has to be unique, then the way a function can be constructed is limited. This is due to the constraint that the variable can be used one time (in the RHS) only.

2.5.2 Amanda

Amanda is a functional language written by Dick Bruin, based on the functional programming language Miranda, which uses the stream I/O model.

The Graphical User Interface (GUI) of Amanda can be addressed by the stream identifier `graphicsout`. Argument to this function is a list of graphics, which are written to the screen. The input events in Amanda are given by a 0-ary function, `eventsin` which is a (possibly infinite) list of input events. The problems with I/O are 'solved' in two ways: The interaction with input and output devices is done using streams, and file handling is done using functions with side effects.

The way user input can be handled in Amanda is by an event loop. From this loop all functions not handling I/O can be called, separating the GUI-handling from the other functionality. In Code 2.29 the function `main` redirects all events `eventsin` to the `doEvents`-function, which results in a list of output events that are written to the screen by the `graphicsout`-function.

```
main :: [graphics]
main = graphicsout (doEvents emptyState eventsin)
```

Code 2.29: Binding streams to event-loop

This `doEvents` function is shown in Code 2.30: it forwards the list to the `doEvent`-function that just handles one event (see Code 2.31 for an example). This improves the readability of the functions because in this case the recursive clause can be given once instead of multiple times.

```
doEvents :: state -> [inputEvent] -> [graphics]
doEvents s (e:es) = g ++ doEvents newS es
                    where
                      (newS,g) = doEvent s e
```

Code 2.30: Function handling List of Events

The `state` which is given to each function as a parameter can be seen as the memory of the program. In this `state` all persistent information about the state of the program is stored.

File handling in Amanda is done in a very simple way. The function `fread` and `fwrite` respectively read and write to a file. These functions address the

```
doEvent :: state -> inputevent -> (state,[graphics])
doEvent s (KeyIn '\e')
    = (s,[GraphQuit])
doEvent s (MouseDown (x,y))
    = (addCircle s (x,y), [GraphEllipse 0 (x,y) 0.4])
doEvent s _
    = (s,[])
```

Code 2.31: Function handling Single Event

content of the file, in contrast to the Clean approach where the files are handled themselves. The file handling is done in a referentially opaque way. As said in Chapter 2.3 this is not a good way to deal with I/O. It is just shifting the problem from a language level to the programmer.

GUI in Amanda is done by a function that takes a list of graphics as an argument. This list of graphics is processed outside the scope of the programmer and shown on the screen. User input, like the keyboard input is handled by the input events. There are different (fixed) kinds of input events.

The way graphics in Amanda are handled can not be extended. There is a function `graphicsout`, but there is no way to extend the type of the events that are given to this function. On the other hand it should be possible to create a new function which writes output to other devices, or other output to the screen.

The input events are members of an algebraic type, and there is no means to extend this type with your own members. It is not possible to extend the events with a `HeatSensorIn` event for instance.

Amanda is a language with limited behaviour with respect to I/O. Only a limited set of I/O operations are supported. It is clear that the programming language is not used in an industrial setting and several peculiarities are not solved. Examples are: some I/O functions running out of memory after a long time of doing nothing; the IDE can not handle directories and file names with spaces; it is not possible to send events over a network. This is one of the reasons the new functional programming language TINA is being developed at the University of Twente (Papegaaij, 2005) and (Hove, 2005, to appear). However, the way user events are handled by an event loop is fairly easy to implement and acts as expected.

The event loop ties all functions together and therefore tends to get less readable. It is easy to separate non-I/O functionality from the user interface functionality, by delegating events to separate functions.

Amanda is not fully referentially transparent and should therefore be used with care. From a language perspective, there is no means to ensure that unexpected behaviour will never take place when using these referentially opaque functions.

Amanda is capable of handling input given by the keyboard and mouse and write information to the screen in a fairly easy way. Furthermore it is possible to access files, although at the cost of not being referentially transparent anymore. From the list of different types of I/O given in Chapter 2.3 Amanda can only handle the first two.

2.5.3 Haskell

Due to syntactic sugar the monads are not very visible in Haskell. Consider for instance the following function `onSave` in Code 2.32 for storing a score in a file. The score is retrieved from the store using `getScore`.

```
onSave :: StoreTp -> IO ()
onSave store = do
                 s <- getScore store
                 saveToFile s
                 return ()
```

Code 2.32: Do Construction

The **do**-notation hides the monads in the function. The monad is only visible in the result of the function. A more explicit monadic representation of this function is with the use of λ-abstractions in Code 2.33. It is clear how monads are used to create an order in this function. Also the fact that every function gives a result (which is not used in some cases) is more clear than in Code 2.32

```
onSave :: StoreTp -> IO ()
onSave store = getScore store >>= (\s ->
               saveToFile s   >>= (\_ ->
               return () )))
```

Code 2.33: Monad Construction

A benefit of monads is the purely functional style. A program can be written almost like an imperative program but still be purely functional. Also the possibility to create a function which is applicable to several monads is useful.

Less positive is that functions must be rewritten, in order to be applicable for monads. Also a programmer must ensure that the evaluation order will not be over restricted by the monads.

I think the monadic approach is quite a strong paradigm which can be used in multiple domains, however it does not feel intuitive. Monadic functions are different because their results are monadic values and can not be composed with non monadic functions directly. When you create a program that handles no input, it is not necessary to create functions which can handle monadic values. However, when you reconsider and in the end do need some user input, then at least a part of the program has to be rewritten.

When functions are changed to handle monadic values this means they have to be rewritten. In order to prevent that all functions have to be rewritten to handle (one-way) monadic values, sometimes the function can be written in the way that is shown in Code 2.34. The function `doSomething`, which has type `doSomething :: String -> a`, can be written without the burden of the **IO** monad, because the function is evaluated 'inside' the I/O monad. This solution is only possible when the I/O-value is read at the 'top-level' of the program. When the I/O-value is read half way, this construction to work around monads is not possible.

```
main :: IO a
main = do
         value <- readIOValue
         return (doSomething value)
```

Code 2.34: No I/O-monad

Monads are a powerful yet rather hard to understand concept. It is hard to keep track of when the value was wrapped into a monad or is a non monadic value. The syntactic sugar in Haskell with the **do**-notation is quite easy, however it encapsulates the exact behaviour and makes it therefore harder to understand fully.

Monads are referentially transparent. Although a 'wrapper' is created when reading from a file, it seems that values can be compared. Reading from a file can result in **IO** *"a"*, **IO** *"b"* and therefore differ. Yet, it is not possible to compare these **IO**-values directly. The **IO** monad ensures that the evaluation of a function can not be wrongly unified with another evaluation of the same function. It is wrong to say that **IO** values can not be compared at all, as is shown in Code 2.35 in which two values of the **IO** monad are compared and is shown whether the values are equal. A common way to look at an (**IO** `a`) value is that it denotes a *computation*, that may perform I/O when executed by the environment and then results in a value of type `a` (Gordon & Hammond, 1995).

Simple terminal I/O is possible in the core of Haskell, for extensive I/O there are several packages to show windows etc. Haskell has a foreign function interface (FFI) (Finne et al., 2003) to interact with the operating system and thereby with other languages. In concurrent Haskell (Peyton Jones, Gordon, & Finne,

```
compareIOValues :: Eq a => IO a -> IO a -> IO ()
compareIOValues x y = do
                        a <- x
                        b <- y
                        putStrLn (show (a==b))
                        return ()
```

Code 2.35: Function comparing to I/O values

1996) it is possible to create processes and communicate between processes.

2.6 Conclusion

Monads are a powerful construction and are most certain useful in functional programming languages, like the **Maybe** monad. Yet the one-way **IO**-monad is not as intuitive as desired. The **IO**-monad can not be removed, therefore in some cases the program will be cluttered with the **IO**-monad, which alters the type definition of the function and the way functions can be composed. When comparing monads to, for instance, streams another disadvantage of monads comes to light: the monad itself. In the streams model there is no construct necessary to wrap values in, like the **IO**-monad.

Uniqueness typing is very straightforward, although quite verbose, because the state of the system must be passed through all functions that perform I/O. Uniqueness typing is not only useful to handle I/O, it is also possible to use uniqueness typing to implement destructive updates. The uniqueness typing ensures that destructive updates are possible because values can not be copied.

Streams are a simple, intuitive approach that can be used to create elegant code without cluttering the complete program with I/O-constructs. Much of the disadvantages of streams given above are directed at the more error-prone synchronous streams. The view of the programs written in the stream model tend to be more event based. Arguably GUIs are more event based than sequential, therefore streams seem to be a better solution to implement GUIs. Furthermore I have the feeling that the learning curve of monads is less steep than the one of streams, indicating that streams can have a more educational purpose.

3

Extending Algebraic Types with Unions

In the previous chapter is shown that one disadvantage of the stream I/O model is that the algebraic types used to model the events can not be extended. In this chapter I will propose a new construction that is aimed at this problem. This new construction, the union of algebraic types can be used to model events and extend them. First I will give a short introduction about algebraic types in general. Then an intuitive notion of the union type is given, along with some of its possible uses. This is followed by the typing rules that are needed to create a syntactic notion of the union type. These typing rules are used to prove some properties for unions, commutativity, associativity and idempotence. In the last section is described how functions defined on unions should behave and can be used.

3.1 Algebraic Types

An algebraic type is a type which consists of one or more constructors with or without arguments. These constructors allow discrimination between those arguments. A simple example of an algebraic type is given in Code 3.1

```
vehicle  ::= Car        brand
           | Motorcycle brand cc
           | Bike

brand == [char]
cc    == num
```

Code 3.1: An example algebraic type

In some cases the discriminating constructor *is* the value, like in `Bike`. Examples of terms of this type are given in Code 3.2.

```
vehicle1 = Bike
vehicle2 = Motorcycle "Yamaha" 750
```

Code 3.2: A term of type `Vehicle`

The equivalent of an algebraic type in set theory is the disjoint union - a set whose elements consist of a tag (equivalent to a constructor) and values in this set (equivalent to the constructor arguments).

3.2 Union Types

In stream I/O algebraic constructors are used to distinguish between events as is described in the stream approach (Section 2.4.4) and its implementation in the Amanda language (Section 2.5.2).

A problem with the use of algebraic types in the stream I/O model is that they can not be extended. Libraries can not be constructed in which new events (based on low level events) are added to the language, because the event types are fixed. Therefore I will add a new construction to a functional programming language which creates the possibility to merge algebraic types. Consider the algebraic types A and B from Code 3.3. When the union of the algebraic types A and B, denoted as $A \tilde{\cup} B$[1] is formed the programmer should be able to address $A \tilde{\cup} B$ as if it were defined like type C in Code 3.4.

```
A ::= C1
    | C2

B ::= C3
    | C4
```

Code 3.3: Algebraic types

```
C ::= C1
    | C2
    | C3
    | C4
```

Code 3.4: Algebraic types merged: `C` behaves like `A` $\tilde{\cup}$ `B`

The union, only defined on algebraic types, is the concatenation of two algebraic types without a discriminating tag to distinguish between values of these types. Types can only be concatenated in a union when the types do not conflict. Type σ and τ conflict when:

$$\exists (C\ \overline{t} \in \sigma)\ .\ \exists (C\ \overline{t'} \in \tau)\ .\ \overline{t} \neq \overline{t'}$$

Where C is the name of the constructors and $\overline{t}$ and $\overline{t'}$ are the arguments of

[1]When the union operator is used in program text, the union operator $\tilde{\cup}$ will be displayed as <U>

constructor C. Informally, types conflict when there is a constructor in both types that do not have the same arguments.

The semantic definition of a union type is:

Definition 3.1.

$$\begin{array}{ll}
\sigma \mathbin{\tilde{\cup}} \tau \textit{ is undefined} & \textit{, if } \sigma \textit{ and } \tau \textit{ conflict} \vee \\
 & \sigma \textit{ is undefined} \vee \\
 & \tau \textit{ is undefined} \\
x : (\sigma \mathbin{\tilde{\cup}} \tau) & \textit{, if } x : \sigma \vee x : \tau
\end{array}$$

Informally, the union over algebraic types is defined as a normal union, when the constructors of type σ and type τ do not conflict. Otherwise the union over algebraic types is not defined. This implies that the $\vee$ in the second clause is the normal inclusive or. This also implies that the algebraic union is idempotent, i.e. it is desired that $A \mathbin{\tilde{\cup}} A$ is equivalent to A. Here equivalence of types, denoted as $\sigma :=: \tau$, is interpreted as follows:

$$\sigma :=: \tau \quad \text{iff} \quad x : \sigma \Leftrightarrow x : \tau$$

It seems not very useful to allow algebraic unions over types that are equivalent. The union, $A \mathbin{\tilde{\cup}} A$ is equivalent to A and does not create more expressive power in the language. Yet the following union, in which also constructors that have the same name and arguments are used in a union, can be quite useful: $(A \mathbin{\tilde{\cup}} B) \mathbin{\tilde{\cup}} (A \mathbin{\tilde{\cup}} C)$. When A is the type specifying core events and B and C are types defined in libraries specifying extensions to this type A then it could occur that in the application these types are needed in a union. One could argue that this should not occur, that the libraries only need their own types. In practice however, this example is likely to occur. It is desirable that the result type of the union of these types, $(A \mathbin{\tilde{\cup}} B) \mathbin{\tilde{\cup}} (A \mathbin{\tilde{\cup}} C)$, is equivalent to $A \mathbin{\tilde{\cup}} B \mathbin{\tilde{\cup}} C$. This boils down to the question whether $A \mathbin{\tilde{\cup}} A$ is equivalent to A. This simplification is possible using the commutative and associative notion of unions, which will be given in Section 3.5.

3.3 Why Union Types

In the stream I/O model, algebraic unions can be used to combine libraries of event types to use these together as if they were all core events. The use of union types creates the possibility to add layers of new event types to facilitate the programmer to create programs in a more abstract way, because low-level constructs can be hidden from the programmer. In Chapter 6 new events are introduced to convey the notion of moving blocks. The IP-addresses and port numbers necessary to send information over the network are conveniently concealed inside the functions handling move-events.

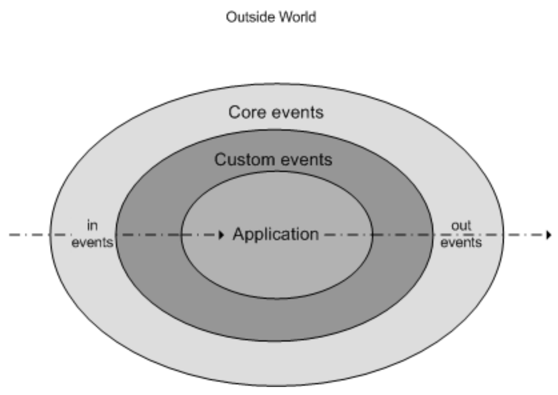


Figure 3.1: A layered event model

In the example given in Chapter 6 `Send` is a core event and `Move` is a custom event. Only core events can be given to and received from the outside world, because the compiler only knows how to handle events that are specified in the compiler. This layered model is illustrated in Figure 3.1. It should be possible to add drivers to the compiler to add new core events, for instance to get information about a sensor. To do this the source of the compiler needs to be extended to know what to do with these new events. In Figure 3.1 the ring representing the core events will be expanded in that case. In the language itself events can be added by making an abstraction over core events. In the example an abstraction is made over the `Send` and `Received` event by the `Move` and `Moved` event respectively. These abstractions can be made using libraries to group these events.

Not only in the stream I/O model but also in other cases it is useful to have union types. In the master's thesis (Tinnemeier, 2006, to appear) an agent language is implemented in a functional language. In this implementation, actions (steps in a plan of the agent, like `If`, `Repeat`) and propositions (facts about the world, like `On x y`, `Clear y`) can be interleaved. In a plan, propositions as well as actions can occur, yet they are obviously not the same. This problem can be solved by adding extra constructors to include propositions in actions by making them a specific kind of action, but this will create more verbose code, with extra (unnecesary) constructors.

Also in the Tinadic parsing method developed by (Kuper, 2006) the use of union types can be very useful as will be seen. Tinadic parsing is based on the notion that specification of a syntax should be defined in a way that most closely resembles the EBNF notation. This Extended Backus Naur Form (EBNF) is a notation describing context-free grammars. Most languages are already described in EBNF, therefore it is easy when a parser can be defined in a format closely related to this notation. In the Tinadic parsing method constructions in the EBNF syntax, like optionality (?) and multiplicity (*,+) are part of an algebraic type along with all non-terminals in the grammar. In Code 3.5 this can be seen. A part of the algebraic type to parse the Amanda language is shown. The first part of the algebraic type, the constructors `Token` until `Rep1`, are constructions from the EBNF syntax, the rest of the algebraic type are the non-terminals of the grammar. Now the grammar can be defined as a function in terms of the algebraic type, a part of the syntax of the Amanda language is shown in Code 3.6. The unary operator <?> is a prefix notation equivalent to the postfix EBNF notation (?). It shows that a simple right hand side of a function is an expression possibly followed by **where** clauses. A specification is either an identifier typed with a type or a type form consisting of a type name and type variables and a type. For a complete EBNF description of the Amanda language see Appendix E.

```
alphabet ::=   Token [char]
             | Check ([char]->bool)
             | Alt   [alphabet] [alphabet]
             | Try   [alphabet] [alphabet]
             | Opt   [alphabet]
             | Rep0  [alphabet]
             | Rep1  [alphabet]

             || Non-terminals
             | Argtype
             | Case
             | Exp
             | Fdef
             | ...
```

Code 3.5: A part of the algebraic type of the Grammar

```
grammar :: alphabet -> [[alphabet]]
grammar
  = ...
  | Simple_rhs    -> [[Exp, <?> [Whdefs]]]
  | Spec          -> [[Identifier, Token "::", Type]
                     ,[Tform, Token "::", Type]
                     ]
  | ...
```

Code 3.6: The grammar of the language

For every grammar, the EBNF constructions need to be defined again. When one EBNF construction is missing from the algebraic type, the parsing will not work. From a software engineering point of view the copy-paste of all EBNF constructors each time is very bad practice with respect to code reuse and maintenance. When these constructs can be defined once and used many times, this would be highly recommended. The use of union types could alleviate this problem by defining `alphabet` in the way specified in Code 3.7.

```
alphabet ::= (ebnf alphabet) <U> nonTerminals

ebnf * ::= Token [char]
         | Check ([char]->bool)
         | Alt   [*] [*]
         | Try   [*] [*]
         | Opt   [*]
         | Rep0  [*]
         | Rep1  [*]
         | List0 [*] *
         | List1 [*] *

nonTerminals ::= Argtype
               | Case
               | Exp
               | Fdef
               | ...
```

Code 3.7: The algebraic Union of the Grammar

3.4 Subtyping

When algebraic data types can be extended by adding constructors a notion of subtyping is created. The subtyping relation that is created when unions are introduced is a bit counter intuitive. With another form of subtyping, namely record extension, extra information (new fields that may contain extra data) are added to the data type. When a union type is introduced by combining two types it seems that extra information (new values) are added. However, this is not true. This can be seen easily when focusing on a single value. When x is of algebraic type A it is clear that x has one of the constructors of A. Yet, when x is of type $A \widetilde{\cup} B$ then x can have constructors of type A or type B. Thus, A is more specific, i.e. a subtype, of $A \widetilde{\cup} B$.

Subtyping relaxes the requirement that functions take arguments of a given type, by allowing arguments of any subtype of that type to be given (Cardelli, 1988). The intuitive notion of a subtyping relation is when for two types σ and τ:

$$\sigma <: \tau, \text{ iff } \forall e.(e : \sigma) \rightarrow (e : \tau)$$

Note that:

$$\sigma :=: \tau, \text{ iff } \sigma <: \tau \wedge \tau <: \sigma$$

This has some consequences on the type checking of functions, for instance the function `showT` is defined on a type σ, but should also work on arguments of type τ when τ is a subtype of σ. This is called contravariance in the argument of the function. Note that the subtyping rule is switched; $\sigma <: \tau \Rightarrow (\tau \rightarrow \alpha) <: (\sigma \rightarrow \alpha)$. In the result type of a function the subtyping rule is not switched, therefore it is called covariant in the result. This rule is not trivial, therefore I will try to illustrate it with an example. Obviously, Int <: Float thus for all e : Int also applies that e : Float. Function f : Float $\rightarrow \alpha$ analogously has also type Int $\rightarrow \alpha$. Therefore Float $\rightarrow \alpha <:$ Int $\rightarrow \alpha$(Cardelli, 1997).

3.4.1 Subtyping Rules

A type system is a system that validates the type of expressions with typing judgements. All these possible type judgements for a type system with subtyping are:

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash \sigma$	σ is a well-formed type in Γ
$\Gamma \vdash \sigma <: \tau$	σ is a subtype of τ in Γ
$\Gamma \vdash e : \sigma$	e is a well-formed term of type σ in Γ

These judgements ($\Gamma \vdash A$), where A is an assertion, are grouped in typing rules. These typing rules consist of a number of premise judgements above the line, with a single conclusion judgement below the line. When all of the premises are satisfied then the conclusion must hold; the number of premises may be zero. This mechanic way of validating types can be implemented in a programming language, whereas the informal notion mentioned in the previous sections can not be implemented in a programming language directly.

$$\frac{\Gamma_1 \vdash A_1 \ldots \Gamma_{n-1} \vdash A_{n-1}}{\Gamma \vdash A_n} \qquad \text{(General form of a type rule)}$$

With the notion of subtyping the typing rules must be extended to handle subtyping in the type system. First three rules are introduced that define the relation between equivalence between types and subtyping. Note that these rules are the syntactic rules for the intuitive notion of subtyping and equivalence given above.

$$\frac{\Gamma \vdash \sigma \quad \Gamma \vdash \tau \quad \Gamma \vdash \sigma <: \tau \quad \Gamma \vdash \tau <: \sigma}{\Gamma \vdash \sigma :=: \tau} \qquad \text{(Eq I)}$$

$$\frac{\Gamma \vdash \sigma \quad \Gamma \vdash \tau \quad \Gamma \vdash \sigma :=: \tau}{\Gamma \vdash \sigma <: \tau} \tag{Eq E$_1$}$$

$$\frac{\Gamma \vdash \sigma \quad \Gamma \vdash \tau \quad \Gamma \vdash \sigma :=: \tau}{\Gamma \vdash \tau <: \sigma} \tag{Eq E$_2$}$$

Now a number of rules are introduced that define the subtyping relation on types. These rules are taken from (Cardelli, 1997). The first rules Sub Refl and Sub Trans induce a partial order between types. The typing rule Val Subs describes that when a value has a specific type, it also has a more general type. The rules Sub Arrow, Sub Tuple, Sub Record and Sub List are needed to define subtyping on constructs in the language, like lists and tuples. Note that there is another way of subtyping for records. The Sub Record rule can also be defined to indicate record a is a subtype of record b if a has at least all the labels of type b, i.e. $\{\texttt{x} :: \textbf{char}, \texttt{y} :: \textbf{bool}\}$ is a subtype of $\{\texttt{x} :: \textbf{char}\}$. This kind of subtyping is *not* part of the union types discussed here. Extensible records (Leijen, 2005) is one paper handling this kind of subtyping.

$$\frac{\Gamma \vdash \sigma}{\Gamma \vdash \sigma <: \sigma} \tag{Sub Refl}$$

$$\frac{\Gamma \vdash \sigma <: \tau \quad \Gamma \vdash \tau <: \upsilon}{\Gamma \vdash \sigma <: \upsilon} \tag{Sub Trans}$$

$$\frac{\Gamma \vdash x : \sigma \quad \sigma <: \tau}{\Gamma \vdash x : \tau} \tag{Val Subs}$$

$$\frac{\Gamma \vdash \sigma' <: \sigma \quad \Gamma \vdash \tau <: \tau'}{\Gamma \vdash (\sigma \to \tau) <: (\sigma' \to \tau')} \tag{Sub Arrow}$$

$$\frac{\Gamma \vdash \sigma' <: \sigma \quad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash (\sigma' \times \tau') <: (\sigma \times \tau)} \tag{Sub Tuple}$$

$$\frac{\Gamma \vdash \forall i \in \{1, \ldots, n\}.\sigma_i <: \tau_i}{\Gamma \vdash \{l_1 : \sigma_1, \ldots, l_n : \sigma_n\} <: \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \tag{Sub Record}$$

$$\frac{\Gamma \vdash \sigma <: \tau}{\Gamma \vdash [\sigma] <: [\tau]} \tag{Sub List}$$

3.4.2 Type Checking

The addition of union types to a functional programming language introduces several new rules on top of the normal rules for a functional programming language as been given in (Cardelli, 1997). First of all rules must be introduced to create a well-formed union type, under the assumption that the types σ and τ do not conflict.

$$\frac{\Gamma \vdash \sigma \quad \Gamma \vdash \tau}{\Gamma \vdash \sigma \mathbin{\widetilde{\cup}} \tau} \qquad \text{(Type Union)}$$

The typing rules to subtyping with respect to unions are shown below. The rule Val Union $\widetilde{G}$ introduces a union type when the two functions are composed in parallel. The second Val Union $\widetilde{F}$ introduces a union in a function when applied to a parallel function composition that gives one type as a result. The rules Sub I_1, Sub I_2 and Sub I_3 create a new assertion about subtypes with respect to union types.

$$\frac{\Gamma \vdash f : \upsilon \to \sigma \quad \Gamma \vdash g : \upsilon \to \tau \quad \Gamma \vdash \sigma \mathbin{\widetilde{\cup}} \tau}{\Gamma \vdash f \mathbin{\widetilde{G}} g : \upsilon \to (\sigma \mathbin{\widetilde{\cup}} \tau)} \qquad \text{(Val Union } \widetilde{G}\text{)}$$

$$\frac{\Gamma \vdash f : \sigma \to \upsilon \quad \Gamma \vdash g : \tau \to \upsilon \quad \Gamma \vdash \sigma \mathbin{\widetilde{\cup}} \tau}{\Gamma \vdash f \mathbin{\widetilde{F}} g : (\sigma \mathbin{\widetilde{\cup}} \tau) \to \upsilon} \qquad \text{(Val Union } \widetilde{F}\text{)}$$

$$\frac{\Gamma \vdash \sigma \quad \Gamma \vdash \tau \quad \Gamma \vdash \sigma \mathbin{\widetilde{\cup}} \tau}{\Gamma \vdash \sigma <: \sigma \mathbin{\widetilde{\cup}} \tau} \qquad \text{(Sub I}_1\text{)}$$

$$\frac{\Gamma \vdash \sigma \quad \Gamma \vdash \tau \quad \Gamma \vdash \sigma \mathbin{\widetilde{\cup}} \tau}{\Gamma \vdash \tau <: \sigma \mathbin{\widetilde{\cup}} \tau} \qquad \text{(Sub I}_2\text{)}$$

$$\frac{\Gamma \vdash \sigma \mathbin{\widetilde{\cup}} \tau \quad \Gamma \vdash \tau \quad \Gamma \vdash \sigma \quad \Gamma \vdash \upsilon \quad \Gamma \vdash \sigma <: \upsilon \quad \Gamma \vdash \tau <: \upsilon}{\Gamma \vdash \sigma \mathbin{\widetilde{\cup}} \tau <: \upsilon} \qquad \text{(Sub I}_3\text{)}$$

The question now is whether these rules fulfill the desired properties of subtyping in the case of union types, i.e:

$$\Gamma \vdash x : (\sigma \mathbin{\widetilde{\cup}} \tau) \quad \Leftrightarrow \quad \Gamma \vdash x : \sigma \vee \Gamma \vdash x : \tau$$

$\Leftarrow$ Suppose $\Gamma \vdash e : \sigma$. Since $\Gamma \vdash \sigma <: (\sigma \mathbin{\widetilde{\cup}} \tau)$ by rule Sub I_1 it follows by Val Subs that $\Gamma \vdash e : (\sigma \mathbin{\widetilde{\cup}} \tau)$. For τ the proof is analogous.

$\Rightarrow$ On a *metalevel* we can prove that the only way to construct an expression $e : \sigma \mathbin{\widetilde{\cup}} \tau$ is with the combination of the rules Val Subs and either Sub I_1 or Sub I_2. Therefore $e : \sigma$ or $e : \tau$.

Therefore it is save to add the rule:

$$\frac{\Gamma \vdash x : \sigma \mathbin{\widetilde{\cup}} \tau}{\Gamma \vdash x : \sigma \text{ or } \Gamma \vdash x : \tau} \qquad (\widetilde{\cup} \text{ Elimination})$$

But this rule is not constructive and therefore not suitable in a type system. On the other hand there is no proof that can be constructed *within* the system.

3.5 Union Properties

Union types should have some properties that are intuitively clear. These properties are shown in Property 3.1, 3.2 and 3.3.

A union must comply to the intuitive notion that a union is commutative and associative. That means for all expressions e, if $e : A \mathbin{\widetilde{\cup}} B$ then e also has type $B \mathbin{\widetilde{\cup}} A$. I.e. $A \mathbin{\widetilde{\cup}} B$ is structurally equivalent ($:=:$) to $B \mathbin{\widetilde{\cup}} A$. Also when an expression e has type $A \mathbin{\widetilde{\cup}} (B \mathbin{\widetilde{\cup}} C)$ then e also has type $(A \mathbin{\widetilde{\cup}} B) \mathbin{\widetilde{\cup}} C$. Because it is possible to have unions of types with the same constructor $\widetilde{\cup}$ should also be idempotent. Note that these properties are met *only* when the union is well defined.

Property 3.1. $A \mathbin{\widetilde{\cup}} A :=: A$ *(Idempotence)*
For all expressions e, if $e : A \mathbin{\widetilde{\cup}} A$ *then* $e : A$*, this is possible because union types may consist of types which contain the same constructor.*

Proof. $A \mathbin{\widetilde{\cup}} A :=: A$ is correct when $A \mathbin{\widetilde{\cup}} A <: A \wedge A <: A \mathbin{\widetilde{\cup}} A$.

- $A <: A \mathbin{\widetilde{\cup}} A$ Follows trivial from the rules Sub I_1 and Sub I_2.
- $A \mathbin{\widetilde{\cup}} A <: A$ With the use of rule Sub I_3

$$\frac{\Gamma \vdash A \mathbin{\widetilde{\cup}} A \quad \Gamma \vdash A <: A \quad \Gamma \vdash A <: A}{\Gamma \vdash A \mathbin{\widetilde{\cup}} A <: A}$$

□

Property 3.2. $A \mathbin{\widetilde{\cup}} B :=: B \mathbin{\widetilde{\cup}} A$ *(Commutativity)*
For all expressions e, if $e : A \mathbin{\widetilde{\cup}} B$ *then* $e : B \mathbin{\widetilde{\cup}} A$*.*

Proof. $A \mathbin{\widetilde{\cup}} B :=: B \mathbin{\widetilde{\cup}} A$ is correct when $A \mathbin{\widetilde{\cup}} B <: B \mathbin{\widetilde{\cup}} A \wedge B \mathbin{\widetilde{\cup}} A <: A \mathbin{\widetilde{\cup}} B$.

- $A \mathbin{\widetilde{\cup}} B <: B \mathbin{\widetilde{\cup}} A$

  $$\frac{\Gamma \vdash A, B, A \mathbin{\widetilde{\cup}} B, B \mathbin{\widetilde{\cup}} A \quad \Gamma \vdash A <: B \mathbin{\widetilde{\cup}} A \quad \Gamma \vdash B <: B \mathbin{\widetilde{\cup}} A}{\Gamma \vdash A \mathbin{\widetilde{\cup}} B <: B \mathbin{\widetilde{\cup}} A}$$

  where $\Gamma \vdash A, B$ is the shorthand notation of $\Gamma \vdash A \quad \Gamma \vdash B$
- $B \mathbin{\widetilde{\cup}} A <: A \mathbin{\widetilde{\cup}} B$ is analogous.

□

Property 3.3. $A \mathbin{\tilde{\cup}} \big(B \mathbin{\tilde{\cup}} C\big) :=: \big(A \mathbin{\tilde{\cup}} B\big) \mathbin{\tilde{\cup}} C$ *(Associativity)*
For all expressions e, if $e : A \mathbin{\tilde{\cup}} (B \mathbin{\tilde{\cup}} C)$ then $e : (A \mathbin{\tilde{\cup}} B) \mathbin{\tilde{\cup}} C$.

Proof. $A \mathbin{\tilde{\cup}} (B \mathbin{\tilde{\cup}} C) :=: (A \mathbin{\tilde{\cup}} B) \mathbin{\tilde{\cup}} C$ is correct when $A \mathbin{\tilde{\cup}} (B \mathbin{\tilde{\cup}} C) <: (A \mathbin{\tilde{\cup}} B) \mathbin{\tilde{\cup}} C \wedge (A \mathbin{\tilde{\cup}} B) \mathbin{\tilde{\cup}} C <: A \mathbin{\tilde{\cup}} (B \mathbin{\tilde{\cup}} C)$

- $A \mathbin{\tilde{\cup}} (B \mathbin{\tilde{\cup}} C) <: (A \mathbin{\tilde{\cup}} B) \mathbin{\tilde{\cup}} C$ is shown in Figure 3.2. Provided that all types are correct, and do not conflict.
- $(A \mathbin{\tilde{\cup}} B) \mathbin{\tilde{\cup}} C <: A \mathbin{\tilde{\cup}} (B \mathbin{\tilde{\cup}} C)$ can be constructed in the same way and will not be shown.

□

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash A}{\Gamma \vdash A <: A\tilde{\cup}B}}{\Gamma \vdash A <: (A\tilde{\cup}B)\tilde{\cup}C} \qquad \dfrac{\Gamma \vdash B\tilde{\cup}C \qquad \dfrac{\dfrac{\Gamma \vdash B}{\Gamma \vdash B <: A\tilde{\cup}B}}{\Gamma \vdash B <: (A\tilde{\cup}B)\tilde{\cup}C} \qquad \dfrac{\Gamma \vdash C}{\Gamma \vdash C <: (A\tilde{\cup}B)\tilde{\cup}C}}{\Gamma \vdash B\tilde{\cup}C <: (A\tilde{\cup}B)\tilde{\cup}C}}{\Gamma \vdash A\tilde{\cup}(B\tilde{\cup}C) <: (A\tilde{\cup}B)\tilde{\cup}C}$$

Figure 3.2: Proof of associativity

3.6 Union Types and Functions

Parallel function composition, the combination of functions using union types, is an important part of the use of algebraic unions. When unions are defined and several functions are defined on all constituents of the union, it should be possible to use these functions over the parallel composition as well.

```
showT1 :: t1 -> [char]
showT1 C1 = "C1"
showT1 C2 = "C2"

showT2 :: t2 -> [char]
showT2 C3 = "C3"
showT2 C4 = "C4"
```

Code 3.8: Show Functions

Suppose $t1$ and $t2$ are defined like in Code 3.9. Furthermore suppose that `showT1` and `showT2` are defined like in Code 3.8. Now suppose $t = t1 \mathbin{\tilde{\cup}} t2$, then one would like to be able to combine `showT1` and `showT2` into a function `showT` that works on the full type t. To do this I will introduce a new operator $\widetilde{F}$ [2] such that `showT` can be defined as: `showT` = `showT1` $\widetilde{F}$ `showT2`.

```
t1 ::= C1 | C2
t2 ::= C3 | C4
```

Code 3.9: Algebraic types

The type of the function operator $\widetilde{F}$ and $\widetilde{G}$ [3] are:

$$\widetilde{F} : (A \to C) \to (B \to C) \to (A \mathbin{\widetilde{\cup}} B) \to C$$
$$\widetilde{G} : (A \to B) \to (A \to C) \to A \to (B \mathbin{\widetilde{\cup}} C)$$

where the function operators $\widetilde{F}$ and $\widetilde{G}$ are defined as:

$$\begin{array}{lll} (f\ \widetilde{F}\ g)\ x & = f\ x & \text{, if } x :: A \\ & = g\ x & \text{, otherwise} \\ (f\ \widetilde{G}\ g)\ x & = f\ x & \text{, if } x \in \mathit{dom}(f) \\ & = g\ x & \text{, otherwise} \end{array}$$

Note that from the definition of $\widetilde{G}$ is clear that when the domains of the functions overlap the function named first in the composition has precedence over the second function. In Code 3.10 the use of the operator $\widetilde{G}$ is shown. The operator $\widetilde{G}$ uses partial functions to create a function that is the parallel composition of two functions of the constituents of the union. A partial function is a function that is only defined on a part of its domain. For instance when `readT1` is defined as a total function as is done in Code 3.11 it is not useful to use this function in a parallel composition, because `readT` will never result in a value of type $t2$, which most likely is unwanted behaviour.

Note that the following operator $\widetilde{+}$ can be constructed using $\widetilde{F}$.

$$\widetilde{+} : (A \to C) \to (B \to D) \to (A \mathbin{\widetilde{\cup}} B) \to (C \mathbin{\widetilde{\cup}} D)$$

Suppose $h = f \mathbin{\widetilde{+}} g$ where $f : A \to C$ and $g : B \to D$. Because of the subtyping rules given in Section 3.4 the type of f is also $f : A \to C \mathbin{\widetilde{\cup}} D$ and the type of g is also $g : B \to C \mathbin{\widetilde{\cup}} D$, thus $h = f\ \widetilde{F}\ g$.

It should be noted that the

[2]When this operator is used in program text, the union operator $\widetilde{F}$ will be displayed as `<F>`

[3]likewise $\widetilde{G}$ will be displayed as `<G>` in program text

```
readT1 :: [char] -> t1
readT1 "C1" = C1
readT1 "C2" = C2

readT2 :: [char] -> t2
readT2 "C3" = C3
readT2 "C4" = C4

readT :: [char] -> t
readT = readT1 <G> readT2
```

Code 3.10: Composing functions with <G>

```
readT1 :: [char] -> t1
readT1 "C1" = C1
readT1 _    = C2
```

Code 3.11: Incorrect function

3.7 Equivalence

There are two major kinds of type systems, the structural type system and the nominative type system. Nominative typing means that two variables have an equivalent type only if they appear either in the same declaration or in declarations that use same type name. Java, C and C++ are examples of such languages. On the other hand, in the structural type system, used by Haskell and Amanda for instance, two terms are considered to have compatible types when the *structure* of the types is identical. In Code 3.12 the values x and y are of the same type, whereas x and y in Code 3.13 are not.

```
a == (num, [char])
b == (num, [char])

x :: a
x = (42, "Hello")

y :: b
y = (42, "Hello")
```

Code 3.12: Structural Typing

Note that structural equivalence corresponds to the definition of equivalence for unions as given in Section 3.2.

```
public class A {
  int i    = 0;
  String s = "";

  public A(int i, String s) { this.i = i; this.s = s; }
}

public class B {
  int i    = 0;
  String s = "";

  public B(int i, String s) { this.i = i; this.s = s; }
}

A x = new A(42,"Hello");
B y = new B(42,"Hello");
```

Code 3.13: Nominative Typing in Java

4

Naïve Implementation of Unions

4.1 Introduction

In order to be able to use streams in a more general way, it must be possible to extend the event types in the application. This can be done with the aid of untagged algebraic unions. In this chapter, I will give a rewriting mechanism to rewrite functions and algebraic types to a format that is executable in a functional programming language environment.

In short this rewriting scheme, and the one in the next chapter, rewrite a language with unions to a language without these unions. Such that every expression in the language with unions has a type, that can be converted to a type in the language without unions.

In this chapter the language Amanda with unions will be rewritten to core Amanda.

4.2 Outline

A preprocessor has been written to convert the parse-tree obtained using the Tinadic Amanda parser, described in Chapter D. The parse tree is processed in the following way:

1. All union types are rewritten to an algebraic type with extra constructors (4.3).
2. All functions that define a parallel function composition with a union as an argument are converted to a 'normal' function that forwards the union

argument to the appropriate function. (4.4).

3. All functions that define a parallel function composition with a union as result type are converted to a 'normal' function by merging the functionality of the functions. There is a distinction between functions that are distinct on the patterns, and those that are distinct on the guards (4.5).

4. All functions that use an argument or give a result which is a union type must be wrapped by adding constructors (4.6).

The rewriting rules are depicted in the following way. The term to be rewritten is denoted: $\mathcal{R}_x[\![\cdot]\!]$ and is shown above the first black line. Between two lines the rewritten term is shown. The enumeration below shows the assumptions made about the rewriting term. The x denotes what kind of rewriting rule is currently being processed. During the transformation the rule may shift (parts) of the expression that must be rewritten to a rule with another name. The overall rule which rewrites the whole program is $\mathcal{R}_A$. This rule rewrites all expressions in the source file. This notation resembles the one used in (Peyton Jones, 1987) and (Oosterhof, 2005). The rewriting rules to evaluate a program without algebraic unions are not shown here. The result of this preprocessing step will be a source code file that can be compiled and evaluated in a functional language.

4.3 The Algebraic Union

Rewriting of an algebraic union is quite simple. It can be done as depicted in Figure 4.1. Thus a union type is rewritten to a normal algebraic type. Note that the use of extra constructors does not solve commutativity and associativity of union types. This must be solved using conversion functions and should be addressed when implementing this in a core functional language. Note that this rewriting rule is not addressed recursively. This is because every union must be defined. The union $T = A \mathbin{\widetilde{\cup}} (B \mathbin{\widetilde{\cup}} C)$ can only be defined as: $T = A \mathbin{\widetilde{\cup}} D$ where $D = B \mathbin{\widetilde{\cup}} C$.

4.4 Functions with a Union as argument

When the type of one of the arguments of the function is a union type the function must be rewritten like in Figure 4.2. The result of the rewriting rule is in Code 4.1. It is clear that the created function after complete rewriting will only delegate its argument to the appropriate function according to its inner value.

$\mathcal{R}_A[\![t = a \texttt{ <U> } b]\!]$
$t ::= \texttt{Nr\_}t\texttt{\_0}\ a$ $\quad\mid \texttt{Nr\_}t\texttt{\_1}\ b$
• The extra constructor $\texttt{Nr\_}t\texttt{\_}i$ is based on the type of the union (t) and a number (i).   • $\texttt{Nr\_}t\texttt{\_}i$ is a new unique constructor.   • Add the tuple (t,y,$\texttt{Nr\_}t\texttt{\_}i$) to the *replacement* list. Where y is the type of the constituent of the union, in this case a or b.

Figure 4.1: Rewriting rule for union introduction

$\mathcal{R}_A[\![f = g \texttt{ <F> } h]\!]$
$f\ (\texttt{Nr\_}t\texttt{\_0}\ x) = \mathcal{R}_P[\![g\ x]\!]$ $f\ (\texttt{Nr\_}t\texttt{\_1}\ x) = \mathcal{R}_P[\![h\ x]\!]$
• Where x is a unique identifier   • the type of f is t −> a where a is an arbitrary type.

Figure 4.2: Rewriting rule for parallel function composition

```
f :: t -> a
f (Nr_t_0 x) = g x
f (Nr_t_1 x) = h x
```

Code 4.1: Function after complete rewriting when the result type is no union

4.5 Functions with a Union as result type

When the result type of the parallel composed function is a union type, the function to be created needs to have all the clauses of the functions that are being composed. There are two cases, either the argument is distinguished by a pattern (Section 4.5.1) or by a guard (Section 4.5.2).

4.5.1 Pattern Functions

When the functions to be merged differ on the way patterns are matched in the clauses, the functions can be merged by just enumerating the clauses of the functions in the new function, as can be seen in Figure 4.3 and Code 4.2 and 4.3. When there is an overlap in patterns between the first and the second function, the first function has precedence over the second function. Note that the language that implements this rewriting rules should be able to enumerate clauses that overlap.

```
f :: char -> t
f = g <F> h

g 'a' = C1
g 'b' = C2

h 'c' = C3
h 'd' = C4
```

Code 4.2: Pattern function

```
f :: char -> t
f 'a' = C1
f 'b' = C2
f 'c' = C3
f 'd' = C4
```

Code 4.3: Rewritten pattern function

4.5.2 Guard Functions

When the function clauses are distinguished by guards then the new function is defined by all clauses of the existing functions. Note that the identifiers in the functions need not be the same and need to be unified in the where clause. The rewriting rule is defined in Figure 4.4 an example is shown in Code 4.4 and 4.5.

4.5.3 Normal Functions

All functions that use unions must be rewritten because the internal representation of the union is different from the internal representation of elements of the

$\mathcal{R}_A[\![f = g \texttt{ <G> } h]\!]$

$f\ x_1 = \texttt{Nr\_}t\texttt{\_0(}\ g_1\ \texttt{)}$
$\vdots$
$f\ x_n = \texttt{Nr\_}t\texttt{\_0(}\ g_n\ \texttt{)}$
$f\ y_1 = \texttt{Nr\_}t\texttt{\_1(}\ h_1\ \texttt{)}$
$\vdots$
$f\ y_k = \texttt{Nr\_}t\texttt{\_1(}\ h_k\ \texttt{)}$

- g is defined in the following way:
  $\texttt{g}\ x_1 = \texttt{g}_1$
  $\vdots$
  $\texttt{g}\ x_n = \texttt{g}_n$
- h is defined in the following way:
  $\texttt{h}\ y_1 = \texttt{h}_1$
  $\vdots$
  $\texttt{h}\ y_k = \texttt{h}_k$
- The type of f is c $->$ t. Where c is not a union type and t is $a\ \widetilde{\cup}\ b$.
- $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ are patterns
- The precedence of function g over function h is defined implicitly by naming the clauses of g before the clauses of h.

Figure 4.3: Rewriting rule for pattern functions

unionelement, i.e. constructors are added to elements of the union to distinguish between the elements. In Figure 4.5 and Figure 4.6 this rules are shown. The rules forward the pattern rewriting and expression rewriting to the $\mathcal{R}_P$-rule.

4.6 Patterns

When an expression is supposed to have a union type, but instead has a type that is an element of a union, a constructor must be added to this expression. The rewriting rule can be seen in Figure 4.7 and Figure 4.8. Note that the rule is defined recursively, because unions might be nested.

$\mathcal{R}_A[\![f = g \texttt{ <G> } h\ ]\!]$
$f\ x = \texttt{Nr\_}t\texttt{\_0(}\ \mathcal{R}_P[\![g_1]\!])\ ,\ \textbf{if}\ b_1$ $\vdots$ $= \texttt{Nr\_}t\texttt{\_0(}\ \mathcal{R}_P[\![g_n]\!])\ ,\ \textbf{if}\ b_n$ $= \texttt{Nr\_}t\texttt{\_1(}\ \mathcal{R}_P[\![h_1]\!])\ ,\ \textbf{if}\ v_1$ $\vdots$ $= \texttt{Nr\_}t\texttt{\_1(}\ \mathcal{R}_P[\![h_k]\!])\ ,\ \textbf{if}\ v_k$ **where** fs gs hs
• g is defined in the following way: $g\ x = g_1\ ,\ \textbf{if}\ b_1$ $\vdots$ $= g_n\,,\ \textbf{if}\ b_n$ **where** gs • h is defined in the following way: $h\ y = h_1\ ,\ \textbf{if}\ v_1$ $\vdots$ $= h_n\,,\ \textbf{if}\ v_k$ **where** hs • gs and hs are possibly empty.   • fs is $y = x$ when the identifier y does not have the same name as x. Otherwise fs is empty.   • If g and h contain the same variables, these must be renamed to unique variables (in function g and h).   • The precedence of function g over function h is defined implicitly by naming the clauses of g before the clauses of h.

Figure 4.4: Rewriting rule for case functions

$\mathcal{R}_A[\![f\ x_1 \ldots x_n = e]\!]$
$f\ \mathcal{R}_P[\![x_1]\!] \ldots\ \mathcal{R}_P[\![x_n]\!] = \mathcal{R}_P[\![e]\!]$

Figure 4.5: Function rewriting rule

```
f :: char -> t
f = g <F> h

g x = C1 , if p = 'a'
    = C2 , if p = 'b'
      where
        p = x

h y = C3 , if q = 'c'
    = C4 , if q = 'd'
      where
        q = y
```

Code 4.4: Case function

```
f :: char -> t
f x = C1 , if p = 'a'
    = C2 , if p = 'b'
    = C3 , if q = 'c'
    = C4 , if q = 'd'
     where
       y = x
       p = x
       q = y
```

Code 4.5: Rewritten case function

$\mathcal{R}_P[\![f\ x_1 \ldots x_n]\!]$
$f\ \mathcal{R}_P[\![x_1]\!] \ldots \mathcal{R}_P[\![x_n]\!]$

Figure 4.6: Function application rewriting rule

$\mathcal{R}_P[\![e]\!]$
$C\ \mathcal{R}_P[\![e_1]\!] \ldots \mathcal{R}_P\ [\![e_n]\!]$
• e should have union type t • e has type a • C is the constructor out of the list *Replacements* where the tuple is (t,a,C).

Figure 4.7: Rewriting rule to add constructors

$\mathcal{R}_P[\![e]\!]$
$C\ \mathcal{R}_P[\![e_1]\!] \ldots \mathcal{R}_P\ [\![e_n]\!]$
• e has type a • There is no b for which $a = b$ in the list *Replacements* where the tuple is (t,b,C).

Figure 4.8: Rewriting rule for constructors that are not part of a union

```
f :: t -> [char]
f C1 = "C1"
f C2 = "C2"
f C3 = "C3"
f C4 = "C4"
```

Code 4.6: Expression

```
f :: t -> [char]
f (Nr_t_0 (C1)) = "C1"
f (Nr_t_0 (C2)) = "C2"
f (Nr_t_1 (C3)) = "C3"
f (Nr_t_1 (C4)) = "C4"
```

Code 4.7: Expression with constructor

4.7 Preprocessor

The rewriting rules are implemented using a preprocessor in Amanda. This preprocessing program rewrites the source code with unions to a file in which unions are replaced, as shown in the rewring rules in this chapter. A full description of the preprocessor is shown in Appendix D.

The preprocessor converts a program which may contain unions along the lines of the rewriting rules given in this chapter to a file that contains no unions. In Chapter 6 it is shown how the program contains unions behaves. The preprocessing step and the compilation make it possible to execute this program without having to rewrite the program. This shows that it is possible to add unions to the language.

A large downside of the preprocessor in Amanda is that the union is not commutative. The use of a type t which is defined as $a \mathbin{\widetilde{\cup}} b$ is different of the use of $b \mathbin{\widetilde{\cup}} a$ when used in functions that are composed. Worst still, functions that are defined on $a \mathbin{\widetilde{\cup}} b$ are not defined on $b \mathbin{\widetilde{\cup}} a$. Furthermore although a function with overlapping domains can be defined according to the rules, Amanda might give an error when functions are merged that have overlapping patterns.

5

Extensive Implementation of Unions

5.1 Type Classes

The rewriting scheme in the previous chapter is implemented in a simple way. This, however, creates some problems that can not be solved immediately, like the fact that unions are not commutative. In this chapter I will show that with some features particular to Haskell it is possible to create unions that are commutative, associative and idempotent. Furthermore I will show that in most cases the implementation of type classes on constituents of unions can be implemented on unions in a default, mechanic way.

In this section and the next two sections I will describe some features that are part of the Glasgow Haskell Compiler. After that I will show that these features can be used to implement unions in Haskell, more specific in the Glasgow Haskell Compiler (GHC).

There are two kinds of polymorphism present in Haskell, *parametric* polymorphism and *ad hoc* polymorphism (Hudak & Fasel, 1992), better known as *overloading*. Parametric polymorphism is the kind of polymorphism that occurs for instance with the function **length**. This function returns the length of a list, regardless of the type of the elements in the list. The type of **length** is therefore `[a]` −> **Int**. Examples of ad hoc polymorphism are

- The literals 1, 2, etc. that are used to represent both fixed and arbitrary precision integers.
- Numeric operators such as + that are often defined to work on many different kind of numbers.
- The equality operator that is defined on many (but not all) types.

Note that these overloadeded behaviours differ for each type, whereas in parametric polymorphism the type truly does not matter. In Haskell type classes are used to structure this overloading. For instance by the type class **Eq** which is defined as shown in Code 5.1. Note the default implementation of `/=` in terms of `==`.

```
class Eq a where
  (==)      :: a -> a -> Bool
  (/=)      :: a -> a -> Bool
  x /= y   = not (x == y)
```

Code 5.1: The type class **Eq**

Now instances for these type class can be implemented. For instance the implementation to define equality over the **Bool**-type is shown in Code 5.2.

```
instance Eq Bool where
  (==) True  True  = True
  (==) False False = True
  (==) _     _     = False
```

Code 5.2: Implementation of **Eq Bool**

It is not only possible to implement type classes on base types, it is also possible to implement type classes on more complex types. Consider the implementation of **Eq**, as shown in Code 5.3 on the algebraic type `Tree`.

```
instance (Eq a) => Eq (Tree a) where
  (==) (Leaf x) (Leaf y)           = x == y
  (==) (Node a1 a2) (Node b1 b2) = (a1 == b1) && (a2 == b2)
  (==) _            _             = False
```

Code 5.3: Implementation of **Eq** `Tree`

From now on it is possible to use the function == from the type class **Eq** on a value of the polymorphic type `Tree a` provided that the type class **Eq** is implemented on `a`, denoted by =>.

A simplified possible definition of the **Ord** type class is shown in Code 5.4. The **Ord** type class specifies functions that create an order between values. As can be seen instances of this type class must also implement the type class **Eq**, with functions `==` (equal) and `/=` (not equal), besides the functions `>`, `<`, `<=` and `>=`. Note that only the function `<=` needs to be defined on types implementing this type class, in which case the default implementations shown here are used for the other functions in **Ord**. Arguments and results of functions can be restricted to types that implement type classes. Compare the Haskell type definition of quicksort in Code 5.5 (the top one) with the Amanda type definition (the bottom one). The Haskell type definition restricts the types that can be sorted using quicksort to types that implement the **Ord** type class. In the Amanda representation no extra constraints can be added to the type of a function, therefore it is possible to sort a list of functions in Amanda, although no specification of the order of functions is given in Amanda. Worse still, in Amanda the equality operation on functions is not referentially transparent. The expression (ˆ2) = (ˆ3) might result randomly in true or false, which obviously is incorrect.

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

  (<)        :: a -> a -> Bool
  (<) x y = (x <= y) && (x /= y)
  (>)        :: a -> a -> Bool
  (>) x y = not (x <= y)
  (>=)       :: a -> a -> Bool
  (>=) x y = not (x <= y) || (x == y)
```

Code 5.4: Simplification of the **Ord** type class

```
qsort :: Ord a => [a] -> [a]

qsort :: [*] -> [*]
```

Code 5.5: Function type restricted by type classes

The type classes restrict the use of the sort function to types that have a meaningful implementation of the ordinality functions. This means, because the type class **Ord** is not implemented on functions, that it is not possible to sort functions.

To help the programmer, some type classes can be *derived*, using the keyword **deriving**. For a lot of types that are introduced by the programmer the implementation for certain type classes are very straightforward. For instance, the default implementation of the **Eq** type class for an algebraic data type is obviously that two algebraic constructors are equivalent when the constructor name is the same and all arguments of the constructor are equivalent. When the programmer needs to implement this for every type, this will become very awkward. Therefore the language alleviates the programmer from implementing this every time again by the **deriving** construct. Type classes that can be derived are for instance the **Show**, **Eq**, **Read** and **Typeable** type class. The **Eq** type class for the type **Bool** could be derived, when it was not already implemented, as done in Code 5.6 although it could also be defined as done above in Code 5.2.

```
data Bool = True
          | False
          deriving (Eq)
```

Code 5.6: The type **Bool** deriving **Eq**

5.2 Generics

In GHC a module can be used that enables *casting* of types. The theoretical background of this generic approach is described in (Lämmel & Peyton Jones, 2003). According to Lämmel & Peyton Jones when a function is written to traverse over a mutually recursive data structure this creates a lot of boilerplate code. To give a notion of what boilerplate code is, consider the following example taken from (Lämmel & Peyton Jones, 2003). Suppose you have to write a function that increases the salary of every person in an organization and persons are stored in a mutually recursive data structure which represents the organization. The code of the function that increases the salary will be dominated by code that traverses the data structure, the boilerplate code, whereas the increasing of the salary will be handled probably in one or two lines of code. In (Lämmel & Peyton Jones, 2003) a technique is introduced to write this boilerplate code once and use it in every function that traverses the data structure.

The implementation of the technique to specify the boilerplate code in one place needs a generic *cast* function. This function `cast` can be used to cast a value of a certain type to another type. Because it is not sure whether the value can be cast to this other type, the result of the `cast` function will be a value of the **Maybe** monad. The definition of the `cast` function is shown in Code 5.7. The function `get` is used to get the type representation of the result of the function. As can be seen the function `cast` depends on the type class **Typeable**. This means the `cast` function is only defined on types that implement the **Typeable** type class. This type class consist of a function `typeOf`, that evaluates a value to a representation of its data type. The **Typeable** type class creates a way to use the type of the function and act according to the type. This type representation is stored in string format and is used in the `cast` function to create a type safe cast function, based on the unsafe cast function that is already part of the Haskell language. In Code 5.8 a number of expressions given to the interactive `GHCi` are shown to illustrate the use of `cast`. The lines starting with > are the expressions given to the `GHCi` whereas the other lines are the results. It might seem that this cast function has not much use, because it boils down to the identity function with another type, yet in the Section 5.4 I will show that this `cast` function can be used to implement unions.

```
cast :: (Typeable a, Typeable b) => a -> Maybe b
cast x = r
         where
          r = if (typeOf x) == (typeOf(get r))
              then Just (unsafeCoerce x)
              else Nothing
          get :: Maybe a -> a
          get x = undefined
```

Code 5.7: Generic cast Function

```
> (cast 'a') :: Maybe Char
Just 'a'
> (cast 'a') :: Maybe Bool
Nothing
> (cast [True,False,True]) :: Maybe [Bool]
Just [True,False,True]
```

Code 5.8: Use of the cast Function

5.3 Pattern Guards

Patterns play an important role in functional programming languages to distinguish the clauses of function definitions. Combined with guards, patterns determine the appropriate function clause based on the value of the argument. They also create very readable and extensible code, in Code 5.9 it is easy to extend the function with another clause that handles a new constructor; Green, Blue, White, etc, without having to alter existing code.

```
toString :: color -> [char]
toString (Yellow   ) = "Yellow"
toString (Black    ) = "Black"
toString (Red      ) = "Red"
toString (RGB r g b) = "#" ++ (itoa r) ++ (itoa g) ++ (itoa b)
```

Code 5.9: Pattern Function

The internal representation of unions ($\tilde{\cup}$) with extra constructors as shown in Chapter 4 lead to problems using patterns. The 'normal' algebraic patterns, like given in Code 5.9 `RGB r g b` can not be used with the implementation of unions with extra algebraic constructors. It might be the case that the pattern belonging to the internal representation of the union is `L(RGB r g b)`. Where `L` is an internal constructor to distinguish between the components of the union. The programmer does not know, and does not want to know this internal representation and can therefore not use the 'normal' algebraic patterns.

To alleviate this problem *pattern guards* (Erwig & Peyton Jones, 2000) will be used. The idea of Erwig & Peyton Jones is to extend guards in a way that guards are able to match patterns and bind variables. The expression on the right hand side of the pattern guard, i.e. on the right hand side of the <− is evaluated and matched against the pattern on the left. If the match fails, the next clause is tried. Note that there is a difference in what way patterns and guards are handled in Amanda and Haskell. In Appendix F this and some other differences between Amanda and Haskell are addressed.

```
--               key -> [(key,value)] -> Maybe value
lookup :: Eq a => a -> [(a, b)] -> Maybe b

myAdd :: [(Int,Int)] -> Int -> Int -> (Maybe Int)
myAdd env x y
        | Just a <- lookup x env
        , Just b <- lookup y env
          = Just (a + b)
        | otherwise
          = Nothing
```

Code 5.10: Use of a Pattern Guard

In Code 5.10 the use of pattern guards is shown. The function **lookup** is a standard function in the **Prelude** of Haskell. When the given list contains the key (a), then the result of **lookup** will be the value belonging to this key, otherwise the result will be **Nothing**. The result of the lookup of the keys x and y are matched against **Just** a and **Just** b respectively. When the lookup of the values of x and y do not result in **Nothing** the result values are bound to a and b. This implies that the expressiveness of guards is extended with the possibility of variable binding and pattern matching. In GHC pattern guards are implemented when using extensions of the compiler. This can be done using the directive `-fglasgow-exts`. The other part of (Erwig & Peyton Jones, 2000) about *transformational patterns*, which could be used also to implement unions instead of pattern guards, is not (yet) implemented in the GHC-compiler.

5.4 Implementing Unions using Pattern Guards and Generics

A *basic* algebraic type is a type that defines algebraic constructors directly, the algebraic type `Tree` in Code 5.11 is an example of a basic algebraic type. A *union* type is a type that is a type that is a composition of two types in a union. An algebraic type therefore is more basic when it contains less unions.

The combination of pattern guards, type classes and generics can hide the internal representation of unions. With generics and pattern guards the values of the most basic types can be retreived from an arbitrary complex union.

```
data Tree = Node Tree Tree
          | Leaf Int
```

Code 5.11: A basic Algebraic Type

Now, values of a union can be addressed in the same way as values of non-union algebraic data types. Thus the programmer does not have to know about the internal representation of unions. To be able to do this a number of rewriting steps must be taken, as presented in the next sections.

The general idea is to define a new type class, named **Algebraic** that must be implemented for each algebraic type. This type class consists of functions to convert the algebraic type to a basic algebraic type, and vice versa, to convert an algebraic type to a union type.

In Figure 5.1 the function that traverses the union down to the most specific (basic) type, named `downCast`, is depicted. Conversely, there is a function `upCast` that is used to create a union type from a basic type. In other words, the traversal down gives a more specific type, i.e. the type of the value is restricted to a part of the union, whereas traversal up gives a more general type, i.e. the type of the whole union.

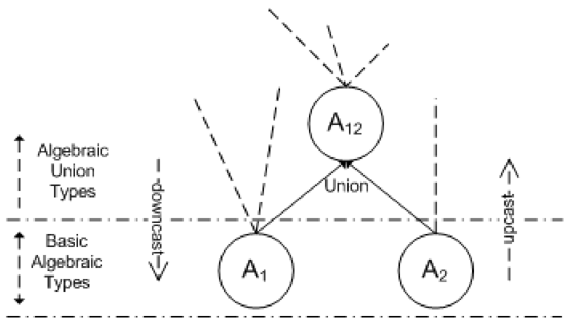


Figure 5.1: Algebraic union

5.4.1 Algebraic Type

Each algebraic data type is a member of the type class **Algebraic**. This type class consists of three functions:

- `upCast` converts a value of a more basic type to a union type.
- `downCast` converts a value of a union type to a more basic type.
- `eq` compares two values that are instance of type class **Algebraic** with each other, to determine whether the values are the same.

The definition and default implementation of type class **Algebraic** is depicted in Code 5.12.

```haskell
class (Eq a, Typeable a) => Algebraic a where

    {- Cast the value of this type to a more general type. -}
    upCast :: (Algebraic b) => b -> Maybe a
    upCast = cast

    {- Cast the value of this type to a more specific type. -}
    downCast :: (Algebraic b) => a -> Maybe b
    downCast = cast

    {- A custom equality function, compares two Algebraic values.-}
    eq :: (Algebraic b) => a -> b -> Bool
    x `eq` y = Just x == (downCast y :: Maybe a)
```
Code 5.12: Type class **Algebraic**

To ensure that the `cast` function can be used, the algebraic type must implement the **Typeable** type class. Every algebraic type needs to be an instance of the **Algebraic** type class, therefore a rewriting rule is needed that implements the **Algebraic** type class for an algebraic type. This rule is given in Figure 5.2. Besides the implementation of the **Algebraic** type class functions are also needed that, based on values of the **Algebraic** type class, may give the arguments of the constructor as result. When the correct constructor is given to this function the result will be the arguments of the function, otherwise the result will be **Nothing**. This rule is shown in Figure 5.3. These functions are needed to replace the algebraic patterns, the rule to rewrite the algebraic patterns to patterns that use these functions is depicted in Figure 5.10.

$\mathcal{R}[\![\textbf{data}\ T = C_1\ \overline{c_1}|\ldots|C_n\ \overline{c_n}]\!]$

$\textbf{data}\ T = C_1\ \overline{c_1}|\ldots|C_n\ \overline{c_n}$

instance Algebraic T

$\mathcal{R}_f([\![C_1\ \overline{c_1}]\!], T)$

$\quad\vdots$

$\mathcal{R}_f([\![C_n\ \overline{c_n}]\!], T)$

- $\overline{c_i}$ is the sequence of all arguments of constructor C_i. This sequence can be empty.

Figure 5.2: Rewriting rule for Algebraic Data Types

$\mathcal{R}_f([\![C\ t_1 \ldots t_n]\!], T)$

```
from$C$ :: (Algebraic a) =>a -> Maybe $(t_1, \ldots, t_n)$
from$C$ $x$
  | Just $(C\ t_1 \ldots t_n)$ <- downCast $x$ :: Maybe $T$)
    = Just $(t_1, \ldots, t_n)$
  | otherwise
    = Nothing
```

- When the number of arguments of constructor C is 0, the result type will be "**Maybe** ()".
- `from`C is a unique identifier.

Figure 5.3: Creation of functions to extract arguments from a constructor

5.4.2 Union

To be able to create unions of algebraic types a new `Union` type is created. The full implementation of this union type is shown in Section A and part of its implementation is shown in Code 5.13.

```
data Union a b = L a
               | R b
```

Code 5.13: The Union Definition

The union type must implement the **Algebraic** type class, to be able to use the union in functions just like other algebraic types. The implementation of the **Algebraic** type class is shown in Code 5.15.

The function `unionIdentify` is a function that creates an **Either** type out of a union and is shown in Code 5.14. Note that this funtion is not commutative. I.e. the function `unionIdentify` will give a different result for `A` ‘`Union`‘ `B` and `B` ‘`Union`‘ `A`. A union can be seen as a specific kind of the **Either** type. It is not possible to use the **Either** type to implement the union because some type classes are already defined on **Either**, which must be defined in another way for a union. Also the internal representation must be concealed, which is not possible for the **Either** type.

```
unionIdentify :: a `Union` b -> Either a b
unionIdentify (L x) = Left  x
unionIdentify (R x) = Right x
```

Code 5.14: The function unionIdentify

```
instance (Algebraic l,Algebraic r)=>Algebraic (l `Union` r) where
  upCast x
    | Just z <- cast   x :: Maybe (l `Union` r)  = Just z
    | Just z <- upCast x :: Maybe l              = Just (L z)
    | Just z <- upCast x :: Maybe r              = Just (R z)
    | otherwise                                  = Nothing

  downCast = (either downCast downCast) . unionIdentify

  eq x y = either eq eq (unionIdentify x) y
```

Code 5.15: The implementation of the Union Type

The `upCast` and `downCast` are now implemented also on all union types. In this way they can be used like a non-union algebraic type.

Now a new union can be declared as in Code 5.16.

```
type T = A `Union` B
         deriving(Eq,Typeable)
```

Code 5.16: Declaration of a union

The union implements the type classes **Show**, **Typeable**, **Algebraic** and **Eq**.

When a function expects a union type as an argument, for instance a function `f` with type (`T` ‘`Union`‘ `S`) −> **String**, it is not possible to apply `f` on a value of the more basic algebraic type `T`. This is behaviour that should be possible though. Obviously when a function works on `T` ‘`Union`‘ `S` it should also work for values of type `T`. The algebraic constructor given as argument, however, is not the same as what is expected in the function. Suppose the algebraic type `T` is defined as `C` **Int**. Now the internal representation of a value of `T` in type `T` ‘`Union`‘ `S` is `L` (`C` **Int**) according to the definition of the `Union`. Obviously the type of these values are not the same. Therefore it is not possible to give `C` **Int** as an argument to `f`. A possible way to solve this is by using the fact that in Haskell an algebraic constructor merely is a *function* that creates an algebraic value. The constructor `C` therefore is a function with type **Int** −> `T`. This constructor-function can be altered to a function that creates a value of an arbitrary type that implements the type class **Algebraic** as shown in Code 5.17. To distinguish between the standard Haskell algebraic constructor function and the new constructor function, the new function is shown in lowercase. The new constructor function creates a value of the type class **Algebraic**. The function that uses a value that is constructed using this constructor function will fix the type of the constructor to a specific type. When `c` is used to give an argument to function `f` the constructor function `c` will return a value of type `T` ‘`Union`‘ `S`, i.e. `L(C` **Int**). This is because the function `upCast` function casts the value to a value of type `T` ‘`Union`‘ `S`.

```
data T = C Int

c :: (Algebraic a) => Int -> a
c x = (fromJust.upCast) (C x)
```

Code 5.17: Algebraic Constructor Function

One could argue that type classes on unions can be implemented using rewriting rules. A rewriting rule to create an instance of a type class for a union type is depicted in Figure 5.4. The instance for this type class for unions can be implemented in a straightforward way. A construction can be added to the language that shifts the implementation of the type class to the language level instead of bothering the programmer with a default implementation, just like the deriving construct. This construct is necessary, instead of implementing the classes by default, because in some cases it can be useful for a programmer to implement the type class in a non-default way. The difference with the **deriving** construct that is already part of the Haskell language, is that the **deriving** construct is used to implement a type class given a certain type, whereas in the derivation proposed above, the union must implement a given type class. One way to enable this derivation is by creating a new construction (`derive`) that can implement type classes for the union in a default way, this is shown in Code 5.18.

```
class A a where
  ...
  derive(Union)
```

Code 5.18: Deriving Type Classes for Union Type

A rule that can implement the type class for a union in a default way is shown in the next rules. The rule given in Figure 5.4 creates the instance for the union and distributes the rewriting of functions to the $\mathcal{R}_i$-rule, given in Figure 5.5, which creates the functions of the type class over the union type.

The $\mathcal{R}_i$ rule creates the function of the type class based on the function type.

$\mathcal{R}_A[\![$**class** $A\ a$ **where**
  $f_1\ ::\ T_1$
  $\vdots$
  $f_n\ ::\ T_n\ ]\!]$

class $A\ a$ **where**
  $f_1\ ::\ T_1$
  $\vdots$
  $f_n\ ::\ T_n$
instance $(A$ `l`, A `r`$)$ `=>` A (`l` ‘`Union`‘ `r`) **where**
  $\mathcal{R}_i([\![f_1\ ::\ T_1]\!],\ a)$
    $\vdots$
  $\mathcal{R}_i([\![f_n\ ::\ T_n]\!],\ a)$

Figure 5.4: The type class rewritten

When the arguments consist of unions, i.e. one of the arguments has type a, new guards are introduced, as shown in Figure 5.6. This rewriting rule uses a new notation:

$$e \lhd \mathsf{guard}_x^+ \leftarrow e'$$

This notation means that when rewriting an expression, the result of the rewriting rule will be the expression e with the expression e' added to the guard x. When x is not specified e' is added to the guard to which expression e belongs.

$\mathcal{R}_i([\![f\ ::\ T_1 \text{ -> } \ldots \text{ -> } T_n]\!],\ a)$

`f` $\mathcal{R}_p([\![x_1]\!],\ T_1,\ a)\ \ldots\ \mathcal{R}_p([\![x_n]\!],\ T_n,\ a)$
    $\mid\ \mathsf{guard}_l = \mathcal{R}_r([\![f\ y_1 \ldots y_n]\!],\ T_n, a)$
    $\mid\ \mathsf{guard}_r = \mathcal{R}_r([\![f\ y_1 \ldots y_n]\!],\ T_n, a)$

- guard_l and guard_r are filled by the $\mathcal{R}_r$-rule
- $y_1, \ldots, y_n$ are unique identifiers corresponding to $x_1, \ldots, x_n$, which are created in the $\mathcal{R}_P$-rule.

Figure 5.5: The functions in the type class rewriten

The rule to rewrite a pattern that has the type of the type class is shown in Figure 5.6.

Patterns that need not be rewritten are a single variable. The name of the variable is replaced to create a more simple rewriting scheme. In the rule depicted in Figure 5.6 the name of the variable needs to be changed, so it is convenient to do this in Figure 5.7 also.

$\mathcal{R}_P(\llbracket x_i \rrbracket,\ a,\ a)$
$y_i \lhd$ $\mathsf{guard}_l^+ \leftarrow$ **Just** y_i <− `downCast` x_i :: **Maybe** `l` $\lhd$ $\mathsf{guard}_r^+ \leftarrow$ **Just** y_i <− `downCast` x_i :: **Maybe** `r`
• y_i is a unique variable.

Figure 5.6: Rewriting the patterns of the type class

$\mathcal{R}_P(\llbracket x_i \rrbracket,\ a,\ b)$
y_i
• x_i is a variable.   • y_i is a new unique variable.   • $a \neq b$.

Figure 5.7: Rewriting patterns that do not have the type of the type class

Note that it is not possible to create a function this way that has other patterns than standard variables, like lists or tuples. The functions given in Code 5.19 can therefore not be created, because the rewriting rules are not given here. However lists and tuples are algebraic constructions, therefore it is easy to implement the typeclass algebraic on these types.

```
f :: [a]          -> String
g :: (String,a)   -> Int
```

Code 5.19: Patterns that can not be handled

The right hand side of functions also needs to be rewritten when the result of the function is **Maybe** a and the type class is instantiated over a. This is shown in Figure 5.8. The function is evaluated with all arguments being elements of either the left or the right side of the union. When the evaluated function yields a value when evaluating the function with only values that are in the internal representation part of the `L` constructor, the result of the whole function will be this value. Otherwise the function is evaluated with values from the internal `R` constructor. To ensure that the result of the function is a value a clause is added, as shown in Figure 5.8.

When the result of the function is not **Maybe** a then the function need not be rewritten as much as in Figure 5.8. The function is evaluated for the arguments given. These arguments will either be values of the left type of the union, or values of the right type of the union, this is forced by the guards. The evaluation of the function therefore is passed to the implementation of the type class on the left or right type of the union (Figure 5.9).

$\mathcal{R}_r([\![\texttt{f}\ y_1 \ldots y_n]\!], \textbf{Maybe}\ a, a)$
$\texttt{upCast}\ z \lhd \textsf{guard}^+ \leftarrow \textbf{Just}\ z <\!-\ f\ y_1 \ldots y_n$
• z is a new unique variable

Figure 5.8: Type class rewritten with union as result type

$\mathcal{R}_r([\![f\ y_1 \ldots y_n]\!], a, b)$
$f\ y_1 \ldots y_n$
• $a \neq b$.

Figure 5.9: Type class rewritten with union as result type

5.4.3 Functions

The mere need for unions is that two distinct types are merged because they share functionality or behaviour. When both types implement a certain type class and therefore functionality, the union must implement the same type class to have this functionality also.

When the arguments of the functions are elements of the union type, it is fairly simple to implement the type class on the union level, for instance the **Show** type class when implemented for both types can be easily implemented for an arbitrary union as well in Code 5.20

```
instance (Show a, Show b) => Show (Union a b) where
  show = (either show show) . unionIdentify
```

Code 5.20: Show

On the other hand when the result of a function is a union and the domains of the functions overlap it is harder to merge these functions. An example of a function that has a union as result type and has overlapping domains, is the function `read` in the type class **Read**. The function `read` has type: **String** −> `a`. The compiler can not determine, based on the string value, which function must be used. It might be the case that when trying the function implementation of the left type of the union results in an error, whereas the implementation of the right type would result in a correct value, or vice versa. Code 5.21 shows one way to merge functions that have a union as a result and have overlapping domains, by giving results wrapped in the **Maybe** monad.

When it is possible to handle exceptions in Haskell without the **IO** monad using **try** and **catch** the **Maybe** type is not necessary anymore. In Haskell it is possible to throw exceptions in all functions, yet they can only be handled with the use of the **IO** monad (Peyton Jones, 2005). The **IO** monad enforces that an error thrown is evaluated, which boils down to the lazy evaluation of Haskell. The **IO** monad, however, is the very thing we want to get rid of. However it

```
class MRead a where
  mread :: String -> Maybe a

instance (Mread l,Mread r) => Mread (l `Union` r) where
  mread s
    | Just y <- (mread s :: Maybe l) = upCast y
    | Just y <- (mread s :: Maybe r) = upCast y
    | otherwise                      = Nothing
```

Code 5.21: Maybe Read

seems that exceptions created in the Haskell compiler can be catched in the calling function. When exceptions can be handled without the **IO** monad the function can be implemented as done in Code 5.22.

```
instance (Read l,Read r) => Read (Union l r) where
  read x = fromJust . upCast (
             try   (read x :: l)
             catch (read x :: r)
           )
```

Code 5.22: Function using Exception handling

The instantiation of type classes as shown in Code 5.21 can result in different behaviour for commuting unions that should be equivalent. The instance declaration given in Figure 5.4 for functions that result in a union may result in different behaviour when the union is defined with the union arguments switched, due to the overlapping domains. In Code 5.23 an implementation of the type class `MRead` is given that overlaps on the domain, when `mread` is used on the union of `T1` and `T2`.

It is clear that the evaluation of the function `mread` on union `` T1 `Union` T2 `` will result in **Just** `A`, **Just** `B` or **Just** `C`. Whereas the union `` T2 `Union` T1 `` will result in **Just** `D`, **Just** `E` or **Just** `F`. Therefore when introducing functions that result in a union, care must be taken with the domains of the functions.

There are several possibilities to handle these overlapping domains, suppose you have the type class `MRead` and a value `x` which will result in **Just** `<value>` for the implementation of this type class for type `A` and for the implementation of the type class for type `B`. There are different solutions to prevent incorrect behaviour of these functions. These solutions can be divided in compile time, the first two, and run time solutions, the other three.

1. Ignore the problem, but give a warning on compile time to the programmer that there can be overlapping function definitions.

```
data T1 = A | B | C deriving(Eq,Typeable)
instance Algebraic T1

data T2 = D | E | F deriving(Eq,Typeable)
instance Algebraic T2

instance MRead T1 where
  mread "A" = Just A
  mread "B" = Just B
  mread _   = Just C

instance MyRead T2 where
  mread "D" = Just D
  mread "E" = Just E
  mread _   = Just F
```

Code 5.23: Type Classes with overlapping domains

2. Try to examine whether there are overlapping instances. There are three cases; (a) it can be deduced from the patterns and guards, from now on called the (selection) criteria, that when there are no overlapping selection criteria, no error or warning is given, and there is no problem with the parallel composition of these functions. (b) it can be concluded from the selection criteria that the criteria overlap, then an error must be given, because the union is not commutative anymore. (c) it is not possible to conclude whether selection criteria overlap - because guards can be arbitrary complex - then a warning must be given that overlapping patterns and guards might occur.
3. Just take the result of the first function that gives a value on runtime. This implies that the function on the union may give different results for `A ‘Union‘ B` and `B ‘Union‘ A`.
4. Give a runtime error message when both functions on the union will result in a value.
5. Give a runtime error message when both functions on the union will result in a *different* value.

The first two solutions can only be implemented in a compiler and are therefore part of future research. The third is implemented in the rewriting rule of Figure 5.4. The fourth and fifth can be implemented rather easily, by altering the code that is generated when a type class is instantiated. The code for the fourth and fifth possibility to handle overlapping instances are respectively shown in Code 5.24 and 5.25.

```
instance (MRead l,MRead r) => MRead (Union l r) where
    mread  e
      | Just x <- a, Just y <- b
        = error ("Overlapping instances for " ++ show e)
      | Just x <- a
        = upCast x
      | Just x <- b
        = upCast x
      | otherwise
        = Nothing
        where
          a = mread e :: Maybe l
          b = mread e :: Maybe r
```

Code 5.24: Give a runtime error on overlapping instances

```
instance (MRead l,MRead r) => MRead (Union l r) where
    mread  e
      | Just x <- a, Just y <- b, not (x `eq` y)
        = error ("Overlapping instances for " ++ show e)
      | Just x <- a
        = upCast x
      | Just x <- b
        = upCast x
      | otherwise
        = Nothing
        where
          a = mread e :: Maybe l
          b = mread e :: Maybe r
```

Code 5.25: Give a runtime error on overlapping instances with different values

5.4.4 Parallel Function Composition using Operators

Besides functions on type classes also other functions might be composed in parallel. This parallel function composition can be done using the same operators as described in Chapter 4.

The operators $\widetilde{F}$ and $\widetilde{+}$ can be defined easily, as shown in Code 5.26. The operator $\widetilde{F}$ should have a synonym `union` to resemble the same notion as the **Either** type with the corresponding function `either`, which is evaluated with two functions and an **Either** value. The first will be used when the value is **Left** `x` the second when the value is **Right** `x`. The `union` function can be used in the same way now.

```
<F> :: (Algebraic a, Algebraic b)
      => (a->c) -> (b->c) -> (a `Union` b) -> c
<F> f g x = (either f g) (unionIdentify x)

union = <F>

<+> :: (Algebraic a, Algebraic b,Algebraic c,Algebraic d)
      => (a->c) -> (b->d) -> (a `Union` b) -> (c `Union` d)
<+> f g x = (<F>) (fromJust.upCast.f) (fromJust.upCast.g) x
```

Code 5.26: Parallel Function Composition Operators

The function `eq` as defined in Code 5.15 can be rewritten in a more simple way as done in Code 5.27

```
eq x y = (union eq eq x) y
```

Code 5.27: The equality operator rewritten

The implementation of the operator $\widetilde{G}$ depends on the choice, listed above, of how functions need to be composed in parallel. One implementation of $\widetilde{G}$, which corresponds to the third solution given, is shown in Code 5.28.

```
<G> :: (Algebraic b,Algebraic c)
    => (a->Maybe b) -> (a->Maybe c) -> a -> Maybe (b `Union` c)
<G> f g x | Just y <- f x = upCast y
          | Just y <- g x = upCast y
          | otherwise     = Nothing
```

Code 5.28: Parallel Function Composition Operator `<G>`

5.4.5 Algebraic Patterns

The use of pattern guards, although elegant, is quite verbose. It is possible to add syntactic sugar to the language, to be able to show it as a construct that is already known to the programmer. The normal representation of algebraic patterns can be rewritten in an automatic way to a pattern guard form. The rewriting rule to rewrite algebraic patterns is shown in Figure 5.11, which is used by the rule depicted in Figure 5.10. In Code 5.29 a function with patterns using algebraic constructors is shown. In Code 5.30 this function is rewritten to a function that uses pattern guards. Note that the rule processes recursively through the algebraic patterns, because algebraic patterns might be nested.

$\mathcal{R}_A[\![f x_1 \ldots x_n = e]\!]$
$f\ \mathcal{R}_{AP}[\![x_1]\!] \ldots \mathcal{R}_{AP}[\![x_n]\!] = e$

Figure 5.10: Algebraic Patterns

$\mathcal{R}_{AP}[\![C\ c_1 \ldots c_n]\!]$
$x \lhd \mathsf{guard}^{+} \leftarrow$ **Just** ($\mathcal{R}_{AP}[\![c_1]\!], \ldots, \mathcal{R}_{AP}[\![c_n]\!]$)<−`from`$C\ x$
• x is a unique variable.

Figure 5.11: Algebraic Patterns rewritten

$\mathcal{R}_{AP}[\![(x{:}xs)]\!]$
$(\mathcal{R}_{AP}[\![x]\!]{:}\mathcal{R}_{AP}[\![xs]\!])$

Figure 5.12: List Pattern rewritten

$\mathcal{R}_{AP}[\![x]\!]$
x
• x is an identifier, _ or a constant.

Figure 5.13: Pattern rewritten

```
doEvent :: InEvent −> String
doEvent (Moved from to      ) = from ++ " −> " ++ to
doEvent (Recieved ip p cont) = "Received " ++ cont
```

Code 5.29: Algebraic Function

```
doEvent :: InEvent −> String
doEvent x
  | Just (from,to)    <− fromMoved x
  = from ++ "−>" ++ to
  | Just (ip,p,cont) <− fromRecieved x
  = "Received " ++ cont
```

Code 5.30: Algebraic Function rewritten

Rewriting composite patterns, i.e. list, record or tuple patterns can be rewritten in a straightforward way. For instance the list pattern can be rewritten as done in Figure 5.12. The implementation of the rules to handle tuples and records are not shown here, but they are also very straightforward. When a pattern need not be rewritten, the rule shown in Figure 5.13 applies.

5.5 Unnamed Unions

With this implementation of a union it is possible to use unnamed unions, i.e. unions that are defined in place in functions. The type definition of the `doEvent` function which is defined in the module Section C.3 can also be given without defining the unions first. Both type definitions are shown in Code 5.31 and 5.32.

```
type InEvent  = MoveInEvent  `Union` CoreInEvent
type OutEvent = MoveOutEvent `Union` CoreOutEvent

doEvent :: State -> InEvent -> (State,[OutEvent])
```

Code 5.31: Type definition with named unions

```
doEvent :: State ->
             (MoveInEvent `Union` CoreInEvent) ->
               (State,[MoveOutEvent `Union` CoreOutEvent])
```

Code 5.32: Type definition with unnamed unions

5.6 Equality of Unions

According to our intuitive notion of unions, unions should be commutative as described in Section 3.7, i.e. `A `Union` B` should be equivalent to `B `Union` A`. In most cases this is true for the implementation given.

The function `doEvent` is defined on type `InEvent`. The definition of `InEvent` is shown in Section C.3 and in Code 5.31. The function `doEvent` will evaluate to the same result, indifferent of the order of the union declaration. Yet, in Haskell cannot be unified by the type checker, therefore it is not possible when a type definition is given to evaluate the function with arguments of the commuting union (Code 5.33). When the type definition is removed, however, the function will evaluate and the result will be the same, because the type checker does not have to unify the types. The same goes for arguments that are not a union value. The expression: `doEvent` *state* (`Received` """" "") will result in the correct result (*state* ,[]) when no type definition is given, but will result in a type error when it is evaluated with the type definition given in Code 5.31.

```
type InEvent2 = CoreInEvent `Union` MoveInEvent
```

Code 5.33: The union that should commute with `InEvent`

6

Example

6.1 Outline of the Example

I have made an example in which the use of (simulated) stream-I/O and algebraic unions are illustrated. The simple example is about a robot and an agent that operates in a blocks world. This blocks world consists of 5 blocks named "A" until "E". These blocks can be placed on top of each other and on the "Floor". A configuration of the blocks world can be seen in Figure 6.1.

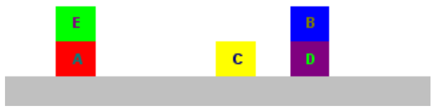


Figure 6.1: A configuration of the blocksworld

The responsibility of the robot is to move blocks from one place to another, when it receives requests from the outside world and gives as result whether the block is moved. The agent is a program that can give this requests to the robot. The flow of events between the programs is illustrated in Figure 6.2.

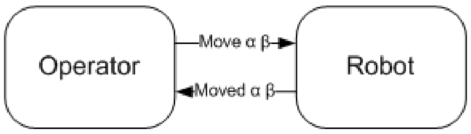


Figure 6.2: The flow of events

The agent program has to make the request to the robot that a block must be moved. It would be nice when the programmer is able to create a new algebraic constructor that is an abstract representation of a move request. One algebraic constructor that models the event to send a request to the robot, and one modeling the event occurring when the robot has performed the move.

6.1.1 Union

In the application, the agent must be able to handle the core events - events like `DrawCircle`, `ButtonClick` and `KeyIn` - and the added move events in the same way. This is done by creating a union between the core events and the move events.

6.1.2 Function Implementations

When the program uses this events two functions must be implemented for both core events and move events. The types that form type `inevent` must implement a function that may convert an incoming event to another event. Also a function must be implemented to transform an outgoing event to a core event.

6.1.3 Event Loop

The event loop is used to specify the behaviour that must occur when handling events that are coming in. The use of unions enables the mixing of core events and move events.

6.2 Example with Algebraic Types

The code used to implement the example using algebraic types with the rewriting rules of Chapter 5 is shown in Appendix B. The union that is needed to merge the different types of events is shown in this code and can also be seen in Code 6.1.

For this types a function must be defined that converts a string to a core event or a move event. This function composition is shown in Code 6.2 along with the implementations of the functions that are being functionally composed. The other way round the function composition to handle both core output events and the move output event is shown in 6.3. This function converts a union to a list of graphics.

```
inevent ::= coreIn <U> moveIn

outevent ::= coreOut <U> moveOut
```

Code 6.1: The definition of the Event Union types

```
string2inevent :: string -> inevent
string2inevent = string2moveIn <F> string2coreIn

string2moveIn :: string -> moveIn
string2moveIn s
   = Moved (getFrom s) (getTo s)
    , if getFrom s ~= None /\ getTo s ~= None

string2coreIn :: [char] -> coreIn
string2coreIn s
  = Ack (getvalue "Event" s) (getvalue "Port" s)
      , if (getvalue "Event" s) = "Subscribe"
  = ...
  = MessageIn (getvalue "Address" s)
              (getvalue "Port" s)
              (getvalue "Contents" s)
      , if (getvalue "Id" s) = "Receive"
```

Code 6.2: Function composition for input events

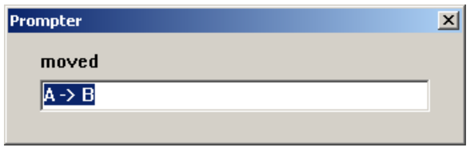
```
handleOutevent :: outevent -> [graphics]
handleOutevent = handleCoreOut <F> handleMoveOut

handleMoveOut :: moveOut -> [graphics]
handleMoveOut (Move f t)
   = handleCoreOut (Send "localhost" "4001" msg)
     where
      msg  = from ++ to
      from = "<From>" ++ bl2str f ++ "</From>"
      to   = "<To>"   ++ bl2str t ++ "</To>"

handleCoreOut :: coreOut -> [graphics]
handleCoreOut (PromptOut x y)
   = [GraphPrompt x y]
handleCoreOut (Send tar port cont)
   = sendCoreOutToJava (Send tar port cont  )
...
handleCoreOut NoOutput
   = []
```

Code 6.3: Function composition for output events

The event loop can be used to handle both the move events and core events, as can be seen in Code 6.4. This function creates an output event when a certain event occurs `Move` and receives an input event `Moved` from the robot when a block is moved. This event is shown to the user with a prompt. In this case the prompt in Figure 6.3 is shown to the user when block "A" is moved on top of block "B".



A -> B

Figure 6.3: A prompt shown when a block is moved

It is not possible to send messages over the network in Amanda, this is simulated by a Java-program and some text-files. The actual implementation of this network communication simulation is not relevant to the implementation that is given here. Therefore it will not be discussed further.

6.3 Example with Type Classes

In Appendix C an implementation of the example of this Chapter is given. Because the Haskell language does not have the stream I/O model the I/O model is simulated using some functions that have the same types as the functions used in the Amanda stream model. The functions **`eventsin`** and **`eventsout`** are functions that respectively give a list of input events and get a list of output events as an argument. The "system" can not handle stream I/O in the sense that it interacts with the environment. It rather is a static simulation of how events are handled.

The union of events can be implemented rather easily by creating a type synonym as shown in Code 6.5. As described in Section 5.5 these unions need not be declared, but can also be used using unnamed unions. Yet the declaration of these synonyms make the type definition of functions simpler.

These event types can be used to handle the events coming in and going out respectively. In order to create events that are most suited to the user a type class is implemented to convert a `CoreInEvent` to a specific inevent and a type class to convert a specific outevent to a `CoreOutEvent`. These classes are shown in Code 6.6

These type classes are implemented for the core and move events. For instance

```
doE :: storeTp -> inevent -> (storeTp,[outevent])
doE s (ButtonClick  w 20 i) = (s,bye)
doE s (MessageIn _ _ t)     = (s,[(PromptOut "received" t)])
doE s (Keyin '\e')          = (s,bye)
doE s (Error _ t)           = (s,[(PromptOut "Error" t)])
doE s (ButtonClick  w 13 i) = (s,[(Move from to)])
                              where
                               from = radio2blockFrom i
                               to   = radio2blockTo i
doE s (Moved f t)           = (s,[PromptOut "moved" msg ])
                              where
                               msg = ((bl2str f) ++
                                      " -> " ++
                                      (bl2str t))

doE s _                     = (s,[NoOutput])
```

Code 6.4: The event loop to handle all input events

```
type InEvent  = MoveInEvent  `Union` CoreInEvent
type OutEvent = MoveOutEvent `Union` CoreOutEvent
```

Code 6.5: Union definition

```
class (Algebraic a) => InEventClass a where
  fromCoreInEvent :: CoreInEvent -> Maybe a

class (Algebraic a) => OutEventClass a where
  toCoreOutEvent :: a -> CoreOutEvent
```

Code 6.6: The event type classes

```
instance OutEventClass MoveOutEvent where
  toCoreOutEvent (Move a b)
      = Send "localhost" "4000" ("MOVE " ++ a ++ " " ++ b)
```

Code 6.7: The implementation of the `OutEventClass` for `MoveOutEvent` the `MoveOutEvent` algebraic type implements the `OutEventClass` as shown in Code 6.7.

Note that the way the implementation of the `InEventClass` type class is not commutative with respect to unions by default. The idea of this type class is to convert an event to the most specific type possible. Only when no specific type can be found, the events of `CoreInEvent` must be returned. It is obvious that the implementation of `InEventClass` for the type `CoreInEvent` will result in a value always. Therefore a union between `CoreInEvent` and some other type, is not commutative. That is, the result of the function `fromCoreInEvent` on the type `CoreInEvent ‘Union‘ MoveInEvent` will result in a different event than the same function on the type `MoveInEvent ‘Union‘ CoreInEvent`. Still, it is possible to reimplement the type class for a union with a `CoreInEvent` to enforce commutativity. This overriding of implementations of type classes is not possible in the standard Haskell compiler, yet with the flag `−fallow −overlapping −instances`, this is possible.

The implementations of the type class `InEventClass`, the default and the more specific ones are shown in Code 6.8. Note that the commutativity between `MoveInEvent ‘Union‘ CoreInEvent` and `MoveInEvent ‘Union‘ CoreInEvent` is preserved now.

In the event loop `MoveEvents` and `CoreEvents` can be addressed in the same way. This is shown in Code 6.9. With the rewriting rules as been given in Chapter 5 the algebraic patterns are rewritten to a form with pattern guards that is shown in the Appendix C.3.

6.4 Conclusion

The conclusion of this small example is that unions can be used to add custom events to the language with very little programming effort. To add new events to the language in both approaches some actions must be taken.

- Introduce new event types, i.e. algebraic constructors.
- Make a union of the custom event types and the core event types
- Implement functions that convert the custom events to standard events - either by type classes or parallel composition of functions - in order to be handled by the language.

```
instance (InEventClass l, InEventClass r)
  => InEventClass (Union l r) where
    fromCoreInEvent e
      | Just x <- fromCoreInEvent e :: Maybe l = upCast x
      | Just x <- fromCoreInEvent e :: Maybe r = upCast x
      | otherwise                              = Nothing

instance (InEventClass a)
    => InEventClass (a `Union` CoreInEvent) where
  fromCoreInEvent e
    | Just x <- fromCoreInEvent e :: Maybe a           = upCast x
    | Just x <- fromCoreInEvent e :: Maybe CoreInEvent = upCast x
    | otherwise                                        = Nothing

instance (InEventClass a)
    => InEventClass (CoreInEvent `Union` a) where
  fromCoreInEvent e
    | Just x <- fromCoreInEvent e :: Maybe a           = upCast x
    | Just x <- fromCoreInEvent e :: Maybe CoreInEvent = upCast x
    | otherwise                                        = Nothing
```

Code 6.8: Implementation of type classes to preserve commutativity

```
doEvent :: (InEventClass a,OutEventClass b)
              => State -> a -> (State,[b])
doEvent s (Moved from to)
  = (s ++ "\n" ++ from ++ "->" ++ to,[])
doEvent s (Received ip p cont)
  = (s ++ "\n" ++ cont                ,[])
```

Code 6.9: Using the events in the event loop

7

Related and Future Work

7.1 Related Work

Union types as introduced in this thesis are studied in type theory to some extent (Pierce, 1991; Barbanera, Dezani-Ciancaglini, & Liguoro, 1995). Yet this papers are mostly focused on the type theoretical background and almost no effort is taken to show an implementation of union types in languages.

In (Löh & Hinze, 2006) a proposal is made that is related to the use of union types I proposed. Löh & Hinze present *open* data types and *open* functions. This means functions and data types can be declared open when they might be extended in the future. One large benefit of open data types and functions over unions are that no type extension has to be made to the (Haskell) language. Furthermore, the proposal that is made about best fit pattern matching can also be used to implement parallel function compositions with overlapping function domains. This best fit pattern matching, opposite to the normal Haskell pattern matching, does not try to find the first pattern that matches the value found, but orders the function clauses with the most specific pattern first and the most generic pattern last. For instance the pattern `C x y`, where `C` is a constructor will be placed above the pattern `_` because it is more specific, but below `C x` 5. Using this best fit pattern matching the clauses of the parallel composed functions with overlapping domains can prove to be useful.

Now I will show that the proposal that is made in the paper of Löh & Hinze can also be modeled using unions and type classes. To do this, I will use the same example as Löh & Hinze use in their paper. This example is about a very simple language of expressions based on algebraic data types.

First we define a type for the expression language, which at first only consists of expressions denoting numbers. For the functions that are implemented on the expressions a type class is introduced, Code 7.1 shows this. Also a function that shows how the expression will be evaluated is shown.

```
type Expr = Number

data Number = N Int
              deriving(Eq, Typeable)

class Expression a where
  eval     :: a -> Int
  toString :: a -> String

instance Expression Number where
  eval (N x) = x
  toString (N x) = show x

expression :: Expr -> String
expression x
  = (toString x) ++ " will evaluate to " ++ ((show.eval) x)
```

Code 7.1: Expression consisting of numbers

Now the expression language is extended with a new expression, the + operator. The code that is altered and extended is shown in Code 7.2. Note that the function `expression` need not be changed.

```
type Expr = Number `Union` Plus

data Plus = Plus Expr Expr

instance Expression Plus where
  eval (Plus x y) = (eval x) + (eval y)
  toString (Plus x y)
   = "(" ++ (toString x) ++ " + " ++ (toString y) ++ ")"
```

Code 7.2: Extend Expression language with +

The use of type classes using `Union`s forces the added constructor `Plus` to implement the type class `Expression` whereas Löh & Hinze open data type and functions will give a runtime error because a pattern is missing that handles the `Plus` constructor.

Furthermore constructors that are added to the open data type can not be used separately, whereas every type that is used to create a union can also be used separately.

A large disadvantage of open data types and open functions is that the type signature of the function is mandatory (Löh & Hinze, 2006, p.2). This implies that type inference is not possible for open functions.

7.2 Future Work

I introduced a new construction to be added to functional programming languages. During the implementation an increasing number of areas occurred in which union types might be useful, I highlighted a few of them. Yet in the future research needs to be done to explore areas in which union types can be used.

Just like in logic the dual of disjunction is conjunction, the dual of a union type is the intersection type. In type theory research has been done to study this notion of union and intersection types (Pierce, 1991; Barbanera et al., 1995). It would be nice to see whether this intersection types can be useful in functional languages and whether several distributive laws emerging from their duality, like distributivity can be proved for implementations in functional languages.

Also some research need to be done to add this construction to the core language. The implementation I used showed that the construction can be implemented in a functional language. However the functionality should be part of the core language, which eliminates labour-intensive pre processing and creates more readable error messages and type information.

The merging of functions of a union might give some problems with overlapping patterns and guards as described in Section 5.6. Further research must be done to find an algorithm that approximates whether clauses overlap. This approximation can then be used to implement the merging of functions with a union type as result value. Also research need to be done to discover what the best solution (5.4.3) is to handle overlapping patterns.

Furthermore in practice should be evaluated which parts of extensible stream I/O are sufficient and what parts can be automated. For instance the union of event types can be automated when the libraries are imported. So maybe this is something the programmer should not worry about. Also the use in practice can produce more real-world examples.

Bibliography

Achten, P., Groningen, J. van, & Plasmeijer, M. (1993). High Level Specification of I/O in Functional Languages. In J. L. et al (Ed.), *Proceedings Glasgow Workshop on Functional Programming.* New York, NY: Springer Verlag.

Barbanera, F., Dezani-Ciancaglini, M., & Liguoro, U. de'. (1995). Intersection and Union Types: Syntax and Semantics. *Information and Computation*, *119*, 202-230.

Barendsen, E., & Smetsers, S. (1993). Conventional and Uniqueness Typing in Graph Rewrite Systems. In *Proceedings of the 13th Conference on the Foundations of Software Technology and Theoretical Computer Science 1993, Bombay.* New York, NY: Springer-Verlag.

Cardelli, L. (1988). Structural Subtyping and the Notion of Power Type. In *POPL* (pp. 70–79). San Diego, California: ACM Press.

Cardelli, L. (1997). Type Systems. In A. Tucker (Ed.), *The Handbook of Computer Science and Engineering.* (chap. 103). Boca Raton, FL: CRC Press.

Erwig, M., & Peyton Jones, S. (2000, September). Pattern Guards and Transformational Patterns. In *ACM SIGPLAN Haskell Workshop.* (Vol. 41). Montréal, Canada: Elsevier.

Finne, S., Henderson, F., Kowalczyk, M., Leijen, D., Marlow, S., Meijer, E., et al. (2003). *The Haskell 98 Foreign Function Interface 1.0 An Addendum to the Haskell 98 report.* Retrieved April 25, 2006, from `http://www.cse.unsw.edu.au/~chak/haskell/ffi/`.

Gordon, A. (1994). *Functional Programming and Input/Output.* New York, NY, USA: Cambridge University Press.

Gordon, A., & Hammond, K. (1995, June). Monadic I/O in Haskell 1.3. *Proceedings of the Haskell Workshop*, 50–68.

Hanus, M., Kuchen, H., & Moreno-Navarro, J. (1995). Curry: A Truly Functional-logic Language. In *ILPS'95 Post Conference Workshop on Declarative Languages for the Future.* Portland (USA): Melbourne University.

Hoon, W. de, Rutten, L., & Eekelen, M. van. (1995). Implementing a Functional Spreadsheet in Clean. *Journal of Functional Programming*, *5*(3), 383-414.

Hove, J. ter. (2005). *The G-Machine.* Internal report. University of Twente.

Hudak, P., & Fasel, J. (1992). A gentle introduction to Haskell. *SIGPLAN Not.*, *27*(5), 1–52.

Hudak, P., & Sundaresh, R. (1989). *On the Expressiveness of Purely Functional I/O Systems.* (Tech. Rep.). Yale University, Box 2158 Yale Station New Haven, CT 06520: Departement of Computer Science.

Hughes, J. (1989). Why Functional Programming Matters. *Computer Journal*, *32*(2), 98–107.

Jones, S. (1995, July). Experiences with Clean I/O. In D. Turner (Ed.), *Proceedings of the 1995 Glasgow Workshop on Functional Programming.* Ullapool, Scotland: Springer.

Kuper, J. (2006). *Tinadic Parsing.* Personal Communication.

Lämmel, R., & Peyton Jones, S. (2003). Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation* (pp. 26–37). New York, NY, USA: ACM Press.

Landin, P. (1965). A correspondence between ALGOL 60 and Church's Lambda-notations: Part II. *Commun. ACM*, *8*(3), 158–167.

Landin, P. (1966). The next 700 programming languages. *Commun. ACM*, *9*(3), 157–166.

Leijen, D. (2005, September). Extensible records with scoped labels. In *Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05)* (pp. 297–312). P.O. Box 80.089, 3058TB Utrecht, the Netherlands.

Leroy, X. (2004). *The Objective Caml system release 3.09.*

Löh, A., & Hinze, R. (2006). *Open Data Types and Open Functions.* (Tech. Rep. No. IAI-TR-2006-2). Universität Bonn: Institut für Informatik III.

Moggi, E. (1989). Computational Lambda-Calculus and Monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989.* (pp. 14–23). Washington, DC: IEEE Computer Society Press.

Müller, M., Müller, T., & Roy, P. V. (1995, December). Multi-paradigm programming in Oz. *Visions for the Future of Logic Programming: Laying the Foundations for a Modern successor of Prolog.* (A Workshop in Association with ILPS'95)

Newbern, J. (2004). *All about monads.* Retrieved January 10, 2006, from `http://www.nomaware.com/monads/html/index.html`.

Noble, R., & Runciman, C. (1994). *Functional Languages and Graphical User Interfaces.* (Tech. Rep. No. YCS-94-223). Heslington, York, Y01 55D, England: Departement of Computer Science, University of York.

Oosterhof, N. (2005). *Application Patterns in Functional Languages.* Master's thesis, University of Twente, The Netherlands.

Papegaaij, E. (2005). *The Tina Language, A report on the specification and implementation.* Internal report. University of Twente.

Perry, N. (1991). *The Implementation of Practical Functional Programming Languages.* Unpublished doctoral dissertation, Imperial College, London.

Peyton Jones, S. (1987). *The Implementation of Functional Programming Languages.* Maylands Avenue, Hemel Hempstead: Prentice-Hall International.

Peyton Jones, S. (2005, July). *Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell.*

Peyton Jones, S., Gordon, A., & Finne, S. (1996). Concurrent Haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* (pp. 295–308). New York, NY, USA: ACM Press.

Peyton Jones, S., & Wadler, P. (1993). Imperative Functional Programming. In *Principles Of Programming Languages.* (pp. 71–84). Department of Computing Science, University of Glasgow: ACM Press.

Pierce, B. (1991, Februari). *Programming with Intersection Types, Union Types, and Polymorphism.* (Technical Report No. CMU-CS-91-106). Pittsburgh (USA): Carnegie Mellon University.

Plasmeijer, M., & Eekelen, M. van. (2001, December). *Concurrent CLEAN Language Report Version 2.0 DRAFT.* Retrieved April 10, 2006, from `http://www.cs.ru.nl/~clean/`.

Russell, D. (1997). *The Design Implications of Monads.* Kingston Upon Thames, Surrey KT2 7LB, UK.

Smetsers, S., Barendsen, E., Eekelen, M. van, & Plasmeijer, M. (1993). Guaranteeing Safe Destructive Updates Through a Type System with Uniqueness Information for Graphs. In H.J.Schneider & H. Ehrig (Eds.), *Dagstuhl seminar on graph transformations in computer science.* (Vol. 776, pp. 358–379). Radboud University Nijmegen: Springer.

Thompson, S. (1992, November). *Formulating Haskell.* (Tech. Rep. Nos. 29–92*). University of Kent, Canterbury, UK: University of Kent, Computing Laboratory.

Tinnemeier, N. (2006). *Provisional: Agents in Functional Programming languages.* Unpublished master's thesis, University of Twente, The Netherlands. To appear.

Wadler, P. (1990). Comprehending monads. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming.* (pp. 61–78). New York, NY, USA: ACM Press.

Wadler, P. (1997). How to declare an imperative. *ACM Computing Surveys, 29*(3), 240–263.

A

Algebraic Type Class and Union Type

```
{-# OPTIONS -fglasgow-exts #-}
{- Use to be able to use pattern guards -}

{- Overall class that defines the type class that must be implemented
by *every* algebraic type.-}
module Algebraic
  (module Data.Generics
  ,Union
  ,Algebraic (..)
  ,unionIdentify
  ,fromJust
  ,union
  ,(<&>)
  ,(<*>)
  ,(<+>)
  )
where

{- To be able to use the class typeable and to use the cast function
-}
import Data.Generics
import Maybe

{- Union type, to store either the left or the right value -}
data Union a b = L a
               | R b
                 deriving(Eq,Typeable)

{- Return an Either type that corresponds to the union type, this
makes it possible to hide the internal constructors. Note that this
function is not commutative -}
unionIdentify :: a `Union` b -> Either a b
unionIdentify (L x) = Left  x
unionIdentify (R x) = Right x
```

```

{- The type class for each algebraic type, note that instances of this
class must also implement Eq and Typeable -}
class (Eq a,Typeable a) => Algebraic a where

    {- Cast the value of this type to a more general type. The default
    implementation is to cast the value to its own type -}
    upCast :: (Algebraic b) => b -> Maybe a
    upCast = cast

    {- Cast the value of this type to a more specific type. The
    default implementation is to cast the value to its own type -}
    downCast :: (Algebraic b) => a -> Maybe b
    downCast = cast

    {- A custom equality function, this function compares two values
    that are the most specific. The default implementation assumes
    that the value of the implementation of this class (the left
    value) is the most specific value and compares this value with a
    value of the other function when this is casted to the same type
    -}
    eq :: (Algebraic b) => a -> b -> Bool
    x `eq` y = Just x == (downCast y :: Maybe a)

{- The union is also part of the Algebraic class, implemented in the
following way. Therefore *every* union is also part of the Algebraic
typeclass -}
instance (Algebraic l, Algebraic r) => Algebraic (l `Union` r) where
  upCast x | Just z <- cast   x :: Maybe (l `Union` r)  = Just z
           | Just z <- upCast x :: Maybe l              = Just (L z)
           | Just z <- upCast x :: Maybe r              = Just (R z)
           | otherwise                                  = Nothing

  downCast = union downCast downCast

  eq x y   = (union eq eq x) y

{- The Union is part of the Show type class, when both parts are -}
instance (Show l, Algebraic l, Show r, Algebraic r)
  => Show (Union l r) where
  show = union show show

{- Equivalent to <&>, which is the Haskell representation of <F> -}
union :: (Algebraic a, Algebraic b)
  => (a->c) -> (b->c) -> ((a `Union` b) -> c)
union f g x = (either f g) (unionIdentify x)

{- Equivalent to union, which is the Haskell representation of <F> -}
(<&>) :: (Algebraic a, Algebraic b)
  => (a->c) -> (b->c) -> ((a `Union` b) -> c)
```

```
(<&>) = union

{- The function <+> -}
(<+>) :: (Algebraic a, Algebraic b,Algebraic c,Algebraic d)
  => (a->c) -> (b->d) -> (a `Union` b) -> (c `Union` d)
f <+> g = union (fromJust.upCast.f) (fromJust.upCast.g)

{- One Haskell representation of <G> -}
(<*>) :: (Algebraic b,Algebraic c)
  => (a->Maybe b) -> (a->Maybe c) -> a -> Maybe (b `Union` c)
(<*>) f g x | Just y <- f x = upCast y
            | Just y <- g x = upCast y
            | otherwise     = Nothing
```

Code A.1: Algebraic Type Class and Union Type

B

Example code using Algebraic Types

```
#import "TinaSim/TinaSim.ama"
#import "TinaSim/TinaEventTypes.ama"
#import "windows.ama"
#import "xml.ama"
#import "moveL.ama"

|| Union of in-event types
inevent ::= coreIn <U> moveIn

|| Union of out-event types
outevent ::= coreOut <U> moveOut

|| Move events
|| Receive which block is moved to where
moveIn ::= Moved block block

|| Request a move of a block
moveOut ::= Move block block

|| convert a xml message in the block to be moved
|| arg1: the xml-message
|| result: the block to be moved
getFrom :: string -> block
getFrom = str2bl.(getvalue "From").getCont

|| convert a xml-message in the block to where must
|| be moved
|| arg1: the xml-message
|| result: the block to where must be moved
getTo :: string -> block
getTo = str2bl.(getvalue "To").getCont

|| convert a string to a move event
|| arg1: the xml-message
|| result: the event occurred
string2moveIn :: string -> moveIn
string2moveIn s
   = Moved (getFrom s) (getTo s)
       , if getFrom s ~= None /\ getTo s ~= None

|| Handle the move event
|| arg1: the event to be handled
|| result: the graphics that are the result of
||         handled events
handleMoveOut :: moveOut -> [graphics]
handleMoveOut (Move f t)
   = handleCoreOut (Send "localhost" "4001" msg)
     where
      msg  = from ++ to
      from = "<From>" ++ bl2str f ++ "</From>"
      to   = "<To>"   ++ bl2str t ++ "</To>"

|| Parallel composition of functions, first try to
|| convert the string to a move event, and then to
|| a normal event
|| arg1: the xml-message representing the event
|| result: the inevent
string2inevent :: string -> inevent
string2inevent = string2moveIn <F> string2coreIn

|| Parallel composition of functions, first handle
|| the core events and then the move events
handleOutevent :: outevent -> [graphics]
handleOutevent = handleCoreOut <F> handleMoveOut

|| Store, keeping track of the number of moves
storeTp ::= {moves :: num}


|| some initial events, to start the program
initEv :: [outevent]
initEv
   = [Subscribe "localhost" "4000"
      ,Graphics "operator" 0 (windowCreate operator)
      ,ShowWindow "operator" True
      ,Graphics "main" 0 [GraphVisible False]
      ]

|| The whole program, moving the events to the
|| doEs-function
main :: bool
main = tinaEventsout
          (initEv ++
            (doEs {moves=0} tinaEventsin)
          )

|| Distibute to single event
doEs :: storeTp -> [inevent] -> [outevent]
doEs s (Noinput:es) = doEs s es
doEs s (e      :es) = out ++ (doEs newS es)
                      where
                       (newS,out) = doE s e


|| single event call
doE :: storeTp -> inevent -> (storeTp,[outevent])
doE s (ButtonClick  w 20 i)
      = (s,bye)
doE s (MessageIn _ _ t)
      = (s,[(PromptOut "received" t)])
doE s (Keyin '\e')
      = (s,bye)
doE s (Error _ t)
      = (s,[(PromptOut "Error" t)])
doE s (ButtonClick  w 13 i)
      = (s,[(Move from to)])
        where
         from = radio2blockFrom i
         to   = radio2blockTo i
doE s (Moved f t)
      = (s,[PromptOut "moved" msg ])
        where
         msg = ((bl2str f) ++
               " -> " ++
               (bl2str t))
doE s _
      = (s,[NoOutput])

|| convert the value of a radio-button to a block
|| that must be moved
radio2blockFrom :: [(ident,[char])] -> block
radio2blockFrom = radio2block 30

|| convert the value of a radio-button to a block
|| where must be moved to
radio2blockTo :: [(ident,[char])] -> block
radio2blockTo   = radio2block 40

|| convert the value of a radio-button to a block
radio2block :: num -> [(ident,[char])] -> block
radio2block n xs
   = (str2bl.toL.decode.(64+).((-n)+).hd) ys
              , if ys ~= []
   = Floor    , otherwise
```

```
     where
      ys = [x|(x,y)<-xs; y="Y"; x>n; x<n+10]
      toL x = [x]

|| events to be send when the program is quit
bye :: [outevent]
bye = [Unsubscribe "localhost" "4000"
      ,Graphics "operator" 0 [GraphQuit]
      ]


|| the GUI of the program
operator :: windowTp
operator = Window "operator"
             ([ Label        99  "Move"] ++
                group1                ++
              [ Label        99  "to"]  ++
              group2                  ++
              [ Button       13  "OK"
              , Button       20  "Stop"
              ]
             )
|| the group of radiobuttons of the from-blocks
group1 :: [windowItem]
group1 = [RadioButton 31 "A"
         ,RadioButton 32 "B"
         ,RadioButton 33 "C"
         ,RadioButton 34 "D"
         ,RadioButton 35 "E" ]

|| the group of radiobuttons of the to-blocks
group2 :: [windowItem]
group2 = [RadioButton 41 "A"
         ,RadioButton 42 "B"
         ,RadioButton 43 "C"
         ,RadioButton 44 "D"
         ,RadioButton 45 "E"
         ,RadioButton 46 "Floor" ]
```

Code B.1: Source code of the Example using Algebraic Types

C

Example Code using Type Classes

C.1 CoreEvents

```
{-# OPTIONS -fglasgow-exts #-}
{-# OPTIONS -fallow-overlapping-instances #-}
module CoreEvents
  (InEventClass(..)
  ,OutEventClass(..)
  ,CoreInEvent(..)
  ,CoreOutEvent(..)
  ,fromReceived
  ,fromFileReq
  ,fromSend
  ,fromFileIn
  )
where
{- Module to specify all core Events -}

import Algebraic
import HelpEvents
{- INEVENT ---------------------------------------------------}

{- CoreInEvent declaration -}
data CoreInEvent = Received String String String
                 | FileIn   String String
                   deriving(Show,Eq,Typeable)

{- InEventClass Class declaration -}
class (Algebraic a) => InEventClass a where
  fromCoreInEvent :: CoreInEvent -> Maybe a

{- CoreInEvent is part of the Algebraic Type Class -}
instance Algebraic CoreInEvent
```

```
{- CoreInEvent is part of the InEventClass Type Class -}
instance InEventClass CoreInEvent where
  fromCoreInEvent = Just

{- Each Union for which elements are part of the InEventClass Type
Class is also part of the InEventClass Type Class -}
instance (InEventClass l, InEventClass r)
  => InEventClass (Union l r) where
    fromCoreInEvent e
      | Just x <- fromCoreInEvent e :: Maybe l = upCast x
      | Just x <- fromCoreInEvent e :: Maybe r = upCast x
      | otherwise                              = Nothing

{- VERY important implementations of the InEventClass, when a union of
a type with CoreInEvent is unioned then the behaviour must be the same
for "a `Union` CoreInEvent" and "CoreInEvent `Union` a" -}
instance (InEventClass a)
    => InEventClass (a `Union` CoreInEvent) where
  fromCoreInEvent e
    | Just x <- fromCoreInEvent e :: Maybe a           = upCast x
    | Just x <- fromCoreInEvent e :: Maybe CoreInEvent = upCast x
    | otherwise                                        = Nothing

instance (InEventClass a)
    => InEventClass (CoreInEvent `Union` a) where
  fromCoreInEvent e
    | Just x <- fromCoreInEvent e :: Maybe a           = upCast x
    | Just x <- fromCoreInEvent e :: Maybe CoreInEvent = upCast x
    | otherwise                                        = Nothing

{- Implementation to get the arguments from the constructors -}
{- Received String String String -}
fromReceived :: (Algebraic a) => a -> Maybe (String,String,String)
fromReceived z
  | Just (Received w x y) <- (downCast z :: Maybe CoreInEvent)
    = Just (w,x,y)
  | otherwise = Nothing

{- FileIn String String -}
fromFileIn :: (Algebraic a) => a -> Maybe (String,String)
fromFileIn z
  | Just (FileIn x y) <- (downCast z :: Maybe CoreInEvent)
    = Just (x,y)
  | otherwise
    = Nothing

{- FUNCTIONS CoreInEvent -}

{- Handle the core in events, create events from the strings -}
handleCoreInEvents :: [String] -> [CoreInEvent]
handleCoreInEvents es = map handleCoreInEvent es

{- Handle the core in events, create a CoreInEvent from a string -}
handleCoreInEvent :: String -> CoreInEvent
handleCoreInEvent s
  | length es == 4 && es!!0 == "RECIEVED"
      = Received (es!!1) (es!!2) (es!!3)
  | length es == 3 && es!!0 == "FILEIN"
      = FileIn (es!!1) (es!!2)
        where
          es = splitString s

{- OUTEVENT--------------------------------------------------------------}

{- CoreOutEvent declaration -}
data CoreOutEvent = Send    String String String
                  | FileReq String
                    deriving(Show,Eq,Typeable)

{- OutEvent Class declaration -}
class (Algebraic a,Show a) => OutEventClass a where
  toCoreOutEvent :: a -> CoreOutEvent

{- CoreOutEvent is part of the Algebraic Type Class -}
instance Algebraic CoreOutEvent

{- CoreOutEvent is part of the OutEventClass Type Class -}
instance OutEventClass CoreOutEvent where
  toCoreOutEvent = id

{- Each Union for which elements are part of the OutEventClass Type
Class is also part of the OutEventClass Type Class -}
instance (OutEventClass a, OutEventClass b)
  => OutEventClass (a `Union` b) where
    toCoreOutEvent
      = union toCoreOutEvent toCoreOutEvent

{- Implementation to get the arguments from the constructors -}
{- Send String String String -}
fromSend :: (Algebraic a) => a -> Maybe (String,String,String)
fromSend z
  | Just (Send w x y) <- (downCast z :: Maybe CoreOutEvent)
    = Just (w,x,y)
  | otherwise
    = Nothing

{- FileReq String -}
fromFileReq :: (Algebraic a) => a -> Maybe String
fromFileReq z
  | Just (FileReq x) <- (downCast z :: Maybe CoreOutEvent)
```

```
    = Just x
  | otherwise
    = Nothing

{- FUNCTIONS CoreOutEvent -}

{- Handle the core out events, Do something in the outside world -}
handleCoreOutEvents :: [CoreOutEvent]-> [String]
handleCoreOutEvents es = map handleCoreOutEvent es

{- Handle the core out events, create a string based on the
constructor -}
handleCoreOutEvent (Send ip port cont)
    = "SEND " ++ ip ++ " " ++ port ++ " " ++ cont
handleCoreOutEvent (FileReq path)
    = "FILEREQUEST " ++ path
```

Code C.1: CoreEvents

C.2 MoveEvents

```
{-# OPTIONS -fglasgow-exts #-}
module MoveEvents
  (MoveInEvent(..)
  ,MoveOutEvent(..)
  ,fromMoved
  ,fromMove
  )
  where
{- Module to specify all Move Events -}

import Algebraic
import HelpEvents
import CoreEvents

{- MoveInEvent declaration -}
data MoveInEvent = Moved String String
                   deriving(Show,Eq,Typeable)

{- MoveOutEvent declaration -}
data MoveOutEvent = Move String String
                    deriving(Show,Eq,Typeable)

{- Implementation to get the arguments from the constructors -}
{- Moved String String -}
fromMoved :: (Algebraic a) => a -> Maybe (String,String)
fromMoved z
  | Just (Moved x y) <- (downCast z :: Maybe MoveInEvent)
    = Just (x,y)
  | otherwise
    = Nothing

{- Move String String -}
fromMove :: (Algebraic a) => a -> Maybe (String,String)
fromMove z
  | Just (Move x y) <- (downCast z :: Maybe MoveOutEvent)
    = Just (x,y)
  | otherwise
    = Nothing

{- Default implementation of Algebraic for MoveInEvent-}
instance Algebraic MoveInEvent

{- Default implementation of Algebraic for MoveOutEvent-}
instance Algebraic MoveOutEvent

{- Implementation of InEventClass for MoveInEvent -}
instance InEventClass MoveInEvent where
  fromCoreInEvent (Received ip p cont)
    | (length es) == 3 && es!!0 == "MOVED"
      = Just (Moved (es!!1) (es!!2))
    | otherwise
      = Nothing
        where
          es = splitString cont

{- Implementation of OutEventClass for MoveOutEvent -}
instance OutEventClass MoveOutEvent where
  toCoreOutEvent (Move a b)
    = Send "localhost" "4000" ("MOVE " ++ a ++ " " ++ b)
```

Code C.2: MoveEvents

C.3 Program

```
{-# OPTIONS -fglasgow-exts #-}
module Program where

import CoreEvents
import MoveEvents
import Algebraic
import MySystem
import Debug.Trace

-- Store the state in a string
type State = String

-- Union Events
```

```
type InEvent  = MoveInEvent  `Union` CoreInEvent
type OutEvent = MoveOutEvent `Union` CoreOutEvent

{- The initial state is an empty string -}
initState = ""

-- Model the stream I/O
main :: String
main
  = eventsout
     (doEvents initState (eventsin :: [InEvent]) :: [OutEvent])

-- Handle all events
doEvents :: (InEventClass a,OutEventClass b)
                 => State -> [a] -> [b]
doEvents s []     = trace s []
doEvents s (e:es) = os ++ (doEvents ns es)
                    where
                     (ns,os) = doEvent s e

-- Handle a single event
doEvent :: (InEventClass a,OutEventClass b)
               => State -> a -> (State,[b])
doEvent s x
  | Just (from,to)   <- fromMoved x
   = (s ++ "\n" ++ from ++ "->" ++ to,[])
  | Just (ip,p,cont) <- fromReceived x
   = (s ++ "\n" ++ cont                ,[])
```

Code C.3: Program

C.4 System modelling Stream I/O

```
module MySystem where

import CoreEvents
import Algebraic

{- Function modeling incoming events -}
eventsin :: (InEventClass a) => [a]
eventsin = map (fromJust.fromCoreInEvent)
            [Received "127.0.0.1" "4000" "MOVED A B"
            ,Received "localhost" "4000" "Dit is een bericht"
            ]

{- Function modeling outgoing events -}
eventsout :: (OutEventClass a) => [a] -> String
eventsout es = show es
```

Code C.4: System modelling Stream I/O

D

Preprocessor

D.1 Outline of the Preprocessor

To be able to use algebraic unions, I created a simple preprocessor for Amanda which is able to parse an extended version of Amanda, with algebraic unions, to the normal Amanda syntax, in which algebraic unions are converted to normal algebraic types. This is done in an ad hoc manner, to show that it is possible to use unions in a functional language.

in Figure D.1 the general outline of this parser is shown. Dedentation tokens are added to the textual representation of the input-file as described in Section D.3. Then the string is lexed by the `Lexer`. After the list of tokens is transformed in a parse tree, with the use of the EBNF-grammar, the parse tree is processed to a new parse tree. In this transformation the added constructs are transformed to constructs that are part of the core language. In the end the parse tree is converted back to a textual representation that can be handled by the Amanda-compiler.

D.2 Main

The overall program that links all parts of the program together. There are two functions that can be used to parse an input file.

- *eval*
  Show the parsed function in a graphical window represented as a tree before it is transformed by the process-part.
- *convert*
  Converts a file given as an argument from the directory `in` to the directory `out`. For instance `convert` ''`board`'' converts `in/board.ama` to

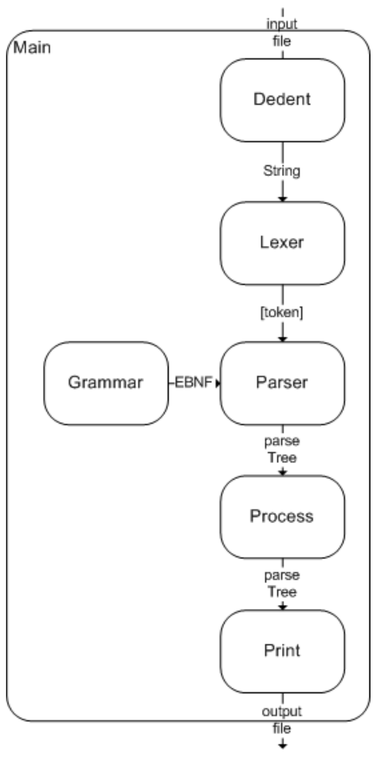


Figure D.1: The outline of the parser

`out/board.ama`.

D.3 Dedent

Adds dedentation tokens to the expression that must be parsed. Dedentation tokens are needed because Amanda, in fact almost all functional programming languages use the off-side rule to define scoping rules. The off-side rule is defined by Landin:

> *Any non-whitespace token to the left of the first such token on the previous line taken to be the start of a new declaration.*

(Landin, 1966)

The token mentioned in the case of Amanda is the ”=”. Because during the parsing whitespace does not matter and it will be removed a token must be added to tell the parser the current definition is ended. This is done in the dedent-function.

Functional programming languages depend on this off-side rule, but the EBNF notation is not suited to specify this off-side rule. Therefore, before the program to be preprocessed is lexed a very simple process adds dedentation tokens to the program. Note that this is done in a very simple, ad hoc way.

- *ded*
  Adds dedent tokens to the string given, based on the offset rule of Amanda.

During the dedentation adding process a list of indices of current indentations is used. At the moment that a new line is processed, the following cases can be distinguished:

1. The first token on the line is right of the last indentation and the line does not contain '='. Do not add an dedentation token and continue with the next line and the same list of indices.
2. The first token on the line is at the right of the last indentation token and the line contains '='. Do not add a dedentation token and add the indentation specified by the '=' to the *front* of the indentation list. Continue with the next line with this new list.
3. The first token on the line is left or at the same place of the last ”=”-token, add a dedentation token and continue with the line and the rest of the indices.

```
f x = y
      where
        y = z
        z = 8

g a = 9
```

Code D.1: Function with dedentation tokens

The dedentation tokens will be added in the following way. Between parentheses the applied case is denoted, corresponding to the cases given above. The indentation list at the start of the application of the case is denoted between the line number and the case.

Line 1 [] (Case 2) Add an indentation index (4) to the list of indices.

Line 2 [4] (Case 1) Proceed

Line 3 [4] (Case 2) Add an indentation index (9) to the list of indices.

Line 4 [9,4] (Case 3) Add a dedentation token and continue with the same line and the rest of the indices

- [4] (Case 2) Add an indentation index (9) to the list of indices

Line 5 [9,4] Proceed

Line 6 [9,4] (Case 3) Add a dedentation token and continue with the same line and the rest of the indices

- [4] (Case 3) Add a dedentation token and continue with the same line and the rest of the indices
- [] (Case 2) Add the indentation index to the list of indices.

Note that this implementation of the dedent tokens is not completely correct, for instance the expression shown in Code D.2 will add a dedentation token before ,6=7], because the "=" that is found on the line above. It is not possible during the lexing phase to determine that this token, between the 5s is a boolean equality operator in stead of a function definition operator.

```
list :: [bool]
list = [4=4
       ,5=5
       ,6=7]
```

Code D.2: Example of incorrect adding of dedentation tokens

D.4 Simple Lexer

The lexer splits the program, which is a large string in a list of strings (tokens). This is done in the following (ad hoc) way:

1. Split the string at all given tokens. Of course the ' '-characters but also '.' and '*'-characters. E.g. "*x*.*y*" will be split into the list ["*x*",".","*y*"]. Although it is tempting to only tokenize a string when a space (' ') occurs, this is not correct. The result of this will be that the expression `x*5` will be seen as the identifier `x*5` in stead of the expression `x` $*$ 5. The EBNF-grammar should provide that the tokens "34", "." and "6" are the real number "34.6".

2. The downside of this tokenizing is that tokens that belong together are also split. like .. , ::= and ++ for instance. These tokens need to be merged to a single token again. This is done by examining the whole token list again to find a sequence of tokens that should be a single token.
3. Remove all comments. For each tuple with a begin-comment token and an end-comment token all tokens are removed which occur between these tokens. In Amanda these tuples are: ("||","\n") and "/*","*/".
4. Remove all tokens that are not necessary anymore, like spaces and newlines. These are already replaced by dedentation-tokens in the dedent process.

D.5 Parser

The actual parser of the tokens is specified in `parser.ama`. The parser uses the language definition given in `grammar.ama` to parse the tokens into a parse tree. The parse tree is an algebraic type, with nodes and leafs. The leafs are the terminals of the EBNF-notation, whereas the nodes are the non terminals of the language. Each node has a list of children, according to the language definition. Each node also has an argument which is the type of the algebraic type specifying the language. This way the type of the node can be distinguished. The simplified parse tree of the expression "`x + y`" can be seen in Code D.3

```
tree = ParseNode Exp
         [ ParseNode Exp   [ ParseLeaf "x" ]
         , ParseNode Infix [ ParseLeaf "+" ]
         , ParseNode Exp   [ ParseLeaf "y" ]
         ]
```

Code D.3: A simple parse tree

D.6 Grammar

In `grammar.ama` the algebraic type is specified in the way that is given in Section 3.2. Furthermore all the EBNF-notations as specified in Appendix E are denoted, as well as the way these algebraic constructors need to be printed when the parse tree is shown in a graphical window.

D.7 Process

The process part of the preprocessing, as shown in Figure D.1 converts a parse tree with the union-constructions to a parse tree that does not have these constructs. In Figure D.2 parts of the program that form the process-part together are shown. These parts are:

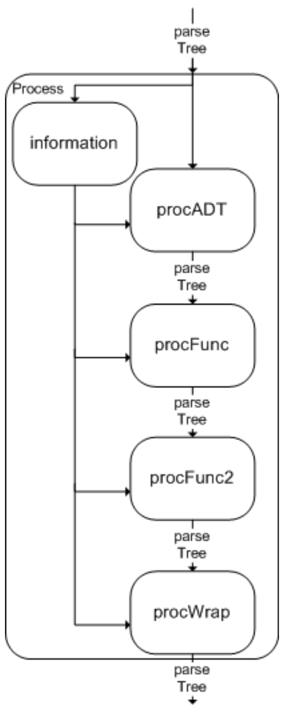


Figure D.2: The outline of the processing

- *Information*
  Information is a function that traverses the parse tree and gets information about:
  - *Union declarations*
    All union declarations are stored along with their elements
  - *Function declarations*
    All new function declarations containing <F> .
  - *Replacements*
    All types part of a union are stored along with their wrapping constructors.

- *procADT*
  The union type is rewritten as described in Chapter 4, the constructors are of the form `Nr_A_0` where `A` is the type and 0 is the index.

- *procFunc*
  All functions that have a union type as an argument are constructed. Provided that the function is written in a curried way, without arguments.

- *procFunc2*
  All functions that have a union as a result type are written. No check is done on the patterns on the left hand side of the function. When functions overlap Amanda will give an error. When two functions are being composed that have different variable names at the left hand side of a function, the variables are made equal in the where clause. Consider the functions `j` and `k` given in Code D.4 that are being composed in function `l`. The generated function `l` is given in Code D.5

```
j :: [char] -> a
j x = A , if x = "A"
    = B , if x = "B"


k :: [char] -> b
k y = C , if y = "C"
    = D , if y = "D"
```

Code D.4: Functions with different variables

```
l x  = (Nr_t_0 (A)), if x = "A"
     = (Nr_t_0 (B)), if x = "B"
     = (Nr_t_1 (C)), if y = "C"
     = (Nr_t_1 (D)), if y = "D"
       where
        y = x
```

Code D.5: Merging of functions with different variables

- *procWrap*
  When the result of a function is a constructor that must be wrapped to belong to a union type this is done in the *ProcWrap* function.

D.8 Postproc

The `post`-function in the `postproc`-file post-processes the printed string. This is done to generate code that is better readable. Code that is generated need not be read when generated in a proper way. Yet it is always better to create code that is readable when generated. Therefore all #R#-strings, that are added by the printing function are again processed and the "=" is printed directly below the previous "=". These #R# tokens are added by the printing function after each case. A case is a function clause with a guard, as can be seen in Appendix E. Function `f` is printed in the string as shown in Code D.6. The way the function is printed after the post-processing step is shown in Code D.7.

```
f x = x  , if x > 0 #R# = 0, otherwise
```

Code D.6: Reindentation

```
f x = x  , if x > 0
    = 0  , otherwise
```

Code D.7: Reindentation

D.9 Printer

The way the parse tree must be printed is specified in the file `printer`. For most algebraic constructors is specified how the functions must be printed. The algebraic constructors that are not specified are just printed in a default way. Two print functions are added to make live easier:

- *cprint*
  Prints the list of parse trees and concatenates them.
- *mprint*
  Prints the list of parse trees and prints a string between all elements.

D.10 Limitations of the Parser

The parser is created in an ad hoc way and some parts are not implemented in an extensible way. The limitations of the preprocessor are:

D.10.1 Parallel function compositions

Functions that are composed in a parallel function composition can not deal with variable with the same name in clauses that are being merged. For instance the functions `f` and `g` can not be merged. This should be possible by renaming the variables, yet in the implementation this is not implemented.

D.10.2 Memory

The preprocessor in some cases is to powerful for the programming language Amanda. This shows sometimes when a file is processed that Amanda will give the error message: "ERROR: out of memory". Maybe that with using the `strict` function the use of the memory could be downsized, yet I did not look into that.

D.10.3 Type Definitions

For each function a type definition is needed to check whether this function should have wrapping constructors. This could be enlightened a bit by assuming that when no type is given the function need not be processed. Yet this is not assumed by the preprocessor.

However Amanda gives an error message when some patterns overlap. Same constant patterns can be declared in different clauses, variables can not be declared in subsequent clauses.

E

Amanda Grammar

Alt	::=	Exp If
Argtype	::=	Typename
	|	”(” Typename (Argtype)$^+$ ”)”
	|	Typevar
	|	ListType
	|	RecordType
	|	TupleType
	|	ArgtypeB
ArgtypeB	::=	”(” Type ”)”
Case	::=	Alt Dedent ”=” Cases
CaseLast	::=	Lastalt (Whdefs)$^?$
Cases	::=	Case
	|	CaseLast
ConstantChar	::=	” ´” All ” ´”
ConstantString	::=	” ´ ´”<< text >> ” ´ ´”
Construct	::=	IDENTIFIER (Argtype)*
	|	”(” Construct ”)” (Argtype)*
Constructs	::=	Construct (”|” Construct)*
Decl	::=	Def
	|	Tdef
	|	Spec

Dedentation	::=	(”;”)* (Dedent)*
Dedent	::=	”#D#”
	|	”;”
Def	::=	Fnform ”=” Rhs
	|	Pat ”=” Rhs
	|	**Fnform ”=” Fdef**
Dollar	::=	”$”
Dontcare	::=	”_”
Dot	::=	”.”
Exp	::=	E1
	|	Prefix1
	|	Infix
E1	::=	Float
	|	E2 (Infix E1)$^?$
E2	::=	(Simple)$^+$
	|	Prefix E1
Fdef	::=	**Exp (”<F>” Exp)**$^+$
Float	::=	Nat Dot Nat
Fnform	::=	Identifier (Formal)*
	|	”(” Fnform ”)” (Formal)*
Formal	::=	Identifier
	|	IDENTIFIER (Formal)*
	|	Literal
	|	Dontcare
	|	TuplePat
	|	ListPat
	|	FormalB
	|	RecordPat
FormalB	::=	”(” Formal ”)”
Generator	::=	Pat ”<-” Exp
Identifier	::=	Lower (Ident)*
IDENTIFIER	::=	Upper (Ident)*
If	::=	”,” ”if” Exp

```
       Infix  ::=  '++' | '–' | ':' | ' ' | '>' | '>=' | '=' | '˜='
                   | '<=' | '<' | '+' | '*' | '!' | 'div' | 'mod'
                   | '.' | '/' | '^' | '-' | '\/' | '/\']
               |   Dollar Identifier
   LambdaExp  ::=  Pat "->" Exp
     Lastalt  ::=  Exp If
               |   Exp Otherwise
      Libdir  ::=  "#" "import" ConstantString
  ListCompre  ::=  "[" Exp "|" Qualifier (";" Qualifier)* "]"
     ListExp  ::=  ListExps
               |   ListCompre
               |   ListRange
    ListExps  ::=  "[" (Exp ("," Exp)* )? "]"
     ListPat  ::=  ListPatFull
               |   ListPatCons
 ListPatCons  ::=  "(" Pat (":" Pat)+ ")"
 ListPatFull  ::=  "[" (Pat ("," Pat)* )? "]"
   ListRange  ::=  "[" Exp ".." (Exp)? "]"
    ListType  ::=  "[" Type "]"
     Literal  ::=  Numeral
               |   ConstantChar
               |   ConstantString
         Nat  ::=  (Digit)+
     Numeral  ::=  Nat
               |   Float
   Otherwise  ::=  "," "otherwise"
         Pat  ::=  ListPat
               |   TuplePat
               |   RecordPat
               |   "(" Pat ")"
               |   Formal
      Prefix  ::=  Prefix1
               |   "-"
     Prefix1  ::=  "#"
               |   "~"
```

Qualifier	::=	Exp
	|	Generator
RecordExp	::=	RecordPat
	|	RecordUpdate
RecordPat	::=	”{” (RecordVarUpd) (”,” RecordVarUpd)* ”}”
RecordSpec	::=	Identifier ”::” Type
	|	Tform ”::” Type
RecordType	::=	”{” (RecordSpec) (”,” RecordSpec)* ”}”
RecordUpdate	::=	Identifier ”&” RecordPat
	|	Identifier ”&” Exp
RecordVarUpd	::=	Identifier ”=” Exp
Rhs	::=	Simple_rhs
	|	Cases
Script	::=	(Libdir | Decl)*
Simple	::=	Identifier
	|	Literal
	|	LambdaExp
	|	SimpleB
	|	ListExp
	|	TupleExp
	|	RecordExp
	|	IDENTIFIER
SimpleB	::=	”(” Infix E1 ”)”
	|	”(” E1 Infix ”)”
	|	”(” (Exp)$^+$ ”)”
Simple_rhs	::=	Exp (Whdefs)$^?$
Spec	::=	Identifier ”::” Type
	|	Tform ”::” Type
Tdecl	::=	Tform ”::=” Tdecls
	|	Tform ”==” Union
Tdecls	::=	Constructs
	|	RecordType
Tdef	::=	Tsynonym
	|	Tdecl
Tform	::=	Typename (Typevar)*

Tsynonym	::=	Tform ”==” Type
TupleExp	::=	”(” Exp (”,” Exp)* ”)”
TuplePat	::=	”(” Pat (”,” Pat)$^+$ ”)”
TupleType	::=	”(” Type (”,” Type)$^+$ ”)”
Type	::=	Argtype (”->” Argtype)*
Typename	::=	Identifier
Typevar	::=	(”$*$”)$^+$
Union	::=	**Tform (”<U>” Tform)**$^+$
Whdefs	::=	”where” (Def Dedent)* Dedent
Lower	::=	’a’ | .. | ’z’
Upper	::=	’A’ | .. | ’Z’
Digit	::=	’0’ | .. | ’9’
Ident	::=	Lower
	|	Upper
	|	Digit
	|	’_’

F

Differences between Amanda and Haskell

In this Appendix I will address some important differences between Amanda and Haskell. Throughout this thesis I used both languages, because I created a rewriting scheme for both languages and I addressed both models of I/O handling.

I will show the differences by some examples. First the use of guards is different, in Amanda the guards are written most right, with ", **if**" or ", **otherwise**". Whereas in Haskell guards are written before the = sign, using `|`. Also note the difference between the equality operator = and ==:

```
fac n
 = 1          , if n = 0
 = n*fac (n-1), otherwise
```

Code F.1: Amanda faculty function

```
fac n
 | n == 0    = 1
 | otherwise = n * fac (n-1)
```

Code F.2: Haskell faculty function

In Amanda types are written with a lowercase letter, whereas in Haskell types are written with an uppercase letter. The types denoted in haskell with a lower case letter are type variables. Amanda uses the sequence of `*` to denote type variables.

```
tree *
  ::= Node (tree *) (tree *)
    | Leaf *
```

Code F.3: Amanda Types

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

Code F.4: Haskell Types

λ-abstractions in both languages are almost identical. However, Haskell uses a \ to specify that the current function is a λ-abstraction.

```
f = x -> y -> x + y
```
Code F.5: Amanda Lambda

```
f = \x -> \y -> x + y
```
Code F.6: Haskell Lambda

There is a difference in the way Haskell and Amanda handle patterns and guards. The evaluation of `foo` (**Just** 4) in Amanda using Code F.7 will result in a run-time error whereas the same expression in Haskell (Code F.8) will evaluate to **False**. This is caused by the way a clause is selected. A function clause in Amanda is chosen when the pattern matches, then the guards are evaluated, whereas in Haskell the pattern and guard must match both to evaluate the expression.

```
foo :: maybe num -> bool
foo (Just x) = True , if x = 0
foo _        = False
```
Code F.7: Amanda Patterns/Guards

```
foo :: Maybe Int -> Bool
foo (Just x) | x == 0 = True
foo _                 = False
```
Code F.8: Haskell Patterns/Guards

In Haskell the type to represent integer values is **Int** and the type to represent floating point numbers is **Float** and **Double**. In Amanda only one type is available for numbers, the type **num**. In Amanda there are two functions to divide numbers. One that returns the integer value of the division / and one that returns the floating point number of the division //.

Furthermore there are some features that are available in Haskell that are not available in Amanda. A list of features in Haskell that are not present in Amanda, without attempting to be exhaustive, are: modules, type classes, generics, ffi (foreign function interface(Finne et al., 2003)) etc.

Algebraic constructors are considered functions in Haskell, whereas in Amanda constructors are just algebraic constructors. The following function `foo` in Code F.9 is valid in Haskell, but can not be expressed in this way in Amanda. In this function the constructor **Just** is used as a function with type **String -> Maybe String**.

```
foo :: Int -> Maybe String
foo = Just . show
```
Code F.9: The constructor used as a function