

Denotation by Transformation

Towards Obtaining a Denotational Semantics by Transformation to Point-free Style

Bernd Braßel and Jan Christiansen

Institute of Computer Science
University of Kiel, 24098 Kiel, Germany
{bbr,jac}@informatik.uni-kiel.de

Abstract. It has often been observed that a *point-free style* of programming provides a more abstract view on programs. We aim to use the gain in abstraction to obtain a denotational semantics for *functional logic languages* in a straightforward way. We propose a set of basic operations based on which arbitrary functional logic programs can be transformed to point-free programs. Surprisingly, the additional features of functional logic languages do require *less* basic operations to obtain point-free programs than known approaches for functional languages. This effect is mostly due to employing so called *function patterns*.

We interpret the basic operations in *relation algebra* to obtain a denotational semantics for the whole point-free subset of functional logic languages. As this subset is connected to the whole language by the proposed transformation this enables a purely algebraic view on the whole language.

A final example illustrates the additional possibilities gained by this approach.

1 Introduction

This work aims at combining the results of several well researched fields. Most notably, these are the fields of *declarative programming* and *relation algebra*. Moreover, the importance of a *point-free* view on *programming* has been emphasized particularly in the applications of category theory to semantics of programming languages. We expect the combination of these fields to be very fruitful. Many results and techniques from the field of relation algebra could be used for the analysis of functional logic programs and the knowledge about the implementation of functional logic languages could be employed to concisely solve relation-algebraic problems. As a concrete example motivating this approach in Section 3 we use algebraic calculations to optimise an operator definition. Due to space reasons the semantics proposed in Section 3 is strict, whereas the transformation developed in Section 2 takes laziness into account. In order to cover laziness one needs to extent the domain of values and to redefine the transposition of a relation as we want to show in future work.

1.1 Functional Logic Languages

We consider a functional logic program as a constructor-based rewriting system, allowing extra variables on the right hand side and so called *function patterns*. This section establishes some of the involved notation, referring to [?] for functional logic programming and [?] for function patterns. For our examples we adopt the syntax of Curry [?].

For a program P , Σ_P is a signature partitioned into two sets, the set of *constructors* \mathcal{C}_P and the set of defined *operations* \mathcal{O}_P . We denote n -ary constructor (operation) symbols by c^n (f^n, g^n) omitting the arity where it is apparent. For a set of (sorted) variables \mathcal{X} , the sets of (well-sorted) *terms* and *constructor terms* are denoted by $\mathcal{T}(\Sigma_P, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}_P, \mathcal{X})$, respectively. The function $var(t)$ yields the set of variables occurring in term t . A term is *linear* if every variable occurs at most once. Sorts and constructors are introduced by a **data** declaration, as shown in Example (1). The “a” in the third declaration denotes that

<code>data Success = Success</code>	[a] is a polymorphic type. We use syntac-
<code>data Bool = True False</code>	tic sugar for list terms, e.g., [True,False]
<code>data [a] = [] a : [a]</code>	instead of (True : (False : [])). Operations
	are defined by <i>rewrite rules</i> of the form

“ $f p_1 \dots p_n = e$ ” where $f^n \in \mathcal{O}_P$ and p_1, \dots, p_n are called *patterns*. The standard way to define an operation in Curry is that each pattern of the rewrite rules is a *constructor term* and each variable occurs not more than once in the whole pattern. In other words, the term (p_1, \dots, p_n) must be a *linear constructor term*. Such standard Curry programs are evaluated by *weakly needed narrowing* [?]. A *narrowing step* is a rewrite step combined with substitutions for extra variables needed to match the left-hand side of an applicable rule. E.g., a narrowing step for Example (2) is `app x [True] → {x ↦ []} [True]`.

<code>app [] ys = ys</code>	In addition to defining rules, <i>type sig-</i>
<code>app (x:xs) ys = x : (app xs ys)</code>	(2) <i>natures</i> are used to declare the sorts
	an operation is defined for. As an ex-

ample, `app :: [a] -> [a] -> [a]` declares that `app` is an operation which maps two (polymorphic) lists to a list. These lists have elements of the same type.

As in Example (2) there might be more than one possible narrowing step. Functional logic languages provide *non-deterministic search* to obtain values in this situation. Non-determinism does not only stem from narrowing but also from operator definitions with overlapping left hand sides.

<code>coin :: Bool</code>	E.g., there are two derivations <code>coin → True</code> and <code>coin →</code>
<code>coin = True</code>	(3) <code>False</code> or, for short, <code>coin → True False</code> . The operation
<code>coin = False</code>	<code>coin</code> is very popular because it can be used to exemplify an
	important feature of functional logic languages: <i>call-time</i>

choice, cf. [?]. With call-time choice non-deterministic choices for arguments are done before application, at least conceptually. In combination with laziness call-time choice is affine to the concept of referential transparency, as illustrated in the next example.

`or :: Bool -> Bool -> Bool`
`or x y = if x then x else y` (4)

For example, employing call-time choice the expression $e := \text{or coin True}$ is evaluated to `True` only, because both occurrences of `x` are substituted with the same value. That is, employing call-time choice there are the derivations $e \rightarrow \text{if True then True else True} \rightarrow \text{True}$ and $e \rightarrow \text{if False then False else True} \rightarrow \text{True}$. In the dual conception, run-time choice, e is reduced to $\text{if True then coin else True} \rightarrow \text{coin} \rightarrow \text{True} \mid \text{False}$ and $\text{if False then coin else True} \rightarrow \text{True}$. That is, the evaluation of e yields non-deterministically `True` or `False`. We have to make sure to capture call-time choice accordingly in Section 3.

An important operation in functional logic languages is the *strict equality* ($=:=$) :: `a -> a -> Success`. The intended meaning is that the equation $e_1 := e_2$ is satisfied iff e_1 and e_2 can be reduced to the same constructor term, see [?] for a detailed discussion. In Curry, satisfying a predicate like the above equation is modelled by a reduction to the special type `Success`, cf. Program (1).

Strict equality can be employed to allow a certain type of non-standard operator definitions. A non-linear left hand side of a rewrite rule $l = e$ with x occurring n times in l can be taken as syntactic sugar for a rule where x is replaced by different variables $x_1 \dots x_n$ in l and e is extended by constraints $(x_1 := x_2), \dots, (x_1 := x_n)$. See [?, 4.1] for a discussion of this transformation.

Finally, *function patterns* [?] allow operator definitions with arbitrary first order patterns.¹ The intended meaning of a function pattern is that only the pattern is evaluated to a constructor term. The argument is evaluated until a unification is possible. Unlike extra variables unified with strict equality ($=:=$) this unification may bind a pattern variable to an unevaluated term. Non-linear patterns are still treated with ($=:=$) as described above.

`last (app xs [x]) = x` (5)

The operation `last` yields the last element of a given list. We apply `last` (5) to the list `[True, False]`, that is, we evaluate the term $e := \text{last [True, False]}$. We get the following reduction:

$$\text{app xs [x]} \rightarrow \{\text{xs} \mapsto \text{y} : \text{ys}\} \text{y} : (\text{app ys [x]}) \rightarrow \{\text{ys} \mapsto []\} [\text{y}, \text{x}] \quad (6)$$

$[\text{y}, \text{x}]$ can be unified with `[True, False]` yielding $e \rightarrow \{\text{xs} \mapsto [\text{True}], \text{x} \mapsto \text{False}\} \text{False}$.

1.2 Point-free Style

The term *point-free* originates from topology where you have points in a space and functions that operate on these points. In functional programming spaces are types, functions are functions and points are the arguments of a function. In *point-free* style you do not explicitly access the points, that is, the arguments of a function. The idea of the *point-free* programming paradigm is to build functions by combining simpler ones. It was introduced by John Backus in his Turing

¹ We will see in the Section 2.2 that the restriction imposed on function patterns in [?] is not necessary to obtain a reasonable semantics for such patterns. Therefore, we will omit it for simplicity.

Award Lecture in 1977 [?]. The counterpart of *point-free* is *point-wise*, that is, functions that explicitly access their arguments. Here, *point-free* programs are based on a couple of *point-wise* primitives.

1.3 Relation Algebra

In this section we present the relation-algebraic basics for Section 3. For a detailed introduction to relation algebra see [?]. Relation algebras can also be defined as a special kind of categories, first in [?].

In this paper we make use of the so called *concrete* relation algebra, in which relations are represented as sets of Cartesian products. We denote the set of all relations with domain X and range Y by $[X \leftrightarrow Y]$. We write $R :: X \leftrightarrow Y$ instead of $R \in [X \leftrightarrow Y]$ and xRy instead of $(x, y) \in R$ and call $X \leftrightarrow Y$ the type of R .

Lattice For each set $[X \leftrightarrow Y]$ the operations \cup (union, join), \cap (intersection, meet), and $\bar{\cdot}$ (complement, negation) together with a greatest element \mathbf{L} (universal relation) and a smallest element \mathbf{O} (empty relation) form a boolean lattice. A lattice provides an induced order denoted by \subseteq .

Union, intersection and complement of the concrete relation algebra are the standard union, intersection and complement operations on sets. The empty relation is the empty set and the universal relation $\mathbf{L} :: X \leftrightarrow Y$ is $X \times Y$. The lattice order is the subset relation. The relation algebra enriches the lattice with two operations \circ (multiplication) and \cdot^\top (inversion, transposition) and a constant \mathbf{I} (identity relation).

The Multiplication of relation algebra is a binary associative operation for which the identity relation \mathbf{I} is the neutral element. For each set X , $\mathbf{I} :: X \leftrightarrow X$ is defined by $x\mathbf{I}x$ for all $x \in X$. The multiplication of two relations $R :: X \leftrightarrow Y$ and $S :: Y \leftrightarrow Z$ is defined by $x(R \circ S)z$ iff there exists $y \in Y$ such that xRy and ySz . It is $R \circ S :: X \leftrightarrow Z$.

The Inversion of a relation $R :: X \leftrightarrow Y$ is defined by $yR^\top x \Leftrightarrow xRy$. It is $R^\top :: Y \leftrightarrow X$. In Section 3 we use some properties of inversion: $(R^\top)^\top = R$, $(R \circ S)^\top = S^\top \circ R^\top$ and $(R \cup S)^\top = R^\top \cup S^\top$.

Direct Products Given a product $X \times Y$ there are two projections which map a pair (u_1, u_2) to its first component u_1 and second component u_2 , respectively. We consider the corresponding projection relations $\pi_1 :: X \times Y \leftrightarrow X$ and $\pi_2 :: X \times Y \leftrightarrow Y$ such that $u\pi_1x \Leftrightarrow x = u_1$ and $u\pi_2y \Leftrightarrow y = u_2$ for $u = (u_1, u_2)$.

The tupling $[\cdot, \cdot]$ of two relations $R :: X \leftrightarrow Y$ and $S :: X \leftrightarrow Z$ is defined by $x[R, S](y, z) \Leftrightarrow xRy \wedge xSz$ and its type is $X \leftrightarrow (Y, Z)$. The parallel composition $\cdot \parallel \cdot$ of two relations $R :: X \leftrightarrow Z$ and $S :: Y \leftrightarrow W$ is defined by $(x, y)(R \parallel S)(z, w) \Leftrightarrow xRz \wedge ySw$ and its type is $(X, Y) \leftrightarrow (Z, W)$. Tupling and the projections are connected by the following properties:

$$S \text{ total} \Rightarrow [R, S] \circ \pi_1 = R \qquad R \text{ total} \Rightarrow [R, S] \circ \pi_2 = S$$

Direct Sums are introduced as a dual concept to direct products. Sums are constructed by the *injections* ι_1 and ι_2 which can be used to define the semantics of constructors.

2 Transformation to Point-free Style

2.1 The Set of “Primitives”

In this section we define a small set of point-wise operations which allow the definition of arbitrary functional logic operations in a point-free style.

Composition of Operations The first such “primitive” is *sequential composition*, occasionally simply referred to as “composition”.

$$\begin{aligned}
 (*) &:: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow a \rightarrow c & (7) & \quad \boxed{f} \text{---} \boxed{g} \\
 (f * g) x &= g (f x)
 \end{aligned}$$

The primitive $(*)$ is a flipped version of $(.)$. Whereas $(f . g)$ reads as “f after g”, $(f * g)$ is more like “f before g”. This is more convenient with regard to our aim of a relation-algebraic treatment of programming semantics. Furthermore, the left-to-right reading provides a very descriptive graphical representation. The composition is visualised by connecting two operations with a line, indicating that the output of one is the input of the other. Such visualisations were also used in connecting functional programs [?] and allegory theory with hardware design [?] and to describe physical structures in general [?]. Simple definitions can be made point-free by using sequential composition, cf. Example (8).

$$\begin{aligned}
 \text{involution } x &= \text{not } (\text{not } x) & (8) & \quad \boxed{\text{not}} \text{---} \boxed{\text{not}} \\
 \text{involution} &= \text{not } * \text{ not}
 \end{aligned}$$

Operations with several arguments are composed by *parallel composition*.

$$\begin{aligned}
 (/) &:: (a \rightarrow c) \rightarrow (b \rightarrow d) \rightarrow (a,b) \rightarrow (c,d) & (9) & \quad \begin{array}{c} \boxed{f} \\ \boxed{g} \end{array} \\
 (f / g) (x,y) &= (f x, g y)
 \end{aligned}$$

Example (10) illustrates the use of parallel composition. The Operation $(*)$ has a higher precedence than $(/)$ making the parenthesis necessary.

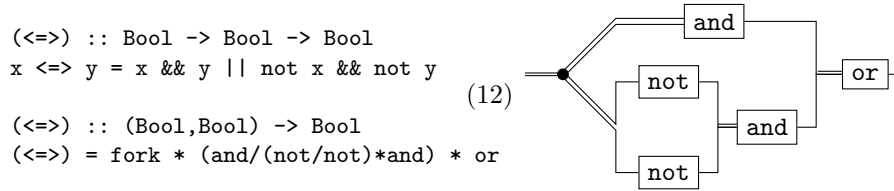
$$\begin{aligned}
 \text{nor} &:: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \\
 \text{nor } x \ y &= \text{not } x \ \&\& \ \text{not } y & (10) & \quad \begin{array}{c} \boxed{\text{not}} \\ \boxed{\text{not}} \end{array} \text{---} \boxed{\text{and}} \\
 \text{nor} &:: (\text{Bool}, \text{Bool}) \rightarrow \text{Bool} \\
 \text{nor} &= (\text{not} / \text{not}) * \text{and}
 \end{aligned}$$

Discretely, we have changed the type of `nor` to a so called “uncurried” version. We use curried operations only when higher order is employed, as discussed in Paragraph “Higher Order”.

Interface Adaption So far, we can express only right linear rules. Sharing arguments is the first of the primitives dealing with what we call “interface adaption”. Interface adaption means that the connectives of two operations have to be copied, joined or reordered in some way. An uncurried and point-free version of the boolean operator “if and only if” (12) can be formulated using $(/)$ and `fork`.

$$\begin{aligned}
 \text{fork} &:: a \rightarrow (a,a) & (11) & \quad \bullet \\
 \text{fork } x &= (x,x)
 \end{aligned}$$

arguments is the first of the primitives dealing with what we call “interface adaption”. Interface adaption means that the connectives of two operations have to be copied, joined or reordered in some way. An uncurried and point-free version of the boolean operator “if and only if” (12) can be formulated using $(/)$ and `fork`.



There are two more primitives for interface adaption. The operator `unit` to “discard a value” and the identity `id` to “pass a value on”. Both are exemplified in the following sections.

```

unit :: _ -> () (13)  ─┤
unit _ = ()

```

```

id :: a -> a (14)  ─●─
id x = x

```

Data Structures, Inversion and Pattern Matching We do not wish to abstract from concrete domains at this point. Later in Section 3, we treat data structures in the standard way of sums and products. Here we define different operations to construct data. Each constructor of the original program will be assigned one operation.

```

nil :: () -> [a]
nil () = []
cons :: (a,[a]) -> [a]
cons (x,xs) = x : xs

```

```

true, false :: () -> Bool
true () = True
false () = False

```

(15)

Note that these operations are again uncurried and that the constants `True`, `False`, and `[]` are extended with an argument. The reason for the latter extension will become apparent soon.

What we have seen so far is a more or less standard treatment of expressing functional programs in a point-free style. To concisely express pattern matching and to combine several rules we employ two additional features of functional *logic* programming, i.e., non-determinism and function patterns.

```

(?) :: a -> a -> a
x ? _ = x
_ ? y = y

```

(16)

```

coin :: () -> Bool
coin = true ? false

```

(17)

As stated in the introduction, overlapping rules in functional logic languages lead to non-deterministic search, cf. [?]. In principal, all non-determinism can be introduced by permitting only a single operation with overlapping rules (?) (16), cf. [?]. We use (?) to combine the rules of a function, cf. Example (17). Note that the introduction of the argument () for constant constructors extends to all definitions of constants.

```

invert :: (a -> b) -> (b -> a)
invert f = f' where f' (f x) = x

```

(18)

Function patterns can be used to invert arbitrary operations. This yields the primitive `invert` defined in (18).

The semantics of function patterns are described in [?] in terms of an possibly infinite set of rewrite rules. We aim at giving a denotational semantics for function patterns for the first time.

The expressive power of function patterns can be estimated by considering that all other logic features can be obtained by using function patterns.

E.g., `invert unit :: () -> a` yields a logic variable when applied to `()` and `invert fork :: (a,a) -> a` performs unification, cf. Section 1.1. Therefore we can define the following useful abbreviations for interface adaption.

$$\begin{array}{l} \text{unknown} :: () \rightarrow a \\ \text{unknown} = \text{invert unit} \end{array} \quad (19) \quad \dashv \quad \begin{array}{l} \text{join} :: (a,a) \rightarrow a \\ \text{join} = \text{invert fork} \end{array} \quad (20) \quad \begin{array}{c} \diagup \\ \bullet \\ \diagdown \end{array}$$

The operations `up` and `down` demand the discarded argument to be an empty tuple which is important for the definition of pattern matching.

$$\begin{array}{l} \text{up} :: (a,()) \rightarrow a \\ \text{up} = (\text{id} / \text{unknown}) * \text{join} \end{array} \quad (21) \quad \begin{array}{l} \text{fst} :: (a,_) \rightarrow a \\ \text{fst} = (\text{id} / \text{unit}) * \text{up} \end{array} \quad (22) \quad \begin{array}{c} \bullet \\ \text{---} \\ | \end{array}$$

$$\begin{array}{l} \text{down} :: ((),a) \rightarrow a \\ \text{down} = (\text{unknown} / \text{id}) * \text{join} \end{array} \quad (23) \quad \begin{array}{l} \text{snd} :: (_,a) \rightarrow a \\ \text{snd} = (\text{unit} / \text{id}) * \text{down} \end{array} \quad (24) \quad \begin{array}{c} | \\ \text{---} \\ \bullet \end{array}$$

The expressive power gained by function patterns is paid with computational overhead, cf. [?]. It is thus desirable to replace a function pattern by an equivalent operation without such patterns. We think that our semantics gives a base on which optimizing procedures can be developed. As a first outlook in this direction we give a derivation of a more efficient version of `last` defined in (5) in Section 3.

$$\begin{array}{l} \text{head} :: [a] \rightarrow a \\ \text{head} = \text{invert cons} * \text{fst} \end{array} \quad \begin{array}{l} \text{tail} :: [a] \rightarrow [a] \\ \text{tail} = \text{invert cons} * \text{snd} \end{array} \quad (25)$$

In addition to (?) to combine rules, the primitive `invert` can be used to express arbitrary pattern matching including function patterns. A constructor pattern is a linear constructor term, cf. Section 1.1. In order to match such a pattern we only have to invert the according constructors and then adapt the result like shown in (25). From this point of view it becomes apparent why constant constructors are extended with an argument: to make them invertible.

There is one last feature concerning pattern matching in connection with laziness. If a value is discarded, e.g., by using `unit`, it is not evaluated. The semantics of pattern matching demands that matching is ensured regardless of whether the resulting variable bindings are used or not. The operations `head` and `tail` defined in (25) use one of the variables bound by the matching and therefore the pattern matching is indeed performed. In general we have to combine several of the primitives introduced so far to achieve the desired evaluation.

$$\begin{array}{l} \text{null} :: [a] \rightarrow \text{Bool} \\ \text{null} [] = \text{True} \\ \text{null} (_:_) = \text{False} \end{array} \quad (26) \quad \begin{array}{l} \text{Example (26) shows a case in which the bind-} \\ \text{ings of the matching are discarded. The point-free} \\ \text{version has to make sure that a) the empty tuple} \\ \text{of (invert nil) and b) the pair resulting from} \end{array}$$

(`invert cons`) are demanded, and not more. The following definition provides these properties.

$$\text{null} = \text{invert nil} * \text{true} ? \text{invert cons} * (\text{unit/unit}) * \text{join} * \text{false} \quad (27)$$

The astute reader might wonder why we introduce non-determinism for a perfectly deterministic operation like the pattern matching of `null`. The reason for this is twofold. 1) From a semantic point of view the non-deterministic branching does not matter. If the matching was indeed deterministic, for a given de-

terministic value all but one branch will finitely (even immediately) fail. 2) In a functional logic language patterns are *not* always deterministic nor treated in a sequential way (like in Haskell). Overlapping patterns induce non-determinism which is easily captured by our approach.

```

member :: [a] -> a
member (x:_) = x
member (_:xs) = member xs

```

(28) For example, the operation `member` defined in (28) non-deterministically relates a list with each of its elements. Without further additions this behaviour is captured by the transformation. The following definition shows a point-free version of `member`.

```
member = invert cons * fst ? invert cons * snd * member (29)
```

Example (28) also illustrates that recursive functions simply stay recursive. There is no need for changes, e.g., a special recursion operator. Complex patterns are treated like complex expressions, i.e., they are composed with `(*)` and `(/)` before inverting the whole expressions. We treat function patterns in the very same way. For example, the function `last` (5) is translated to:

```
last = invert ((id / (id/nil) * cons) * app) * snd * up (30)
```

Higher Order In order to introduce higher-order operations we need to adapt the well known pair `apply` and `curry` to our setting. A first point to consider is

```

apply :: (a -> b, a) -> b
apply (f, x) = f x

```

(31) that values of type `a` correspond to operations of type `() -> a`. Because higher-order operations should be first class objects we need to

translate them in the same way. An operation of type `(a -> b)` must become an object of type `() -> (a -> b)` when used as an argument of an operation. If we assume this kind of translation we can define `apply` and `curry` straightforward.

```

curry :: (() -> (a, b) -> c) -> () -> (a -> b -> c)
curry f = \ () x y -> f () (x, y)

```

(32)

```

map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs

```

(33) The step to obtain the first curried version of a given function cannot be formulated in an equally general way because of call-time choice. This is illus-

trated by a standard example of a higher-order operation in Example (33). We can already translate `map` with the primitives introduced so far, adding `apply`.

```

map :: (a -> b, [a]) -> [b]
map = invert (id / nil) * nil
      ? invert (id / cons) * adapt * ((apply / map) * cons)

```

(34)

We assume `adapt` to map the tuple structure `(f, (x, xs))` to `((f, x), (f, xs))`. We omit its concrete definition by means of `id`, `unit`, `invert` and `fork`. We want to map the operation `not` on the list `[False, True]`.

```

not = invert true * false ? invert false * true
listFalseTrue = fork * (false / fork * (true / nil) * cons) * cons
mapNot = fork * (curryNot / listFalseTrue) * map

```

(35)

What should `curryNot :: () -> (Bool -> Bool)` be defined as? A first version

might be `curryNot = const not`. But evaluating `mapNot ()` yields no solution. The reason is call-time choice. Because `f` is a variable the choice whether `f` is the operation “invert true * false” or “invert false * true” is made consistently for all applications of `f`. But this decision has to be made anew for each application of `f`. This can be achieved by η -expansion.

`curryNot () x = not x` (36) Using definition (36) (`mapNot ()`) evaluates to `[True,False]` as intended. The example shows that a second version of each operation which will be applied higher order is needed. We have now illustrated all the point-wise primitives necessary to translate arbitrary Curry programs: `(*)` (7), `(/)` (9), `fork` (11), `unit` (13), `id` (14), `(?)` (16), `invert`(18), `apply` (31) and `curry` (32). The next section presents a formal definition of the transformation.

2.2 Obtaining Point-free Style in General

Modern functional logic languages provide syntactic sugar to formulate very concise and readable code, e.g., the `if · then · else` used in Example (4). In the following, we will consider the core language defined in Figure 1. We distinguishing *partial* and *full* applications in order to model higher order. For example, the Curry expression `map not []` is represented as `(map (PC not) [])`. For our programs, we assume sort correctness. Furthermore, we assume that the source program contains no unary operations nor unary constructors in order to avoid the concept of an “unary tuple”. Figure 2 shows how expressions are trans-

$P ::= R^*$		{program}
$R ::= f^n p_1 \dots p_n = e$	$f \in \mathcal{O}_P$	{rule}
$p ::= x$	$x \in \mathcal{X}$	{(pattern) variable}
$(s^n p_1 \dots p_n)$	$s \in \Sigma_P$	{complex pattern}
$e ::= x$	$x \in \mathcal{X}$	{variable}
$(s^n e_1 \dots e_n)$	$s \in \Sigma_P$	{full application}
$(PC s^n)$	$s \in \Sigma_P$	{partial application}
$(AP e_1 e_2)$		{higher order application}

Fig. 1. The Core Language

$exp(x)$	=	$(PC\ id)$
$exp(s^0)$	=	$(PC\ s^1)$
$exp(s^n e_1 \dots e_n)$	=	$(exp(e_1)/\dots/exp(e_n))* (PC\ s^n) \ n > 1$
$exp(PC\ s^n)$	=	$\underbrace{curry(\dots (curry\ cs)\dots)}_{n-1}$
$exp(AP\ e_1\ e_2)$	=	$(exp(e_1)/exp(e_2))* (PC\ apply)$

Fig. 2. Transforming Expressions and Patterns

lated. Note that constants are replaced by unary symbols of the same name, cf. the discussion of Examples (15) and (25) in the previous section. Furthermore, the partial application of `s` is replaced by an application of a new symbol `cs` as motivated in paragraph “Higher Order” above. The operation `cs` is defined in

Figure 7. Because patterns are effectively a subset of expressions, the rules of Figure 2 are also used to translate patterns.

$$\begin{aligned}
\mathit{int}(x) &= x \\
\mathit{int}(s\ e_1\ e_2\ \dots\ e_n) &= (\mathit{int}(e_1), (\mathit{int}(e_2), (\dots, \mathit{int}(e_n)) \dots)) \\
\mathit{int}(\mathbf{PC}\ s^n) &= () \\
\mathit{int}(\mathbf{AP}\ e_1\ e_2) &= (\mathit{int}(e_1), \mathit{int}(e_2))
\end{aligned}$$

Fig. 3. Obtaining Interfaces

Next we present the general approach to “Interface Adaption”, cf. the paragraph of the same name above. An *interface* is an abstraction from the actual structure of an expression. That is, it is a tree with the same branching structure as the original expression which contains variables that occur in the expression. The simple mapping from expressions to interfaces is depicted in Figure 3. We denote complex interfaces, i.e. those not in $\mathcal{X} \cup \{()\}$ by $i, i_1, i_2 \dots$ and by $\mathit{var}(i)$ the set of all variables occurring in i .

$$\begin{aligned}
\mathit{sel}(x, ()) &= (\mathbf{PC}\ \mathit{id}) \\
\mathit{sel}(x, y) &= \begin{cases} (\mathbf{PC}\ \mathit{id}) & , \text{ if } x = y \\ (\mathbf{PC}\ \mathit{unit}) & , \text{ otherwise} \end{cases} \\
\mathit{sel}(x, (i, i')) &= (\mathit{sel}(x, i) / \mathit{sel}(x, i')) * \left\{ \begin{array}{l} ((\mathbf{PC}\ \mathit{id}) / u) * , \text{ if } x \in \mathit{var}(i) \wedge x \notin \mathit{var}(i') \\ (u / (\mathbf{PC}\ \mathit{id})) * , \text{ if } x \notin \mathit{var}(i) \wedge x \in \mathit{var}(i') \\ , \text{ otherwise} \end{array} \right\} j. \\
u = (\mathbf{invert}\ (\mathbf{PC}\ \mathit{unit})) & \quad j = (\mathbf{invert}\ (\mathbf{PC}\ \mathit{fork})) \\
\mathit{adapt}(i, ()) &= \mathit{sel}(y, i) \text{ where } y \notin \mathit{var}(i) \\
\mathit{adapt}(i, x) &= \mathit{sel}(x, i) \\
\mathit{adapt}(i, (i_1, i_2)) &= (\mathbf{PC}\ \mathit{fork}) * (\mathit{adapt}(i, i_1) / \mathit{adapt}(i, i_2))
\end{aligned}$$

Fig. 4. Variable Selection and Interface Adaption

Selecting variables from a given interface and adapting two interfaces is defined in Figure 4. Each occurrence of the selected variable is passed on by **id**, while all other variables are discarded by **unit**. The remaining operations introduced by $\mathit{sel}(\cdot, \cdot)$ ensure that the whole interface structure is matched. This is important in order to demand the whole pattern matching as discussed in connection with Example (26). The effect of mapping $\mathit{adapt}(\cdot, \cdot)$ is twofold. First, an application of the mapping $\mathit{sel}(\cdot, \cdot)$ is introduced for all leaves of the interface adapted to. Second, the incoming argument is copied as often as needed by employing the primitive **fork**. The examples given in the previous section often contain a more simple interface adaption. In the appendix, which will not be part of the final version of this paper but will be made available online as a technical report, we provide a set of simplification rules along with detailed examples. As these rules do not provide any additional insights we omit them here.

Extra variables on the right-hand side are added by the mapping $\mathit{addfree}(\cdot, \cdot)$, defined in Figure 5. For each variable a call to **unknown** alias **invert unit** is introduced as explained in Section 2.1. The rules of a defined operation are trans-

$$\begin{aligned}
\text{addfree}(i_1, i_2) &= \underbrace{(\text{free} \times \dots \times (\text{free} \times \text{id}) \dots)}_n * \text{adapt}(i'_1, i_2) \\
\text{where } \text{free} &= (\text{unit} * (\text{invert unit})) \\
i'_1 &= (x_1, \dots, (x_n, i_1) \dots) \\
\{x_1, \dots, x_n\} &= \text{var}(i_2) \setminus \text{var}(i_1) \\
e \times e' &= \text{fork}*(e/e')
\end{aligned}$$

Fig. 5. Adding Logic Variables

$$\begin{aligned}
\text{rule}(f \ p_1 \ \dots \ p_n = e) &= (\text{invert} (\text{exp}(p_1) / \dots / \text{exp}(p_n))) * \text{adp} * \text{exp}(e) \\
\text{where } \text{adp} &= \text{addfree}(\text{int}((p_1, \dots, p_n)), \text{int}(e))
\end{aligned}$$

Fig. 6. Transforming Rules

formed according to Figure 6. The general technique is: invert the transformed pattern then apply interface adaption and finally transform the body of the rule.

$$\begin{aligned}
\text{cons}(c^0) &= c \ () = c' \\
\text{cons}(c^{n>1}) &= c \ ((x_1, x_2), \dots, x_n) = c' \ x_1 \ \dots \ x_n \\
\text{cOp}(f) &= c f \ () \ x = f \ x \\
\text{op}(f_i) &= f = \text{rule}(r_{i_1}) \ ? \ \dots \ ? \ \text{rule}(r_{i_{n_i}}) \\
\text{prog}(P) &= \text{prims} \ \text{op}(f_1) \ \dots \ \text{op}(f_n) \ \text{cOp}(f_1) \ \dots \ \text{cOp}(f_n) \ \text{cons}(c_1) \ \dots \ \text{cons}(c_m)
\end{aligned}$$

Fig. 7. Transforming Programs

Finally, the definitions of Figure 7 are employed to transform an entire program P where prims are the definitions of $(*)$ (7), $(/)$ (9), fork (11), unit (13), id (14), $(?)$ (16), invert (18), apply (31) and curry (32). In Figure 7 we assume that $\mathcal{O}_P = \{f_1, \dots, f_n\}$ and $\mathcal{C}_P = \{c_1, \dots, c_m\}$ and that for each $f_i \in \mathcal{O}_P$, $r_{i_1} \dots r_{i_{n_i}}$ to be the rules defining f_i in P . As discussed with Example (15), a new function is introduced for each constructor in \mathcal{C}_P in the first mapping of Figure 7. For simplicity, a new constructor symbol is introduced rather than a new function symbol. cOp in Figure 7 introduces the new functions needed to realize higher order, cf. Examples (33) and (36). op combines the rules defining an operation by $(?)$, cf. Example (17). In the last equation all three mappings are combined along with the definitions of the primitive operations to obtain the whole result. The resulting program P' is a program with the following signatures, where rc replaces constants by unary symbols, i.e., $\text{rc}(M) = \{s^1 \mid s^0 \in M\}$.

$$\begin{aligned}
\mathcal{O}_{P'} &= \text{rc}(\mathcal{O}_P \cup \{cf \mid f \in \mathcal{O}_P\} \cup \mathcal{C}_P) \cup \text{Prim} \\
\mathcal{C}_{P'} &= \text{rc}(\{c' \mid c \in \mathcal{C}_P\}) \\
\text{Prim} &= \{(*)^3, (/)^3, (?)^2, \text{fork}^1, \text{id}^1, \text{unit}^1, \text{invert}^1, \text{apply}^1, \text{curry}^1\}
\end{aligned}$$

Of course the question arises, how to give an account of the correctness of the transformation. Although the translation using $(*)$ and $(/)$ is standard [?], our use of function patterns is a new technique. For future work, we therefore plan to give a stronger formalisation of the operational behaviour of function patterns

than [?] and prove the correctness in the framework of an operational semantics like the one introduced in [?].

3 Obtaining Semantics

We define a denotational semantics for Curry by defining a semantics for all point-wise primitives where $assocr = [\pi_1 \circ \pi_1, [\pi_1 \circ \pi_2, \pi_2]]$.

$$\begin{aligned}
\llbracket \mathbf{f} * \mathbf{g} \rrbracket &= \llbracket \mathbf{f} \rrbracket \circ \llbracket \mathbf{g} \rrbracket & \llbracket \mathbf{f} / \mathbf{g} \rrbracket &= \llbracket \mathbf{f} \rrbracket \parallel \llbracket \mathbf{g} \rrbracket \\
\llbracket \mathbf{f} ? \mathbf{g} \rrbracket &= \llbracket \mathbf{f} \rrbracket \cup \llbracket \mathbf{g} \rrbracket \\
\llbracket \mathbf{fork} \rrbracket &= [I, I] & \llbracket \mathbf{invert} \mathbf{f} \rrbracket &= \llbracket \mathbf{f} \rrbracket^\top \\
\llbracket \mathbf{id} \rrbracket &= I & \llbracket \mathbf{unit} \rrbracket &= L \\
\llbracket \mathbf{apply} \rrbracket &= (I \parallel [I, L]) \circ [I, I]^\top \circ \pi_2 & \llbracket \mathbf{curry} \mathbf{f} \rrbracket &= L \circ [I, \llbracket \mathbf{f} \rrbracket] \circ assocr
\end{aligned}$$

There is no one-to-one relation between the semantics of **apply** and **curry** and the corresponding point-wise primitives. Higher order is essentially modelled by manipulating a relational representation of the graph of an operation. We only define the semantics of **apply** and **curry** here to show that we can provide semantics for the full set of primitives. Constructors are represented by injections. That is, if a data type has n constructors and C is the k -th constructor we define the semantics of C by an injection to the k -th position of an n -ary sum. For example, the constructors **true** and **false** defined in (15) are represented as $tt := I + O$ and $ff := O + I$, respectively. In consequence, for the non-deterministic operation **coin**, cf. (17), holds: $\llbracket \mathbf{coin} \rrbracket = \llbracket \mathbf{true} \rrbracket \cup \llbracket \mathbf{false} \rrbracket = tt \cup ff = I + I$. In the model of the concrete relation algebra, cf. Section 1.3, this can be represented by the set $\{((), tt), ((), ff)\}$.

Although the name might suggest thus, call-time choice does not coincide with strictness. It is the way that direct products are defined in relation algebra which implies that the proposed semantics indeed models call-time choice. For example, the semantics of a shared **coin**, i.e., $\llbracket \mathbf{coin} * \mathbf{fork} \rrbracket$ is $\{((), tt), ((), ff)\} \circ [I, I]$. By definition of \circ , I and $[\cdot, \cdot]$ this denotes the set $\{((), (tt, tt)), ((), (ff, ff))\}$.

In general all sharing is introduced by **fork**. The semantics would be run-time choice iff the two expressions $\mathbf{f} * \mathbf{fork}$ and $\mathbf{fork} * (\mathbf{f} / \mathbf{f})$ are equal regardless of \mathbf{f} . In contrast, in our semantics the following two properties hold.

$$R \circ [I, I] \subseteq [I, I] \circ (R \parallel R) \quad R \text{ univalent} \Leftrightarrow R \circ [I, I] = [I, I] \circ (R \parallel R)$$

Note that the given semantics is strict. The conceptual work on extending the given framework to provide a lazy semantics is advanced but not finished yet. The main reason to present a strict semantics is to show the promising new possibilities gained by giving an algebraic derivation of an optimised version of the definition of **last** introduced in (30).

For readability we will use fonts to distinguish between syntax and semantics instead of adding brackets. Names written in *italics* are the semantics of the same term written in *typewriter*, e.g., “*cons*” denotes the semantics of **cons**. First, we

provide the semantics of **last** defined in (30) and **app** defined in (2). Note that $fst = \pi_1$ and $snd = \pi_2$.

$$\begin{aligned} last &= ((\mathbf{1} \parallel (\mathbf{1} \parallel nil) \circ cons) \circ app)^\top \circ snd \circ up \\ app &= (nil \parallel \mathbf{1})^\top \circ down \cup (cons \parallel \mathbf{1})^\top \circ assocr \circ (\mathbf{1} \parallel app) \circ cons \end{aligned}$$

We want to calculate an operation called **last'**, that has the same semantics as **last** but does not use a function pattern. On the right hand side of each step, we state the relation-algebraic law that is applied.

$$\begin{aligned} last &= (\mathbf{1} \parallel (\mathbf{1} \parallel nil) \circ cons \circ app)^\top \circ snd \circ up & (R \circ S)^\top &= S^\top \circ R^\top \\ &= app^\top \circ (\mathbf{1} \parallel (\mathbf{1} \parallel nil) \circ cons)^\top \circ snd \circ up & (R \parallel S)^\top &= R^\top \parallel S^\top \\ &= app^\top \circ (\mathbf{1}^\top \parallel ((\mathbf{1} \parallel nil) \circ cons)^\top) \circ snd \circ up & (R \parallel S) \circ snd &= snd \circ S \\ &= app^\top \circ snd \circ ((\mathbf{1} \parallel nil) \circ cons)^\top \circ up \end{aligned}$$

Now we invert the semantics of the operation **app**. We split **app** in two parts, app_1 and app_2 . These are the semantics of the two rules of **app**. We simplify each rule of **app** separately. We use the abbreviation $assocl = [[\pi_1, \pi_2 \circ \pi_1], \pi_2 \circ \pi_2]$.

$$\begin{aligned} app^\top &= (app_1 \cup app_2)^\top & (R \cup S)^\top &= R^\top \cup S^\top \\ &= app_1^\top \cup app_2^\top \end{aligned}$$

$$\begin{aligned} app_1^\top &= ((nil \parallel \mathbf{1})^\top \circ down)^\top & (R \circ S)^\top &= S^\top \circ R^\top \\ &= down^\top \circ (nil \parallel \mathbf{1})^{\top\top} & R^{\top\top} &= R \\ &= down^\top \circ (nil \parallel \mathbf{1}) \end{aligned}$$

$$\begin{aligned} app_2^\top &= ((cons \parallel \mathbf{1})^\top \circ assocr \circ (\mathbf{1} \parallel app) \circ cons)^\top & (R \circ S)^\top &= S^\top \circ R^\top \\ &= cons^\top \circ (\mathbf{1} \parallel app)^\top \circ assocr^\top \circ (cons \parallel \mathbf{1})^{\top\top} & R^{\top\top} &= R \\ &= cons^\top \circ (\mathbf{1} \parallel app)^\top \circ assocr^\top \circ (cons \parallel \mathbf{1}) & (R \parallel S)^\top &= R^\top \parallel S^\top \\ &= cons^\top \circ (\mathbf{1}^\top \parallel app^\top) \circ assocr^\top \circ (cons \parallel \mathbf{1}) & \mathbf{1}^\top &= \mathbf{1} \\ &= cons^\top \circ (\mathbf{1} \parallel app^\top) \circ assocr^\top \circ (cons \parallel \mathbf{1}) & assocr^\top &= assocl \\ &= cons^\top \circ (\mathbf{1} \parallel app^\top) \circ assocl \circ (cons \parallel \mathbf{1}) \end{aligned}$$

In the next step we substitute the expression app^\top in **last** by its definition. Again, we can treat the two arguments of the union separately.

$$\begin{aligned} last &= app^\top \circ snd \circ ((\mathbf{1} \parallel nil) \circ cons)^\top \circ up & \text{def. of } app^\top \\ &= (app_1^\top \cup app_2^\top) \circ snd \circ ((\mathbf{1} \parallel nil) \circ cons)^\top \circ up & (R \cup S) \circ T &= R \circ T \cup S \circ T \\ &= (app_1^\top \circ snd \circ ((\mathbf{1} \parallel nil) \circ cons)^\top \circ up) \\ &\quad \cup (app_2^\top \circ snd \circ ((\mathbf{1} \parallel nil) \circ cons)^\top \circ up) \\ &= last_1 \cup last_2 \end{aligned}$$

$$\begin{aligned}
last_1 &= app_1^\top \circ snd \circ ((I \parallel nil) \circ cons)^\top \circ up && \text{def. of } app_1^\top \\
&= down^\top \circ (nil \parallel I) \circ snd \circ ((I \parallel nil) \circ cons)^\top \circ up && (R \parallel S) \circ snd = snd \circ S \\
&= down^\top \circ snd \circ I \circ ((I \parallel nil) \circ cons)^\top \circ up && down^\top \circ snd = I \\
&= I \circ I \circ ((I \parallel nil) \circ cons)^\top \circ up && I \circ R = R \\
&= ((I \parallel nil) \circ cons)^\top \circ up
\end{aligned}$$

By applying the transformation in the other direction we get the following rule.

$$\mathbf{last}' \ (x: []) = x$$

Now we substitute the right hand side of app_2^\top in the definition of $last_2$ and transform the resulting term.

$$\begin{aligned}
last_2 &= app_2^\top \circ snd \circ ((I \parallel nil) \circ cons)^\top \circ up \\
&\quad \text{def of } app_2^\top \\
&= cons^\top \circ (I \parallel app^\top) \circ assocl \circ (cons \parallel I) \circ snd \circ ((I \parallel nil) \circ cons)^\top \circ up \\
&\quad (R \parallel S) \circ snd = snd \circ S \\
&= cons^\top \circ (I \parallel app^\top) \circ assocl \circ snd \circ I \circ ((I \parallel nil) \circ cons)^\top \circ up \\
&\quad assocl \circ snd = snd \circ snd \\
&= cons^\top \circ (I \parallel app^\top) \circ snd \circ snd \circ I \circ ((I \parallel nil) \circ cons)^\top \circ up \\
&\quad (R \parallel S) \circ snd = snd \circ S \\
&= cons^\top \circ snd \circ app^\top \circ snd \circ I \circ ((I \parallel nil) \circ cons)^\top \circ up \\
&\quad I \circ R = R \\
&= cons^\top \circ snd \circ \underbrace{app^\top \circ snd \circ ((I \parallel nil) \circ cons)^\top \circ up}_{last}
\end{aligned}$$

By applying the inverse transformation we get a second rule for \mathbf{last}' that contains a recursive call. Now we unite the results for $last_1$ and $last_2$ and get a definition of \mathbf{last} that does not use a function pattern.

$$\begin{aligned}
\mathbf{last}' \ [x] &= x \\
\mathbf{last}' \ (_ : xs) &= \mathbf{last}' \ xs
\end{aligned}$$

4 Related and Future Work

We are aware that many works on all three topics connected in this paper exist, i.e., on the semantics of functional logic programming languages, the point-free programming style and relation algebra. Here, we can only relate our work to a small selection.

Cunha, Pinto and Proença [?,?] present a framework for transformations into point-free style. They present a library for point-free programming in Haskell

which is similar to our primitives. Furthermore, they have developed a program that transforms arbitrary Haskell programs into point-free style and they present a tool for manipulating point-free programs. Although their focus is different from ours, especially [?] provided valuable insights, e.g., that the opportunities for automatic reasoning about programs is not as straightforward as the formalism might suggest. Therefore we aim to use our approach for program analysis like [?] and to prove (with only half automatic tool support) the correctness of optimisations for functional logic languages.

The book “Algebra of Programming” by Bird and de Moor [?] has been very influential for this work. They present a calculus for the algebraic manipulation of functional programs. We hope that we could give an idea that the framework of functional *logic* languages is an even more natural and promising field for this style of reasoning about programs. The elementary difference is the existence of non-determinism. Whereas in [?] every inversion and every non-deterministic definition resulting from inversion *must* be eliminated, the framework of functional logic languages allows much less restricted use of algebraic methods. The same is true a fortiori for approaches like [?] that aim on deriving a functional definition to compute the inversion of a given function definition.

Regarding the denotational semantics of functional (logic) languages, we want to relate our approach especially with two papers as future work. [?] proposes a denotational semantics for a functional language employing relation algebra. [?] provides a denotational semantics for functional logic languages based on cones. There are many interesting extensions to the framework of [?] which we want to investigate. However, our work presents a promising step towards covering function patterns for the first time.

Apart from relating with the existing approaches, we plan to extend our approach to cover laziness. Conceptual work for this extension already exists. Naturally, we want to work on the correctness of the proposed transformation, most suitably by using a framework like [?]. Furthermore, a formal relation to the semantics of [?] would further ensure the validity of the approach. In the long term we would like to use algebraic methods like demonstrated in Section 3 for program analysis and optimisation.