Application Patterns in Functional Languages

Nikolaas N. Oosterhof

2005

Application Patterns in Functional Languages

by

Nikolaas N. Oosterhof

a Thesis

written under the supervision of

dr. ir. Jan Kuper

dr. Maarten M. Fokkinga   drs. Joeri van Ruth

submitted to the

department of Computer Science

in partial fulfilment of
the requirements for the degree of

Ingenieur

equivalent to a Master's degree

in

Computer Science

University of Twente
Enschede, The Netherlands
2005

Supervisor committee:

dr. ir. Jan Kuper	Primary supervisor
dr. Maarten M. Fokkinga	Secundary supervisor
drs. Joeri van Ruth	Secundary supervisor

This thesis can be cited as:
Oosterhof, Nikolaas N. (2005). *Application Patterns in Functional Languages.* Master's thesis, University of Twente, The Netherlands.

The author may be contacted at `n.n.oosterhof@student.uva.nl`

This thesis is set in Computer Modern Roman by the author using LaTeX.

To my parents

Abstract

Most contemporary pure functional languages provide support for *patterns* in function definitions. Examples of common patterns are the identifier, constant, tuple, list algebraic, `n+k` and as-pattern.

This thesis introduces a new kind of patterns, called application patterns. Such patterns consist of a function applied to arguments: they are of the form `(f` x_1 `...` x_n`)`. When such a pattern is matched against an actual argument, inverse functions are used to find the binding of variables to values. A theoretical framework is provided that accomodates for defining multiple *generalized inverse functions* (for returning different sets of arguments) for one function. These inverse functions can be available in the system, derived by the system or defined by the programmer. A notation is introduced so that in a definition's left hand side identifiers can be used that are bound in the context. It is established that application patterns are universal in the sense that they include constant, tuple, list, algebraic, `n+k` and as-patterns.

This thesis describes an algoritm that translates functional program code with application patterns to program code without application patterns that can be run on an interpreter. It also provides a proof-of-concept implementation of this algoritm in a functional language.

Keywords: Functional programming, pattern matching, inverse function, application pattern

Contents

Acknowledgements

I would like to thank my supervisors Jan Kuper, Maarten Fokkinga and Joeri van Ruth for their continuous support and valuable remarks, suggestions and feedback.

The important roles of both Jan Kuper and Philip Hölzenspies can hardly by underestimated. I think the three of us contributed evenly important idea's for the development of the idea of application patterns. We had many long meetings with great discussions about syntax, semantics, esthetics, pragmatics and many other aspects of the concept. Jan and Philip, it has been a privilege working with you.

Jan and Philip should also be credited for initiating the Tina development group. It was this group, with great contributions by Jillis te Hove, Emond Papegaaij, Arjan Boeijink, Ruben Smelik and Berteun Damman that revived my enthousiasm for functional programming. This group was very enthusiastic and open-minded so that the idea for application patterns (in its most basic form, at that time) could develop. Thank you guys for the nice meetings and the great atmosphere.

Finally I would like to thank my parents Dick and Janet, my brother Chris and my sisters Jantine and Dianne for the important role they fulfill in my Life, the Universe, and Everything. I would like to dedicate my work to all five, but as there is also another thesis in Philosophy of Science, Technology and Society, *Thinking Machines That Feel: the Role of Emotions in Artificial Intelligence Research*, I considered an even split of dedication for the two generations most appropriate. Therefore, this thesis is dedicated to my parents.

Chapter 1

Introducing patterns

This chapter provides the necessary background for the remainder of this thesis. It provides a short introduction in the world of functions, function definitions in functional programming and pattern matching. The structure is as follows: in Section 1.1 the concept of a *function* is described, both from a mathematical and a functional programming perspective. In functional programming, patterns are important in function definitions. Therefore, Section 1.2 provides an overview is given of typical patterns in functional programming.

1.1 Functions

In this section a brief overview of functions is given. First mathematical functions are described, followed by functions and their definitions in functional languages.

1.1.1 Mathematical functions

In mathematics, *functions* are objects that map elements in one set to sets to elements in another set. If A and B are sets, then the relation $f \subset A \times B$ is called a *partial function* if, for every $x \in A$, there is at most one $y \in B$ so that $(x, y) \in f$. Instead of $(x, y) \in f$, we write $f(x) = y$ or $f\ x = y$, and say that $y \in B$ is the *image* or *result* of f applied to $x \in A$. Furthermore, A and B are called the domain and codomain, and we write $f : A \to B$.

A partial function $f : A \to B$ is a *total function* if, for every $x \in A$, there is one $y \in B$ so that $f(x) = y$. Except when stated otherwise, *function* indicates a *partial function*.

A total function $f : A \to B$ is the *total inverse* of the total function $g : B \to A$ if, for every x and y, $(x, y) \in f$ if, and only if, $(y, x) \in g$. We then write $g^{-1} = f$. If f is the total inverse of g, then g is also the total inverse of f, and we say that f and g are *inverse functions*.

Sometimes two functions are not inverse, although they would be if their domain and codomain are restricted. A function $f : A \to B$ is the *partial inverse* of the function $g : C \to D$ if, for every $x, \hat{x} \in B \cap C$ and every $y \in A \cap D$, it holds that if $(x, y) \in g$ and $(y, \hat{x}) \in f$ then $x = \hat{x}$. That is, a function and its partial inverse may not be defined for the same (co)domain, but for their 'collective' (co)domain they must agree on the values they map. When no ambiguities can arise, we say losely that f is the *inverse* of g and write $g^{-1} = f$.

Example. Consider the functions in Table 1.1, whose domain and codomain are indicated. The functions f, g, h and sin are total functions. For g a total inverse function exists, whereas for f, h, tan and sin only an partial inverse exists.

Table 1.1: Some functions and their inverses

function	(co)domain	restricted (co)domain	inverse function
$f : m \mapsto m + 1$	$\mathbb{N} \to \mathbb{N}$	$\mathbb{N} \to \mathbb{N}^*$	$n \mapsto n - 1$
$g : x \mapsto 3x + 5$	$\mathbb{Q} \to \mathbb{Q}$		$y \mapsto \frac{y-5}{3}$
$h : x \mapsto x^2 - 2x - 3$	$\mathbb{R} \to \mathbb{R}$	$[1, \infty\rangle \to [1, \infty\rangle$	$y \mapsto 1 + \sqrt{4 + y}$
tan	$\mathbb{R} \to \mathbb{R}$	$\langle -\frac{\pi}{2}, \frac{\pi}{2}\rangle \to \mathbb{R}$	arctan
sin	$\mathbb{R} \to \mathbb{R}$	$[-\frac{\pi}{2}, \frac{\pi}{2}] \to [-1, 1]$	arcsin

The concept of functions can is important in functional languages. In such languages functions can be defined, applied to arguments and the result evaluated by a computer program.

1.1.2 Functions in functional languages

Functional progamming is an area of computer science where pure functional (computer) programs consists largely of, function definitions. Examples of such functional languages are Haskell, Clean, Gofer and Miranda. Although in mathematics it may be sufficient that a function *exists* with certain properties, in functional programming functions are defined in an *algoritmic fashion*. That is, given a function f and a value x, the image of x under f (if it exists) can be calculated in finite time using only a limited number of well-defined rules.

This thesis will not describe all features of pure functional programming in detail. For a good introduction the reader is refered to the books by Thompson (1995) and Plasmijer and Eekelen (1993).

A Miranda-like syntax is used throughout this thesis. The definition of a function `g` with k arguments takes the form

```
g pat₁ pat₂ ... patₖ
 = value₁ , if guard₁
 = value₂ , if guard₂
 ⋮
 where
   where-clauses

g pat′₁ pat′₂ ... pat′ₖ
 = ...
 ⋮
```

```
g pat″₁ pat″₂ ... pat″ₖ
 = ...
 ⋮
⋮
```

When g is applied to *actual arguments* (i.e., values) $\mathtt{a}_1$, ..., $\mathtt{a}_k$, the result is evaluated as follows. First the *left hand side* is considered, that is the part up to and including the last pattern $\mathtt{pat}_k$. The actual arguments are checked for having the right form, by *matching* them against the patterns $\mathtt{pat}_1$, ..., $\mathtt{pat}_k$ (more on pattern matching is said in the next section). If the patterns match, identifiers may be bound by these patterns and evaluation proceeds in the *right hand side*. The guards $\mathtt{guard}_1$, $\mathtt{guard}_2$, ... are evaluated, and the first $\mathtt{guard}_i$ that evaluates to `True` the corresponding $\mathtt{value}_i$ is returned as the result of the evaluation. If there is only one `guard`, the `if,` $\mathtt{guard}_1$ may be omitted. The last guard may also be `otherwise`, which is equivalent to `if True`

Both the $\mathtt{value}_*$'s and $\mathtt{guard}_*$'s may contain identifiers that are bound by patterns or function definitions in the *where-clauses*. Contrary to patterns in the left hand side of a definition, patterns in a where-clause may lead to a runtime exception if they do not match their right hand side.

It is possible, though, that the patterns $\mathtt{pat}_1$, ..., $\mathtt{pat}_k$ do not match, i.e. a pattern is refused. However, g's definition can consist of multiple *equations*, each with different patterns: $\mathtt{pat}_*$'s, $\mathtt{pat}'_*$'s, $\mathtt{pat}''_*$'s, and so on. For evaluation, the first equation whose patterns match is used. That means that if one of the $\mathtt{pat}_*$ patterns does not match, the $\mathtt{pat}'_*$'s are tried, then the $\mathtt{pat}''_*$'s, and so on, until an equation is found in which all patterns match the actual arguments. If no patterns match, evaluation is halted and a runtime exception is thrown.

The next section shows examples of common patterns in functional languages.

1.2 Pattern matching

In this section an overview of different types of patterns are given as they are used in languages such as Haskell (Peyton Jones, 2003) , Clean (Plasmeijer & Eekelen, 2001) and Miranda (Thompson, 1995). The syntax between these languages may differ slightly, but Miranda and in one case Amanda are taken as a starting point.

1.2.1 Identifier pattern

The *identifier pattern* is the most simple type of pattern. An actual arguments always matches and is bound to the identifier.

Example. Consider the function `max2`, that returns the greatest of two numbers. Note the first line, the *type definition*, that states that `max2` takes two numbers as arguments and returns a number as well.

```
max2 :: num → num → num
```

```
max2 x y = x, if x > y
         = y, otherwise
```

Using this definition, the maximum of the numbers `3` and `5` is computed easily.

```
max2 3 5
    { patterns match: x := 3, y := 5 }
 ⇒  3, if 3 > 5
    { 3 > 5 ⇒  False, first guard fails }
 ⇒  5, if True
 ⇒  5
```

Comments are shown between `{` curly braces `}`. Here the binding of `x` to `3` is denoted by `x := 3`. Binding `x` to `3` means [1] that each occurence of `x` can be replaced by `3` (except when `x` bound again to some other value in a where clause).

1.2.2 Constant pattern

Another pattern is the *constant pattern* that checks whether an actual argument equals a constant. This pattern refuses the actual argument if it is not equal to the constant.

Example. The function `isSemiVowel` takes a character as its arguments and returns whether this character is a semi-vowel.

```
isSemiVowel :: char → bool
isSemiVowel 'w' = True
isSemiVowel 'y' = True
isSemiVowel _   = False
```

This definition consists of three equations. The last equation contains the `_` pattern, which is an identifier pattern and may replaced by any 'fresh' identifier that is not bound elsewhere.

With this definition, it is easily determined that the `'y'` character is a semi-vowel.

```
isSemiVowel 'y'
    { 'y' does not equal 'w'; refuse first equation }
    { 'y  equals 'y', second pattern matches}
 ⇒  True
```

[1] As is the case in any *referential transparant* language such as Miranda, Clean and Haskell

1.2.3 List pattern

Lists are sequences of elements of the same type. Elements in a list are separated by comma's and enclosed by brackets. Some examples of lists are

```
[]
[5, 12, 13]
[[True, False], [False], []]
['t', 'i', 'n', 'a']
```

List of characters allow for a special notation using double quotes, so that the last list can also be written `"tina"`.

Any list can be considered as either the empty list, denoted `Nil` or `[]`, or as some head element `x` followed by a tail list `xs`. In the latter case we write `x:xs` using the *cons* operator ‘`:`’. Thus, we can write the list `[5, 12, 13]` as `5:(12:(13:Nil))`. The parenthesis are optional, so that `5:12:13:Nil` is also valid. In the list `x:xs` it is required that the elements in `xs` have the same type as `x`.

A list patterns is of the form `h:t`, where `h` and `t` are patterns that match the head and tail, respectively. Note that patterns may be *nested*: a pattern may contain another pattern. A list pattern is refused if the actual argument is the empty list, or if the head or tail pattern does not match.

Example. The `join2` function, also written infix ‘`++`’, joins two lists. It is defined recursively by

```
join2 :: [*] → [*] → [*]
join2 []     ys = ys
join2 (x:xs) ys = x : (join2 xs ys)
```

The lists `[1, 2]` and `[3, 4, 5]` are joined as follows.

```
join2 [1, 2] [3, 4, 5]
    { [1, 2] does not equal []; refuse first pattern}
    { second pattern matches; x := 1, xs := 2 }
 ⇒  1 : (join2 [2] [3, 4, 5]
       { [2] does not equal []; refuse first pattern }
       { second pattern matches; x := 2, xs := [] }
 ⇒  1 : 2 : (join2 [] [3, 4, 5])
       { [] equals [], first pattern matches }
 ⇒  1 : 2 : [3, 4, 5]
 ⇒  [1, 2, 3, 4, 5]
```

List patterns allow for powerful computations, as the following example shows.

Example. Consider the `prime` function that generates the list of primes. Its definition contains a list pattern (as well as a list comprehension, but that is not discussed here).

```
primes = sieve [2..]
sieve (p:x) = p : sieve [n | n <- x; n mod p > 0]
```

The list of primes is infinite and thus can never be evaluated completely. However, this definition does allow for the evaluation of an initial sublist of the list of primes. For example,

```
take 5 primes
 ⇒* [2, 3, 5, 8, 13]
```

1.2.4 Tuple pattern

A *tuple* consists of a finite number of elements, possibly of different types. Elements are seperated by comma's en enclosed by brackets, and a tuple with n elements is called an n-tuple. Some examples are

```
(1, 2)
('t', 1, 'n', 'a')
((1, 'a'), ('z', 2))
```

A tuple matches a tuple pattern if they have the same number of elements and each element matches the respective pattern in the tuple.

Example. A complex number $a + bi$ $(a, b \in \mathbb{R})$ can be represented by a 2-tuple (*pair*) of numbers `(a, b)`. The definition for multiplication of two complex numbers is straightforward.

```
complex == (num, num)

multComplex:: complex → complex → complex
multComplex (a0, b0) (a1, b1)
 = (a0*a1 - b0*b1, a0*b1 + a1*b0)
```

The complex product of $(3 + 4i) \cdot (-1 + 2i)$ is computed by

```
multComplex (3,4) (-1, 2)
    { a0 := 3, b0 := 4, a1 := -1, b1 := 2 }
 ⇒  (3 * -1 - 4 * 2, 3 * 2 + -1 * 4 }
 ⇒  (-11, 2)
```

resulting in $-11 + 2i$.

1.2.5 Algebraic pattern

Algebraic types allow for labelling values of different types with a *constructor* and then joining them in a type. A constructor is written with an uppercase identifier and is defined to be accompanied by a finite number of values called *arguments*. Examples of definitions with algebraic types and values are

```
bool     ::= True | False
functionalProgrammingIsObsolete :: bool
functionalProgrammingIsObsolete = False

binTree ::= Leaf num | Node binTree binTree
myTree = Node (Leaf 3) (Node (Leaf 4) (Leaf 8))

maybe * ::= Just * | Nothing
```

In these definitions `True`, `False`, `Leaf`, `Node`, `Just` and `Nothing` are the *constructors*, and they have zero, zero, one, two, one and zero *arguments*, respectively.

The `binTree` is a binary tree that contains numbers. With the `maybe` type one can specify that a result *is not* defined (`Nothing`), or that the result *is* defined and has value v (`Just` v). In the `maybe` definition the asterisk '`*`' is a type variable, used for polymorphism, so that both `Just 3` (of type `maybe num`) and `Just "ab"` (of type `maybe [char]`) are valid expressions. One example of its use is given in Chapter 3.

An algebraic pattern matches an algebraic value if it has the same constructor and each argument matches. Note that constants, lists and tuples can be expressed using algebraic types.

Example. Consider finding the sum of all numbers in a binary tree.

```
sumBinTree (Leaf n)   = n
sumBinTree (Node x y) = sumBinTree x + sumBinTree y
```

Now summing up values in `myTree` proceeds as follows:

```
sumBinTree myTree
 ⇒ sumBinTree (Node (Leaf 3) (Node (Leaf 4) (Leaf 8)))
   { Node (Leaf 3) (Node (Leaf 4) (Leaf 8))
       does not match Leaf x; refuse first equation}
   { Node (Leaf 3) (Node (Leaf 4) (Leaf 8))
       matches Node x y; x := Leaf 3,
                         y := Node (Leaf 4) (Leaf 8) }
 ⇒ sumBinTree (Leaf 3)
      + sumBinTree (Node (Leaf 4) (Leaf 8))
   { Leaf 3 matches Leaf n; n := 3 }
 ⇒ 3 + sumBinTree (Node (Leaf 4) (Leaf 8))
 ⇒* 3 + (4 + 8)
```

```
⇒* 15
```

1.2.6 n+k pattern

The `n+k` pattern is a pattern that is only applicable to numbers. In its most common form (e.g., Miranda and Gofer), it only matches nonnegative integers. In an `n+k` pattern, the `k` is a constant number and `n` is een identifier. When applied to an actual argument `a`, `n` is bound to `a-k`. Intuitively we can understand this as 'solving' the equation `n+k=a` for `n`, which leads to `n=a-k`.

Example. The `power` function for natural numbers can be defined by the use of an `n+k` pattern. In this example `k=1`.

```
power _ 0      = 1
power b (n+1)  = b * power b n
```

Evaluation of 3^2 proceeds as follows

```
power 3 2
    { 0 does not equal 2; refuse first equation }
    { patterns match: b := 3, n := (2-1) }
 ⇒  3 * power 3 1
    { 0 does not equal 1; refuse first equation }
    { patterns match: b := 3, n := (1-1) }
 ⇒  3 * 3 * power 3 0
    { 0 equals 0; use first equation }
 ⇒  3 * 3 * 1
 ⇒  9
```

An extension to this pattern is available in the Gofer language, as a result of a discussion on the Haskell mailinglist initiated by Tony Davie (as cited by Jones, 1991). Gofer allows for `c * p` and `p + k` patterns, where `c` > 1 and `k` > 0 are constants. It extends the syntax using a grammer so that these patterns can be nested.

$$\textit{pattern} \rightarrow \ldots \mid \textit{pattern} + \textit{integer} \mid \textit{integer} * \textit{pattern}$$

The semantics of a nested pattern of this form is comparable to that of the standard `n+k` pattern.

Example. Tony Davie (as cited by Jones, 1991) gives the following more efficient definition of the `power` function

```
power' x 0         = 1
power' x (2*n)     = xn * xn
                     where xn = power' x n
power' x (2*n+1)   = x * power' x (2*n)
```

The second and third clause use a `c*p` and a `c*p+k` pattern (with `c=2` and `k=1`), respectively. Evalution of 2^2 using this definition goes as follows:

```
power' 2 2
 ⇒  power' 2 2
    { 0 does not equal 2, first equation fails }
    { 2 equals 2*n for n := 1
 ⇒  xn * xn
    where
      xn
       = power' 2 1
      { 1 does not equal 0, first equation fails }
      { 1 does not equal 2*n for any n, second
          equation fails }
      { 1 equals 2*n+1 for n := 0 }
           ⇒  2 * power' 2 0
           ⇒  2
 ⇒  2 * 2
 ⇒  4
```

Note that this pattern matches only if the to-be-bound identifier can be given a non-negative integer value.

1.2.7 As pattern

The as pattern allows for binding identifiers multiple times to (parts of) the actual argument. That is, if `pat` is a pattern, `(x=pat)` is a binding pattern that binds `x` to `pat` if `pat` matches the actual argument (in Haskell the `x@pat` notation is used). It is assumed that `x` is a free identifier.

Example. The function `headListTail` gets a non-empty list as argument. It returns the head of the list, followed by the complete list, followed by the tail of the list.

```
headListTail (list=(x:xs)) = x : list ++ tail
```

Evaluation of this function applied to the list `abcde` leads to

```
headListTail "abcde"
   { list pattern matches: x := 'a', xs := "bcde" }
   { binding pattern matches: list := "abcde"
 ⇒  'a' : "abcde" ++ "bcde"
 ⇒  "aabcdebcde"
```

1.2.8 Equivalence pattern

The equivalence pattern allows for multiple occurences of an identifier in one equation. Not every language supports this: Haskell and Miranda allows for such patterns but not Amanda. An equivalence pattern matches if every occurence of an identifier agrees on a value for that identifier.

Example. Consider the function that determines the greatest common devisor of two numbers.

```
gcd x x = x
gcd x y = gcd (x-y) y, if x > y
        = gcd x (y-x), otherwise
```

The first line shows an example of an equivalence pattern, stating that the greatest common divisors of two equal numbers is that number itself. Using this definition, the greatest common divisor of `18` and `9` is calculated as

```
gcd 18 9
     { x := 18, x := 9, conflict in first equation, no
          match }
     { x := 18, y := 9, second equation matches }
     { 18 > 9 ⇒  True }
 ⇒  gcd (18-9) 9
 ⇒  gcd 9 9
     { x := 9, x := 9 -- no conflict, matches }
 ⇒  9
```

Except when stated otherwise, in the remainder of this thesis it is assumed that any function definition does not contain an equivalence pattern. However, as we will see in Section 2.4.2, using a caret notation equivalence patterns can be mimicked while still each identifier is bound only once.

Several researchers have suggested new forms of pattern matching that are beyond standard pattern matching described so far. In the next chapter yet another form of pattern matching is introduced. The relation with other proposed pattern matching extensions is discussed in Section 4.2.

1.3 Conclusion

This chapter gave an overview of function definitions and standard pattern matching. Function definitions may contain patterns, guards, expressions and `where` clauses. Examples of existing patterns are the identifier, constant, list, algebraic, tuple, `n+k`, as and equivalence pattern.

In the next chapter a new type of patterns is introduced: application patterns.

Chapter 2

Application patterns

2.1 Introduction

In the previous section the necessary background about functions and pattern matching was discussed. In this chapter application patterns are described. The structure of this chapter is as follows: Section 2.2 introduces the concept of application patterns, initially in a basic form that is extended with refutability. Section 2.3 adds application patterns with an arbitrary number of arguments. In Section 2.4 some refinements are discussed. Section 2.5 describes how application patterns can be considered as a general form of most other patterns. Section 2.6 lists the definition of inverse functions for many standard functions. Finally the use of application patterns is discussed in Section 2.7.

2.2 Application patterns

Application patterns as presented here were introduced by Oosterhof, Hölzenspies, and Kuper (2005). This section describes such patterns, starting with a basic from to which refutability is added.

2.2.1 Basic application pattern

In its most simple form, an application pattern is a pattern of the form `f x`. As any other pattern, application patterns can be used in both the left hand side of function definitions and in where clauses. The use of an application pattern requires that an inverse function $\mathtt{f}^{-1}$ is defined. Such an inverse may be available as a standard function of the programming language (see 2.6), it can be derived by the systems (a discussion of this subject is beyond the scope of this thesis) or it must be defined by the programmer. For now it must be assured that for an actual argument a, $\mathtt{f}\ (\mathtt{f}^{-1} a) = a$ for every value for which `f` and $\mathtt{f}^{-1}$ are defined.

When an actual argument `b` is matched against a pattern `f x`, `x` is bound to the value of the inverse function $\mathtt{f}^{-1}$ applied to `b`. The intuition behind this procedure is that

$$\mathtt{f\ x} = \mathtt{b} \quad \Longleftrightarrow \quad \mathtt{x} = \mathtt{f}^{-1}\ \mathtt{b}$$

Note that matching against the pattern `f x` *does not* involve checking its syntactic structure. Contrary, an application pattern matches against the semantic value of the actual argument. Application patterns can be nested and used together with any other kind of pattern.

Example. The `succ` function computes the successor of any non-negative integer and has an inverse function $\texttt{succ}^{-1}$. Their definitions are trivial.

```
succ n    = n + 1, if n ≥ 0

succ⁻¹ n = n - 1, if n ≥ 1
```

Now consider yet another definition of the `power` function, one that uses an application pattern.

```
power'' _ 0        = 1
power'' b (succ n) = b * power'' b n
```

In the second equation, the second argument `succ n` is an application pattern that consists of the function `succ` applied to an argument `n`. Note that this pattern resembles an `n+k` pattern. Evaluation of `power'' 2 5` now proceeds as follows.

```
power'' 2 5
    { 5 does not equal 0, refuse first equation }
    { b := 2
      { 'solve'
           succ n = 5
                n = succ⁻¹ 5
                  ⇒ 4      }
      n := 4                             }
 ⇒  2 * power'' 2 4
 ⇒* 32
```

Example. The `zip` function zips a pair of lists into a list of pairs and has an inverse $\texttt{zip}^{-1}$.

```
zip :: ([*], [**]) → [(*, **)]
zip ((x:xs), (y:ys)) = (x,y) : zip (xs, ys)
zip _                = []

zip⁻¹ :: [(*, **)] → ([*], [**])
zip⁻¹ ((x,y) : xys) = (x:xs, y:ys)
                      where
                        (xs, ys) = zip⁻¹ xys
zip⁻¹ []            = ([], [])
```

Note that $\texttt{zip}^{-1}$ always returns lists of equal length, whereas `zip` is also defined for pairs of lists with unequal length. This means that $\texttt{zip}^{-1}$ is a partial inverse of the function `zip`.

Suppose a list of (x, y) coordinates is given. The function `upperLeft` returns the coordinates of the upperleft corner of the smallest rectangle (with sides parallel to the x and y axis) that contains all points of the list. It can be defined with an application pattern.

```
upperLeft :: [(num, num)] → (num, num)
upperLeft (zip (xs, ys)) = (min xs, max ys)
```

Here the `zip (xs, ys)` is an application pattern that consists of the function `zip` applied to the pair `(xs, ys)`. For an intuitive understanding of this definition, suppose `upperLeft` is applied to an actual argument `coords`. The meaning of the application pattern `zip (xs, ys)` is that, for some lists of x-coordinates `xs` and y-coordinates `ys`, `coords` is the result of zipping these two lists. The rectangles' upperleft corner is calculated by taking the minimum and maximum from the lists of x-coordinates `xs` and y-coordinates `ys`, respectively.

The upperleft point for the list of coordinates `[(1,4), (-2,3), (5,2)]` is evaluated as follows.

```
upperLeft [(1,4), (-2,3), (5,2)]
    { { 'solve'
           zip (xs, ys) = [(1,4), (-2,3), (5,2)]
               (xs, ys) = zip⁻¹ [(1,4), (-2,3), (5,2)]
                          ⇒* ([1,-2,5], [4,3,2])   }
      xs := [1,-2,5], ys := [4,3,2]                   }
⇒  (min [1,-2,5], max [4,3,2])
⇒  (-2, 4)
```

2.2.2 Refutable application patterns

Just like some other patterns, application patterns can be refutable. Suppose that the pattern `f x` is matched against an actual argument `a`. If for all possible values of `x`, `a` cannot be the result of `f x`, the pattern is refused. This is indicated by a partial definition of $\texttt{f}^{-1}$. In Chapter 3 it is shown how this approach allows for rewriting code that supports refutable application patterns.

Example. Consider the builtin sine function `sin`. Since the range of this function is $[-1, 1]$, its inverse $\texttt{sin}^{-1}$ is only defined for values in this range.

```
sin⁻¹ x = arcsin x, if -1 ≤ x ∧ x ≤ 1
```

Now consider the definition of `h` that contains an application pattern with the sine function.

```
h (sin a) = a * a
h x       = x - 2
```

When applied to an actual argument `b`, it depends on the value of this argument which of the two equations is used. When `h` is applied to $0.866 \approx \sqrt{3}/2$, the first equation is used which results in $1.0965 \approx \pi^2/9$.

```
h 0.866
    { { 'solve'
           sin a = 0.866
               a = sin⁻¹ 0.866
                     { -1 ≤ 0.866 ∧ 0.866 ≤ 1 ⇒ True,
                         guard succeeds }
                    ⇒ 1.0471                        }
      a := 1.0471                                   }
 ⇒ 1.0471 * 1.0471
 ⇒ 1.0965
```

However, when applied to a value with a greater absolute value than one, the second equation is used and its value is decreased by two.

```
h 100
    { { 'solve'
           sin a = 100
               a = sin⁻¹ 100
                     { -1 ≤ 100 ∧ 100 ≤ 1 ⇒ False,
                        guard fails }
         solving fails                             }}
    { second pattern matches: x := 100 }
 ⇒ 100 - 2
 ⇒ 98
```

Note that despite the fact that the application pattern `sin a` does not match `100`, no runtime exception is thrown but the next equation is tried. In the examples shown so far only function applications with one argument were used. In the next section this is extended to an arbitrary number of arguments.

2.3 Currying application patterns

In this section curried application patterns are discussed. The notions of a *cousin* and *generalized inverse* of a function are introduced, as well as a backtic notations that allows for defining generalized inverses.

2.3.1 Currying

Many functions take multiple arguments; we have already seen this in the `max2` and `power` functions. However, a function that takes n arguments can be seen as a higher order function that takes one argument and returns a function that takes $n-1$ arguments. This new function can again be applied to one argument, yielding another function that takes $n-2$ arguments, and so on. After $n-1$ steps all arguments are handled. Considering functions with multiple functions as sequences of a number of higher order functions that all take one argument is called *currying*.

Example. The `power` function takes two arguments, a base b and an an exponent x, and returns a number b^x. It can be considered as a curried function, with type

```
power :: num → (num → num)
```

If `power` is applied to an actual base argument (say `2`), the result is a function

```
power 2 :: (num → num)
```

that takes one argument and returns two-to-the-power-of-that-argument. When `power 2` is applied to another actual argument (say `5`) the result is the value `32`.

```
power 2 5 :: num
power 2 5 ⇒ 32
```

However, applying the idea of application patterns directly to curried functions would yield a typing problem. A function $f : A \to B$ takes one argument, and its inverse is of type $f^{-1} : B \to A$. But what is the type of, say, the inverse of the `power` function? Using the rule for functions with one argument would yield

```
!!! power⁻¹ :: (num → num) → num
```

but this would mean that the inverse of the `power` function would take a higher function as its argument. This approach has two problems, the first being that equality of higher order functions is undecidable (this is discussed in more detail in Section 4.3.2). The second problem is that is unclear what the *meaning* of such an inverse would be. However, for the power function meaningful inverse-like functions *can* be defined, because if the result b^x and one of the arguments (b or x) is known, the other argument can be found back:

$$\begin{aligned} b^x = s \quad & \Longleftrightarrow \quad b = s^{1/x} \\ & \Longleftrightarrow \quad x = \frac{\ln s}{\ln b} \end{aligned}$$

Thus, we need a generalized form of inverse functions. For this we need the concept of *cousins* of a function.

2.3.2 Cousins of functions

Suppose f is a function that takes n arguments

$$f : A_0 \to \cdots \to A_{n-1} \to B$$

with

$$f x_0 \ \cdots \ x_{n-1} = \mathit{expr}$$

For now it is assumed that the A_*'s and B types do not contain an '$\to$'—but for a discussion of the possibilities for such functions, see Section 4.3.2.

Note that we can partition the list of indices $0, \ldots, n-1$ into two sublists $i_1, \ldots, i_k$ and $j_1, \ldots, j_m$ that both keep their indices in order. We write

$$[i_1, \ldots, i_k] \cup [j_1, \ldots, j_m] = [0, \ldots, n-1]$$

where the $\cup$ operator merges two ordered lists into a new ordered list.

The function

$$f^c_{i_1,\ldots,i_k} : A_{j_1} \to \cdots \to A_{j_m} \to (A_{i_1}, \ldots, A_{i_k}) \to B$$

so that

$$f^c_{i_1,\ldots,i_k} x_{j_1} \ \cdots \ x_{j_m} \ (x_{i_1}, \ldots, x_{i_k}) = f \ x_0 \ \cdots \ x_{n-1}$$

is called a *cousin* of f with respect to $i_1, \ldots, i_k$. Thus a cousin of f more or less calculates the same as f itself, it only takes its argument in a different order.

The *trivial cousin of* f is the cousin of f with respect no none of its arguments and equals f itself: $f = f^c_\emptyset$ (so that $k = 0$, $m = n$ and $j_p = p - 1$).

Example. The repeat function takes a number n and an element c as its arguments and returns a list with n times the element c.

```
rep :: num → * → [*]
rep 0     c = []
rep (n+1) c = c : rep n c
```

The cousin of `rep` with respect to the first and second argument is

```
rep^c_{1,2} :: (num, *) →[*]
rep^c_{1,2} (0, c) = []
rep^c_{1,2} (n+1, c) = c : rep^c_{1,2} (n, c)
```

Example. The `join3` function takes three lists and joins them.

```
join3 :: [*] → [*] → [*] → [*]
join3 x y z = x ++ y ++ z
```

It has multiple cousins, two of them being

```
join3^c_2   x z y     = x ++ y ++ z
join3^c_1,3 y (x, z) = x ++ y ++ z
```

2.3.3 Generalized inverse

The concept of cousins, as described in the previous section, allows for defining inverse functions for curried functions.

Consider the function

$$f : A_0 \to \cdots \to A_{n-1} \to B$$

that has a cousin

$$f^c_{i_1,\ldots,i_k} : A_{j_1} \to \cdots \to A_{j_m} \to (A_{i_1}, \ldots, A_{i_k}) \to B$$

with respect to $i_1, \ldots, i_k$ (hence $[i_1, \ldots, i_k] \cup [j_1, \ldots, j_m] = [0, \ldots, n-1]$).

The *generalized inverse* of f with respect to $i_1, \ldots, i_k$ is a function

$$f^{-1}_{i_1,\ldots,i_k} : A_{j_1} \to \cdots \to A_{j_m} \to B \to (A_{i_1}, \ldots, A_{i_k})$$

such that

$$f^{-1}_{i_1,\ldots,i_k} x_{j_1} \ \cdots \ x_{j_m} y = (x_{i_1} \ \ldots, x_{i_k})$$

if and only if

$$f^{c}_{i_1,\ldots,i_k} x_{j_1} \ \cdots \ x_{j_m} (x_{i_1} \ \ldots, x_{i_k}) = y$$

that is, if and only if

$$f \ x_0 \ \cdots \ x_{n-1} = y$$

We call such a function an *inverse (function) on its $i_1 - 1$st, ..., and $i_k - 1$th argument.*

Note that both $f^c_{i_1,\ldots,i_k}$ and $f^{-1}_{i_1,\ldots,i_k}$ can be parametrized by m parameters. In a parametrized form, these two functions are *inverse functions*, i.e.

$$f^{-1}_{i_1,\ldots,i_k} x_{j_1} \ \cdots \ x_{j_m} = (f^c_{i_1,\ldots,i_k} x_{j_1} \ \cdots \ x_{j_m})^{-1}$$

The *trivial inverse of* f is the inverse of f on none of its arguments. It has the type

$$f^{-1}_{\emptyset} : A_0 \to \cdots \to A_{n-1} \to B \to ()$$

and is informally specified by

$$f^{-1}_{\emptyset} \ x_0 \ \ldots \ x_{n-1} \ \mathit{expr} = (), \mathsf{if} f x_0 \ \ldots \ x_{n-1} = \mathit{expr}$$

Note that his inverse is not defined for values in which the guard is *false*.

For ease of notation, we write the generalized inverse function

$$f^{-1}_{i_1,\ldots,i_k}$$

in Ascii using a *backtic notation* as

```
f‘[ i1, ..., ik ]
```

Note that the backtic ‘ is not an operator: `f‘[` i_1`, ...,` i_k`]` must be considered as a (systematically named) identifier. With this notation (generalized) inverse functions can be defined that are used in matching application patterns.

Example. Since a function can have many cousins, it can have many inverses too. For the power function two meaningul inverse functions can be defined, namely an inverse on its first argument

```
power‘[0] :: num → num → num
power‘[0] x s = s ^ (1 // x)
```

and an inverse on its second argument

```
power‘[1] :: num → num → num
power‘[1] b s = (ln s) // (ln b)
```

Both can be definied simultaniously.

The `sqroot` function $\sqrt{\cdot}$ takes the square root from a non-negative number. Its definition contains an application pattern with the power function.

```
sqroot (power x 2) = x
```

For the evaluation of $\sqrt{81}$, it must be decided *which* inverse of the `power` function must be used. Since in the application pattern `power x 2` the first argument is a variable whereas the second is known (a constant), we choose for the inverse on the first argument `power‘[0]`

```
sqroot 81
 { { 'solve'
       power x 2 = 81
       { choose power‘[0] }
             x   = power‘[0] 2 81
                   ⇒ 9          }
   x := 9                               }
 ⇒ 9
```

Thus, when multiple inverses are defined the choice *which* inverse is choosen depends on which arguments are known. If multiple inverses are possible, it is the programmer's responsability to ensure that it does not matter which inverse function is choosen.

Example. The repeat function has an inverse on its first and second argument. Given a list, `rep‘[0,1]` returns a pair with the length of the list and the first element—but only if all elements in the list are equal. Otherwise the `rep‘[0,1]` is not defined. The `rep‘[0,1]` function is not defined for an empty list, since the typing system requires that the type of the repeated element is known. Thus, strictly speaking, `rep‘[0,1]` is only a partial inverse of `rep`.

```
rep‘[0,1] :: [*] → (num, *)
rep‘[0,1] [x] = (1, x)
rep‘[0,1] (x : rep n y) = (n+1, x), if x=y
```

Note that the second equation contains a (nested) application pattern with the repeat function itself. Thus, this definition of `rep` is recursive.

Consider the function `f` that is defined with an application pattern with `rep`.

```
f (rep n x)   = x : rep n '.'
```

The evaluation of `f` applied to a list of two `'a'`’s uses the inverse function `rep‘[0,1]`

```
f "aaa"
    { 'solve'
        rep n x = "aa"
          (n,x) = rep‘[0,1] "aa"
                   { "aa" does not match [x] }
                   { "aa" might match x : rep n' y
                     x' := 'a', rep n' y := "a"
                     { 'solve'
                         rep n' y = "a"
                           (n',y) = rep‘[0,1] "a"
                                  ⇒ (1, 'a') }
                     n' := 1, y := 'a'
                     { x'=y ⇒ True }           }
                   ⇒ (n'+1, x')
                   ⇒ (2, 'a')
          n := 2, x := 'a'                     }
 ⇒ 'a' : rep 2 '.'
 ⇒ "a.."
```

From this definition, two other inverse `rep` functions can be derived. The first is used when the repeated element is known but the length of the list must be computed,

```
rep‘[0] :: * → [*] → num
rep‘[0] y (rep n x) = n, if x=y
```

whereas the other is used if the length of the list is known and the repeated element must be computed.

```
rep‘[1] :: num → [*] → *
rep‘[1] n (rep m x) = x, if m=n
```

Both definitions, when applied to actual arguments, would make implicit use of `rep‘[0,1]`. Note that the definition for `rep‘[0]` is not defined for an empty list, although in this case such a definition would make sense. A separate definition of `rep‘[0]` would fix this issue.

In this section the basic application pattern was introduced. Refutability and the concept of generalized inverses that allow for multiple inverses were added. The next section discusses some refinements to these ideas.

2.4 Some refinements

This section describes how application patterns could be used in `where` clauses, lambda abstractions and list comprehensions. Also the *caret* notation is introduced. This notation allows for more flexible use of bound identifiers in patterns. Finally extraction functions are discussed.

2.4.1 Pattern expressions

So far we have only dealt with application patterns in the left hand side of function definitions. Since ordinary patterns can also be used in lambda abstractions, `where` clauses and list comprehensions, it is proposed that application patterns can be used in these expressions. Some examples are:

where clauses such as an application pattern that binds `n` in

```
...
  where
    2^n = 8
```

(Note that the above might be read as a definition for the power function using a constant pattern 2. This issue is addressed in the next section.);

lambda abstractions such as the function that maps every number 2^n to the number n^2

```
(2^n → n^2) 256
```

and

list comprehensions such as the list of numbers

```
[ n^2 | 2^n <- [1..8]]
```

With application patterns in function definitions the semantics of application patterns in lambda abstractions and list comprehensions are easily defined. Here it is left as an exercise for the reader. The implementation of this functionality is discussed in Section 3.4.6

Some ambiguities may arise in application patterns. To address this issue a caret notation is introduced in the next section.

2.4.2 Caret notation

Application patterns allow for more flexible syntax in function definitions. However, their use may lead to ambiguities. A caret notation is introduced for identifiers that indicates that an identifier (or its inverse function, in the case of an application pattern) must be retrieved from the context.

Example. Consider Haskell (Peyton Jones, 2003), where an expression of the form `n+k` can be used in the left hand side of a where clause, as in

```
...
  where
    n + 1 = expr
```

Now this reads as a (re)definition of the addition function in a where clause. But with this syntax, how could an `n+k` pattern be used in a `where` clause? The solution that has been chosen in Haskell is to surround the expression `n + 1` by parenthesis, as in

```
...
  where
    (n + 1) = expr
```

A similar problem would arise for the use of an application pattern in a `where` clause. Rather than surrounding the pattern by parenthesis (which adds even more semantics to these tokens), an alternative solution is proposed: the caret `^` identifier-prefix notation. To indicate that a function application should be used *to bind arguments* (instead of defining the function itself) the function name must be prefixed by a caret `^`. Obviously the prefixed function identifier must have an inverse defined in the context.

Example. With the caret notation the definition of `sqroot'` could be written

```
sqroot' x
 = y
 where
   ^power y 2 = x
```

where the caret indicates that the inverse of the `power` function is retrieved from the context. Note that such ambiguities cannot only arise in `where` clauses, not in the left hand side of function definitions, in lambda abstractions or in list comprehensions. In these cases the caret is optional at the function identifier.

Use of the caret for operators would make expressions less readible. Since operators will rarely be redefined in a `where` clause, it is proposed here that the caret is left out for operators too.

Example. Use of an application function with the addition function `+` in a where clause, as in

```
| f y = x + y
|   where
|     x ^+ 2 = y
```

results in a less readible `^+` operator. Note that the same problem applies to other operators, which would be written as, for instance, `^++`, `^:`, `^!` and even `^^`. As the caret is left out for operators, the definition of `f` can be written

```
| f y = x + y
|   where
|     x + 2 = y
```

Sometimes it may be desired to use identifiers from the context in an application pattern. To allow for this the caret notation is extended to any identifier, not just a function name in a function application. The semantics of an identifier `^i` is a constant with the value of `i` derived from the context (assumed that that `^i` *not the function identifier in an application* `^iargs` .

Example. The definition of the factorial function can also be written

```
fac ^zero = 1
fac n     = n * fac (n-1)

zero = 0
```

Example. Yet another definition of square root is

```
| sqroot'' (power x ^two) = x

| two = 2
```

The caret notation must be used with care, as it allows for different semantics of almost identifical expressions, especially in `where` clauses.

Example. Suppose the function `f` takes two arguments in the context

```
x = 2

f‘[1]    x s = ...
f‘[0,1]    s = ...
```

where the inverse functions `f‘[0]` and `f‘[0,1]` are both non-refutable.

Table 2.1 shows variants of its use (with and without carets) in a where clause, and for each variant an equivalent `where` clause without application patterns. This is an example of what is presented in Section 3.3 more generally, namely an approach to rewrite code with function definitions and where clauses into equivelant code without application patterns.

Table 2.1: Some `where` clauses and equivalent clauses

Example `where` clause	Equivalent `where` clause without application patterns and carets

`...`
`  where`
`    f ^x y = expr`
	`...`
	`  where`
	`    f 2 y = expr`

`...`
`  where`
`    ^f x y = expr`
	`...`
	`  where`
	`    (x, y) = f‘[0,1] expr`

`...`
`  where`
`    ^f ^x y = expr`
	`...`
	`  where`
	`    y = f‘[1] 2 expr`

As mentioned before, in a nested application pattern like `g (f x y)` the caret may be left out in front of `f`, since no ambiguity can arise. Thus, such a pattern is equivalent to `g (^f x y)`.

The caret notation can be extended marginally by assuming that in a left hand side of a function definition, the *context of an identifier* also includes other patterns. Equivalence patterns are not supported by all languages[1]. The caret notation allows for mimicking such a pattern.

[1] Amanda is a clear example. Without extra compiler support, Haskell also requires that in the left hand side ‘The set of patterns must be linear—no variable may appear more than once in the set.’ (Peyton Jones, 2003)

Example. Consider the following definition of the function that computes the greatest common divisor for two positive integers.

```
gcd x        ^x       = x
gcd (x + ^y)  y       = gcd x y
gcd x        (x + ^y) = gcd x y
```

The first equation contains an identifier pattern and a constant pattern that together mimick an equivalence pattern. In the first pattern `x` is bound the actual argument, in the second pattern the actual argument is compared to the value that is bound in the first pattern, i.e. it is compared to the first actual argument. The second equation contains an `n+k` pattern. In this pattern the value of `y` is used from the context to compute `x`. This means that `y` is bound in the second pattern *before* the first pattern is evaluated.

In general, since application patterns may be refutable, the caret notation as proposed here may yield a different order of pattern matching during evaluation. Consider the definition of `f` in

```
f (power ^x y) (sin x) = x * y
f _            _       = 0
```

that contains two application patterns. During evaluation, the first pattern requires that the value for `x` is known. This value must be retrieved from the context, in this case from the second pattern `h x`. Thus, first the second pattern must be matched against the second argument, and only then the first pattern can be matched. If the second pattern in the first equation does not match the second equation is used.

In brief, with the caret notation it is denoted explicitly which identifiers are bound and where. It allows for mimicking equivalence patterns and adds a limited amount of power to existing pattern matching.

2.4.3 Extraction functions

So far we have assumed that defined inverse functions are really the inverse of some function, and that such inverse functions can be defined more or less uniquely. However, for application patterns both assumptions are not really required. In fact, the programmer may go as far as he wants, as far as he assures that the defined inverse functions are used properly.

Example. The `upperLeft` function

```
upperLeft :: [(num, num)] → (num, num)
upperLeft (zip (xs, ys)) = (min xs, max ys)
```

contains an application pattern with the `zip` function. For its use the definition of `zip‘[0]` is required. However, the definition of the `zip` function is not really required for this definition of `upperLeft` to work. If an application patterns contains a function that is not a partial inverse, this function is called an *extraction function*.

2.4.4 Programmer's responsibilities

Another issue is the use of an application pattern for which different inverse functions can be defined for the same set of arguments.

Example. The function `join2` joins two strings

```
join2 x y = x ++ y
```

for which clearly two inverses can be defined, one for each argument.

```
join2'[0] y s = x, if y = z
                where
                  (x, z) = split ((#s) - (#y)) s

join2'[1] x s = y, if x = z
                where
                  (y, z) = split (#x) s
```

However, an inverse on both inverses is possible too, but for this multiple definitions are possible. That is, if the list `s` is the result of `x ++ y`, the list `s` can be split at, e.g., the beginning, middle or end to yield original values for the lists `x` and `y`. These variants can be defined by

```
join2'[0,1]   s = ([], s)

join2'[0,1]'  s = split ( (#s) / 2) s

join2'[0,1]'' s = (s, [])
```

All three are partial inverse functions of `join2`, since for any lists `x` and `y` it holds that

$$
\begin{array}{lll}
\texttt{join2`[0,1]\ \ \ s} \Rightarrow^* \texttt{(x, y)} & \Longrightarrow & \texttt{join2 x y} \Rightarrow^* \texttt{s} \\
\texttt{join2`[0,1]'\ \ s} \Rightarrow^* \texttt{(x, y)} & \Longrightarrow & \texttt{join2 x y} \Rightarrow^* \texttt{s} \\
\texttt{join2`[0,1]'' s} \Rightarrow^* \texttt{(x, y)} & \Longrightarrow & \texttt{join2 x y} \Rightarrow^* \texttt{s}
\end{array}
$$

However, application patterns that rely on such definitions must be used with great care. Consider the following definition of `mergeSort` that sorts a list.

```
merge :: [*] → [*] → [*]
merge (x:xs) (y:ys) = x : merge xs (y:ys), if x < y
                    = y : merge (x:xs) ys, otherwise
merge  xs     ys    = xs ++ ys

mergeSort :: [*] → [*]
mergeSort []          = []
mergeSort [x]         = [x]
mergeSort (join2' x y) = merge (mergeSort x)
                               (mergeSort y)
```

The last equation of the definition of `mergeSort` contains an application pattern of the `join2'` function. This means that when `mergeSort` is applied to an actual list argument with length two or more, the inverse function ``join2`[0,1]'`` is used to split the argument in two lists of (almost) equal length. However, the reader may verify that if the application pattern would contain either the `join2` or the `join2''` function, use of the `mergeSort` function would lead to an infinite loop for any list with length greater than one. In this case, renaming `join2'` to, e.g., `join2halves` would avoid confusion and ease debugging.

These examples show that the programmer has a great responsibility in the definitions of inverse functions and in the use (and misuse) of application patterns.

In this section applications patterns in expressions was discussed, as well as the caret notation and programmer's responsabilities. With these considerations in mind, in the next section it is shown how application patterns relate to other patterns.

2.5 Application patterns as a generalization

In this section it is shown how application patterns can be seen as a general form of most of the other patterns discussed in Section 1.2. I consider this as mostly of theoretical importance: is is not my intention to really rewrite all patterns by application patterns, not in least because performance may suffer from such a procedure.

2.5.1 List pattern, revisited

A list pattern `x:xs` matches any non-empty list, binding `x` to the head and `xs` to the tail of that list. To avoid confusion, I define the cons operator `:` as a function with a full name:

```
cons :: * → [*] → [*]
cons x xs = x:xs
```

This functions has an inverse on both its arguments that accepts all lists except the empty list.

```
cons‘[0,1] xs = (hd xs, tl xs), if xs ≠ []
```

Thus, the list pattern `x:xs` can also be expressed by the application pattern `cons x xs`[2].

2.5.2 Algebraic pattern, revisited

In any algebraic pattern, the constructor can be regarded as an injective function on all of its arguments.

[2]A list pattern can also be expressed by an algebraic pattern. Since it is shown that algebraic patterns are a special kind of application patterns, this provides another way to show that list patterns can be expressed by an application pattern.

Example. Miranda and Amanda do not support this, but Haskell provides direct support for such a construction. In this approach, the division tree type

```
divTree ::= Lit num | Div divTree divTree
```

defines two functions that have the types

```
Lit :: num → divTree

Div :: divTree → divTree → divTree
```

These functions are injective on all of their arguments, hence the inverses

```
Lit‘[0] :: divTree → num

Div‘[0,1] :: divTree → (divTree, divTree)
```

exist and their definitions follow directly from the type definition of `divTree`. This approach can be used for any algebraic pattern; hence the algebraic pattern can be seen as a special kind of application pattern. Note that the list pattern is also a special case.

2.5.3 Tuple pattern, revisited

A tuple pattern can be expressed by an algebraic pattern, as a tuple type can be expressed by an algebraic type.

Example. The most general 3-tuple type `(*, **, ***)` is expressed by the algebraic type

```
threeTuple * ** *** ::= ThreeTuple * ** ***
```

With this algebraic type, the tuple pattern `(2, x, True)` where `x` is a free identifier is expressed by

```
ThreeTuple 2 x True
```

and it can be matched as any other algebraic pattern.

2.5.4 `n+k` pattern, revisited

With application patterns, an `n+k` pattern can be considered as just an application with the addition function

```
plus x y = x + y
```

For the `plus` function an inverse function on its first argument can be defined.

```
plus‘[0] k n = n - k, if k ≥ 0 ∧ n ≥ k
```

Note that a variant of this pattern that matches any value (positive or negative) and allows for any value of `k` (positive or negative) can be obtained by removing the guard in this definition.

Likewise, an `c*p` pattern can be seen as an application with the multiplication function

```
times x y = x * y
```

for which the inverse on its second argument is defined by

```
times‘[1] c p = p / c, if c > 0 ∧ divRem p c = 0
```

The variant that matches any value ($\neq 0$) can be obtained by replacing the guard by the condition `c` $\neq$ `0`.

2.5.5 Constant pattern, revisited

The constant pattern only checks an actual argument for having a certain constant value. Since a constant can be considered as a constructor without arguments this translates directly to inverse functions. For example, the constants

```
3 :: num
False :: bool
'w' :: char
```

have inverses that informally may be written

```
42‘[] x     = (), if x = 42

False‘[] x = (), if ~x

'w'‘[] x   = (), if x = 'w'
```

Note that these inverses, if they match, only returns the empty tuple `()`, which means that a match will not bind any identifiers.

The previous patterns could be expressed directly by application patterns. The `as` pattern can be expressed indirectly by use of an helper function.

2.5.6 As pattern, revisited

The as pattern allows for binding identifiers multiple times to (parts of) the actual argument. It can be expressed indirectly by using the `theSame` function

```
theSame x y = x, if x = y
```

that has an inverse on both its arguments

```
theSame‘[0,1] x = (x, x)
```

With this definition, the as pattern `x=pat` (in Haskell: `x@pat`) can be expressed by the application pattern `theSame x pat`.

Example. The definition

```
headListTail (list=(x:xs)) = x : list ++ tail
```

is rewritten into

```
headListTail (theSame list (x:xs)) = x : list ++ tail
```

2.5.7 Equivalence pattern, revisited

Equivalence patterns cannot be expressed by application patterns. However, the caret notation does allow for rewriting an equivalence pattern into an identifier pattern and one or more constant patterns. That is, suppose `x` occurs multiple times in a list of patterns. Then every occurance of `x` *except one* can be replaced by the constant pattern `^x` that has the value of `x`. We have seen an example already in Section 2.4.2.

To conclude, application patterns extended with the caret notation can be seen as a general form of all patterns except for the identifier pattern.

This overview showed how existing patterns can be seen as special cases of application patterns. In the next section it is shown that for many standard functions one or more inverse functions can be defined.

2.6 Standard inverse functions

In this section inverse function definitions are given for standard functions. It is shown that for many standard functions one or more inverses exist. All their definitions can be made part of a standard library for inverse functions.

2.6.1 Arithmetic operators

For most arithmetic operators one or two inverses exist. For each function its type, as well as its inverses are given. For clearity I use alphanumeric identifiers but indicate the operator characters between brackets.

The inverse definition for the modulo operator `%` is debatable, as well as that for the `abs` function that is more like a guard that checks for a non-negative argument. As discussed in Section 2.5.4 about the `n+k` and `p*c` pattern, for addition and multiplication the check for a positive argument is language choice. Here I leave these checks out.

```
|| negate [- (prefix)]
|| neg :: num → num
neg‘[0] x = neg x

|| addition [+]
|| plus :: num → num → num
plus‘[0] y s = s - y

plus‘[1] x s = s - x

|| subtraction [- (infix)]
|| minus :: num → num → num
minus‘[0] y s = s + y
minus‘[1] x s = s - x

|| multiplication [*]
|| times :: num → num → num
times‘[0] y s = s / y, if y ≠ 0
              = 0    , if s = 0

times‘[1] x s = s / x, if x ≠ 0
              = 0    , if s = 0

|| division [/]
|| div :: num → num → num
div‘[0] y s = s * y

div‘[1] x s = x / s, if s ≠ 0

|| modulo [%]
|| mod :: num → num → num
mod‘[1] x s = hd fs, if fs ≠ []
            where
              fs = [ i | i <- [(s+1)..]
                   ; (x-s) mod i = 0    ]

|| absolute value
|| abs :: num → num
abs‘[0] x = x, if x ≥ 0
```

```

|| natural logaritm
|| ln :: num → num
ln'[0] x = e ^ x

|| e power
|| exp :: num → num
exp'[0] x = ln x, if x > 0

|| power [^]
|| power :: num → num → num
power'[0] x s = s ^ (1 // x)

power'[1] b s = (ln s) // (ln b)
```

2.6.2 Goniometric functions

Inverses for the goniometric functions are easily defined. As the arcsin and arccos functions may not be available (as is the case in Amanda), they are expressed using the arctan function.

```
|| sine
|| sin :: num → num
sin'[0] x = atan (x // (1-x^2)^0.5 ), if abs x ≤ 1

|| cosine
|| cos :: num → num
cos'[0] x = atan ( (1-x^2)^0.5 // x), if abs x ≤ 1

|| tangent
|| tan :: num → num
tan'[0] x = atan x

|| inverse sine
|| arcsin :: num → num
arcsin'[0] x = sin x, if abs x ≤ pi//2

|| inverse cosine
|| arccos :: num → num
arccos'[0] x = cos x, if abs x ≤ pi//2

|| inverse tangent
|| tan :: num → num
arctan'[0] x = tan x, if remainder x pi ≠ pi//2
```

2.6.3 Numerical functions

The prime, fibonacci and factorial functions are well-known examples of functions that calculate the n-th number that adheres to some condition. They can be defined in an easily-readable (but not necesarily efficient) way by

```
prime :: num → num → num
prime n = (sieve [2..])!n
        where
          sieve (p:x) = p : sieve [n | n <- x; n mod p
              > 0]

fib :: num → num → num
fib  0    = 1
fib  1    = 1
fib (n+2) = fibonacci n + fibonacci (n+1)

fac :: num → num
fac 0 = 1
fac n = n * fac (n-1)
```

These function have in common that, from a certain value on, they all increase strictly monotonically. Thus, to define the inverse `prime‘[0]` applied to some argument `y`, we can try for an increasing value `x` whether `prime x` evaluates to `y`. Now only a suitable starting value for `x` (`0` would be fine), as well as an halting condition (for when we know that the value for `x` has grown too large) have to be defined. The same approach can be used to define the inverse functions `fib‘[0]` and `fac‘[1]`.

The helper definition `invGenTest` provides the desired functionality. It takes as arguments the original function (for which the inverse is required), a successor function that increases the starting value, a testing function that indicates whether the starting value has grown to large, and an initial value `x`. It returns a function that takes an image `y` and returns `Just x` if there is some value of `x` for which `y` is the image, or `Nothing` is no such value for `x` exists. The use of the `maybe` type is to allow the inverse function to be a partial definition, so that its implicit in a pattern allows for refutability.

```
invGenTest :: (*→**) → (*→*) → (*→**→bool) → *
    → ** → maybe *
invGenTest f next stop x y
 = Just x, if f x = y
 = invGenTest f next stop (next x) y, if ~(stop x y)
```

Now inverses for the prime, fibonacci and factorial functions are defined by

```
prime‘[0] y = fromJust valMb, if valMb ≠ Nothing
              where
                valMb = invGenTest prime (+1) (>) 0 y

```

```
fib‘[0] y = fromJust valMb, if valMb ≠ Nothing
            where
              valMb = invGenTest fib (+1) (>) 1 y

fac‘[0] y = fromJust valMb, if valMb ≠ Nothing
            where
              valMb = invGenTest fac (+1) facStop 1 y
              facStop x y = x-1 > log y
```

Note that, since `fib 0` = `fib 1` and `fac 0` = `fac 1`, the starting values for `x` starts at `1` since the fibonacci and factorial functions only increase strictly monotonically from `1` on. Note further that the factorial function has a rather efficient testing functions that increases `x` faster than the successor function `+1`.

Finally it must be remarked that other definitions for these inverses may be more efficient.

Example. The fibonacci numbers have the property that

$$\mathit{fib}\ n \quad = \quad \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$$

where

$$\phi = \frac{\sqrt{5}+1}{2}, \qquad \hat{\phi} = \frac{\sqrt{5}-1}{2}$$

as is easily shown by induction. Since *fib* n approaches $\phi^n/\sqrt{5}$ asymptotically, a more efficient definition for the inverse fibonacci function is

```
fib‘[0]’ 1 = 1
fib‘[0]’ n = val, if n > 1 ∧ fib val = n
           where
             val = ceiling (log n // log phi)
                 where
                   phi = (1+5^0.5)//2
```

Thus, use of the `invGenTest` function in inverse function definitions may be easy, but sometimes more efficient definitions exist.

2.6.4 List manipulation

For many list manipulation functions inverse functions can be defined, most of them being straightforward.

Note that for the `join3‘[1]` definition—that joins three lists—the first occurence of the separating string is choosen. That is, if the lists `^y` and `^y’` are known while for certain lists it holds that

`join3 x ^y z = s` *and* `join3 x’ ^y’ z’ = s` then matching `join3 p ^y r` against an actual argument will bind `p` to the shortest list in `x` and `x'` (and `q` to the longest list in `z` and `z'`). In other words, `join3` is non-greedy. As we will see in Section 2.7.2, this decision allows for simple string parsing

```
|| reverse a list
|| reverse :: [*] → [*]
reverse`[0] x = reverse x

|| constitute a list from a head and a tail list
|| cons :: * → [*] → [*]
cons`[0] xs = (hd x, tl x), if x ≠ []

|| join two lists [++] (a.k.a. join2)
|| join :: [*] → [*] → [*]
join`[0] y s = x, if y = z
            where
              (x, z) = split ((#s) - (#y)) s

join`[1] x s = y, if x = z
            where
              (y, z) = split (#x) s

|| join three lists
|| join3 :: [*] → [*] → [*] → [*]
join3`[1] x z s = y, if x=x' ∧ z=z'
               where
                 (x', yz) = split (#x) s
                 (y, z')  = split (#ys - #z) ys

join3`[0,2] y s = hd xzs, if xzs ≠ []
               where
                 xzs = [ (x, z)
                       | i <- [0..(#s - #y - 1)]
                       ; (x, yz) = split i s
                       ; (y', z) = split (#y) yz
                       ; y = y'
                       ]
|| zip a pair of lists
|| zip :: ([*], [**]) → [(*, **)]
zip`[0] :: [(*, **)] → ([*], [**])
zip`[0] ((x,y) : xys) = (x:xs, y:ys)
                     where
                       (xs, ys) = zip^-1 xys
zip`[0] []            = ([], [])

|| zip two lists
|| zip2 :: [*] → [**] → [(*, **)]
zip2`[0,1] xs = zip`[0] xs
```

```

|| take the first n elements
|| take :: num → [*] → [*]
take‘[0] x s = len, if take len s = x
             where
               len = #x

|| drop the first n elements
|| drop :: num → [*] → [*]
drop‘[0] y s = len, if drop len s = y
             where
               len = #y

|| splits a list at a certain index
|| split :: num → [*] → ([*], [**])
split‘[0,1] (xs, ys) = (#xs, xs ++ ys)

|| get (n+1)-th element from list [!]
|| index :: [*] → num → *
index‘[1] x s = hd is, if is ≠ []
              where
                is = [ i | i <- [0..(#s-1)]; s!i = x]

|| repeat an element n times
|| rep :: num → * → [*]
rep‘[0,1] :: [*] → (num, *)
rep‘[0,1] [x] = (1, x)
rep‘[0,1] (x : rep n y) = (n+1, x), if x=y
```

2.6.5 Conversion functions

Conversion functions mainly transform values from one type to another. Since this transformation is already defined in both directions, the inverse definitions are rather trivial. Note that the definition of `lines‘[0]` is recursive and rather symmetrical with that of `unlines‘[0]`.

```
|| transform string to integer
|| atoi :: [char] → num
atoi‘[0] s = itoa s

|| transform integer to string
|| itoa :: num → [char]
atoi‘[0] s = atoi s, if filter (member "-.0123456789")
    s = s!\\
                         ∧ count ’.’ s ≤ 1!\\
                         ∧ count ’-’ (tl s) = 0!\\
           where count x = (#) . filter (=x)!\\

```

```
|| get character ascii code
|| code :: char → num
code‘[0] s = decode s, if 0 ≤ s ∧ s ≤ 255

|| get character with certain ascii code
|| decode :: num → char
decode‘[0] s = code s

|| splits a string based on newline characters
|| lines :: [char] → [[char]]
lines‘[0] (join3 x "\n" (lines xs)) = x:xs
lines‘[0] x                         = [x]

|| joins a list of lists, adding newline characters
|| unlines :: [[char]] → [char]
unlines‘[0] (x:xs) = join3 x "\n" (join3 xs)
unlines‘[0] []     = []
```

In brief, for many standard functions an inverse can be defined.

2.7 The use of application patterns

In this section a brief overview of the use of application patterns is given. Besides the theoretical importance that application patterns are a general form (Section 2.5) they also yield practical implications.

2.7.1 More readible definitions

Using the application pattern, function definitions may become more readible. That is, if in a function definition first some trivial operation must be performed on an argument, this operation can be placed in the right hand side of the definition.

Example. In the following definitions, first a simple operation is applied to an actual argument.

```
f (sin alpha) = ... alpha ...
g (2*n)      = ... n ...
h (itoa s)    = ... s ...
k (ln x)      = ... x ...
```

For example, if `f` is applied to an actual argument *a*, the function `sin‘[0]` is applied to *a* and the result bound to `alpha`. Note that `f` is only defined for values between `-1` and `1`, inclusive. Likewise, the functions `g`, `h` and `k` use, when applied to an actual argument, the inverse functions `*‘[1]`, `itoa‘[0]` and `ln‘[0]` in order to bind `n`, `s` and `x`, respectively.

Example. Since nested patterns are possible too, the definition like

```
f x = ... p ..., if abs x ≤ 1
     where
       p = 5 - arcsin x
```

can be written more readible as

```
f (sin (5-p)) = ... p ...
```

Example. Another example of a better readible function definition example is the `upperLeft` function

```
upperLeft :: [(num, num)] → (num, num)
upperLeft (zip (xs, ys)) = (min xs, max ys)
```

2.7.2 Simple string parsing

A nice application for application patterns is simple string parsing. Suppose that, from a long input string `s`, some substrings must be bound to the identifiers $x_1, \ldots, x_k$ which are seperated by known substrings $c_1, \ldots, c_{k-1}$. Then these identifiers can be bound by matching the pattern

$$\texttt{join3}\ x_1\ c_1\ (\texttt{join3}\ x_2\ c_2\ (\ldots\ (\texttt{join3}\ x_{k-1}\ c_{k-1}\ x_k)\ldots))$$

against the input string `s`, since during evaluation the inverse `join3‘[0,2]` is used to bind the x_*'s. If `s` starts or ends with a known substring, then the beginning or end can be matched by the use of the `join2` function.

Example. Consider the function `vec3length` that parses the length of a string that represents a three-dimensional vector `(x,y,z)`. For any other string it returns zero. Informally it would be defined by

```
vec3length ( "(" ++ (itoa x) ++ "," ++ (itoa y) ++ ","
     ++ (itoa z) ++ "")" )
 = (x^2 + y^2 + z^2) ^ 0.5

vec3length _ = 0
```

Now clearly the use of `++` will not work, but this definition can automatically be rewritten into

```
vec3length (join2 "("
                  ( join3 (itoa x)
                          ","
                          ( join3 (itoa y)
                                  ","
                                  ( join2 z ")"  ))))
 = (x^2 + y^2 + z^2) ^ 0.5

vec3length _ 0
```

where the `join2‘[1]`, `join3‘[0,2]` and `join2‘[0]` inverse functions are used when `vec3length` is applied to an actual argument. With this definition, the length of the vector represented by the string `"(2,3,6)"` evaluates to `7`, whereas a bad-formed string evaluates to `0`.

```
vec3length "(2,3,6)"
 ⇒ 7

veclength "not a vector"
 ⇒ 0
```

Thus, such constructions may allow for better readible and easier comprehendible definitions.

Besides these considerations, inverse function definitions allow for indicating that different functions (the function itself and its inverses) are semantically related. In addition, the caret notation allows for some extra expressive power in patterns. It must be noted, though, that application pattern syntax and semantics may take some time to get used to for the programmer.

2.8 Conclusion

This chapter described application patterns. The concept of generalized inverses was introduced, as well as a new caret notation that adds expressive power to patterns. Application patterns together with the caret notation

Chapter 3

The Application Pattern Compiler

3.1 Introduction

In the previous chapter application patterns were discussed. This chapter will discuss rewriting application patterns into semantically equivalent runnable code.

In Section 3.2 this rewriting it is made intuitive to the reader by discussing some examples. In Section 3.3 a general rewriting algoritm for application pattern is sketched. This sketch can be seen as both providing an implementation of rewriting the rewriting algoritm *and* providing the semantics of application patterns. In Section 3.4 the implementation of this rewriting algoritm is discussed.

3.2 Intuitively rewriting application patterns

In this section I describe intuitively how an application pattern can be rewritten. In order to describe all features of the rewriting algoritm, I will take the same approach as in Chapter 1. First rewriting of the basic application pattern is described. Then refutability is added by using the `maybe` type. Finally generalized inverse functions are discussed.

3.2.1 Basic pattern

This section describes rewriting a basic pattern `f x`. Recall the `upperLeft` example from Section 2.2.1.

```
upperLeft :: [(num, num)] → (num, num)
upperLeft (zip (xs, ys)) = (min xs, max ys)
```

How would one rewrite this definition without application patterns? The idea is that during evaluation of `upperLeft` a, where a is an actual argument, the *equation*

$$\texttt{zip (xs, ys)} = a$$

is solved for `xs` and `ys` by applying the inverse function `zip‘[0]` to both sides of this equation:

$$\texttt{(xs, ys)} = \texttt{zip‘[0]}\ a$$

Now the solution proposed here for rewriting the definition of `upperLeft` is to introduce a new (free) identifier, say `var`, that takes the role of the actual argument. The argument `(xs, ys)` in the application pattern is then bound in a where clause:

```
upperLeft' :: [(num, num)] → (num, num)
upperLeft' var = (min xs, max ys)
                where
                  (xs, ys) = zip‘[0] var
```

3.2.2 Adding refutability

The basic rewriting procedure in the previous section does not take into account that an application pattern may be refutable. This section describes rewriting a basic pattern `f x` where `f` may be partially defined (yielding refutability). Consider the definition

```
h (sin a) = a * a
h x       = x - 2
```

This definition uses that the inverse sine function is partially defined, namely only for values in $[-1, 1]$:

```
sin‘[0] x = arcsin x, if abs x ≤ 1
```

However, the evaluation of say `sin‘[0] 2` produces a runtime exception. In order to avoid such exceptions, first the definition of `sin‘[0]` is rewritten (indicated by the `_Mb` suffix) so that it returns a value of the `maybe` type:

```
sin_Mb‘[0] x = Just arcsin x, if abs x ≤1
             = Nothing , otherwise
```

This rewriting can be performed automatically, as is shown further in this chapter. To apply the `sin_Mb‘[0]` function to an argument `var`, first it must be checked that its value is not `Nothing`. If its value is not `Nothing` (that is, the inverse function is defined for the argument `var`), it is `Just` *`something`* where this *`something`* is the desired result.

Thus, to rewrite the first equation in the definition of `h` two variables are introduced: a `var` variable that replaces the pattern `sin a`, and a `match` variable that checks whether the inverse function `sin_Mb‘[0]` is defined for the value of `var`. This results in

```
h' var   = a * a, if match  = Nothing
           where
             match = sin_Mb[0] var
             Just var = match

h' x       = x - 2
```

As the first equation now contains an identifier pattern, the second equation has become unreachable. Thus the two equations must be merged, where care must be taken that the arguments of `h` match. Here the occurence `x` in the second equation can be renamed to `var`, resulting in

```
h' var = a * a, if match ≠ Nothing
           where
             match = sin_Mb[0] var
             Just var = match

h' var  = var - 2, otherwise
```

3.2.3 Adding refutability

With refutability added in the previous section, now it is time to describe rewriting application patterns with multiple arguments. This section describes rewriting a an application pattern `f` $\mathtt{x}_0$ `...` $\mathtt{x}_{n-1}$ where `f` may be partially defined. Consider the definition

```
firstAndThird (join5  x  ^hyph  "_"  "-"  z)
 =  "[" ++ x ++ "|" ++ z ++ "]"

hyph = "_"
```

where the application pattern `join5 x ^hyph "_" "-" z` contains five arguments. The second argument `^hyph`, the third argument `"_"` and the fifth argument are *known*, i.e. constant or retrieved from the context. The first argument `x` and the fifth argument `z` must be bound.

Now suppose the `join5` function joins five lists

```
join5 a b c d e = concat [a, b, c, d, e]
```

and has an inverse function defined with respect to its first, third and fifth argument

```
join5'[0,2,4] :: [*] → [*] → [*] → ([*],[*],[*])
join5'[0,2,4] b d s = ...
```

that can be automatically be rewritten into a definition that returns the `maybe` type

```
join5_Mb‘[0,2,4] :: [*] → [*] → [*] → maybe ([*],[*],[*])
join5_Mb‘[0,2,4] b d s = ...
```

Rewriting as definition $*$ is informally specified by

$$
\begin{array}{lcl}
\texttt{f‘[}a_1\texttt{,}\cdots\texttt{,}a_k\texttt{]}x_1\ \cdots\ x_m & & \texttt{f\_Mb‘[}a_1\texttt{,}\cdots\texttt{,}a_k\texttt{]}x_1\ \cdots\ x_m \\
= v_1\texttt{, if } g_1 & & = \texttt{Just}v_1\texttt{, if}g_1 \\
\vdots \quad\quad \vdots & \stackrel{*}{\Rightarrow} & \vdots \quad\quad \vdots \\
= v_n\texttt{, if } g_n & & = \texttt{Just}v_n\texttt{, if}g_n \\
 & & = \texttt{Nothing, otherwise}
\end{array}
$$

and the resulting definition can be added to the existing definitions.

Clearly we must use this inverse function so solve for the arguments `x` and `z` in the application pattern. The first step is the same as in the previous section: a new identifier `var` replaces the application pattern:

```
firstAndThird' var
 =  "[" ++ x ++ "|" ++ z ++ "]"
```

Note that the arguments in the application pattern `join5 x ^hyph "_" "-" z ")"` can be partitioned into three lists,

1. the arguments that are *both* required by `join5_Mb‘[0,2,4]` *and* known: `^hyph` and `"-"`;
2. the arguments that are provided by `join5_Mb‘[0,2,4]` and that must be bound: `x` and `z`; and
3. the arguments that are provided by `join5_Mb‘[0,2,4]` but are already known: `"_"`. These arguments must be checked for their value.

Now the inverse `join5_Mb‘[0,2,4]` is applied to the arguments from the first set, together with a fresh identifier that replaces the whole application patterns. The result is again bound to a fresh matching identifier `match`

```
firstAndThird' var
 =  "[" ++ x ++ "|" ++ z ++ "]", if match ~= Nothing
 where
    match = join5_Mb‘[0,2,4] ^hyph "-" var
```

The arguments in the other two lists are bound using the result. For members of the third set, arguments that are provided by `join5_Mb‘[0,2,4]` but are already known: `"_"`, new identifiers are introduced that are checked for the right value by adding a guard to the definition. Here a new identifier `var1` is introduced that is checked for being equal to the third argument `"_"`.

```
firstAndThird' var
 =  "[" ++ x ++ "|" ++ z ++ "]", if match ≠ Nothing
                                   ∧ var1 = "_"
 where
   match = join5_Mb'[0,2,4] ^hyph "-" var
   Just (x, var1, z) = match
```

(Note: in the next section an algoritm is presented that treats elements in the last two lists alike).

Finally one issue must be resolved: what if during rewriting an application pattern one can choose between multiple inverse functions that all provide the necessary arguments (and possibly others as well)? Some options are

Choose the first inverse that is defined. This solution is most easily implemented.

Choose the smallest set of arguments that is provided by the inverse.

Choose the largest set of arguments that is provided by the inverse.

Choose a random defined inverse.

Either of the ones above repeatedly by allowing one inverse to fail (because it is not defined) and then try another one.

The different options may yield different evalution results, for instance when one inverse is defined for a smaller domain than another one. The pros and cons of each of these options are subject to further research.

In this section an intuitive overview was given on rewriting single application patterns. In the next section a general rewriting algoritm is discussed that rewrites all kinds of patterns that may be nested and overlap.

3.3 A rewriting algoritm

This section sketches rewriting algoritm that can be used for rewriting code with application patterns into code without application patterns.

3.3.1 Overview

Peyton Jones (1987) describes a compiler that translates function definitions with pattern matching into case-expressions that can be efficiently evaluated. This is not the approach of the algoritm described here, as this algoritm translates function definitions with pattern matching *that may include application patterns* into function definitions with pattern matching *that does not include application patterns*. In addition, the algoritm described here does not incorporate optimizations that are described by Peyton Jones such as a per-column rewriting.

The compiler described by Peyton Jones provides support for

- Overlapping patterns

- Nested patterns
- Constant patterns
- Multiple arguments
- Non-exhaustive sets of equations
- Conditional equations
- Repeated variables

The rewriting algoritm for application patterns, as described here, provides support for all these constructions too. For ordinary (i.e., non-application) patterns the algoritm translates pattern correctly, that is in accordance with the semantics of pattern matching as described by Peyton Jones (1987). Therefore, one may consider application patterns as an extension to current pattern matching whose semantics are defined by the rewriting algoritm.

3.3.2 Rewriting patterns

In essence, the algoritm allows for rewriting all kinds of patterns into simpler patterns. Patterns in the resulting code are either identifier patterns, nested tuple patterns or non-nested algebraic patterns. In this section rewriting all kinds of *patterns* is described (except the ones that can be expressed by application patterns as described in Section 2.5), whereas the next section will cover rewriting function definitions. One may wonder why rewrite not just the application patterns and leave the other patterns as they are. However, since application patterns and all other patterns can occur nested, to avoid runtime exceptions all patterns must be rewritten.

In the previous section application patterns where rewritten by adding guards and `where` clauses to function definitions; the rewriting algoritm is based on the same principle. The notation

$$\begin{aligned}&\mathit{Rewr}[\![\ \mathtt{e}\ ]\!]\\ &=\ \mathtt{e'}\ \lhd \mathsf{guard}^{+} \leftarrow \mathtt{g}\\ &\qquad\ \lhd \mathsf{wheres}^{+} \leftarrow \mathtt{ws}\end{aligned}$$

means that the algoritm rewrites the pattern `e` is into `e'`, with the guard `g` and the `where` clauses ws added to the definition in which the expression occurs.

The predicate $\mathsf{isKnown}(\cdot)$ is used to indicate that an expression is *known* (in its context), i.e. that it contains no free identifiers.

It is assumed that all identifiers that are bound in patterns are *different* (but the caret notation allows for mimicking the equivalence pattern: see Section 2.5.7). For application patterns it is assumed that the caret prefix is used in the function name.

Constant

A constant is replaced by a new identifier whose value is compared to the constant.

Rewr ⟦ `c` ⟧
= `var` ◁ guard$^+$ ⇠ `var = c`

- isKnown(`c`)
- `var` is a new identifier

Example. The definition

```
fac (1 + 2 + 3 + 4 - 10) = 1
```

is rewritten into

```
fac var = 1, if var = 1 + 2 + 3 + 4 - 10
```

Identifier

Rewriting the identifier pattern is trivial, as it is the identifier itself.

Rewr ⟦ `i` ⟧
`= i`

- `i` is an identifier

Function application

For an application pattern the inverse is applied to its required arguments. The arguments provided by the inverse are (in their rewritten form) bound to the result.

$$
\begin{array}{l}
\mathit{Rewr}[\![\ \texttt{\^{}f}\ \texttt{x}_0\ \ldots\ \texttt{x}_{n-1}\ ]\!] \\
\quad = \texttt{var}\ \lhd \textsf{guard}^{+} \leftarrow \texttt{match} \neq \texttt{Nothing} \\
\qquad\qquad \lhd \textsf{wheres}^{+} \leftarrow \texttt{match} = \texttt{f\_Mb`[}\ i_1,\ \ldots,\ i_k\ \texttt{]}\ \texttt{x}_{j_1}\ \ldots\ \texttt{x}_{j_m} \\
\qquad\qquad\qquad\qquad \texttt{Just}\ (\ \mathit{Rewr}[\![\ \texttt{x}_{i_1}\ ]\!],\ \ldots,\ \mathit{Rewr}[\![\ \texttt{x}_{i_k}\ ]\!]\ ) \\
\qquad\qquad\qquad\qquad\quad = \texttt{match}
\end{array}
$$

- $\texttt{x}_*$'s are patterns
- $\forall p \in \{1, \ldots, j\}$: isKnown($\texttt{x}_p$)
- The inverse `f‘[`i_1`, ...,`i_k`]` is defined for m arguments[a]
- The indices of provided arguments i_* and indices of required arguments j_* partition the list of all argument indices[b]:

  $$[i_1, \ldots, i_k] \cup [j_1, \ldots, j_m] = [0, \ldots, n-1]$$

  where the $\cup$ operator merges two ordered lists into a new ordered list.
- `var` and `match` are new identifiers

[a] In Section 3.2.3 it is shown how the inverse `f_Mb‘[`i_1`, ...,`i_k`]` can be derived automatically

[b] It If multiple combinations of lists fulfill this condition, multiple strategies are possible to pick one; see Section 3.2.3

This scheme convers algebraic values, in which constructors yield an inverse on all their arguments so that $k = n$, $m = 0$ and $i_p = i - 1$ (see also Section 2.5.2). This scheme also applies to tuples, lists and constants because they can be represented by algebraic values.

Example. The definition

```
gcd x    (^x + y) = gcd x y
```

contains an application pattern `^x + y` of the addition function (fully written `plus`), for which the first argument is known and the second to be bound. It is rewritten into

```
gcd x    var
 = gcd x y, if match ≠ Nothing
 where
   match = plus_Mb‘[1] ^x var
   Just y = match
```

where it is assumed that the inverse function `plus‘[1]` is defined.

3.3.3 Special cases in rewriting

It was shown in Section 2.5 how algebraic, tuple, list and constant patterns are special cases of an application pattern. This means that they can be rewritten using the schemas given in the previous section. However, using these schemas may not yield the most readible code. In this section alternative rewriting schemas are given that are more easily readible and implementable. The reader may verify that the schemas given agree with the more general forms described previously.

Constructor

For a constructor first the type of the constructor is checked. Then its arguments (in their rewritten form) are bound to the actual argument using the constructor.

$$\begin{array}{l}
\mathit{Rewr}[\![\ \texttt{C}\ \ \texttt{x}_0\ \ \texttt{...}\ \ \texttt{x}_{n-1}\ ]\!] \\
\quad \texttt{= var}\ \lhd \mathsf{guard}^+ \leftarrow\texttt{--isConstr var} \\
\qquad\qquad \lhd \mathsf{wheres}^+ \leftarrow\texttt{--isConstr (C}\ \underbrace{\texttt{_ ... _}}_{n}\texttt{) = True} \\
\qquad\qquad\qquad\qquad \texttt{isConstr _ = False} \\
\qquad\qquad\qquad\qquad \texttt{C}\ \ \mathit{Rewr}[\![\ \texttt{x}_{i_1}\ ]\!]\ \ \texttt{...}\ \ \mathit{Rewr}[\![\ \texttt{x}_{i_{n-1}}\ ]\!]\ \ \texttt{= var}
\end{array}$$

- `c` is a constructor
- $\texttt{x}_*$'s are patterns
- `var` and `isConstr` are new identifiers

Example. The definition

```
sumTree (Node x y) = sumTree x + sumTree y
```

is rewritten into

```
sumTree var
 = sumTree x + sumTree y, if isConstr var
 where
   isConstr (Node _ _) = True
   isConstr _          = False
   Constr x y = var
```

Note that a constant can be seen as a constructor without arguments. This is also in agreement with the rewriting scheme for rewriting a function application patterns with $k = m = 0$.

Tuple

For a tuple the arguments (in their rewritten form) can be rewritten directly, since a tuple pattern is non-refutable.

$Rewr[\![$ `(x`$_0$`, ..., x`$_{n-1}$`)` $]\!]$
`=` `(` $Rewr[\![$ `x`$_{i_1}$ $]\!]$ `, ...,` $Rewr[\![$ `x`$_{i_{n-1}}$ $]\!]$ `)`

- `x`$_*$'s are patterns

Identifier bound in the context

An identifier bound in the context is treated like a constant pattern. This is a special case of rewriting a constant because it must hold that isKnown(`^i`) in the context of `i`.

$Rewr[\![$ `^i` $]\!]$
`= var` $\lhd$ guard$^+$ ←`--var = i`

- `i` is an identifier that is bound in the context.
- `var` is a new identifier

As described earlier, in a left hand side of a function definition the context also includes identifiers that are bound in patterns.

Example. The definition

```
gcd ^x x = x
```

is rewritten into

```
gcd var x = x, if var = x
```

3.3.4 Rewriting definitions

In the previous section it was described how patterns could be rewritten into equivalent patterns without application patterns, where some extra guards and `where` clauses might me added to the function definition.

For adding these guards and `where` clauses to a definition, either of two cases hold:

- The definition's left hand side consists of an ordinary function with one or more (rewritten into identifier) patterns as arguments.

  The extra guards are added to each of the existing guards. Note that the order is important: first the extra guards must be tested, in the left-to-right, outside-in order corresponding to the rewritten pattern, followed by the existing guards (there is one exception: as shown in Section 2.4.2 the caret notation allows for situations in which patterns are matched in a different order).

The extra `where` clauses are added to the list of `where` clauses in the function definition.

- The definition's left hand side is itself a pattern in a `where` clause.

  The extra guards are ignored, because is is assumed that they match. Note that improper use may yield a runtime exception, just like for any other pattern used in a `where` clause

  The extra `where` clauses are added to the list of `where` clauses the pattern is part of.

All rewriting examples given so far are examples to the first case. An example of the second case is

```
p ys = x + #xs
     where
       (x:xs) = ys
```

that is rewritten into

```
p ys = x + #xs
     where
       var
        = ys
       match
        = inv_cons_Mb_0_1 var
       Just ((x, xs))
        = match
```

Note that the pattern `x:xs` is replaced by the identifier `var`. `x` and `xs` are bound in the same `where` clause as where `var` is bound. There is no guard that checks whether `match` equals `Nothing`. This means that `p []` would yield a runtime exception, just like the original definition.

Intuitively, the algoritm described above agrees with the semantics of pattern matching as described by Peyton Jones (1987). A proof for this falls outside the scope of this thesis.

For functions with more than one equation it may be the case that non-identifier patterns are rewritten into identifier patterns. This can make other equations unreachable. Therefore the equations must be merged , which requires that the identifiers in the function's arguments match. This can be achieved by renaming identifiers or binding them in a `where` clause.

3.3.5 Conclusion

In this section an algoritm was sketched for rewriting application patterns. This algoritm provides support for

Overlapping patterns as multiple guards may evaluate to `True`

Nested patterns as patterns are rewritten recursively

Constant patterns as a transformation for this case is defined

Multiple arguments even within application patterns, by rewriting the arguments one by one

Non-exhaustive sets of equations which may cause all guards to evaluate to `False`

Conditional equations by the use of guards

Repeated variables supported by the use of the `caret` notation

The next section describes the implementation of this algoritm.

3.4 The application pattern compiler

This section describes the requirements, design and implementation of the application pattern compiler. This compiler rewrites definitions with application patterns into code witout application patterns according to the algoritm sketched in the previous sections.

3.4.1 Requirements

This section discusses the requirements for the pattern match compiler.

The ultimate goal of the pattern match compiler is to provide the ability to execute code with application patterns. A working implementation would show conclusivily that application patterns are implementable. Furthermore it would allow for more programs written with application patterns, so that it can be tested in real-world applications. In brief, the primary goal of the pattern match compiler is to provide a *proof-of-concept*. The requirements are summarized by

Executable code. The pattern match compiler should allow for running programs that are specified using application patterns.

Substantial language. It should support a sufficient powerful functional language. Not all features have to be working, but all basic functionality should be available

Performance is not an issue. This is a proof-of-concept, thus performance is not an important factor. Using the compiler should just not take too long.

Not fool proof. Likewise, the compiler has not to be foolproof. It should at least be able to use properly written code.

With these requirements in mind, the design is discussed in the next section.

3.4.2 Design

To fullfill the requirements specified in the previous section, it is sufficient to implement the algoritm described in Section 3.3. This algoritm must be provided with properly parsed input. The result must be written into interpretable source code. In addition, proper use of the algoritm requires some pre- and postprocessing.

The output of the compiler, in the form of source code, must be interpreted by an existing interpreter. A drawback is that debugging may be more cumbersome, since if the original code contains errors this may result in an error during rewriting or when the rewritten code is loaded into the interpreter. However, since the compiler is only a proof-of-concept this is not a great advantage.

3.4.3 Implementation language

The chosen implementation language is Amanda, which supports all common functional programming features. As input language I defined a AMANDA-corelanguage, which is a substantial subset of the Amanda language providing support for the following features:

- Constant expressions of type `num`, `char` and `bool`).
- Compound expressions: lists, tuples, algebraic values.
- Constant, list, tuple, algebraic and identifier patterns.
- Identifiers and function applications.
- Prefix and infix operator expressions with priorities.
- Function definitions with multiple equations, clauses and (nested) `where` clauses.

However, there is no support for:

- Record, lambda and list comprehension expressions
- As patterns or equivalence patterns (without the caret notation)
- Type denotations or definitions
- Interpreter directives such as import statements

To this language support for caret notation together with application patterns are added, resulting in AMANDA$^{\mathcal{AP}}$. In this language it is required that inverse functions are defined at the top level with all its arguments mentioned explicitly[1]. In addition, inverse functions for operators must use predefined full names.

The AMANDA$^{\mathcal{AP}}$ language is definitely powerful enough to meet the requirements in the previous section. In 3.4.6 a sketch is given how record, lambda and list comprehensions could be implemented, but the actual implementation is beyond the scope of this proof-of-concept prototype.

[1]The reason is that the compiler cannot deduce types: it just counts the number of arguments to deduce the total number of arguments of the function it is the inverse of

3.4.4 Implemenation

The complete transformation is divided in a number of steps.

Lexer. The lexer splits the input file into a list of elementary items and attaches a label to them. In this case, the lexer also keeps tracks of indentation (the *offside rule*).

Parser. Based on the lexer output, the parser recognizes structure in sequences of lexed items and represents this structure using an algebraic datatype. After this step, infix and prefix operators are written by their full names.

Add carets. As sometimes the use of the caret `^` prefix is optional, these carets are added to both nested function applications and operator function applications in left hand sides of definitions.

Syntactic rewrite `++`. An extra feature is that patterns with the `++` operator are automatically rewritten into application patterns with the `join2` and `join3` functions (see Section 2.7.2). This rewriting is specific to the `++` operator.

Rename identifiers. In order to avoid problems with identifiers that are redefined at multiple levels (such as in nested where clauses), all identifiers except the function names at the top level are renamed to a new unique name.

Rewrite application patterns. This is the step that actually performes the rewriting described by the algoritm presented in Section 3.3. The rewriting schemes from Section 3.3 are used almost literally.

Merge equations. After rewriting some patterns may be lost, so that equations in function definitions become unreachable. These equations are merged.

Add `maybe` type for inverse functions. Besides the transformation itself, definitions for the inverse functions must be added (Section 3.2.3) so that they return the `maybe` type.

Create legal identifiers. Since identifiers with the backtic and caret notation are illegal identifiers in AMANDA-core, these are systematically renamed into legal identifiers.

Pretty print code. Finally the resulting definitions are printed as more or less human readible source code.

The lexer and parser are inspired by the grammar reported by Papegaaij (2005). For the lexer, parser, rename identifiers and rewrite application patterns, I made extensive use of monad constructions as described by Wadler (1992).

Together, these steps perform the complete transformation from AMANDA$^{\mathcal{AP}}$ to AMANDA-core. The implementation of each step is described in Appendix A.

3.4.5 Results

The application pattern compilers works in rewriting code with application patterns. The only problem is that Amanda crashes for larger input files due to an unexplainable but reproducable memory problem. An example of input and output is provided in Appendix A.3. The output is accepted by Amanda (if the proper algebraic type definitions are added manually), the functions can be run and produce the expected results. This shows that application patterns can indeed be implemented and used in functional languages.

3.4.6 Extensions

The application pattern compiler provides no support for partial records, lambda abstractions and list comprehensions. In Amanda all these expressions may contain patterns. In this section I give a sketch how support for such patterns may be provided.

Partial records

Partial records whose type definition contains n fields can be written by an n-tuple with elements of the `maybe`type. Matching against fields in such a record is matching against the expressions `Just ...` and `Nothing`.

Lambda abstractions

In amanda a lambda abstraction takes the form

```
(pat₁ → x₁ | pat₂ → x₂ | ... | patₖ → xₖ)
```

where the `pat`$_*$'s are patterns that are matched against an actual argument. The pipes `|` seperate alternatives (thus allowing for case expressions).

Such an expression can be replaced by a free identifier, say `f`, that is defined by

```
f pat₁ = x₁
f pat₂ = x₂
...
f patₖ = xₖ
```

Clearly this approach can also be used in the case of multiple arguments.

3.4.7 List comprehensions

List comprehensions are easily rewritten using the *list monad*. This monads consists of the definitions of the *unit* and *bind* operators

◇ `monadLst.ama` ◇

```
unitLst x = [x]

bindLst (x:xs) f = f x : bindLst xs f
bindLst []     f = []
```

Now a list comprehension is easily rewritten by the use of these operators and lambda abstractions.

Example. As an example, consider the lambda abstraction

```
[ f x z
| sin p, qs) <- xs
; y <- qs
; check p y
; z + 10 <- zs ]
```

where the lists `xs` and `zs` are bound in the context. The patterns are underlined in this definition. First the guard `check p y` can be rewritten into the generator `_ <- if (check p y)[[]] []`. Now the lambda abstraction is rewritten using the monad operators into

```
  concat (xs             $bindLst ( (sin p, qs) →
  concat (qs             $bindLst ( y           →
  concat (if (check p y) [[]] []
                         $bindLst ( _           →
  concat (zs             $bindLst ( (z+10)      →
  unitLst (f x z)
                                  | _ → [] ))
                                  | _ → [] ))
                                  | _ → [] ))
                                  | _ → [] ))
```

Thus, in every generator the pattern is matched against elements in the list. If matching fails the empty list is returned. The lambda abstractions in this new expression can be rewritten using the approach described above.

3.4.8 Choose carets

A possible extension would be to allow application patterns without carets, so that 'real' application patterns would be possible. A sketch for an algoritm is as follows: for a left hande side without carets one may try all possible caret additions. Since each identifier should occur only once without a caret, the number of combinations is rather limited. For each combination it can be tried whether a selection for inverse function exists that binds all required identifiers.

3.4.9 Integration

The application pattern is a stand-alone application that works seperately from the interpreter. For practical use an integration with the compiler or interpreter would improve usability. This may be achieved by an integration with an pattern match compiler. Other desirable features are support for a complete functional language (not a subset), type checking and no memory limitations. In how far application patterns allow for optimizations like in ordindary pattern matching (see Peyton Jones, 1987) is a point of further research.

3.5 Conclusion

In this chapter a rewriting algoritm for application patterns as sketched and described. Also the implementation of this algoritm, the Application Pattern Compiler, was described. The next chapter discusses further research.

Chapter 4

The future for application patterns

4.1 Introduction

This chapter describes the relation with other work on extensions to pattern matching, and gives some suggestions for further research.

4.2 Related work

In this section the relation with other proposed pattern matching extensions is discussed.

One paper is by Tullsen (2000) who uses inverses of algebraic constructors, but not more generally for other (non-injective) functions. In addition, his paper has a different aim, namely introducing patterns as first class language constructs.

Another paper is by Broberg, Farre, and Svenningsson (2004) who propose an extension of Haskell with regular expression patterns. Such patterns allow, amongst other things, for more flexible string parsing. The use of `join2` and `join3` as described in Section 2.7.2 are simple cases of what their regular expression patterns can handle. Whether their more complicated constructions allow for easy translation in application patterns is a point of further research.

The paper that comes closest to the idea of application patterns is by Erwig and Peyton Jones (2000) who propose to extend Haskell with *pattern guards*. Such constructions allow for pattern matching and adding guards intertwined. They give the example (translated to Miranda syntax) of a function that looks up two values in a mapping,

```
clunky env var1 var2
 = val1 + val2, if ok1 ∧ ok2
 = var1 + var2, otherwise
 where
   m1 = lookup env var1
```

```
   m2 = lookup env var2
   ok1 = isJust m1
   ok2 = isJust m2
   Just val1 = m1
   Just val2 = m2
```

Using their proposed pattern guards, this definition can be written

```
clunky env var1 var2
 | Just val1 <- lookup env var1
 , Just val2 <- lookup env var2
 = val1 + val2
 otherwise = var1 + var2
```

where the `<-` operator tries to fit a pattern into a value.

Such constructions can, to some degree, be rewritten using application patterns. To accomplish this we use a 'trick' that allows for matching patterns and binding identifiers. The functions

```
soThat‘[0,1] x = (x, undef)
matchPat‘[0] x _ = x
```

are used for 'creating room' for binding arguments and for actually matching patterns, respectively. Note that these two functions are each other's inverse. The definition above would translate (using infix notation) into

```
clunky env var1 (var2                          $soThat (
 (Just val1 $matchPat lookup ^env ^var1) $soThat (
 (Just val2 $matchPat lookup ^env ^var2)          )))
 = val1 + val2

clunky env var1 var2
 = var1 + var2
```

which strongly resembles the definition with conditional guards above (for the caret notation see Section 2.4.2). Admittingly it is a bit combersome that the function arguments `env`, `var1` and `var2` must be given twice, and also that `var2` is enclosed by parenthesis.

It is easy to add guards as well by using the definition

```
guardPat‘[0] x _ = undef, if x
```

and by binding the result to the wildcard identifier. For example, the definition

```
f x | [y] <- x
    , y > 3
    , Just z <- h y
  = ...
```

can be translated into

```
f (x                              $soThat (
      [y]     $matchPat ^x      $soThat (
      _       $guardPat ^y > 3  $soThat (
      Just z  $matchPat ^h ^y             ))))
  = ...
```

Conversely, application patterns can also be expressed using pattern guards by mimicking the rewriting algoritm described in Section 3.3. Consider the definition

```
p (f (g x) y) (h z 2)
 = x ^ y + z
```

with the inverses `f‘[0,1]`, `g‘[0]`, and `h‘[0]` defined. The rewriting algoritm would yield

```
p var var1
 = x ^ y + z, if match ≠ Nothing ∧ match1 ≠ Nothing ∧
    match2 ≠ Nothing
  where
     match          = f‘[0,1] var
     Just (var2, y) = match
     match1         = g‘[0] var2
     Just x         = match1
     match2         = h‘[0] 2 var1
     Just z         = match2
```

With pattern guards one would write

```
p var var1
 | Just (var2, y) <- f‘[0,1] var
 | Just x         <- g‘[0] var2
 | Just z         <- h‘[0] 2 var1
 = x ^ y + z
```

which can be considered as more readible than what the rewriting algoritm produces. However, the programmer would have to choose *himself* which inverse to choose and in what order patterns are matched.

These examples suggest that with some effort application patterns and pattern guards can be expressed in one another.

4.3 Further research

4.3.1 Lazyness and evaluation order

Current pattern matching allows for lazy evaluation: when an actual argument is matched against a pattern, the argument is only evaluated as far as necesssary do decide whether the pattern matches. For matching algebraic patterns this means that first the constructor of an actual argument is computed before any of its arguments are evaluted.

Application patterns are a generalization of existing patterns, but when they are used to express existing patterns they agree on lazyness behaviour during evaluation. This can easily be verified by considering the code produced by the application pattern described in Section 3.4. For application patterns that are beyond existing patterns, the programmer ultimately decides how 'lazy' an argument can be matched during evaluation.

4.3.2 Higher order functions

So far only application patterns with ordinary values, i.e. non-higher-order functions are discussed. However, also some higher order functions can be related to inverse functions. Two examples are the map and function composition functions, for which it holds that

$$(map\ f)^{-1}\ xs = map\ f^{-1}\ xs$$

$$(f_1 \circ \cdots \circ f_n)^{-1} = f_n^{-1} \circ \cdots \circ f_1^{-1}$$

Now one might consider writing (`funcomp` refers to function composition $\circ$)

```
!!! map'[1] f ys = map f'[0] ys

!!! funcomp'[0] g h = h . g'[0]
!!! funcomp'[1] f h = f'[0] . h
```

but this is clearly wrong since the backtic ' is not an operator and thus the identifiers `f'[0]` and `g'[0]` cannot be used here.

To a very limited extend one might resolve this by introducing a function

```
inv :: ($\alpha$ → $\beta$) → ($\beta$ \to $\alpha$)
```

that returns the inverse for every function, so that one may write

```
!!! map'[1] f ys = map (inv f) ys

!!! funcomp'[0] g h = h $funcomp (inv g)
!!! funcomp'[1] f h = (inv f) $funcomp h
```

Now such an `inv` function would be possible in principe but can be used only very limited because function equality is not decidable. In Hindley and

Table 4.1: Several notions of function equality for `g x = 2 * x` in Amanda

	`g = g` ⇒ `True`
`h1 = g`	`h1 = g` ⇒ `True`
`h2 x = 2 * x`	`h2 = g` ⇒ `False`
`h3 y = 2 * y`	`h3 = g` ⇒ `False`
`h4 x = x + x`	`h4 = g` ⇒ `False`

Seldin (1986) the prove is mentioned that, in general, the equivalence of two λ-expressions cannot be decided. Since functional languages are implementations of the λ-calculus, this means that also the equality of two function definitions cannot be decided.

This means that the only proper approach is to use a conservative notion of function equality. The problem is that how the interpreter or compiler decides when two functions are equivalent is an *implementation* issue, not a *language* issue. For example, Table 4.1 shows different notation of equality to the function `g x = 2 * x` in Amanda.

This approach is even less useful for curried functions. Suppose for a function f that takes two arguments it holds that

$$\mathit{foldl}\, f\ z\ [x_1, x_2, \ldots, x_n] \quad = \quad f \ldots (f\ (f\ z\ x_1) x_2) \ldots x_n := y$$

Suppose further that f has an inverse on its first argument $f^{-1}_{[0]}$ (in backtic notation `f‘[0]`). Then it holds that

$$\mathit{foldr}\, f^{-1}_{[0]}\ y\ [x_1, x_2, \ldots, x_n] \quad = z$$

since

$$\begin{aligned}
& \mathit{foldr} f^{-1}_{[0]}\ y\ [x_1, x_2, \ldots, x_n] \\
= \;& f^{-1}_{[0]}\ x_1 (f^{-1}_{[0]}\ x_2\ (f^{-1}_{[0]} \ldots (f^{-1}_{[0]}\ x_n\ y) \ldots)) \\
= \;& f^{-1}_{[0]}\ x_1\ (f^{-1}_{[0]}\ x_2\ (f^{-1}_{[0]} \ldots (f^{-1}_{[0]}\ x_n\ (f \ldots (f\ (f\ z\ x\ {}_1)\ x_2) \ldots x_n)) \ldots)) \\
= \;& f^{-1}_{[0]}\ x_1\ (f^{-1}_{[0]}\ x_2\ (f^{-1}_{[0]} \ldots (f\ \ldots (f\ (f\ z\ x_1)\ x_2) \ldots) \ldots)) \\
\vdots \;& \\
= \;& f^{-1}_{[0]}\ x_1\ (f\ z\ x1) \\
= \;& z
\end{aligned}$$

Now the only way to define an inverse function for the $foldl$ function would be introducing an function that gets a function that takes two arguments as an argument and yields the inverse for that function on its first argument. For functions that takes more arguments we would need generalized inverse functions for each kind of cousin, with a whole family of inverse functions as a result. Since all these functions must adhere a restrictive sense of function equality, in my opinion it is doubtful how useful this would be. This is however a point of further research.

4.3.3 More sophisticated pattern failures

A final point for further research is approaches that allow for more sophisticated pattern matches and misses. Hölzenspies (2005) has written a runtime pattern matcher that allows for searching for suitable inverses, together with a notion of undefined-ness for functions. In what respect such an approach allows for more powerful evaluation strategies is a point of further research.

4.4 Conclusion

In this chapter related work was described. The *pattern guards* proposal is most related in that performing manually the rewriting algoritm described in the previous chapter is more easily. Application patterns seem able to express pattern guards.

It is proposed that further research should focus on lazyness and evaluation order issues in pattern matching. Another point that deserves attention is to what extent application patterns are useful for higher order functions.

Appendix A

Implementation of the Application Pattern Compiler

This Appendix is provided as a seperate document, entitled *Application Patterns in Functional Languages: Appendix A: Implementation of the Application Pattern Compiler*

Appendix B

Example input and output

B.0.1 Example input

◇ progIn.amapc ◇

```
f (2+x) 3 = x^2

g (Node x y) = g x + g y
g (Leaf x)  = x

plus'[0] x s = s - x, if x ≥ 0 ∧ s ≥ x
plus'[1] x s = s - x, if x ≥ 0 ∧ s ≥ x

k (Leaf (n+2)) (Node (Leaf (1+x)) (Leaf (1+ ^x))) = n ^ x
k _ _ = 1

l (Leaf x) (Leaf (^x + 1)) = 3

f 1 2 = 3

h u = z^2
    where
      z + ^k = x
             where
               x + (^y+2) = y ^ u
                      where
                        y + ^k = u+1
      k = 14

i p = (x, y)
    where
      (Just x, Just y) = (p, p)

j (Just x, (y, z)) = x + y + z
```

```

m u = y
    where
      y + ^z = z + 1 + u
             where
               z = 3

n (x+1) = fac (fac x)
    where
        fac 0 = 1
        fac n = n * fac(n-1)

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

x = 3

p ys = x + #xs
     where
       (x:xs) = ys

cons`[0,1] xs = (hd xs, tl xs), if xs ≠ []

q ys = f ys + #ys
     where
       f (x:xs) = 1
       f _      = 2

r (Leaf n) (Node (Leaf (^n+2)) (Leaf (x+^n)))
 = n * x
r (Leaf 2) _
 = 3
r _ (Leaf n)
 = n^2
r _ dontcare
 = 4

t (2*2) = 3
t (2+x) = x, if x > 10
t _     = 0 - 1


w xs = (x, x)
     where
       x = (fst . cons`[0,1]) xs


gcd x         ^x = x
gcd (x + ^y)   y = gcd x y
gcd x       (^x + y) = gcd x y
```

B.0.2 Example output

◇ progOut.ama ◇

```
maybe * ::= Just * | Nothing

tree ::= Node tree tree | Leaf num

f var var1
 = x1 ^ 2, if (match ≠ Nothing) ∧ (var1 = 3)
 = 3     , if (var14 = 1) ∧ (var15 = 2)
   where
      (var14, var15)
       = (var, var1)
      match
       = inv_plus_Mb_1 2 var
      Just ((x1))
       = match

g var2
 = (g x2) + (g y), if is_Node var2
 = x3            , if is_Leaf var3
   where
      (var3)
       = (var2)
      is_Node (Node _ _)
       = True
      is_Node _
       = False
      Node x2 y
       = var2
      is_Leaf (Leaf _)
       = True
      is_Leaf _
       = False
      Leaf x3
       = var3

inv_plus_0 x4 s
 = s - x4, if (x4 ≥ 0) ∧ (s ≥ x4)

inv_plus_Mb_0 x4 s
 = Just (s - x4), if (x4 ≥ 0) ∧ (s ≥ x4)
 = Nothing      , otherwise

inv_plus_1 x5 s1
```

```
 = s1 - x5, if (x5 ≥ 0) ∧ (s1 ≥ x5)

inv_plus_Mb_1 x5 s1
 = Just (s1 - x5), if (x5 ≥ 0) ∧ (s1 ≥ x5)
 = Nothing       , otherwise

k var4 var6
 = n1 ^ x6, if (is_Leaf1 var4) ∧ ((match1 ≠ Nothing) ∧ ((is_Node1 var6) ∧ ((is_Leaf2 var7) ∧ ((match2 ≠ Nothing) ∧ ((is_Leaf3 var9) ∧ (var10 = (1 + x6))))))))
 = 1      , otherwise
  where
    (_, _)
     = (var4, var6)
    match1
     = inv_plus_Mb_0 2 var5
    Just ((n1))
     = match1
    is_Leaf1 (Leaf _)
     = True
    is_Leaf1 _
     = False
    Leaf var5
     = var4
    match2
     = inv_plus_Mb_1 1 var8
    Just ((x6))
     = match2
    is_Leaf2 (Leaf _)
     = True
    is_Leaf2 _
     = False
    Leaf var8
     = var7
    is_Leaf3 (Leaf _)
     = True
    is_Leaf3 _
     = False
    Leaf var10
     = var9
    is_Node1 (Node _ _)
     = True
    is_Node1 _
     = False
    Node var7 var9
     = var6

l var11 var12
```

```
 = 3, if (is_Leaf4 var11) ∧ ((is_Leaf5 var12) ∧ (var13
     = (x7 + 1)))
  where
     is_Leaf4 (Leaf _)
      = True
     is_Leaf4 _
      = False
     Leaf x7
      = var11
     is_Leaf5 (Leaf _)
      = True
     is_Leaf5 _
      = False
     Leaf var13
      = var12

h u
 = z ^ 2
  where
     var16
      = x8
     match3
      = inv_plus_Mb_0 k1 var16
     Just ((z))
      = match3
     var17
      = y1 ^ u
     match4
      = inv_plus_Mb_0 (y1 + 2) var17
     Just ((x8))
      = match4
     var18
      = u + 1
     match5
      = inv_plus_Mb_0 k1 var18
     Just ((y1))
      = match5
     k1
      = 14

i p1
 = (x9, y2)
  where
     (var19, var20)
      = (p1, p1)
     is_Just (Just _)
      = True
     is_Just _
      = False
     Just x9
```

```
      = var19
     is_Just1 (Just _)
      = True
     is_Just1 _
      = False
     Just y2
      = var20

j ((var21, (y3, z1)))
 = x10 + (y3 + z1), if is_Just2 var21
  where
     is_Just2 (Just _)
      = True
     is_Just2 _
      = False
     Just x10
      = var21

m u1
 = y4
  where
     var22
      = z2 + (1 + u1)
     match6
      = inv_plus_Mb_0 z2 var22
     Just ((y4))
      = match6
     z2
      = 3

n var23
 = fac1 (fac1 x11), if match7 ≠ Nothing
  where
     fac1 var24
      = 1                   , if var24 = 0
      = n2 * (fac1 (n2 - 1)), otherwise
       where
          (n2)
           = (var24)
     match7
      = inv_plus_Mb_0 1 var23
     Just ((x11))
      = match7

fac var25
 = 1                  , if var25 = 0
 = n3 * (fac (n3 - 1)), otherwise
  where
     (n3)
      = (var25)
```

```

x
 = 3

p ys
 = x12 + (# xs)
  where
     var26
      = ys
     match8
      = inv_cons_Mb_0_1 var26
     Just ((x12, xs))
      = match8

inv_cons_0_1 xs1
 = (hd xs1, tl xs1), if xs1 ≠ Nil

inv_cons_Mb_0_1 xs1
 = Just ((hd xs1, tl xs1)), if xs1 ≠ Nil
 = Nothing                , otherwise

q ys1
 = (f1 ys1) + (# ys1)
  where
     f1 var27
      = 1, if match9 ≠ Nothing
      = 2, otherwise
       where
          (_)
           = (var27)
          match9
           = inv_cons_Mb_0_1 var27
          Just ((x13, xs2))
           = match9

r var28 var29
 = n4 * x14, if (is_Leaf6 var28) ∧ ((is_Node2 var29) ∧
     ((is_Leaf7 var30) ∧ ((var31 = (n4 + 2)) ∧ ((
    is_Leaf8 var32) ∧ (match10 ≠ Nothing)))))
 = 3       , if (is_Leaf9 var34) ∧ (var35 = 2)
 = n5 ^ 2  , if is_Leaf10 var36
 = 4       , otherwise
  where
     (_, dontcare)
      = (var28, var29)
     (_, var36)
      = (var28, var29)
     (var34, _)
      = (var28, var29)
     is_Leaf6 (Leaf _)
```

```
      = True
     is_Leaf6 _
      = False
     Leaf n4
      = var28
     is_Leaf7 (Leaf _)
      = True
     is_Leaf7 _
      = False
     Leaf var31
      = var30
     match10
      = inv_plus_Mb_0 n4 var33
     Just ((x14))
      = match10
     is_Leaf8 (Leaf _)
      = True
     is_Leaf8 _
      = False
     Leaf var33
      = var32
     is_Node2 (Node _ _)
      = True
     is_Node2 _
      = False
     Node var30 var32
      = var29
     is_Leaf9 (Leaf _)
      = True
     is_Leaf9 _
      = False
     Leaf var35
      = var34
     is_Leaf10 (Leaf _)
      = True
     is_Leaf10 _
      = False
     Leaf n5
      = var36

t var37
 = 3    , if var37 = (2 * 2)
 = x15  , if (match11 ≠ Nothing) ∧ (x15 > 10)
 = 0 - 1, otherwise
  where
     (_)
      = (var37)
     (var38)
      = (var37)
     match11
```

```
      = inv_plus_Mb_1 2 var38
     Just ((x15))
      = match11

w xs3
 = (x16, x16)
  where
     x16
      = (fst . inv_cons_0_1) xs3

gcd x17 var39
 = x17       , if var39 = x17
 = gcd x18 y5, if match12 ≠ Nothing
 = gcd x19 y6, if match13 ≠ Nothing
  where
     (x19, var41)
      = (x17, var39)
     (var40, y5)
      = (x17, var39)
     match12
      = inv_plus_Mb_0 y5 var40
     Just ((x18))
      = match12
     match13
      = inv_plus_Mb_1 x19 var41
     Just ((y6))
      = match13
```

B.1 Creative Commons Attribution-ShareAlike License Version 2.0

Attribution-ShareAlike 2.0

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

License

1. Definitions
    1. "Collective Work" means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.
    2. "Derivative Work" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.
    3. "Licensor" means the individual or entity that offers the Work under the terms of this License.
    4. "Original Author" means the individual or entity who created the Work.
    5. "Work" means the copyrightable work of authorship offered under the terms of this License.
    6. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.
    7. "License Elements" means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.
2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

    1. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;
    2. to create and reproduce Derivative Works;
    3. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;
    4. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.
    5. For the avoidance of doubt, where the work is a musical composition:
        1. Performance Royalties Under Blanket Licenses. Licensor waives the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work.
        2. Mechanical Rights and Statutory Royalties. Licensor waives the exclusive right to collect, whether individually or via a music rights society or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).
    6. Webcasting Rights and Statutory Royalties. For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

    The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

    1. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not

require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any reference to such Licensor or the Original Author, as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any reference to such Licensor or the Original Author, as requested.

2. You may distribute, publicly display, publicly perform, or publicly digitally perform a Derivative Work only under the terms of this License, a later version of this License with the same License Elements as this License, or a Creative Commons iCommons license that contains the same License Elements as this License (e.g. Attribution-ShareAlike 2.0 Japan). You must include a copy of, or the Uniform Resource Identifier for, this License or other license specified in the previous sentence with every copy or phonorecord of each Derivative Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Derivative Works that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder, and You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Derivative Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Derivative Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Derivative Work itself to be made subject to the terms of this License.

3. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Derivative Works or Collective Works, You must keep intact all copyright notices for the Work and give the Original Author credit reasonable to the medium or means You are utilizing by conveying the name (or pseudonym if applicable) of the Original Author if supplied; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.

5. Representations, Warranties and Disclaimer
UNLESS OTHERWISE AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE MATERIALS, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY

TO YOU.

6. Limitation on Liability.
EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

    1. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works or Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

    2. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

    1. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

    2. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

    3. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

    4. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

    5. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

Bibliography

Broberg, N., Farre, A., & Svenningsson, J. (2004). Regular expression patterns. 67–78. Retrieved on July 14, 2005, from `http://www.cs.chalmers.se/~d00nibro/harp/harp.pdf`.

Erwig, M., & Peyton Jones, S. (2000). Pattern guards and transformational patterns. Retrieved on September 16, 2005, from `http://research.microsoft.com/~simonpj/Papers/pat.ps.gz`.

Hindley, J. R., & Seldin, J. P. (1986). *Introduction to combinators and λ-calculus.* Cambridge University Press.

Hölzenspies, P. K. F. (2005). (personal communication)

Jones, M. P. (1991). *Gofer 2.21 release notes.* Retrieved on July 7, 2005, from `http://www-i2.informatik.rwth-aachen.de/Teaching/Praktikum/SWPSS96/Gofer/rel221.dvi`.

Oosterhof, N. N., Hölzenspies, P. K. F., & Kuper, J. (2005). Application patterns. In *Proceedings of trends in functional programming 2005, Tallinn, Estonia.*

Papegaaij, E. (2005). The Tina language: A report on the specification and implementation. (Personal communication)

Peyton Jones, S. (1987). *The implementation of functional programming languages.* Retrieved on June 23, 2005, from `http://research.microsoft.com/Users/simonpj/papers/slpj-book-1987/slpj-book-1987.pdf`. Prentice Hall.

Peyton Jones, S. (Ed.). (2003). *Haskell 98 language and libraries: The revised report.* Retrieved on May 14, 2005, from `http://www.haskell.org/definition/haskell98-report.pdf`. Cambridge University Press.

Plasmeijer, R., & Eekelen, M. van. (2001). *Version 2.0 language report* [Draft]. Retrieved on June 2, 2005, from `ftp://ftp.cs.kun.nl/pub/Clean/Clean20/doc/CleanRep2.0.pdf`. Department of Software Technology, University of Nijmegen, The Netherlands.

Plasmijer, R., & Eekelen, M. van. (1993). *Functional programming and parallel graph rewriting.* Amsterdam [etc.]: Addison-Wesley.

Thompson, S. (1995). *Miranda: The Craft of Functional Programming.* Addison Wesley.

Tullsen, M. (2000). First class patterns. In *Practical aspects of declarative languages, second international workshop,padl 2000* (Vol. 1753, pp. 1–15). Springer-Verlag.

Wadler, P. (1992). Monads for functional programming. *118.* Retrieved on June 27, 2005, from `http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/marktoberdorf.pdf`.

Application Patterns in Functional Languages

Appendix A: Implementation of the Application Pattern Compiler

by

Nikolaas N. Oosterhof

University of Twente
Enschede, The Netherlands
2005

The author may be contacted at `n.n.oosterhof@student.uva.nl`

Contents

Appendix A

Implementation of the Application Pattern Compiler

A.1 Introduction

This appendix contains the implementation of the Application Pattern Compiler, a compiler for the Amanda functional language that rewrites application patterns. The actual implementation is given in Section A.2, describing the lexer, parser, preprocessor, renaming, rewriting, postprocessing and printing. The implementation makes extensive use of monads which are described in Section A.3. Some general utility functions are given in Section A.4.

A.2 The application pattern compiler

A.2.1 The main file

◇ main.ama ◇

```
/* the Application Pattern Compiler: a program that translates
 * a functional language with application patterns into semantic
 * equivalent runnable code.
 *
 * This is the main file.
 *
 * Copyright (c) 2005   Nikolaas N. Oosterhof
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
```

```
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#import "util.ama"

#import "monadMb.ama"
#import "monadSt.ama"
#import "monadSts.ama"
#import "monadId.ama"

#import "lexer.ama"
#import "parser.ama"
#import "preproc.ama"
#import "uniqidfs.ama"
#import "rewrite.ama"
#import "postproc.ama"
#import "prettyPrint.ama"

|| the main function
apc :: string → string
apc =    print
       . postproc
       . defsIdfProperNaming
       . defsAddMbInverse
       . mergeDefs
       . rewrDefs
       . uniqidfs
       . preproc
       . parser
       . lexer


|| reading input file and writing output file
|| (the required preamble is added manually)
apcIO fileIn fileOut
 =       fwrite  fileOut premable
   $seq (fappend fileOut . apc . fread) fileIn
 where
   premable =     "maybe * ::= Just * | Nothing \n\n"
               ++ "thenMb Nothing _ = Nothing\n"
               ++ "thenMb _       x = x\n\n"
               ++ "tree ::= Node tree tree | Leaf num\n\n"

```

```
|| an example
apcIOEx
 = apcIO "progIn.amapc" "progOut.ama"
```

A.2.2 The lexer

◇ lexer.ama ◇

```
/* the Application Pattern Compiler: a program that translates
 * a functional language with application patterns into semantic
 * equivalent runnable code.
 *
 * This file contains the lexer.
 *
 * Copyright (c) 2005   Nikolaas N. Oosterhof
 */

/*
 * The lexer state
 *********************************/
lexerTp == (num, string)

lexerSt * ::= { remaining :: [*]    || remaining chars to be lexed
              , pos :: num          || horizontal position
              , vpos :: num         || vertical position
              , offSides :: [num]   || stack of indent positions
              , inDef :: bool       || already seen the first '='-char on this
                  line?
              }

nilLexer s = {remaining = s, pos=0, offSides=[], vpos = 0, inDef = False}

getLexerSt s0 = unitSts s0
setLexerSt s1 s0 = unitSts s10 s10
                 where s10 = s1 & s0

/*
 * Spotting items
 *********************************/
|| spot single element
spot c (st={remaining=[]     }) = unitSts False   st
spot c (st={remaining=(x:xs)}) = unitSts (c = x) st

|| spot multiple elements
spots [c]    = spot c
spots (c:cs) = spot c  $bindSts (b  →
               ifSts b (spots cs    )
                       (unitSts False)
```

```
                                  )

item (st={remaining=(x:xs), pos=pos_})
 = [(x, st & {remaining=xs, pos=pos_+1})]

item _
 = []


lit c st
 = item_st, if item_st ≠ [] ∧ x = c
 = []      , otherwise
 where item_st  = item st
       [(x, _)] = item_st

anylit cs st = concat [ lit c st | c <- nodup cs ]

lits [c]    = lit c $bindSts (x → unitSts [x])

lits (c:cs) = lit  c  $bindSts (x  →
              lits cs $bindSts (xs →
              unitSts (x:xs)    ))



untilLits str
 = (lits str
   ) $biasedOrSts
   ( item $bindSts (c → untilLits str $bindSts (cs → unitSts (c:cs)))
   )

/*
 * Processing indentation
 *
 * Assumption: every line consists of:
 *
 *      blank^* dedent^* [non-blank (any)^*] newLine
 *
 *  A comment is /not/ a blankitem
 *********************************/

/*   dedentLit requires sufficient dedents before any non-blank literal
     01234567890123456 7...
     ... = ...
           where
               ... = ...
             ... --→{for this expression: position p=8, stack=[14,6,4]}

*/

```

```
dedentLit (st={remaining=[], offSides=(x:xs)})
 = [(("dedent", ""), st & { offSides = xs})]

dedentLit (st={pos=p,offSides=(x:xs)})
 = [(("dedent", ""), st & { offSides = xs})], if p < x
 = [], otherwise


dedentLit _
 = []


/* Process an indent token (currently only "=")
   Assumption: the first occurence of that token is indeed the indent-version
               this means that "f (p=(x:xs)) = ..." can not be parsed
               to provide for this: - add a parenthesis counter to the state
                                    - write a left/right parenthesis parser that
                                      adjusts that counter
                                    - give these parser higher priority dan
                                      indentLit
 */


indentLit
 = (biasedOrsSts . map lits . domain) indentNames  $bindSts (tk →
   makeIndent tk)


makeIndent tk (st={pos=p, offSides=offSides_, inDef=False})
 = fromJust
   (  (  (
              ( hdMb offSides_           $bindMb (x →
                    ( ( x = p )              $guardMb (
                      unitMb (unitSts (lookUp indentNames tk ++ redentSuffix, tk)
                                      (st & {inDef=True}))
                                                            )
                    ) $alternativeMb
                    ( ( x < p )              $guardMb (
                      unitMb (unitSts (lookUp indentNames tk, tk)
                                      (st & {inDef=True, offSides = p:offSides_})
                                         )
                                                            )
                    ) $orJust
                    ( zeroSts st
                    )                                )
              ) $orJust
              ( unitSts (lookUp indentNames tk, tk)
                        (st & {inDef=True, offSides = p:offSides_})
              )                                )
          ) $orJust
```

```
          ( zeroSts st
          )
    )

makeIndent _ st = zeroSts st


whereLit    = getStateSts          $bindSts ( (st={pos=pos,offSides=offSides_}) →
              lits "where"         $thenSts (
              updStateSts ( & { inDef=False,offSides = pos:offSides_} )
                                   $thenSts (
              unitSts ("where", "where")                            )))


newLineLit = lits "\n"        $thenSts (
             getStateSts       $bindSts ( (st={vpos=vpos}) →
             updStateSts ( & {inDef=False,pos=0,vpos = vpos+1} ) $thenSts (
             unitSts ("newline", "\n")                  )))



eofLit (st={remaining=[], offSides=[]}) = (unitSts ("eof", [])) st
eofLit st                               = zeroSts st

/*
 * Defining operators, delimeters and keywords
 *
 * - is is assumed that these identifier names ("neg", "lnot", "length",
 *    "cons", ...) are /not defined/ by the programmer
 * - however, for inverse definitions these names /must/ be used by the
 *    programmer ("neq`[0]", "lnot`[0]", "length`[0]", "cons`[0,1]", ...)
 *********************************/

prefixNames = [ ("-", "neg")
              , ("~", "lnot")
              , ("#", "length")
              ]


infixNames = [ (":", "cons")
             , ("++", "join")
             , ("--", "nioj")
             , ("\\/", "lor")
             , ("∧\", "land")
             , ("~", "lnot")
             , ("<", "lt")
             , ("<=", "lte")
             , ("=", "eq")
             , ("≠", "neq")
             , ("≥", "geq")
```

```
                , (">", "gt")
                , ("+", "plus")
                , ("-", "minus")
                , ("*", "times")
                , ("//","floatdivide")
                , ("/", "divide")
                , ("%", "modulo")
                , ('d':"iv", "divide") || } fake amanda parser; it must
                , ('m':"od", "modulo") || } have a bad hack here
                , ("^", "power")
                , (".", "funcomp")
                , ("!", "index")
                ]

delimNames = [ ("|" , "pipe")
             , (";" , "semicolon")
             , ("=" , "eq")
             , ("{" , "lcurly")
             , ("}" , "rcurly")
             , ("(" , "lparen")
             , (")" , "rparen")
             , ("[" , "lbrack")
             , ("]" , "rbrack")
             , ("," , "comma")
             , ("→", "larrow")
             , ("<-", "rarrow")
             ]

indentNames = [ ("::="  , "typeDef")
              , ("::"   , "typeDecl")
              , ("=="   , "typeEq")
              , ("="    , "funDef")
              ]

redentSuffix = "_redent"

litSuffix = ""

keywordNames = map dupl [ "if"
                        , "otherwise"
                        ]
             where dupl x = (x, x)



/*
 * Lexing standard literals
 ********************************/

lexChar   = lexSpecialChar $biasedOrSts
```

```
            (item $checkSts (~.member "\'\""))


lexSpecialChar = ((lit '\\')  $thenSts (
                  (item $checkSts (member (domain specialChars)))
                        $doSts (lookUp specialChars)
                                        ))

|| characters that are escaped by a backslash '\'
specialChars = [ ('n', '\n')
               , ('b', '\b')
               , ('t', '\t')
               , ('r', '\r')
               , ('a', '\a')
               , ('\\', '\\')
               , ('\'','\'')  ||note: we don't allow "'" or '"';
                              ||      use "\'" and '\"' instead
               , ('\"','\"')
               ]

charLit = seqSts [ lit '\''
                , lexChar
                , lit '\''
                ]
          $bindSts (c → unitSts ("char", c))


failLit (st = {pos=pos_, vpos=vpos_, remaining=remaining_})
 = error errorMsg
 where
   errorMsg =    "Lexer: unrecognised character sequence at line "
              ++ itoa vpos_ ++ ":" ++ itoa pos_
              ++ "\nRemaining character sequence: "
              ++ ( if (#remaining_ > remLength) (take remLength remaining_ ++ "
                 [...]")
                                                 remaining_
                 )
            where remLength = 100



stringLit = seqSts [ lits "\""
                  , oneOrMoreSts lexChar
                  , lits "\""
                  ]
            $doSts concat $bindSts (s → unitSts ("string", s))


lexDigit = item $checkSts isDigit
         where isDigit c = ('0' <= c ∧ c <= '9')
```

```
numLit = ( (lit '-' $thenSts lexPosFloat) $bindSts (x →
           unitSts ("num", '-':x)                   )
         ) $biasedOrSts
         ( lexPosFloat                  $bindSts (x →
           unitSts ("num", x)                       )
         )

lexPosFloat = lexNat             $bindSts (x →
              ( lit '.'              $thenSts (
                lexNat               $bindSts (y →
                unitSts (x ++ "." ++ y)        ))
              ) $biasedOrSts
              ( unitSts (x)
              )                            )


lexNat = oneOrMoreSts lexDigit

chars m n = map decode [ i | cm := code m; cn := code n; i<-[cm..cn]]

lexIdf = startLexIdf
         $biasedOrSts
         anylit (  chars '0' '9'
                ++ chars 'A' 'Z'
                )

startLexIdf = anylit ( "_" ++ chars 'a' 'z' )

idfLit = ( lit '^'                        $thenSts (
           startLexIdf                    $bindSts (i  →
           zeroOrMoreSts lexIdf            $bindSts (is →
           unitSts ( "ctxIdf", i:is)                )))
         ) $biasedOrSts
         ( (anylit ("_" ++ chars 'a' 'z')) $bindSts (i  →
           (zeroOrMoreSts lexIdf)          $bindSts (is →
             ( invFunSignature                 $bindSts (sign →
               unitSts ("invIdf", i:is++sign)             )
             ) $biasedOrSts
             ( unitSts ("idf", i:is)
             )                                      ))
         )

invFunSignature = seqSts [ lits "`["
                         , (lexNat $encloseSts (lits ",")) $doSts concat
                         , lits "]"
                         ]
                  $doSts concat
```

```
constrLit = (anylit (chars 'A' 'Z')) $bindSts (i  →
            (oneOrMoreSts lexIdf)     $bindSts (is →
             unitSts ("constrIdf", i:is)              ))


lineCommentLit = lits "||"                      $thenSts (
                 newLineLit                     $thenSts (
                 unitSts ("comment", [])                  ))

lexBlank = item $checkSts (member " ")

blankLit = (oneOrMoreSts lexBlank) $thenSts (
           unitSts ("blank", [])            )

preprocLit = biasedOrsSts (map lits ["#import", "#synonym", "#operator"])
                                                $bindSts ( xs       →
             zeroOrMoreSts lexBlank             $thenSts (
             stringLit                          $bindSts ( (_, ys) →
             zeroOrMoreSts lexBlank             $thenSts (
             newLineLit                         $thenSts (
             unitSts ("preproc", xs ++ " " ++ ys)         )))))

/*
 * Definition of priority of literals
 *
 * For some items this order is /very/ important (a.o. indentation)
 *********************************/

litMapping
 = [ ("blank"      ++ litSuffix, blankLit      )
   , ("where"      ++ litSuffix, whereLit      )
   , ("indent"     ++ litSuffix, indentLit     )
   , ("dedent"     ++ litSuffix, dedentLit     )
   , ("lineComment"++ litSuffix, lineCommentLit)
   , ("preproc"    ++ litSuffix, preprocLit    )
   ] ++ makeMapping keywordNames ++
   [ ("idf"        ++ litSuffix, idfLit        )
   ] ++ makeMapping (prefixNames ++ infixNames ++ delimNames) ++
   [ ("char"       ++ litSuffix, charLit       )
   , ("stringLit"  ++ litSuffix, stringLit     )
   , ("num"        ++ litSuffix, numLit        )
   , ("constr"     ++ litSuffix, constrLit     )
   , ("newline"    ++ litSuffix, newLineLit    )
   , ("fail"       ++ litSuffix, failLit       )
   ]

|| For this implementation, we want to ignore some literals
ignoreNames = ["blank", "lineComment", "preproc", "newline"]
```

```
greaterLength (x, _) (y, _) = #x > #y

filterIgnore = filter (~.member ignoreNames.fst)

makeMapping stuff = [ (name ++ litSuffix, lits x $thenSts (unitSts (name, x)))
                    | (x, name) <- mergeSortWith greaterLength stuff ]
                  where


otherMapping = [ (name ++ litSuffix, lits x $thenSts (unitSts (name, x)))
               | (x, name) <- otherNames ]

otherNames = mergeSortWith greaterLength (   prefixNames
                                          ++ infixNames
                                          ++ delimNames
                                          ++ keywordNames)

programLit
 = altStarSts (range litMapping) eofLit

lexer :: string → [(string, string)]
lexer = filterIgnore . fst . hd . programLit . nilLexer
```

A.2.3 The parser

◇ parser.ama ◇

```
/* the Application Pattern Compiler: a program that translates
 * a functional language with application patterns into semantic
 * equivalent runnable code.
 *
 * This file contains the parser.
 *
 * Copyright (c) 2005   Nikolaas N. Oosterhof
 */

/*
 * Definition of types for definitions and expressions
 *********************************/
defTp
 ::= FunDef exprTp               || {left hand side}
            [(exprTp, exprTp)]   ||  = [{value1, guard1}, ..., {valueN, guardN}]
            [defTp]              ||  where {subDef1, ..., subDefK}

exprTp
 ::= FA [exprTp]            || FA [f, x1, ..., xN]     is  f x1 ... xN
                            ||    where  f  must be an *Idf
   | Constr [char] [exprTp] || Constr "" [x1, ..., xN] is an N-tuple
```

```
                              || Constr c   []             is the constant c
                              || Constr C   [x1, ..., xN] is C x1 ... xN
   | Idf [char]               || Idf i                     is the identifier i
   | InvIdf [num] [char]      || InvIdf [s1, ..., sK] f  is f'[s0, ..., sK]
   | CtxIdf [char]            || CtxIdf i                  is the identifier ^i
                              ||                             (bound in context)

/*
 * Parsing items
 *******************************/
parserSt * ::= { remainingP :: [*] }
nilParser s = { remainingP = s}


token t (st = {remainingP = ((x,y):xs)})
 = unitSts y (st & {remainingP = xs}), if x = t
 = []      , otherwise

token _ _ = []

anyToken ts st = concat [ token t st | t <- nodup ts ]

/*
 * Parsing standard literals
 *******************************/

litExpr = anyToken ["char", "num", "string"] $doSts (x → Constr x [])


listExpr = token "lbrack"                          $thenSts (
                ( ( ( expr $separateSts (token "comma") )
                          $doSts       makeListExpr
                  )                                    $bindSts (xs →
                  token "rbrack"                       $thenSts (
                  unitSts xs                                     ))
                ) $biasedOrSts
                ( token "rbrack"                       $thenSts (
                  unitSts (makeListExpr [] ))                    )
                                                          )
         where
           makeListExpr []     = Constr "Nil" []
           makeListExpr (x:xs) = FA [Idf "cons", x, makeListExpr xs]

idfExpr = ( token "ctxIdf" $bindSts (x →
            unitSts (CtxIdf x)       )
          ) $biasedOrSts
          ( token "invIdf" $doSts makeInvFunSignature
          ) $biasedOrSts
          ( token "idf" $bindSts (x →
            unitSts (Idf x)       )
```

```
           )

faExpr     = atomExpr               $bindSts (i  →
               ( oneOrMoreSts atomExpr  $bindSts (xs →
                 unitSts (FA (i:xs))              )
               )       )

constrExpr = ( token "constrIdf"     $bindSts (c  →
               zeroOrMoreSts (constrExpr $biasedOrSts atomExpr)
                                     $bindSts (xs →
               unitSts (Constr c xs)           ))
             )

|| parse an inverse function signature f‘[...]
|| a bit ugly, as it uses the lexerState.
makeInvFunSignature =   fst
                      . hd
                      .  ( oneOrMoreSts lexIdf               $bindSts (is →
                           lits "‘["                          $thenSts (
                           (lexNat $separateSts (lits ",")
                                  $doSts       (map atoi) )  $bindSts (sign →
                           lit ’]’                            $thenSts (
                           unitSts (InvIdf sign is)                    ))))
                         )
                      . nilLexer

/*
   Priorities of prefix and infix operators
 */
preinfixOps = [ ( []   , [":", "++", "--"] )
              , ( []   , ["\\/"] )
              , ( []   , ["∧\"] )
              , ( ["~"], [] )
              , ( []   , ["<", "<=", "=", "≠", "≥", ">"] )
              , ( []   , ["+", "-"] )
              , ( ["-"], [])
              , ( []   , ["*", "//", "/", ’d’:"iv", ’m’:"od"])
              , ( []   , ["^"] )
              , ( []   , ["."] )
              , ( ["#"], [] )
              , ( []   , ["!"] )
              ]

|| quite elegant solution :-)
expr = makePreInfixExpr expr simpleExpr preinfixOps

makePreInfixExpr _ finalExpr []
 = finalExpr

makePreInfixExpr curExpr finalExpr ((prefixes, infixes):xs)
```

```
 = prefixExpr $biasedOrSts infixExpr
 where
   prefixExpr = biasedOrsSts [ token tname             $bindSts (x →
                                 curExpr               $bindSts (c →
                                 unitSts (FA [Idf tname, c])       ))
                             | t <- prefixes
                             ; tname := lookUp prefixNames t
                             ]
   infixExpr  = nextExpr                  $bindSts (c →
                biasedOrsSts ( [ ( token tname            $bindSts (x →
                                    curExpr               $bindSts (d →
                                    unitSts (FA [Idf tname, c, d])   ))
                                  )
                               | t <- infixes
                               ; tname := lookUp infixNames t
                               ] ++ [unitSts c]
                             )                    )
   nextExpr = makePreInfixExpr nextExpr finalExpr xs


simpleExpr = atomExpr $orSts faExpr $orSts constrExpr

eofExpr (st={remainingP=[]})
 = unitSts [] st
eofExpr st
 = zeroSts st

atomExpr = ( token "lparen"                    $thenSts (
               ( token "rparen"                $thenSts (
                 unitSts (Constr "" [])                  )
               ) $biasedOrSts
               ( expr                              $bindSts (x →
                   ( token "comma"                     $thenSts (
                     expr $separateSts (token "comma")  $bindSts (xs →
                     token "rparen"                    $thenSts (
                     unitSts (Constr [] (x:xs))                 )))
                   ) $biasedOrSts
                   ( token "rparen"                    $thenSts (
                     unitSts x                                  )
                   )                                        )
               )                                       )
           ) $orSts idfExpr $orSts listExpr $orSts litExpr


funcDefinition = expr                             $bindSts (x →
                 token "funDef"                   $thenSts (
                 funcClause $separateSts (token "funDef_redent")
                                                  $bindSts (cs →
                   ( token "where"                    $thenSts (
                     oneOrMoreSts funcDefinition        $bindSts (ws →
```

```
                     token "dedent"                      $thenSts (
                     token "dedent"                      $thenSts (
                     unitSts (FunDef x cs ws)                      ))))
                   ) $biasedOrSts
                   ( token "dedent"                      $thenSts (
                     unitSts (FunDef x cs [] )                      )
                   )                                         )))


funcClause = expr           $bindSts (x →
               ( token "comma"   $thenSts (
                   ( token "if"        $thenSts (
                     expr              $bindSts (y →
                     unitSts (x, y)              ))
                   ) $biasedOrSts
                   ( token "otherwise" $thenSts (
                     unitSts (x, Constr "True" []))
                   )                       )
               ) $orSts
               ( unitSts (x, Constr "True" []))
                                     )

program :: stateSts (parserSt (string, string)) [defTp]
program = oneOrMoreSts funcDefinition $bindSts (x →
          eofExpr                     $thenSts (
          unitSts x                             ))

|| In case of an parsing error the message is not really informative :-|
parser :: [(string, string)] → [defTp]
parser s
 = error "Nothing to parse: empty input"           , if s = []
 = error "Parsing error, somewhere in the program.", if parses = []
 = fst (hd parses)                                 , otherwise
 where
   parses = program (nilParser s)
```

A.2.4 The preprocessor

◇ preproc.ama ◇

```
/* the Application Pattern Compiler: a program that translates
 * a functional language with application patterns into semantic
 * equivalent runnable code.
 *
 * This file contains the preprocessor.
 *
 * Copyright (c) 2005   Nikolaas N. Oosterhof
 */
```

```

/*
 * Rewriting left hand side and/or complete definitions
 *******************************/

rewriteLHside rw (FunDef f vgs ws)
 = FunDef (rw f) vgs (map (rewriteLHside rw) ws)

rewriteLRHside rw (FunDef f vgs ws)
 = rewriteLHside rw (FunDef f
                             (map (mapPair rw) vgs)
                             (map (rewriteLRHside rw) ws)
                    )




rewriteExpr rw (FA fargs)      = rw (FA (map (rewriteExpr rw) fargs))
rewriteExpr rw (Constr c args) = rw (Constr c (map (rewriteExpr rw) args))
rewriteExpr rw i               = rw i

/*
 * Rewriting join operators so that e.g. the application pattern
 *     x ++ "," ++ y ++ "," ++ z
 * is rewritten into
 *     join3 x "," (join3 y "," z)
 * making it suitable for solving for x, y and z by using the inverse join`[0,2]
 *
 *******************************/
rewriteJoin3 (f=(FA [CtxIdf "join", FA [CtxIdf "join", x, y], z]))
 = FA (CtxIdf "join3" : [x, y, z]), if isKnown y
 = f, otherwise

rewriteJoin3 (f=(FA [CtxIdf "join", x, FA [CtxIdf "join", y, z]]))
 = FA (CtxIdf "join3" : [x, y, z]), if isKnown y
 = f, otherwise

rewriteJoin3 otherExpr
= otherExpr


isKnown (Idf _) = False
isKnown (Constr _ args) = forAll args isKnown
isKnown (FA (_:args)) = forAll args isKnown
isKnown (CtxIdf _) = True
isKnown (InvIdf _ _) = True


/*
 * Add the caret in an application pattern in expressions where it
```

```
 * may be omitted by the programmer, i.e.
 *  - for operators
 *  - in nested function applications
 ********************************/
addCtxFA nested (FA (i:args))
 = FA (i1:args1)
 where
   i1 = makeCtxIdf (nested \/ isOperator) i
      where
        isOperator = ~(empty [j | (_, j) <- infixNames++prefixNames; Idf j = i])
        makeCtxIdf True (Idf i) = CtxIdf i
        makeCtxIdf _    i       = i
   args1 = map (addCtxFA True) args

addCtxFA nested (Constr c args)
 = Constr c args1
 where
   args1 = map (addCtxFA nested) args

addCtxFA _ i
 = i


rewrLHsides funcs =   map (rewriteLHside (limiterates funcs))
preproc = rewrLHsides [addCtxFA False, rewriteJoin3]
```

A.2.5 Creating unique identifiers

◇ uniqidfs.ama ◇

```
/* the Application Pattern Compiler: a program that translates
 * a functional language with application patterns into semantic
 * equivalent runnable code.
 *
 * This file contains the part that renames identifiers to unique ones.
 *
 * Copyright (c) 2005   Nikolaas N. Oosterhof
 */

/*
 * Retrieving primary and secundary identifiers:
 * - primary: the identifiers that are bound in a definition
 * - secundary: arguments for the primary identifiers
 *
 ********************************/

primSecIdfs atTop (FA (CtxIdf f:args))
 = map (primSecIdfs atTop)         args $bindId ( idfs →
```

```
  unitId (concatPair ( idfs))     )


primSecIdfs atTop (FA (f:args))
 = primSecIdfs True f                   $bindId ( idf  →
   map (primSecIdfs False)         args $bindId ( idfs →
   unitId (concatPair ( idf : idfs))     ))

primSecIdfs _ (CtxIdf i)
 = ([], [])

primSecIdfs atTop (InvIdf _ _)
 = ([], [])

primSecIdfs atTop (Constr _ args)
 = map (primSecIdfs atTop) args $bindId (idfs →
   unitId (concatPair idfs))

primSecIdfs atTop (Idf i)
 = ([i], []), if atTop
 = ([], [i]), otherwise

|| assumption: all identifiers are different
exprPrimSecIdfs atTop f s
 = ((mapPair (filter ((~) . empty)) . primSecIdfs atTop) f, s)


defsPrimSecIdfs atTop defs s
 = (pairAsSet (concatPair [ fst (exprPrimSecIdfs atTop fargs nilPair) | FunDef
    fargs _ _ <- defs ]), s)


/*
 * Renaming identifiers so they are bound only once in the whole program.
 * A stack is used for the different scopes.
 *********************************/
popSubs ((_:subs), used) = unitSt subs (subs, used)

pushSubs old new (subs, used)
 = (allSubs, (allSubs, newUsed))
 where
   allSubs = (zip2 old new) : subs
   newUsed = asSet (used ++ old ++ new)


subsDefPrimIdfs defs (s=(subs, _))
 = (map (subsDefIdfs subs) defs, s)

subsDefIdfs subs (FunDef f vgs ws)
```

```
 = FunDef (subsExprIdf subs f) ((map . mapPair) (subsExprIdf subs) vgs) (map (
     subsDefIdfs subs) ws)

subsSecIdfs f (s=(subs, used))
 = (subsExprIdf subs f, s)



subsVgsExprIdfs vgs (subs, used)
 = ((map . mapPair) (subsExprIdf subs) vgs, (subs, used))

subsExprIdf subs (FA fargs)
 = FA (map (subsExprIdf subs) fargs)

subsExprIdf subs (Constr c args)
 = Constr c (map (subsExprIdf subs) args)

subsExprIdf subs (Idf i)
 = Idf i1, if found ≠ []
 = Idf i, otherwise
 where
   found=lookUpAll (concat subs) i
   (i1:_)=found

subsExprIdf subs (CtxIdf i)
 = CtxIdf i1, if found ≠ []
 = CtxIdf i, otherwise
 where
   found=lookUpAll (concat subs) i
   (i1:_)=found

subsExprIdf _ i
 = i

/*
   Creating fresh identifiers
 */
numString n = combinations (bcxyz : rep (a:bcxyz) (n-1))
               where
                 (a:bcxyz) = map decode [code '0'..code '9']

bdNewIdf i (st=(x, used))
 = unitSt newIdf (x, used ++ [newIdf])
 where
   (newIdf:_) = filter ((~).member used)
                        (map (i++) ("" : concatmap numString [1..]))



/*
   The actual renaming.
```

```
   (bd refers to Barendrecht without any apparant reason)
 */

bds :: [defTp] → stateT ([[(string,string)]], [string]) [defTp]
||     <input>              <substitutions>    <bound>   <output>

bds defs
 = defsPrimSecIdfs True defs                           $bindSt ( (prim, _) →
   bdNewIdfs prim                                      $bindSt (prim1 →
   pushSubs prim prim1                                 $thenSt (
   subsDefPrimIdfs defs                                $bindSt (defs1 →
   traverse bd defs1                                   $bindSt (defs2 →
   unitSt defs2         )))))

bd :: defTp → stateT ([[(string,string)]], [string]) defTp
bd (FunDef f vgs ws)
 = exprPrimSecIdfs True f                              $bindSt ( (_, sec) →
   bdNewIdfs sec                                       $bindSt (sec1 →
   pushSubs sec sec1                                   $thenSt (
   bds ws                                              $bindSt (ws1 →
   subsVgsExprIdfs vgs                                 $bindSt (vgs1 →
   subsSecIdfs f                                       $bindSt (f1 →
   popSubs                                             $thenSt (
   popSubs                                             $thenSt (
   unitSt (FunDef f1 vgs1 ws1)                                   ))))))))

/*
   rename definitions

   renaming wildcard "_" is necessary: if left out, the definition

   f _ 0 = ...
   f x y = ...x...y...

   is rewritten to something like

   f _ var2 = ...
   f x y    = ...x...y..

   which during merging leads to the where clause

   ...
    where (x, y) = (_, var)

    so that x is not bound properly.
 */

bddefs defs
 = traverse bd defs ([[]], wildCardIdfName : prim)
 where
```

```
  ((prim, _), _) = defsPrimSecIdfs True defs ([], [])


bdNewIdfs = traverse bdNewIdf

uniqidfs :: stateT [defTp] [string]
uniqidfs defs
 = (used1, defs1)
 where
   (defs1, (_, used1)) = bddefs defs
```

A.2.6 The actual rewriting

◇ rewrite.ama ◇

```
/* the Application Pattern Compiler: a program that translates
 * a functional language with application patterns into semantic
 * equivalent runnable code.
 *
 * This file does the actual rewriting.
 *
 * Copyright (c) 2005   Nikolaas N. Oosterhof
 */

idfSep = "_"
varPrefix = "var"
matchPrefix = "match"

maybeSfx = "_Mb"

invPrefix="inv"

newIdfPrefix = "idf_"

wildCardIdfName = "_"
wildCardIdf = Idf wildCardIdfName

/*
 * The rewriting state
 ********************************/

patTp
 ::= { wheres     :: [defTp]                        || where clauses to be added
     , guards     :: [exprTp]                       || guards to be added
     , invFunDefs :: [([char], ([num], [num]))]     || list of (f, i*'s, j*'s
                                                    || - i*'s: indices provided args
                                                    || - j*'s:    "    required  "
     , usedIdfs   :: [string]                       || bound identifiers
```

```
     , provided   :: [(exprTp, exprTp)]          || provided identifiers and condition
     , required   :: [(exprTp, exprTp)]          || required identifiers and condition
     }

nilPat
 = { wheres    = []
   , guards    = []
   , invFunDefs = []
   , usedIdfs  = []
   , provided = []
   , required = []
   }

initPatSt (used, defs)
 =   nilPat
   & { invFunDefs = [(i, (s, [0..#s+(#as)-2]--s))
                    | FunDef (FA ((InvIdf s i):as)) _ _ <- defs]
     , usedIdfs = used }

define f g
 = FunDef f [(g, Constr "True" [])] []

/*
   Updating the state
 */
addGuard x (st={guards=guards})
 = (x, st & {guards=guards ++ [x]})

addWheres xs (st={wheres=wheres})
 = (xs, st & {wheres=wheres ++ xs})


addRequired xs g (st={required=required})
 = (xs, st & {required=required ++ [ (i, g) | i <- concatmap ctxIdfs xs ]})


addProvided xs g (st={provided=provided})
 = (xs, st & {provided=provided ++ [ (i, g) | i <- xs ]})

|| find context identifiers (these are the only that may cause problems)
ctxIdfs (FA (_:args))
 = concatmap ctxIdfs args

ctxIdfs (Constr _ args)
 = concatmap ctxIdfs args

ctxIdfs (CtxIdf i)
 = [Idf i]
```

```

ctxIdfs _
 = []



/*
   Adds created guards to existing guards
   - returns new guards and created where clauses
   - as a side effect all created guards and where clauses so far
     are removed from the current state
 */

gatherVgsWs vgs st
 = unitSt (vgs1, ws1) (st & emptySt)
 where
   { wheres     = wheres
   , guards     = guards
   , provided   = provided
   , required   = required
   } = st
   vgs1 = [(v, foldr makeAnd g
                (fixPatternOrder provided required guards))
          | (v, g) <- vgs]
   ws1 = wheres
   emptySt = {wheres=[], guards=[], provided=[], required=[]}

makeAnd x y = FA [Idf "land", x, y]

/*
   Fixes order of pattern matching, if necessary. This is only necessary for
   identifiers used in the context that are bound in a pattern to the right
   in of the current pattern (in a lhs)

   For example, in the definition

     f (p x ^y) (q y)

   first y must be resolved (through q`[0], assumed that it is defined),
   then x must be resolved  (through p`[0],   "                         " ).

   The order is fixed by moving guards to the left, if necessary.

   Note: there is /no check/ for cyclic dependancies.
 */

fixPatternOrder provided required guards
 = map fst patFixed
  where
    patDeps = transClose [ (reqgd, provgd)
```

```
                          | (prov, provgd) <- provided
                          ; (req , reqgd ) <- required
                          ; prov = req
                          ]
    patMDPs = [ (m, (d, p))
              | m, p <- guards, [0..]
              ; d := lookUpAll patDeps m
              ]
    patFixed = mergeSortWith patOrd patMDPs

/*
   for the proper order of matching patterns
   - first consider dependencies
   - if no dependencies exist, use the ordinary (left-to-right) order

   in a tuple (m, (d, p)): m is the matching guard
                           d are other matchings guards m depends on
                           p is the position of the pattern (p is in [0..])
 */

patOrd (m0, (d0, p0)) (m1, (d1, p1))
 =       (d01 ∧ ~d10)
   \/  ~(d10 ∧ ~d01)
   \/    (p1 > p0)
 where
   d01 = member d1 m0
   d10 = member d0 m1



/*
   Create new identifier
 */

patNewIdf prefix (s={usedIdfs=usedIdfs})
 = (Idf newIdf, s & {usedIdfs=(newIdf : usedIdfs)})
 where
   allIdfs = map (prefix++) ("" : concatmap numString [1..])
   (newIdf:_) = filter (~.(member usedIdfs)) allIdfs

patNewIdfs = traverse patNewIdf

/*
   Find inverse function definition
   - if none is found a runtime exception is thrown
   - if multiple definitions are found the first is taken
 */
patInvFunDef i args (s={invFunDefs=invFunDefs})
 = error ("No suitable inverse defined for " ++ i), if rps = []
 = (hd rps, s), otherwise
```

```
 where
   rps = [ (invIdfAsMb (InvIdf provi i), prov, req)
         | (provi, reqi) <- lookUpAll invFunDefs i
         ; req := reqi $fromList args
         ; forAll req isKnown
         ; prov := provi $fromList args
         ]


invIdfAsMb (InvIdf c i)
 = InvIdf c (i ++ maybeSfx)

/*
 * Rewriting a pattern
 ********************************/

|| Application pattern
rewrPat (pat=(FA (CtxIdf i:args)))
 = patNewIdfs ["var"]             $bindSt ( [var] →
   addGuard (FA [Idf "eq", var, pat])
                                  $thenSt (
   unitSt var                               )), if forAll args isKnown

 = patNewIdfs ["var", "match"]    $bindSt ( [var, match] →
   unitSt (FA [Idf "neq", match, Constr "Nothing" []])
                                  $bindSt ( guard →
   addGuard guard $thenSt (
   patInvFunDef i args            $bindSt ( (finv, prov, req) →
   traverse rewrPat prov          $bindSt ( prov1 →
   addWheres [ define match
                      (FA (finv:req++[var]))
             , define (Constr "Just" [Constr "" prov1])
                      match
             ]                    $thenSt (
   addRequired req guard          $thenSt (
   addProvided prov1 guard        $thenSt (
   unitSt var                               ))))))))

|| Constant application pattern (that binds no identifiers)
rewrPat (pat=(FA (f:args)))
 = patNewIdfs ["var"]             $bindSt ( [var] →
   addGuard (FA [Idf "eq", var, pat])
                                  $thenSt (
   unitSt var                               )), if forAll args isKnown
 = error "Oops! why rewrite function application?", otherwise

|| Tuple
rewrPat (Constr "" args)
 = traverse rewrPat args          $bindSt ( args1 →
   unitSt (Constr "" args1)                 )
```

```
|| Constant
rewrPat (pat=(Constr c []))
 = patNewIdfs ["var"]             $bindSt ( [var] →
   addGuard (FA [Idf "eq", var, pat])
                                  $thenSt (
   unitSt var                               ))

|| Constructor with ≥ 1 argument
rewrPat (Constr c args)
 = patNewIdfs ["var", "is_" ++ c] $bindSt ( [var, isConstr] →
   unitSt (FA [isConstr, var])    $bindSt ( guard →
   addGuard guard                 $thenSt (
   traverse rewrPat args          $bindSt ( args1 →
   addWheres [ define (FA [isConstr, Constr c (rep wildCardIdf (#args))])
                      (Constr "True" [])
             , define (FA [isConstr, wildCardIdf])
                      (Constr "False" [])
             , define (Constr c args1)
                      var
             ]                    $thenSt (
   addProvided args1 guard        $thenSt (
   unitSt var                               ))))))

|| Identifier bound from context: treat like a constant
rewrPat ((CtxIdf i))
 = patNewIdfs ["var"]             $bindSt ( [var] →
   addGuard (FA [Idf "eq", var, Idf i])
                                  $thenSt (
   unitSt var                               ))

|| Any other pattern, i.e. only an identifier
rewrPat i
 = unitSt i



/*
 * Rewriting a definition
 *********************************/

getMakePats (pat=(FA ((CtxIdf i):args)))
 = ( [pat], ([p] → p), False )

getMakePats (FA (i:args))
 = ( args, (a → FA (i:a)), True)

getMakePats pat
 = ( [pat], ([p] → p), False )
```

```
getPats expr
 = pats
 where
   (pats, _, _) = getMakePats expr

rewrDef (FunDef fun vgs ws)
 = unitSt (getMakePats fun)     $bindSt ( (pats, makePat, outSide) →
   traverse rewrPat pats        $bindSt ( pats1 →
   gatherVgsWs vgs              $bindSt ( (vgs1, ws1) →
   traverse rewrDef ws          $bindSt ( ws2 →
   unitSt (if outSide [(FunDef (makePat pats1) vgs1 (concat ws2 ++ ws1))]
                       ((FunDef (makePat pats1) vgs []) : ws1 ++ concat ws2)
                           ||note: use old vgs as we dont check for pattern match
          )                                ))))

rewrDefs (ud=(used, defs))
 = (concat . fst . traverse rewrDef defs . initPatSt) ud
```

A.2.7 The postprocessing

◇ postproc.ama ◇

```
/* the Application Pattern Compiler: a program that translates
 * a functional language with application patterns into semantic
 * equivalent runnable code.
 *
 * This file contains the postprocessor.
 *
 * Copyright (c) 2005   Nikolaas N. Oosterhof
 */


/*
 * Comparing patterns to decide whether definitions must be merged.
 *
 * This is necessary because replacing non-identier patterns by
 * identifier patterns can make subsequent equations in function
 * definitions unreachable
 *********************************/

equalPat (FA (f:xs)) (FA (g:ys))
 = f = g ∧ and (map2 equalPat xs ys)
 where

equalPat (Constr c xs) (Constr d ys)
 = and ( (c=d) : map2 equalPat xs ys )

equalPat (Idf _)      (Idf _)      = True
```

```
equalPat (CtxIdf _)   (CtxIdf _)   = True
equalPat (InvIdf _ _) (InvIdf _ _) = True

equalPat _ _ = False

equalDefPat f g
 = fp = gp ∧ equalPat x y
 where
   (x, y) = mapPair getFun (f, g)
          where
            getFun (FunDef f _ _) = f
   (fp, gp) = mapPair (fst . primSecIdfs True) (x, y)


/*
 * Combining two definitions
 *******************************/
joinDefs (FunDef f0 vgs0 ws0) (FunDef f1 vgs1 ws1)
 = FunDef f0
          (vgs0 ++ vgs1)
          (alignPats : ws0 ++ ws1)
 where
   (pats0, pats1) = mapPair getPats (f0, f1)
   alignPats = define (Constr "" pats1) (Constr "" pats0)

combineDefs (d:ds) = foldl joinDefs d ds

iterateDefs f defs
 = f [ FunDef fun vgs (iterateDefs f ws)
     | FunDef fun vgs  ws <- defs
     ]
mergeDefs :: [defTp] → [defTp]
mergeDefs = iterateDefs ((map combineDefs) . (partition equalDefPat))

/*
 * For each inverse function definition, add an extra definition
 * that returns a value of the maybe type
 *
 * That is, for each definition
 *
 *      f‘[...] :: ... → tp
 *      f‘[...] x1 ... xK
 *        = v1, if g1
 *        ...
 *        = vN, if gN
 *
 * we add an extra definition
 *
 *      f_Mb‘[...] :: ... → maybe tp
 *      f_Mb‘[...] x1 ... xK
```

```
 *        = Just v1, if g1
 *        ...
 *        = Just vN, if gN
 *        = Nothing, otherwise
 *
 *********************************/

addMbInverse (def=(FunDef (FA ((f=(InvIdf _ _)):args)) vgs ws))
 = [def, defMb]
 where
   defMb = FunDef (FA ((invIdfAsMb f) : args))
                  (map addJust vgs ++ [nothingOtherwise])
                  ws || assume no inverses are defined in where clauses
         where
           addJust (v, g) = (Constr "Just" [v], g)
           nothingOtherwise = (Constr "Nothing" [], Constr "True" [])

addMbInverse def
 = [def]

defsAddMbInverse :: [defTp] → [defTp]
defsAddMbInverse
 = concatmap addMbInverse

/*
 * Rename identifers of the form InvIdf and CtxIdf
 * so that they are understood by the Amanda compiler
 *********************************/
defsIdfProperNaming
 = map (rewriteLRHside (rewriteExpr exprIdfProperNaming))

exprIdfProperNaming (InvIdf s i)
 = Idf (   invPrefix
        ++ idfSep
        ++ i
        ++ idfSep
        ++ (enclose "" idfSep "" (map itoa s))
       )

exprIdfProperNaming (CtxIdf i)
 = Idf i

exprIdfProperNaming i
 = i

/*
 * Very limited postprocessing
 * Replace expressions "x ∧ True" and "True ∧ x" by "x"
                       "(x)"                     by "x"
 *********************************/
```

```
rewrAndTrue (FA [Idf "land", Constr "True" [], x]) = x

rewrAndTrue (FA [Idf "land", x, Constr "True" []]) = x

rewrAndTrue x = x


rewrSingleConstr (Constr "" [x]) = x

rewrSingleConstr x               = x

postproc :: [defTp] → [defTp]
postproc = map (rewriteLRHside (limiterates (map rewriteExpr funcs)))
         where
           funcs = [rewrAndTrue, rewrSingleConstr]
```

A.2.8 Printing the output nicely

◇ prettyPrint.ama ◇

```
/* the Application Pattern Compiler: a program that translates
 * a functional language with application patterns into semantic
 * equivalent runnable code.
 *
 * This file contains the prettyPrint functionality.
 *
 * Copyright (c) 2005   Nikolaas N. Oosterhof
 */

/*
 * Positioning of blocks of text above and next to each other
 *******************************/

nextTo a [] = a
nextTo a [[]] = a
nextTo [] b = b
nextTo [[]] b = b
nextTo a b = [ la ++ lb | la, lb <- filla, fillb ]
           where
             (xa, xb) = mapPair (max . map #) (a, b)
             (ya, yb) = mapPair (#)           (a, b)
             filla = fill (rep ' ' xa) height (map (fill ' ' xa) a)
             fillb = fill (rep ' ' xb) height (map (fill ' ' xb) b)
             height = max2 ya yb

nextTos = foldr nextTo []
```

```
above a [] = a
above [] b = b
above  a b = filla ++ fillb
           where
             (xa, xb) = mapPair (max . map #) (a, b)
             filla = map (fill ' ' width) a
             fillb = map (fill ' ' width) b
             width = max2 xa xb

aboves = foldr above []

fill atom length line = line ++ rep atom (length - (#line))


optAbove a [] = []
optAbove a [[]] = [[]]
optAbove a b = a $above b

optNextTo a [] = []
optNextTo a [[]] = [[]]
optNextTo a b = a $nextTo b, otherwise

/*
 * Printing definitions and expressions nicely
 ********************************/

|| the boolean denotes whether the result must not be enclosed by parenthesis
generic prettyPrint :: bool → * → [char]

prettyPrint _ (Idf f)
 = f

prettyPrint _ (CtxIdf f)
 = '^':f

prettyPrint _ (InvIdf sign f)
 = f ++ "`" ++ (filter (≠' ') . toString) sign

prettyPrint True (Constr [] xs)
 = enclose "(" ", " ")" (map (prettyPrint True) xs)

prettyPrint _ (Constr i [])
 = i

prettyPrint True (Constr i xs)
 = prettyPrint True ((Idf i):xs)

prettyPrint True (FA [Idf f, arg])
 = prettyPrint True (Constr fshort [arg])
 where
```

```
   fshort = fromJust ( ( lookUpMb (invert prefixNames) f
                       ) $orJust f
                     )

prettyPrint True (FA [Idf f, arg0,arg1])
 = prettyPrint True farg01
 where
   farg01 = fromJust ( ( lookUpMb (invert infixNames) f   $bindMb (fshort →
                         unitMb [arg0, Idf fshort, arg1]           )
                       ) $orJust [Idf f, arg0, arg1]
                     )

prettyPrint True (FA (f:args))
 = prettyPrint True (f:args)

prettyPrint b (FunDef f vgs ws)
 = unlines (        [printF]
             $above printVgs
             $above (            ["  where"]
                       $optAbove (["     "] $optNextTo printWs)
                    )
           )
 where
   printF   = prettyPrint b f
   printVgs = prettyPrintVgs b vgs
   printWs  = aboves (map (lines . prettyPrint b) ws)

prettyPrintVgs b vgs
 = nextTos [ rep defEq (#vgs)
           , map (prettyPrint b . fst) vgs
           , guards
           ]
 where
  guards = [ guard
           | (_, g), i <- vgs, [1..]
           ; guard := if (g = ifTrue)
                         (if (#vgs = 1)
                             ""
                             (if (i = (#vgs))
                                 ", otherwise"
                                 (", if " ++ prettyPrint b g)
                             )
                         )
                         (", if " ++ prettyPrint b g)
           ]
   defEq = " = "
   ifTrue = Constr "True" []

prettyPrint b (x, y)
 = prettyPrint True x ++ " " ++ prettyPrint True y
```

```

prettyPrint b ((d=FunDef f cgs ws):ds)
 = (unlines . aboves . map ((++[" "]) . lines . prettyPrint True)) (d:ds)

prettyPrint True []
 = []

prettyPrint True (x:xs)
 = enclose "" " " "" (map (prettyPrint False) (x:xs))

prettyPrint False y
 = "(" ++ prettyPrint True y ++ ")"

prettyPrint _ x
 = toString x


|| Remove spaces at the end of each line
crop = unlines . map cropLine . lines
     where
           cropLine = reverse . (dropwhile (=' ')) . reverse


print :: [defTp] → string
print = crop . prettyPrint True
```

A.3 Introducing monads

◇ monadMb.ama ◇

```
/*
 * Maybe monad
 *
 * implementation inspired by Philip Wadler (1992), MfFP
 *
 * 11-2005   Nikolaas N. Oosterhof
 */

maybe * ::= Nothing | Just *

unitMb :: * → maybe *
unitMb a = Just a

zeroMb = Nothing

bindMb :: maybe * → (* → maybe **) → maybe **
bindMb (Nothing) _ = Nothing
```

```
bindMb (Just a) f = f a

thenMb :: maybe * → maybe ** → maybe **
thenMb a b = bindMb a (_ → b)

alternativeMb (Just a)  _ = Just a
alternativeMb (Nothing) b = b

alternativesMb = foldl alternativeMb Nothing

guardMb False _ = Nothing
guardMb True  a = a

hdMb [] = Nothing
hdMb (a:as) = Just a

tlMb [] = Nothing
tlMb (a:as) = Just as

hdtlMb [] = Nothing
hdtlMb (a:as) = Just (a, as)



lastMb [] = Nothing
lastMb [x]    = Just x
lastMb (_:xs) = lastMb xs

frontMb [] = Nothing
frontMb [x] = Just []
frontMb (x:xs) = frontMb xs $bindMb (frontxs → unitMb (x:frontxs))


fromJust (Just x) = x
fromJust _        = error "fromJust"

fromJusts = map fromJust . filter (≠Nothing)

orJust a b = a $alternativeMb (unitMb b)

isNothing Nothing = True
isNothing _       = False

areNothing = and . map isNothing

lookUpMb []           _   = Nothing
lookUpMb ((x,y):xys) key = ((x=key) $guardMb unitMb y)
                            $alternativeMb lookUpMb xys key

lookMbUpMb xs key = key                            $bindMb (x →
```

```
                    hdMb (filter ((=x).fst) xs)  $bindMb ( (_, y) →
                    unitMb y                              ))
```

◇ monadSt.ama ◇

```
/*
 * State-transition monad
 *
 * implementation inspired by Philip Wadler (1992), MfFP
 *
 * 11-2005   Nikolaas N. Oosterhof
 */

stateT * ** == * → (**, *)

unitSt :: ** → stateT * **
unitSt x st = (x, st)

bindSt :: stateT * ** → (** → stateT * ***) → stateT * ***
bindSt p q s
 = q x s1
 where (x, s1) = p s

thenSt p q
 = p $bindSt (_ → q)

getSt :: stateT * **
getSt st = (st, st)

setSt st _ = (st, st)
updSt f = getSt         $bindSt (s  →
          unitSt (f s) $bindSt (s1 →
          setSt s1               ))

traverse :: (** → stateT * ***) → [**] → stateT * [***]
traverse f []     st = ([], st)
traverse f (x:xs) st = (x1:xs2, st2)
                     where
                       (x1,  st1) = f x st
                       (xs2, st2) = traverse f xs st1
```

◇ monadSts.ama ◇

```
/*
 * State-transition-with-choice monad
 *
```

```
 * implementation inspired by Philip Wadler (1992), MfFP
 *
 * 11-2005   Nikolaas N. Oosterhof
 */


stateSts * ** == * → [(**, *)]

unitSts :: ** → stateSts * **
unitSts x st = [(x, st)]

zeroSts x = []

bindSts :: stateSts * ** → (** → stateSts * ***) → stateSts * ***
bindSts p q st0
 = [ (y, st2) | (x, st1) <- p st0; (y, st2) <- q x st1]

thenSts :: stateSts * ** → stateSts * *** → stateSts * ***
thenSts p q = bindSts p (_ → q)

getStateSts st = unitSts st st

updStateSts f st0 = setStateSts (f st0) st0

setStateSts st _ = unitSts st st


orSts :: stateSts * ** → stateSts * ** → stateSts * **
orSts p q st0 = p st0 ++ q st0

biasedOrSts p q st0 = q st0, if pst0 = []
                    = pst0 , otherwise
                    where pst0 = p st0

orsSts       = foldr orSts       zeroSts
biasedOrsSts = foldr biasedOrSts zeroSts

orUnitSts p altval = biasedOrSts p (unitSts altval)


checkSts p check = p $bindSts (x → if (check x) (unitSts x) (zeroSts))

guardSts check p = checkSts p check

|| parse smallest list "p p ... p q"
untilSts p q = (q)
               $biasedOrSts
               ( p            $bindSts (x  →
                 untilSts p q  $bindSts (xs →
                 unitSts (x : xs )      ))
```

```
              )


|| parse smalles list "p p ... p q", return pair (p's, q)
upToSts p q = (q $bindSts (y → unitSts ([], y)))
              $biasedOrSts
              ( p             $bindSts (x        →
                upToSts p q   $bindSts ((xs,y) →
                unitSts (x:xs, y)        ))
              )

|| parse  "p q p ... p q p"
encloseSts p q = p               $bindSts (x →
                   ( q               $bindSts (y  →
                     encloseSts p q $bindSts (xs →
                     unitSts (x:y:xs)        ))
                   ) $orUnitSts [x]     )

|| parse "p q p ... p q p", but leave out the q's in the result

separateSts p q = p              $bindSts (x →
                  ( q              $thenSts (
                    separateSts p q $bindSts (xs →
                    unitSts (x:xs)        ))
                  ) $orUnitSts [x]     )


|| parse p0 p1 ... pN

seqSts []     = unitSts []
seqSts (p:ps) = p          $bindSts (x  →
               (seqSts ps) $bindSts (xs →
               unitSts (x:xs)       ))

doSts p f = p $bindSts (x → (unitSts (f x)))

foldrSts f zero p = (p                  $bindSts (x →
                     foldrSts f zero p $bindSts (xs →
                     unitSts (f x xs))           ))
                   $biasedOrSts (unitSts zero)


zeroOrMoreSts p = oneOrMoreSts p
                 $biasedOrSts (unitSts [])

|| parse longest list p^+

oneOrMoreSts p = p               $bindSts (x  →
                  ( oneOrMoreSts p  $bindSts (xs →
                    unitSts (x:xs)            )
```

```
                   ) $biasedOrSts
                   ( unitSts [x]
                   )                      )




ifSts = if

|| altStarSts parsers stopparser
altStarSts ps q
 = ( q $thenSts (unitSts [])
   ) $biasedOrSts
   ( biasedOrsSts ps $bindSts (x  →
     altStarSts ps q $bindSts (xs →
     unitSts (x:xs)           ))
   )

valsSts = map fst
```

◇ monadId.ama ◇

```
/*
 * The trivial identify monad
 */

bindId x f = f x
unitId x = x
```

A.4 Some utility functions

◇ util.ama ◇

```
/*
 * Utility functions
 *
 * 11-2005   Nikolaas N. Oosterhof
 */

string == [char]

lookUp []          key = error ("Lookup: not found key " ++ toString key)
lookUp ((x, y):ys) key = y              , if x = key
                       = lookUp ys key , otherwise

```

```
lookUpAll []           key = []
lookUpAll ((x, y):ys) key = y : more, if x = key
                          = more    , otherwise
                          where
                            more = lookUpAll ys key


doMapping []           key f = []
doMapping ((x, y):ys) key f = (x, f y) : more, if x = key
                            = (x, y)   : more, otherwise
                        where
                          more = doMapping ys key f

update []          key z = [(key, z)]
update ((x, y):ys) key z = (x, y) : update ys key z, if x ≠ key
                         = (x, z) : ys    , otherwise



unitMapping x y = [(x, y)]

joinMapping xs ys = [ (key, (nodup . concat) (lookUpAll (xs ++ ys) key)) | key <- (nodup . opPair (++) . mapPair domain) (xs, ys) ]

joinMappings = foldr joinMapping []


filterDomain f xys = [ (x, y) | (x, y) <- xys; f x ]

filterRange f xys = [ (x, y) | (x, y) <- xys; f y ]

mapMapping f xs = [ (x, f y) | (x, y) <- xs]

appendMapping xs key s = doMapping xs key (++s)

domain = map fst
range = map snd

invert = map swapPair
       where swapPair (x, y) = (y, x)

concatmap f = concat . (map f)



front [a] = []
front (a:as) = a : front as

last [a] = a
last (_:as) = last as
```

```


nand a b = ~(a ∧ b)
nor  a b = ~(a \/ b)

id x = x


asSet = sort . nodup

cup x y = asSet (x ++ y)
cap x y = xy -- ( (xy -- x) $cup (xy -- y) )
        where
          xy = x $cup y

nilPair = ([], [])
mapPair f (a, b) = (f a, f b)
opPair  f (a, b) = a $f b
joinPair (a, b) (c, d) = (a ++ c, b ++ d)
concatPair = foldr joinPair nilPair

mapFst f (a, b) = (f a, b)
mapSnd f (a, b) = (a, f b)


foldrPair (f, g) (a, b) xys = (foldr f a xs, foldr g b ys)
                              where (xs, ys) = unzip xys



|| 'cross-product' of list of lists, using lists instead of tuples
|| combinations [[1,2], [3,4,5], [6]] === [[1, 3, 6], [1, 4, 6], [1, 5, 6], [2,
   3, 6], [2, 4, 6], [2, 5, 6]]
combinations [a]    = [ [x] | x <- a]
combinations (a:as) = [ x : y | x <- a ; y <- combinations as]
combinations []     = []

unitC [x]    = [[c] | c <- x]
unitC (x:xs) = [ i : j | i <- x; j <- unitC xs ]

bindC [x]    f = [ [f i] | i <- x]
bindC (x:xs) f = [ f i : j | i <- x; j <- bindC xs f]

unzip :: [(*, **)] → ([*], [**])
unzip []           = ([], [])
unzip ((x,y):xys) = (x:xs, y:ys)
                    where (xs, ys) = unzip xys


sqnc []     x = x
```

```
sqnc (f:fs) x = sqnc fs (f x)

generic size :: * → num
size (n=num) = 1
size (c=char) = 1
size (b=bool) = 1
size [] = 0
size (x:xs) = size x + size xs
size (x, y) = size x + size y

generic enclose :: [char] → [char] → [char] → * → [char]
enclose left mid right []     = left ++ right
enclose left mid right [x]    = left ++ toString x ++ right
enclose left mid right (x:xs) = left ++ toString x ++ concatmap ((mid++).toString) xs ++ right


generic toString :: * → [char]
toString (n=num)        = itoa n
toString (b=bool)       = if b "True" "False"
toString ((c=char):cs) = c:cs
toString (c=char)       = ['\'', c, '\'']
toString []             = "[]"
toString (x:xs)         = enclose "[" ", " "]" (x:xs)
toString (x, y)         = concat ["(", toString x,", ", toString y, ")"]
toString (x, y, z)      = concat ["(", toString x,", ", toString y, ",", toString z, ")"]

idxs xs = [ i | i, j <- nats 0, xs]

pam []     _ = []
pam (f:fs) x = f x : pam fs x

biasedJoin [] y = y
biasedJoin x  _ = x


generic equals :: * → * → bool
equals (x=bool) y = x = y
equals (x=num)  y = x = y
equals (x=char) y = x = y
equals [] []      = True
equals [] _       = False
equals _ []       = False
equals (x:xs) (y:ys) = equals x y ∧ equals xs ys

pipe []     _  = []
pipe (f:fs) s = out : pipe fs s2
               where (out, s2) = f s
```

```
limit (x:y:ys) = x, if x = y
               = limit (y:ys), otherwise

limit [x]    = x



forAll xs f = and (map f xs)

limiterate f = limit . (iterate f)

limiterates [] x = x
limiterates (ffs=(f:fs)) x
 = x                    , if lims = x
 = limiterates ffs lims, otherwise
 where
   lims = limiterates fs (limiterate f x)

pairAsSet = mapPair (sort . nodup . filter ((~) . empty))

fromList is xs = [xs!i | i <- is]

case x f = f x

splitOn :: (* → bool) → [*] → ([*], [*])
splitOn f []     = nilPair
splitOn f (x:xs) = px $joinPair splitOn f xs
                 where
                   px = ([x], [] ), if f x
                      = ([] , [x]), otherwise

|| partitions a list of elements
|| based on an equivalence relation
partition :: (* → * → bool) → [*] → [[*]]
partition f []
 = []

partition f (x:xs)
 = (x:alike) : partition f different
 where
   (alike, different) = splitOn (f x) xs

map2 :: (* → ** → ***) → [*] → [**] → [***]
map2 f xs ys = [ f x y | (x, y) <- zip2 xs ys ]


mergeSortWith f [] = []
mergeSortWith f [x] = [x]
mergeSortWith f xs = (joinListsWith f . mapPair (mergeSortWith f) . splitList) xs
                   where mapPair f (x, y) = (f x, f y)
```

```
splitList xs = split (#xs/2) xs


joinListsWith _ (xs, [])          = xs
joinListsWith _ ([], ys)          = ys
joinListsWith f ((x:xs), (y:ys)) = x : joinListsWith f (xs, y:ys), if f x y
                                  = y : joinListsWith f (x:xs, ys), otherwise

listElemDo f g []     = []
listElemDo f g (x:xs) = g x ++ listElemDo f g xs, if f x
                      =   x :  listElemDo f g xs, otherwise


transStep xs = nodup (xs ++ [(x, z) | (x, y1) <- xs; (y2, z) <- xs; y1 = y2 ])

transClose = limiterate transStep
```

A.5 Creative Commons Attribution-ShareAlike License Version 2.0

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

    1. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;
    2. to create and reproduce Derivative Works;
    3. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;
    4. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.
    5. For the avoidance of doubt, where the work is a musical composition:
        1. Performance Royalties Under Blanket Licenses. Licensor waives the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work.
        2. Mechanical Rights and Statutory Royalties. Licensor waives the exclusive right to collect, whether individually or via a music rights society or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).
    6. Webcasting Rights and Statutory Royalties. For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

    The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

    1. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not

require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any reference to such Licensor or the Original Author, as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any reference to such Licensor or the Original Author, as requested.

2. You may distribute, publicly display, publicly perform, or publicly digitally perform a Derivative Work only under the terms of this License, a later version of this License with the same License Elements as this License, or a Creative Commons iCommons license that contains the same License Elements as this License (e.g. Attribution-ShareAlike 2.0 Japan). You must include a copy of, or the Uniform Resource Identifier for, this License or other license specified in the previous sentence with every copy or phonorecord of each Derivative Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Derivative Works that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder, and You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Derivative Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Derivative Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Derivative Work itself to be made subject to the terms of this License.

3. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Derivative Works or Collective Works, You must keep intact all copyright notices for the Work and give the Original Author credit reasonable to the medium or means You are utilizing by conveying the name (or pseudonym if applicable) of the Original Author if supplied; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.

5. Representations, Warranties and Disclaimer
UNLESS OTHERWISE AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE MATERIALS, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY

TO YOU.

6. Limitation on Liability.
EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

    1. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works or Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

    2. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

    1. Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

    2. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

    3. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

    4. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

    5. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.