

# Denotation by Transformation<sup>\*</sup>

## 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. Here we propose a set of basic operations based on which arbitrary functional logic programs can be transformed to point-free programs. The semantics of the resulting programs are strict but, nevertheless, the semantics of the original program is preserved.

There is a one-to-one mapping from the primitives introduced by the transformation to operations in relation algebra. This mapping can be extended to obtain a relation algebraic model for the whole program. This yields a denotational semantics which is on one hand closely related to point-free functional logic programs and on the other hand connects to the well-developed field of algebraic logic including automatic proving.

## 1 Introduction

The importance of a *point-free* view on *programming* has been emphasized particularly in the applications of category theory to semantics of programming languages. The concrete advantage of point-free style is the possibility to treat programs adhering to it in an *algebraic way*. The goals of an algebraic approach are mainly: a) elegance of the provided formalism which in practice directly results in what we call “proof economy” and b) the possibility to employ automatic proof procedures. Both goals have received a serious damper in the recent development of the field [?,?]. There we find the disillusioning summary that the state of the art “contradicts the general assumption that calculations in this [point-free] style are more amenable to mechanization” [?, Chapter 8]. In our opinion the situation can be improved by two means: 1.) develop transformations to point-free style which keep much more structure of the original programs and 2.) connect to a well-developed theory with an existing approach to automated proving. The first point is important to enable the formulation of the lemmas needed in any substantial proof and even more important to understand the

---

<sup>\*</sup> This work has been partially supported by the DFG under grant Ha 2457/1-2.

resulting proof or a counter example, respectively. The interested reader is referred to Appendix A for an example comparing the readability of our approach to two others, [?] and [?]. The second point is realized by providing a semantics for the point-free programs within the framework of relation algebra. The semantics of the whole program is directly obtained by interpreting the primitive operations, which were introduced by the transformation, as relation algebraic operations. Relation algebra is a well-developed field of algebraic logic [?] for which approaches to proof automation have been developed, cf, e.g., [?]. In this paper we describe the transformation to point free style (Section 2) and prove its correctness (Section 3). The relation algebraic model we developed for our programs is described in [?].

### 1.1 Functional Logic Programming Languages

We consider a functional logic program as a constructor-based rewriting system, allowing extra variables on the right hand side. This section establishes some of the involved notation, which is mostly following [?,?]. For our examples we adopt the syntax of Curry [?] although the transformation we develop is compatible with other functional logic languages like Toy [?].

$\Sigma_P$  is the signature of a program  $P$  partitioned into two sets, the set of *constructors*  $C_P$  and the set of defined *operations*  $O_P$ . 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 variables  $\mathcal{X}$ , the sets of *terms* and *constructor terms* are denoted by  $\mathcal{T}(\Sigma_P, \mathcal{X})$  and  $\mathcal{T}(C_P, \mathcal{X})$ , 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. A linear constructor term is called a *pattern*. Constructors are introduced by a **data** declaration, as shown in Example (1). The “**a**” in the third declaration denotes that **[a]** is a polymorphic type. Operations are defined by *rewrite rules* of the form “ $f p_1 \dots p_n = e$ ” where  $f^n \in O_P$  and  $p_1, \dots, p_n$  are *patterns*. The right hand side  $e$  may contain *extra variables*, i.e., variables which do not occur in the patterns of

```

data Success = Success
data Bool = True | False (1)
data [a] = [] | a : [a]

```

the left hand side. To cope with extra variables rewriting is extended to *narrowing* [?]. In our context this means that a *narrowing step* is a rewrite step that includes the replacement of extra variables by constructor terms. We call such a replacement a *constructor substitution* and denote it by  $\sigma$  or by  $\theta$ . The set of all constructor substitutions is denoted by  $CSubst$ .

```

app [] ys = ys
app (x:xs) ys = x : (app xs ys) (2)

```

A possible narrowing step for Example 2 is  $\text{app } x \text{ [True]} \rightarrow \{x \mapsto []\} \text{ [True]}$ , where  $x$  is an extra variable.

In addition to defining rules, *type signatures* are used to declare the sorts of an operation. For example,  $\text{app} :: [a] \rightarrow [a] \rightarrow [a]$  declares that **app** 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

`coin :: Bool`      also from operator definitions with overlapping left hand  
`coin = True`    (3) sides. For Example (3), there are two derivations `coin →`  
`coin = False`      `True` and `coin → False` or, for short, `coin → True | False`.

## 2 Transformation to Point-free Style By Example

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. The term was introduced by John Backus in his Turing Award Lecture in 1977 [?]. The counterpart of *point-free* is *point-wise*, that is, functions that explicitly access their arguments. 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. We present only the idea of the transformation here and give a formal definition in the next section.

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

$$\begin{array}{l}
 (*) :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow a \rightarrow c \\
 (f * g) x = g (f x)
 \end{array} \quad (4) \quad \begin{array}{c} \boxed{f} \text{---} \boxed{g} \end{array}$$

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. Simple definitions can be made point-free using sequential composition (5).

$$\begin{array}{l}
 \text{involution } x = \text{not } (\text{not } x) \\
 \text{involution} = \text{not} * \text{not}
 \end{array} \quad (5) \quad \begin{array}{c} \boxed{\text{not}} \text{---} \boxed{\text{not}} \end{array}$$

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


$$\begin{array}{l}
 (/) :: (a \rightarrow c) \rightarrow (b \rightarrow d) \rightarrow (a,b) \rightarrow (c,d) \\
 (f / g) (x,y) = (f x, g y)
 \end{array} \quad (6) \quad \begin{array}{c} \boxed{f} \\ \text{---} \\ \boxed{g} \end{array}$$

Example (7) shows the use of parallel composition. All primitives are right associative. Instead of using precedences we use parenthesis to increase readability.

$$\begin{array}{l}
 \text{nor} :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \\
 \text{nor } x \ y = \text{not } x \ \&\& \ \text{not } y \\
 \\
 \text{nor} :: (\text{Bool},\text{Bool}) \rightarrow \text{Bool} \\
 \text{nor} = (\text{not} / \text{not}) * \text{and}
 \end{array} \quad (7) \quad \begin{array}{c} \boxed{\text{not}} \\ \text{---} \\ \boxed{\text{not}} \end{array} \text{---} \boxed{\text{and}}$$

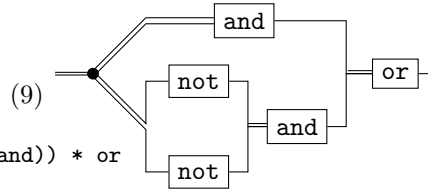
We have effectively changed the type of `nor` to a so called “uncurried” version. That is, instead of taking a pair of arguments it takes only a single argument. Multiple arguments are combined by using tuples. We use curried operations only when higher order is employed, as discussed in Paragraph “Higher 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 or reordered in some way. An uncurried and point-free version of “if and only if” (9) can be formulated using `fork`.

`fork :: a -> (a,a)` (8)   
`fork x = (x,x)`


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 or reordered in some way. An uncurried and point-free version of “if and only if” (9) can be formulated using `fork`.

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





`(<=>) :: (Bool,Bool) -> Bool`  
`(<=>) = fork * (and / ((not / not) * and)) * or`

There are four more primitives for interface adaption. The operator `unit` to “discard a value”, the identity `id` to “pass a value on” and `fst` and `snd` to “select a value”. All these primitives are exemplified in the following sections.

`unit :: a -> ()` (10)   
`unit x = ()`

`id :: a -> a` (11)   
`id x = x`

`fst :: (a,b) -> a` (12)   
`fst (x,y) = x`

`snd :: (a,b) -> b` (13)   
`snd (x,y) = y`

**Data Structures and Pattern Matching** We do not wish to abstract from concrete domains in order to make the resulting programs more readable. We replace constructor definitions by their uncurried versions. For instance, instead of the declarations for `[a]` and `Bool` from Example (1) we obtain the declarations shown in Example (14).

`data List a = Nil ()`  
`| Cons (a,List a)` (14)  
`data Bool = True ()`  
`| False ()`

replace constructor definitions by their uncurried versions. For instance, instead of the declarations for `[a]` and `Bool` from Example (1) we obtain the declarations shown in Example (14).

Note that for uniformity also the constants `True`, `False`, and `Nil` are extended with an argument. This simplifies the definitions of destructors in the following.

To express pattern matching we introduce a *destructor* for every constructor. This operation inverts the constructor, i.e., it peels off the outermost constructor and yields its arguments. Example (15) presents the destructors corresponding to the constructors from Example (14).

`invNil :: List a -> ()` `invTrue, invFalse :: Bool -> ()`  
`invNil (Nil x) = x` `invTrue (True x) = x`  
`invCons :: List a -> (a,List a)` `invFalse (False x) = x` (15)  
`invCons (Cons x) = x`

Now it becomes apparent why constant constructors are extended with an argument: to make them invertible. Note also that the definition of the destructors

follows a very simple pattern. Using the destructor `invCons` and `fst` and `snd` the standard functions `head` and `tail` can easily be expressed.

$$\begin{array}{ll} \text{head} :: \text{List } a \rightarrow a & \text{tail} :: \text{List } a \rightarrow \text{List } a \\ \text{head} = \text{invCons} * \text{fst} & \text{tail} = \text{invCons} * \text{snd} \end{array} \quad (16)$$

To combine several rules we employ an additional feature of functional *logic* programming, i.e., non-determinism. The operator `(?)` allows a very elegant way of expressing pattern matching in a point-free style.

$$\begin{array}{ll} (?) :: (a \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow b & \\ (f ? g) x = f x & (17) \quad \text{coin} :: () \rightarrow \text{Bool} \\ (f ? g) x = g x & \text{coin} = \text{True} ? \text{False} \end{array} \quad (18)$$

As stated in the introduction, overlapping rules in functional logic languages lead to non-deterministic search [?]. In principle, all non-determinism can be introduced by a single operation with overlapping rules (17). We use `(?)` to combine the rules of a function (18). Note that the introduction of the argument `()` for constant constructors extends to all definitions of constants. Example (19) shows the point-free version of `null` which tests whether a list is empty or not.

$$\text{null} = (\text{invNil} * \text{True}) ? (\text{invCons} * \text{unit} * \text{False}) \quad (19)$$

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 deterministic 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. For example,

$$\begin{array}{ll} \text{member} :: [a] \rightarrow a & \text{ily captured by our approach. For example,} \\ \text{member } (x:xs) = x & (20) \text{ the operation } \text{member} \text{ defined in (20) non-} \\ \text{member } (x:xs) = \text{member } xs & \text{deterministically relates a list with each of} \\ & \text{its elements. Without further additions this} \end{array}$$

behaviour is captured by the transformation. The following definition shows a point-free version of `member`. It also illustrates that recursive functions simply stay recursive. There is no need for changes, e.g., a special recursion operator.

$$\text{member} = (\text{invCons} * \text{fst}) ? (\text{invCons} * \text{snd} * \text{member}) \quad (21)$$

$$\begin{array}{ll} \text{unknown} :: () \rightarrow a & \text{There is one more feature that is specific to func-} \\ \text{unknown } () = x & (22) \text{ tional logic languages: free variables. To introduce} \\ \text{where } x \text{ free} & \text{free variables we employ the primitive } \text{unknown} \text{ (22).} \\ & \text{The keyword } \text{free} \text{ is used to define extra variables.} \end{array}$$

**Higher Order** In order to introduce higher-order operations we need to adapt the well-known pair `apply` and `curry` to our setting.

$$\begin{array}{ll} \text{apply} :: (a \rightarrow b, a) \rightarrow b & (23) \quad \text{curry} :: ((a, b) \rightarrow c) \rightarrow a \rightarrow b \rightarrow c \\ \text{apply } (f, x) = f x & \text{curry } f \ x \ y = \text{apply } (f, (x, y)) \end{array} \quad (24)$$

The operations `apply` and `curry` are very similar to the original operations. In the

point-free world `apply` takes a tuple of arguments and `curry` uses this `apply` operation instead of the predefined application. We illustrate the use of `apply` by a standard example of a higher-order operation in Example (25). We assume `adapt` to map the tuple structure  $(f, (x, xs))$  to  $((f, x), (f, xs))$  and omit its concrete definition by means of `(/)`, `fst`, `snd` and `fork`.

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

(25)

```
map :: (a -> b, [a]) -> [b]
map = ((id / invNil) * unit * Nil)
      ? ((id / invCons) * adapt * (apply / map) * Cons)
```

(26)

We map the operation `not` on the list `Cons (False (), Cons (True ()), Nil ())`.

```
val :: a -> () -> a
val f () = f
```

(27)

We have to consider that values of type `a` in the original program correspond to operations of type `() -> a` in the point-free program. For example, the point-free version of `coin :: Bool` is `coin :: () -> Bool`. 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)`. Therefore we introduce the primitive `val` which takes a higher order object and yields a value.<sup>1</sup> By using `val` in the definition of `mapNot` we get a higher order value representing the operation `not`. We do not need to use `curry` because `not` takes only one argument and is therefore uncurried. The application `(mapNot ())` evaluates to `Cons (False ()), Cons (True ()), Nil ())` as intended.

```
not = (invTrue * False) ? (invFalse * True)
listFalseTrue = fork * (False / (fork * (True / Nil) * Cons)) * Cons
mapNot = fork * (val not / listFalseTrue) * map
```

(28)

We have illustrated all the point-wise primitives necessary to translate arbitrary functional logic programs: `(*)` (4), `(/)` (6), `fork` (8), `unit` (10), `id` (11), `(?)` (17), `fst` (12), `snd` (13), `unknown` (22), `apply` (23) and `curry` (24).

### 3 Obtaining Point-free Style in General

In this section we give a formal definition of the transformation that was motivated in Section 2. There is a necessary preliminary step to the transformation described so far. This step is quite interesting in itself, as it transforms a lazy functional logic program to a strict functional logic program nevertheless preserving its denotational semantics. The main idea is to replace laziness by the non-deterministic choice of whether to evaluate a given function application or not. Whereas such a transformation might be a bad idea from an operational point of view, it is very convenient for semantic purposes. As we will see, it is straightforward to prove the main properties of the transformation to strict programs with respect to existing semantics of functional logic languages.

<sup>1</sup> Incidentally, the type of `val` is more general and does not only work for higher order types. In an extended setting with base types we also use it to wrap, e.g., integers.



Whenever the program is not apparent, we write  $\mapsto_P$  for a step with respect to program  $P$ . By  $\mapsto^*$  we denote the reflexive transitive closure of  $\mapsto$ . By  $\llbracket e \rrbracket^\perp$  we denote the set  $\{t \mid e \mapsto^* t, t \in \mathcal{T}(C_P \cup \{\perp\}, \mathcal{X})\}$  and we call each  $t$  in this set a normal form. Note that by definition normal forms are constructor terms.

Rule **B** is used to model laziness. It allows the replacement of unneeded expressions by  $\perp$ .

### 3.2 Transformation to Strict Programs

We define a transformation  $str(\cdot)$  of arbitrary lazy programs into strict programs preserving its semantics. We achieve this by adding the possibility to abort the evaluation of any function application non-deterministically. We get a strict program by replacing each application  $(e_1 e_2)$  by a call to the higher order function  $(\mathbf{app} e_1 e_2)$ . Function  $\mathbf{app}$  takes two expressions and evaluates both to head normal form before applying the first to the second, see the rules of  $\mathbf{app}$  below. As *all* applications, including constructor applications, are replaced by  $\mathbf{app}$ , the resulting program is strict. The non-deterministic abortion of evaluation is achieved by adding an additional rule to each operation definition. The right hand side of this rule contains only the new constructor symbol  $\mathbf{U}^0$  (short for unevaluated). The point is that unevaluated expressions can be non-deterministically replaced by the new constructor  $\mathbf{U}$ . In this sense, non-determinism is more general than laziness. This fact can be used to obtain simple semantic approaches to functional logic languages. The following transformation yields a strict functional logic programs.

$$\begin{aligned} str(P) = & \{l = str(r) \mid l = r \in P\} \cup \{f x_1 \dots x_n = \mathbf{U} \mid f^n \in O_P\} \\ & \cup \{\mathbf{app} (s x_1 \dots x_n) (c y_1 \dots y_m) = (s x_1 \dots x_n) (c y_1 \dots y_m) \\ & \quad \mid s^i \in \Sigma_P, i > n, c^m \in C_P \cup \{\mathbf{U}\}\} \\ & \cup \{\mathbf{app} \mathbf{U} (c x_1 \dots x_n) = \mathbf{U} \mid c^n \in C_P \cup \{\mathbf{U}\}, \} \end{aligned}$$

$$str(x) = x, x \in \mathcal{X} \quad str(s) = s, s \in \Sigma_P \quad str(e_1 e_2) = \mathbf{app} str(e_1) str(e_2)$$

Transformation  $str(\cdot)$  preserve semantics in the sense that programs resulting from it yield the newly introduced constructor  $\mathbf{U}$  instead of  $\perp$ . All other values are identical. Indeed, the way transformed programs work is very similar to the behavior of the semantics of Figure 1. We prove this by stating three simple observations which then allow us to formulate a very tight correspondence between the original program with lazy semantics and the transformed program with strict semantics.

The first observation formalizes our claim that the resulting programs are indeed strict. Recall that rule **B** of Figure 1 is responsible for modelling laziness. Therefore strictness means the following: if rule **B** was used in a derivation  $str(e) \mapsto_{str(P)}^* t$  of an expression  $e$  to a normal form  $t$  then  $t = \perp$ . Equivalently if there is a derivation  $str(e) \mapsto_{str(P)}^* t$  with  $t \neq \perp$  then rule **B** is not used in this derivation. By  $\mapsto_{\mathcal{L}}$  we denote a derivation without rule **B**, i.e., with rule **OR** only and by  $\llbracket e \rrbracket$  we denote the set  $\{t \mid e \mapsto_{\mathcal{L}}^* t, t \in \mathcal{T}(C_P, \mathcal{X})\}$ .

**Proposition 1.**  $\llbracket str(e) \rrbracket_{str(P)}^\perp \setminus \{\perp\} = \llbracket str(e) \rrbracket_{str(P)}$

*Proof (Idea).* Structural induction on  $\mathcal{C}$  shows that  $str(\mathcal{C}[e']) \mapsto str(\mathcal{C}[\perp]) \mapsto^* t$  implies  $t = \perp$ .

We have seen that rule **B** is useless to derive any value but  $\perp$ . On the other hand it is worth noting that if we do not use that rule, strict application ( $\mathbf{app} e_1 e_2$ ) is identical to standard application ( $e_1 e_2$ ).

**Proposition 2.**  $(e_1 e_2) \mapsto_{\not\perp}^* v$  iff  $(\mathbf{app} e_1 e_2) \mapsto_{\not\perp}^* v$

*Proof (Idea).* For all  $e_1, e_2 \neq \perp$   $\mathbf{app} e_1 e_2$  is defined as  $(e_1 e_2)$

We alter rule **B** slightly and replace it by the following rule **B'**. By  $\rightsquigarrow$  we denote the rewrite relation where all applications of **B** are replaced by proofs for **B'**.

$$(\mathbf{B}') \mathcal{C}[(f t_1 \dots t_n)] \rightsquigarrow \mathcal{C}[\perp] \quad \begin{array}{l} \mathcal{C} \in \mathit{Ctx}, f^n \in O_P \vee (n = 1 \wedge f = \perp), \\ t_1, \dots, t_n \in \mathcal{T}(C_P \cup \{\perp\}, \mathcal{X}) \end{array}$$

The difference between the two rules is that applications can only be discarded if its arguments are constructor terms (with  $\perp$ ) and that applications of  $\perp$  to such terms have to be replaced by  $\perp$  one by one. It is easy to prove that the relations defined by  $\mapsto$  and  $\rightsquigarrow$  are identical.

**Proposition 3.**  $e \mapsto^* t$  iff  $e \rightsquigarrow^* t$

*Proof (Idea).* Before discarding the whole expression discard all sub terms.

The notation  $\mapsto_{\not\perp}$  introduced above also excludes the use of variant **B'**. We are now ready to prove the strong correspondence between the original program  $P$  and the transformed program  $str(P)$ .

**Lemma 1.** For all programs  $P$  and all expressions  $e \in Expr_P$  holds  $e \mapsto_P^* t$  iff  $str(e^\not\perp) \mapsto_{str(P)}^* t^\not\perp$  where  $\not\perp$  replaces all occurrences of  $\perp$  by  $\mathbf{U}$ .

*Proof.* By Proposition 1 we may consider the derivation  $str(e^\not\perp) \mapsto_{str(P)}^* t^\not\perp$  instead of  $str(e^\not\perp) \mapsto_{str(P)}^* t^\not\perp$ . By Proposition 2 we may treat the appearances of  $(\mathbf{app} e_1 e_2)$  in the  $\mapsto_{\not\perp}^*$ -derivation as  $(e_1 e_2)$ . This means essentially that  $str(e^\not\perp) \mapsto_{str(P)}^* t^\not\perp$  iff  $e^\not\perp \mapsto_{P'}^* t^\not\perp$  where  $P' := P \cup \{f x_1 \dots x_n = \mathbf{U} \mid f^n \in O_P\}$ . Adding Proposition 3 this means we only need to show that  $e \rightsquigarrow_P^* t$  iff  $e^\not\perp \mapsto_{P'}^* t^\not\perp$ . Between these derivations there is such a close correspondence that there is a one-to-one mapping between them. We show this by induction on the length  $n$  of both derivations.

The base case  $n = 0$  holds trivially. For the induction we distinguish two cases:  
Case 1: Rule **B'** corresponds to one of the new rules.

$$\begin{array}{l} \mathcal{C}[f t_1, \dots, t_n] \rightsquigarrow_P \mathcal{C}[\perp] \quad \text{iff} \\ \mathcal{C}[f t_1, \dots, t_n]^\not\perp = \mathcal{C}[f t_1^\not\perp, \dots, t_n^\not\perp]^\not\perp \mapsto_{P'} \mathcal{C}[\mathbf{U}]^\not\perp = \mathcal{C}[\perp]^\not\perp \end{array}$$

Case 2: Applications of the original rules are unchanged.

$$\begin{array}{l} \mathcal{C}[f t_1 \theta, \dots, t_n \theta] \rightsquigarrow_P \mathcal{C}[r \theta] \quad \text{iff} \\ \mathcal{C}[f t_1, \dots, t_n]^\not\perp = \mathcal{C}[f t_1 \theta^\not\perp, \dots, t_n \theta^\not\perp]^\not\perp \mapsto_{P'} \mathcal{C}[r \theta^\not\perp]^\not\perp = \mathcal{C}[r \theta]^\not\perp \end{array}$$

Since with  $\theta$  being a constructor substitution, also  $(\cdot)^\not\perp \circ \theta$  is in  $CSubst$ .

### 3.3 Transformation to Point-free Style

We use an intermediate transformation which yields programs that contain only function definitions in first order style with the single exception of `curry f x y = apply (f, (x,y))`, cf. the paragraph “Higher Order” in Section 2. We have stated in the introduction that after transforming a program the only source of non-determinism is the operation (?). Likewise, in uncurried programs all higher order functions stem from `curry`. There is a simple transformation to map functions defined with  $n$  patterns  $p_1 \dots p_n$  to functions with one argument which is a tuple  $(p_1, \dots, p_n)$  employing the function pair `curry` and `apply` to preserve higher order semantics. We have formalized such a transformation but feel that the procedure is well known such that we present that transformation in Appendix B only. Uncurried programs allow us to abstract from the arity of all functions but the primitives we introduce.

In this section we present the general transformation of programs into point-free style. First we introduce the notion of an interface of an expression. The *interface* of an expression is an abstraction from its actual structure. An interface is a tree with the same branching structure as the expression and this tree contains the variables that occur in the expression. The mapping from expressions to interfaces is defined as follows, where  $s$  ranges over symbols in  $\Sigma_P$ :

$$\text{int}(s) = () \quad \text{int}(x) = x \quad \text{int}(s (e_1, \dots, e_n)) = (\text{int}(e_1), \dots, \text{int}(e_n))$$

This mapping is frequently used throughout the definition of the transformation. We denote complex interfaces, i.e. those not in  $\mathcal{X} \cup \{()\}$  by  $i, i_1, i_2 \dots$  and by  $\text{var}(i)$  the set of all variables occurring in  $i$ . For example, the interface of the right hand side of the original definition of  $\langle \Rightarrow \rangle$  in Example (9) is  $((x, y), (x, y))$ . The first use of interfaces is *variable selection* defined as follows:

$$\begin{aligned} \text{sel}(x, x) &= \text{id} \\ \text{sel}(x, (i, i')) &= \begin{cases} \text{fst} * \text{sel}(x, i) & , \text{ if } x \in \text{var}(i) \wedge x \notin \text{var}(i') \\ \text{snd} * \text{sel}(x, i') & , \text{ if } x \notin \text{var}(i) \wedge x \in \text{var}(i') \end{cases} \end{aligned}$$

Each occurrence of the selected variable is passed on by `id`, while all other variables are discarded by `fst` and `snd`. Note that this definition requires that variable  $x$  actually occurs in interface  $i$ . To get an intuition about what variable selection is used for reconsider Example (16). The definition of `head` has been derived from `head (x:xs) = x` whose uncurried version is `head (Cons (x,xs)) = x`. To finally yield  $\mathbf{x}$  we select  $\mathbf{x}$  from the interface of `Cons (x,xs)` which simply is  $(\mathbf{x}, \mathbf{xs})$ . The result is `fst * id` which can be simplified to `fst`.

The next lemma states in general that  $\text{sel}(x, i)$  selects the correct sub-term of any substitution of interface  $i$ .

**Lemma 2.** *For all interfaces  $i$ ,  $\theta \in C\text{Subst}$  and  $C \in C\text{ntxt}$   $x \in \text{var}(i)$  implies  $C[\text{sel}(x, i) i\theta] \mapsto^* C[x\theta]$ .*

*Proof (Idea).* Structural induction on  $i$ .

On the basis of  $sel(x, i)$  we define the general approach to “Interface Adaption”, cf. the paragraph of the same name in Section 2 as follows:

$$adapt(i, i') = \begin{cases} \mathbf{id} & , \text{ if } i = i' \\ adapt'(i, i') & \text{ otherwise} \end{cases}$$

$$\begin{aligned} adapt'(i, ()) &= \mathbf{unit} \\ adapt'(i, x) &= sel(x, i) \\ adapt'(i, (i_1, i_2)) &= \mathbf{fork} * (adapt(i, i_1)/adapt(i, i_2)) \end{aligned}$$

The effect of mapping  $adapt(\cdot, \cdot)$  is twofold. First, an application of the mapping  $sel(\cdot, \cdot)$  is introduced for every leaf of the interface adapted to. Second, the incoming argument is copied as often as needed by employing the primitive  $\mathbf{fork}$ . For instance, for the definition of  $\langle \Leftarrow \Rightarrow \rangle$  in Example 9 the interface  $(x, y)$  is adapted to  $((x, y), (x, y))$  and the result is  $\mathbf{fork} * (\mathbf{id}/\mathbf{id})$ . This expression can be shown to be equivalent to  $\mathbf{fork}$  which is what one would expect for this example.

The next lemma states that mapping  $adapt(i, i')$  indeed yields an interface adaption from interface  $i$  to  $i'$ . More concretely, if  $adapt(i, i')$  is applied to a tuple of the form  $i$  the result is a tuple of form  $i'$ . Note that Lemma 3 requires that the variables of the target interface are a subset of the variables of the argument interface.

**Lemma 3.** *For all interfaces  $i, i'$  with  $var(i') \subseteq var(i)$ , all  $\theta \in CSubst$  and all contexts  $\mathcal{C}: \mathcal{C}[adapt(i, i') i\theta] \mapsto^* \mathcal{C}[i'\theta]$*

*Proof.* By structural induction on  $i'$ .

Case 1  $i = i'$  :

$$\mathcal{C}[adapt(i, i') i\theta] \{ \text{def } adapt(\cdot, \cdot) \} = \mathcal{C}[\mathbf{id} i\theta] \{ \text{def } \mathbf{id} \} \mapsto \mathcal{C}[i\theta] = \mathcal{C}[i'\theta]$$

Case 2  $i \neq i'$  :

$$\mathcal{C}[adapt(i, i') i\theta] \{ \text{def } adapt(\cdot, \cdot) \} = \mathcal{C}[adapt'(i, i') i\theta]$$

$i' = ()$  :

$$\mathcal{C}[adapt'(i, ()) i\theta] \{ \text{def } adapt'(\cdot, \cdot) \} = \mathcal{C}[\mathbf{unit} i\theta] \{ \text{def } \mathbf{unit} \} \mapsto \mathcal{C}[()] = \mathcal{C}[i'\theta]$$

$i' = x$  :

$$\mathcal{C}[adapt'(i, i') i\theta] \{ \text{def } adapt'(\cdot, \cdot) \} = \mathcal{C}[sel(x, i) i\theta] \{ \text{Lemma 2} \} \mapsto^* \mathcal{C}[x\theta]$$

$i' = (i_1, i_2)$  :

$$\begin{aligned} & \mathcal{C}[adapt'(i, i') i\theta] \\ \{ \text{def } adapt'(\cdot, \cdot) \} &= \mathcal{C}[(\mathbf{fork} * (adapt(i, i_1)/adapt(i, i_2))) i\theta] \\ \{ \text{def } * \} &\mapsto \mathcal{C}[(adapt(i, i_1)/adapt(i, i_2)) (\mathbf{fork} i\theta)] \\ \{ \text{def } \mathbf{fork} \} &\mapsto \mathcal{C}[(adapt(i, i_1)/adapt(i, i_2)) (i\theta, i\theta)] \\ \{ \text{def } / \} &\mapsto \mathcal{C}[(adapt(i, i_1) i\theta, adapt(i, i_2) i\theta)] \\ \{ \text{ind. hypothesis} \} &\mapsto^* \mathcal{C}[(i_1\theta, i_2\theta)] = \mathcal{C}[i\theta] \end{aligned}$$

The last missing step of interface adaption is to introduce extra variables by function  $\mathbf{unknown}$ , cf. the last paragraph of Section 2. We assume  $\times$  to be a left associative operator symbol. The required applications of  $\mathbf{unknown}$  are introduced

by the mapping  $addfree(\cdot, \cdot)$ , defined as follows:

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

**Proposition 4.** For all interfaces  $i: \mathcal{C}[\underbrace{((free \times \dots \times free) i)}_n] \mapsto^* \mathcal{C}[(x_1, \dots, x_n)]$  where  $x_k, x_j \notin var(\mathcal{C}[i])$  and  $x_k \neq x_j$  for all  $k \in \{1, \dots, n\}$ .

*Proof (Idea).* Induction on  $n$  using the definitions of  $(/)$ ,  $\mathbf{unit}$  and  $\mathbf{unknown}$ .

The following lemma extends Lemma 3 by the introduction of extra variables.

**Lemma 4.** For all interfaces  $i_1, i_2$ , all substitutions  $\theta$  and all contexts  $\mathcal{C}$ :  $\mathcal{C}[addfree(i_1, i_2) i_1\theta] \mapsto^* \mathcal{C}[i_2\theta]$ .

*Proof (Idea).* The proof connects Proposition 4 and Lemma 3.

The next step in the transformation to point-free programs is the transformation of expressions. Applications are replaced by the operator  $(*)$  and the arguments are combined by  $(/)$ . Higher order functions, i.e., single symbols  $(s)$  are made values by using  $\mathbf{val}$ . Expressions are translated as follows:

$$\begin{aligned}
exp(s ()) &= s \\
exp(s (e_1, \dots, e_{n>0})) &= (exp(e_1)/\dots/exp(e_n)) * s \\
exp(s) &= \mathbf{val}(s) \\
exp(x) &= \mathbf{id}
\end{aligned}$$

The next lemma states that an application of  $exp(e)$  to the interface of  $e$  can be reduced to the original expression.

**Lemma 5.** For all expressions  $e$ ,  $\theta \in CSubst$  and contexts  $\mathcal{C}$ :  $\mathcal{C}[exp(e) int(e)\theta] \mapsto^* \mathcal{C}[e\theta]$ .

*Proof (Idea).* Structural Induction on  $e$ .

Next we define the transformation of pattern matching to point-free style. The mapping  $invert(\cdot)$  is very similar to  $exp(\cdot)$ . All occurrences of constructors  $c$  are replaced by the corresponding destructors  $invC$ .

$$\begin{aligned}
invert(c ()) &= invC \\
invert(c (e_1, \dots, e_{n>0})) &= invC * (invert(e_1)/\dots/invert(e_n)) \\
invert(x) &= \mathbf{id}
\end{aligned}$$

Furthermore the destructors  $invC$  are applied *before* the resulting arguments are processed. The next lemma states that the application of  $invert(p)$  yields the argument of a constructor if it matches the pattern and fails otherwise.

**Lemma 6.** *Let  $p$  be a linear pattern,  $e$  a term and  $\mathcal{C}$  a context. If there exists a  $\theta \in CSubst$  with  $e = p\theta$  then  $\mathcal{C}[invert(p) e] \mapsto^* \mathcal{C}[int(p)\theta]$  and otherwise  $\llbracket \mathcal{C}[invert(p) e] \rrbracket = \emptyset$ .*

*Proof.* By induction on the structure of  $p$ .

$$p = x : \mathcal{C}[invert(x) e] = \mathcal{C}[id e] \mapsto \mathcal{C}[e] = \mathcal{C}[x\{x \mapsto e\}] = \mathcal{C}[int(x)\{x \mapsto e\}]$$

$$\begin{aligned} p = c(p_1, \dots, p_n) : \mathcal{C}[invert(c(p_1, \dots, p_n)) e] \\ \{ \text{def } invert(\cdot) \} &= \mathcal{C}[(invC * (invert(p_1) / \dots / invert(p_n))) e] \\ \{ \text{def } * \} &\mapsto \mathcal{C}[(invert(p_1) / \dots / invert(p_n)) (invC e)] \end{aligned}$$

case 1  $e = c(e_1, \dots, e_n)$

$$\begin{aligned} \{ \text{def } invC \} &\mapsto \mathcal{C}[(invert(p_1) / \dots / invert(p_n)) (e_1, \dots, e_n)] \\ \{ \text{def } / \} &\mapsto^* \mathcal{C}[(invert(p_1) e_1, \dots, invert(p_n) e_n)] \end{aligned}$$

case 1.1 there exists a  $\theta$  with  $e = p\theta$  which implies  $e_1 = p_1\theta \wedge \dots \wedge e_n = p_n\theta$

$$\begin{aligned} \{ \text{ind. hypothesis} \} &\mapsto^* \mathcal{C}[(int(p_1)\theta, \dots, int(p_n)\theta)] \\ \{ \theta \in CSubst \} &\mapsto \mathcal{C}[(int(p_1), \dots, int(p_n))\theta] \\ \{ \text{def } int(\cdot) \} &= \mathcal{C}[int(p)\theta] \end{aligned}$$

case 1.2  $e_i$  and  $p_i$  not unifiable for an  $i \in \{1, \dots, n\}$

By induction hypothesis  $(invert(p_i) e_i)$  is not reducible to a constructor normal form. As all applications are strict this implies

$$\llbracket \mathcal{C}[(invert(p_1) / \dots / invert(p_n)) (invC e)] \rrbracket = \emptyset$$

case 2  $e \neq c(e_1, \dots, e_n)$

By definition  $(invC e)$  does not have a constructor normal form. As all applications are strict this implies  $\llbracket \mathcal{C}[(invert(p_1) / \dots / invert(p_n)) (invC e)] \rrbracket = \emptyset$

The general technique of the transformation of rules is: invert the pattern then apply interface adaption and finally transform the body of the rule. The rules of an operation are transformed as follows:

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

The following lemma extends the pattern matching Lemma 6 to rules.

**Lemma 7.** *Let  $p = (p_1, \dots, p_n)$  and  $f p = r$  be a rule in  $P$ . Let  $e_1, \dots, e_n$  be terms,  $e = (e_1, \dots, e_n)$  and  $a = \mathcal{C}[rule(f p = r) e]$ .*

*If there exists  $\theta \in CSubst$  with  $e = p\theta$ , then  $a \mapsto^* \mathcal{C}[r\theta]$  and otherwise  $\llbracket a \rrbracket = \emptyset$ .*

*Proof (Idea).* Unfold definition of  $rule(\cdot)$  until Lemma 6 is applicable.

We can finally define how a whole program is transformed. The signature of the resulting program  $P'$  is an extension of the original one. For every constructor symbol  $c$  we introduce a new unary symbol  $invC$ , the corresponding destructor.

$$\begin{aligned} O_{P'} &= O_P \cup \{invC^1 \mid c^1 \in C_P\} \cup Prim \\ C_{P'} &= C_P \\ Prim &= \{(*)^3, (/)^3, (?)^3, fork^1, id^1, unit^1, fst^1, snd^1, val^2, curry^2, apply^1\} \end{aligned}$$

In the following definition  $prims$  are the operation definitions of  $(*)$  (4),  $(/)$  (6),  $fork$  (8),  $unit$  (10),  $id$  (11),  $unknown$  (22),  $fst$  (12),  $snd$  (13), and  $(?)$  (17).

$$\begin{aligned} op(f) &= f = rule(r_1) ? \dots ? rule(r_n) \text{ where } \{r_1, \dots, r_n\} = \{f p = r \in P\} \\ prog(P) &= prims \cup \{op(f) \mid f \in O_P\} \cup \{invC(c \mathbf{x}) = \mathbf{x} \mid c \in C_P\} \end{aligned}$$

Finally, we can put together the insights about the transformation.

**Theorem 1.** *Let  $P$  be a program. Then  $\llbracket P \rrbracket^{\perp \neq} = \llbracket \text{prog}(\text{str}(P)) \rrbracket$ .*

*Proof.* By Lemma 1 we have that the semantics of  $\text{str}(P)$  is equivalent to the semantics of  $P$  when replacing  $\perp$  with the special constructor  $\mathbb{U}$ , i.e.,  $\llbracket \text{str}(P) \rrbracket^{\perp \neq} = \llbracket P \rrbracket^{\perp \neq}$ . Also by Lemma 1 strict and lazy semantics for  $\text{str}(P)$  are equivalent, i.e.,  $\llbracket \text{str}(P) \rrbracket^{\perp \neq} = \llbracket \text{str}(P) \rrbracket$ . We now prove that there is a derivation  $e \mapsto^* t$  to a normal form  $t$  in the program  $\text{str}(P)$  iff there is a derivation  $(\text{exp}(e) \text{ int}(e)) \mapsto^* t$  in the transformed program  $\text{prog}(\text{str}(P))$ .

( $\Rightarrow$ ): By induction on the length  $n$  of the derivation  $e \mapsto^* t$ .

$n = 0$ : By Lemma 5 we have  $(\text{exp}(t) \text{ int}(t)) \mapsto^* t$ .

$n + 1$ : The derivation is of the form  $\mathcal{C}[f(p_1, \dots, p_n)\theta] \mapsto \mathcal{C}[r\theta] \mapsto^* t$ . By Lemma 7 the existence of  $\theta$  yields  $\mathcal{C}[\text{rule}(f(p_1, \dots, p_n) = r) (p_1, \dots, p_n)\theta] \mapsto^* \mathcal{C}[r\theta]$ . Therefore the induction hypothesis ensures the claim.

( $\Leftarrow$ ): By induction on the number  $n$  of applications of functions  $f \in O_P$  (i.e., excluding the applications of primitives introduced by the transformation).

$n = 0$ : By definition  $\text{exp}(e)$  contains exactly as many applications of functions  $f \in O_P$  as  $e$ . Naturally, the primitive functions do not apply functions of the original program  $P$ . As the semantics is strict such that all functions in  $e$  will actually be applied,  $n = 0$  implies that  $e$  is a constructor term. Lemma 5 yields therefore  $e = t$  and the derivation  $e \mapsto^* e$  in  $P$  exists trivially.

$n + 1$ : By definition of the transformation  $\text{prog}$  all functions of the resulting program but (?) have only a single rule. Therefore their application is the only source of non-determinism. Moreover, (?) is only introduced to combine the transformed rules of a function of the original program. Because of this and by Lemma 7 derivations in the transformed program are always of the form

$$\mathcal{C}[f p\theta] \mapsto \mathcal{C}[(r_1 ? \dots ? r_m) p\theta] \mapsto \mathcal{C}[r_i p\theta]$$

where  $f^n \in O_P$ ,  $p = (x_1, \dots, x_n)$ ,  $i \in \{1, \dots, m\}$  and  $r_i = \text{rule}(f(p_1, \dots, p_n) = e_i)$ .

By Lemma 7 the existence of a derivation to a normal form  $t$  implies that there exists a constructor substitution  $\sigma$  with  $p\theta = (p_1, \dots, p_n)\sigma$  and that the derivation above can be continued as:

$$\mathcal{C}[r_i (p_1, \dots, p_n)\sigma] \mapsto^* \mathcal{C}[e_i\sigma] \mapsto^* t$$

By induction hypothesis we may assume that there exists a derivation  $D$  corresponding to  $\mathcal{C}[e_i\sigma] \mapsto^* t$ . And, all in all, we can construct the following corresponding derivation in the original program:

$$\mathcal{C}[f(p_1, \dots, p_n)\sigma] \mapsto \underbrace{\mathcal{C}[e_i\sigma] \mapsto^* t}_D$$

## 4 Related and Future Work

Cunha, Pinto and Proença [?,?] present a framework for transformations of functional programs into point-free style. They implemented a library for point-free programming in Haskell and transform Haskell programs into point-free programs which are based on this library. Conceptually, their approach first

transforms a subset of Haskell to a simply-typed  $\lambda$ -calculus, and back to a Haskell program. Because of the intermediate transformation to  $\lambda$ -calculus, the resulting programs bear only a remote resemblance to the original. In contrast, one of our aims is to keep the resulting programs close to the original. For example, we preserve the recursive structure of the program instead of expressing it by primitive recursion operators and we keep the data types and definitions of the original program instead of transforming them into generic sum and product types.

There is a lot of work to employ category theory in order to enable the algebraic manipulation of *functional programs* from which we only mention [?]. We have the intuition 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 [?] and similar works 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.

As mentioned in the introduction we aim at using automatic proving like presented in [?] to prove for example the correctness of transformations. Furthermore we hope that the well-known area of relation algebra provides new insights into functional logic programming.

[?] presents a semantics for a functional language employing relation algebra. We want to investigate the relation with our approach as future work.

One of our future goals is to extend the presented approach to cover *function patterns* [?] for the first time. Function patterns allow operator definitions with arbitrary first order patterns. There seems to be a close correlation between function patterns and the inversion operator of relation algebra. The use of function patterns allows to generate even simpler point-free programs than presented here. Function patterns can be used to express arbitrary pattern matching by inverting the corresponding expression. Furthermore by employing function patterns we could decrease the number of primitives introduced by the transformation. For example, we could define `unknown` as the inversion of `unit`.

## A Comparing Transformations to Point-free Style

The following program is transformed by three different approaches, resulting in a different degree of readability.

```
data Nat = Zero | Succ Nat

len :: [a] -> Nat
len []   = Zero
len (h:t) = Succ (len t)
```

Result according to [?]:

```
len :: [a] -> Nat
len = hylo (L :: Mu (:+:) (Const One) Id)
  (app . (((curry (app . ((either . ((curry (inn . (inl . bang))))
/\ (curry (inn . (inr . snd)))))) /\ snd))) . bang) /\ id))
  (app . (((curry (app . ((either . ((curry (inl . bang))
/\ (curry (inr . (snd . snd)))))) /\ (out . snd)))) . bang) /\ id))
```

A simplification with the SimpliFree tool [?]:

```
len : List A -> Nat
len = hylo (\x-> case x (\y-> in (inl )) (\y-> in (inr y)))
  (\x-> (out x) (\y-> inl ) (\y-> inr (snd y)))
```

The presented Transformation:

```
data Nat = Zero () | Succ Nat | U ()

len :: List a -> Nat
len = invNil * Zero
  ? invCons * snd * len * Succ
  ? unit * U
```

## B Uncurried Programs

$$\begin{aligned}
\Sigma_{fo(P)} &= \{s^1 \mid s \in \Sigma_P\} \cup \{\mathbf{apply}^1, \mathbf{curry}^3\} \\
fo(P) &= \{f(fo(p_1), \dots, fo(p_n)) = fo(r) \mid f p_1 \dots p_n = r \in P\} \\
&\cup \{\mathbf{apply}(f, x) = f x, \mathbf{curry} f x y = \mathbf{apply}(f, (x, y))\} \\
fo(s^0) &= s^1() \\
fo(s^{n>0} e_1 \dots e_m) &= \begin{cases} \mathbf{curry}(\dots(\mathbf{curry} s^1)\dots) & , \text{ if } m = 0 \\ \underbrace{\mathbf{curry}(\dots(\mathbf{curry} s^1)\dots)}_{n-1} & , \text{ if } m = n \\ s^1(fo(e_1), \dots, fo(e_m)) & , \text{ if } m = n \\ \mathbf{apply}(fo(s^n e_1 \dots e_{m-1}), fo(e_m)) & , \text{ otherwise} \end{cases} \\
fo(x e_1 \dots e_m) &= \begin{cases} x & , \text{ if } m = 0 \\ \mathbf{apply}(fo(x e_1 \dots e_{m-1}), fo(e_m)) & , \text{ if } m > 0 \end{cases}
\end{aligned}$$

**Fig. 2.** Uncurrying Programs

Figure 2 shows a standard transformation of a program to a program with uncurried functions and constructors only. All functions and constructors of the resulting program are unary. The following lemma states that `curry` and `apply` correctly apply a higher order function to a pair of arguments.

**Proposition 5.** *For all symbols  $s$  and all  $n$  we have that*

$$\mathcal{C}[\underbrace{\mathbf{apply}(\dots \mathbf{apply}(\mathbf{curry}(\dots(\mathbf{curry} s^1)\dots), e_1)\dots, e_n)}_{n-1}] = \mathcal{C}[s^1((e_1, e_2), \dots, e_n)].$$

*Proof.*  $n = 1$  :  $\mathcal{C}[\mathbf{apply}(s^1, e_1)] = \mathcal{C}[s^1 e_1]$

$n + 1$  : Let  $cs = \underbrace{\mathbf{curry}(\dots(\mathbf{curry} s^1)\dots)}_{n-1}$  and

$$\mathcal{C}' = \mathcal{C}[\underbrace{\mathbf{apply}(\dots \mathbf{apply}([], e_3)\dots, e_{n+1})}_{n-1}].$$

$$\begin{aligned}
&\mathcal{C}'[(\mathbf{apply}(\mathbf{apply}(\mathbf{curry} cs, e_1), e_2))] \\
\{\text{ def apply } \} &\mapsto \mathcal{C}'[(\mathbf{apply}(\mathbf{curry} cs, e_1)) e_2] \\
\{\text{ def apply } \} &\mapsto \mathcal{C}'[(\mathbf{curry} cs) e_1] e_2 \\
\{\text{ def curry } \} &\mapsto \mathcal{C}'[\mathbf{apply}(cs, (e_1, e_2))] \\
\{\text{ induction hypothesis } \} &\mapsto \mathcal{C}[s^1((e_1, e_2), \dots, e_n)]
\end{aligned}$$

## C Detailed Proofs

*Proof (Proposition 1).* Direction ( $\supseteq$ ) is obvious. For ( $\subseteq$ ) we consider a derivation  $str(\mathcal{C}[e']) \mapsto str(\mathcal{C}[\perp]) \mapsto^* t$  and show by induction on the structure of  $\mathcal{C}$  that  $t = \perp$ .

$$\begin{aligned}
\mathcal{C} = [] &: str(\mathcal{C}[\perp]) = \perp \mapsto^* \perp \\
\mathcal{C} = (e \ \mathcal{C}') &: str(e \ \mathcal{C}') = (\mathbf{app} \ str(e) \ str(\mathcal{C}')) \\
&\{\text{induction hypothesis}\} \mapsto^* (\mathbf{app} \ str(e) \ \perp) \\
&\{\mathbf{app} \ \text{undefined for this application}\} \mapsto \perp \\
\mathcal{C} = (\mathcal{C}' \ e) &: str((\mathcal{C}' \ e)) = (\mathbf{app} \ str(\mathcal{C}') \ str(e)) \\
\text{case 1: } str(e) \mapsto^* \perp &: (\mathbf{app} \ str(e) \ \perp) \{\mathbf{app} \ \text{undefined}\} \mapsto \perp \\
\text{case 2: } str(e) \mapsto^* t \neq \perp &: (\mathbf{app} \ str(e) \ t) \{\text{ind. hyp.}\} \mapsto^* (\mathbf{app} \ \perp \ t) \\
&\{\mathbf{app} \ \text{undefined}\} \mapsto (\perp \ t) \mapsto \perp
\end{aligned}$$

*Proof (Proposition 2).* Because rule **B** is the only one which introduces the symbol  $\perp$  the two derivations  $e_1 \mapsto_{\mathcal{L}}^* t_1 := (s \ \overline{e'_i})$  and  $e_2 \mapsto_{\mathcal{L}}^* t_2 := (c \ \overline{e''_j})$  exist for appropriate  $s, c, \overline{e'_i}, \overline{e''_j}$ . Furthermore, as the programs are assumed well-sorted, the arity of  $s$  must be greater than  $i$ . Thus, for every such  $t_1, t_2$  the application  $(\mathbf{app} \ t_1 \ t_2)$  is defined as  $(t_1 \ t_2)$ .

*Proof (Proposition 3).* Direction ( $\Leftarrow$ ) is obvious since **B'** is a restricted version of **B**.

( $\Rightarrow$ ): A structural induction shows that every term  $e$  has a normal form  $t \in \mathcal{T}(C_P \cup \{\perp\}, \mathcal{X})$  with respect to  $\mapsto^*$  and, thus, rule **B'** can be applied after rewriting  $e_1, \dots, e_n$  to normal form.

$$\begin{aligned}
e = c^0 &: e = t \\
e = (e_1 \ e_2) &: \text{By induction hypothesis we have } e_1 \mapsto^* t_1 \text{ and } e_2 \mapsto^* t_2 \text{ with } \\
&t_1, t_2 \in \mathcal{T}(C_P \cup \{\perp\}, \mathcal{X}). \\
\text{case 1 } t_1 = \perp &: (\perp \ t_2) \mapsto \perp \{\text{def } \mathbf{B}'\} \\
\text{case 2 } t_1 = s \ u_1 \ \dots \ u_m &\text{ with } s^{n > m} \in \Sigma_P: \\
\text{case 2.1 } n = m + 1 \wedge s \in O_P &: s \ u_1 \ \dots \ u_m \ t_2 \mapsto \perp \{\text{def } \mathbf{B}'\} \\
\text{case 2.2 } n > m + 1 \vee s \in C_P &: s \ u_1 \ \dots \ u_m \ t_2 \in \mathcal{T}(C_P \cup \{\perp\}, \mathcal{X})
\end{aligned}$$

*Proof (Lemma 2).* By structural induction on  $i$ .

$$i = x : \mathcal{C}[sel(x, x) \ x\theta] \{\text{def } sel(\cdot, \cdot)\} = \mathcal{C}[\mathbf{id} \ x\theta] \{\text{def } \mathbf{id}\} \mapsto \mathcal{C}[x\theta]$$

$$i = (i_1, i_2), x \in var(i_2) \wedge x \notin var(i_1) :$$

$$\begin{aligned}
&\mathcal{C}[sel(x, i) \ i\theta] \\
\{\text{def } sel(\cdot, \cdot)\} &\mapsto \mathcal{C}[(\mathbf{fst} \ * \ sel(x, i_1)) \ i\theta] \\
\{\text{def } *\} &\mapsto \mathcal{C}[sel(x, i_1) \ (\mathbf{fst} \ i\theta)] \\
\{\text{def } \mathbf{fst}\} &\mapsto \mathcal{C}[sel(x, i_1) \ i_1\theta] \\
\{\text{ind. hypothesis}\} &\mapsto^* \mathcal{C}[x\theta]
\end{aligned}$$

$i = (i_1, i_2), x \notin var(i_2) \wedge x \in var(i_1) :$  analog to the previous case with **snd** instead of **fst**.

*Proof (Proposition 4).* induction on  $n$ .  $n = 1$  :

$$\begin{array}{l} \mathcal{C}[free\ i] \quad \{ def\ free \} \mapsto \mathcal{C}[(unit * unknown)\ i] \\ \{ def * \} \mapsto \mathcal{C}[unknown\ (unit\ i)] \quad \{ def\ unit \} \mapsto \mathcal{C}[unknown\ ()] \\ \{ def\ unknown \} \mapsto \mathcal{C}[x] \end{array}$$

where  $x \notin var(\mathcal{C}[i])$

$$n + 1 : \text{Let } fs = \underbrace{free \times \dots \times free}_{n-1}.$$

$$\begin{array}{l} \mathcal{C}[(fs \times free)\ i] \quad \{ def.\ of\ \times \} \mapsto \mathcal{C}[(fork * (fs/free))\ i] \\ \{ def * \} \mapsto \mathcal{C}[(fs/free)\ (fork\ i)] \\ \{ def\ fork \} \mapsto \mathcal{C}[(fs/free)\ (i, i)] \\ \{ def\ / \} \mapsto \mathcal{C}[(fs\ i, free\ i)] \\ \{ def\ free \} \mapsto \mathcal{C}[(fs\ i, (unit * unknown)\ i)] \\ \{ def\ unit \} \mapsto \mathcal{C}[(fs\ i, unknown\ (unit\ i))] \\ \{ def\ unknown \} \mapsto \mathcal{C}[(fs\ i, unknown\ ())] \\ \{ def\ unknown \} \mapsto \mathcal{C}[(fs\ i, x_{n+1})] \\ \{ ind.\ hypothesis \} \mapsto \mathcal{C}[(x_1, \dots, x_n, x_{n+1})] \end{array}$$

where  $x_n \notin var(\mathcal{C}[(fs\ i, i)])$

*Proof (Lemma 4).*

$$\begin{array}{l} \mathcal{C}[addfree(i_1, i_2)\ i_1\theta] \\ \{ def\ addfree(\cdot, \cdot) \} = \mathcal{C}[(\underbrace{free \times \dots \times free}_n * id) * adapt(i'_1, i_2)]\ i_1\theta \\ \\ \{ def * \} \mapsto \mathcal{C}[adapt(i'_1, i_2)\ ((free \times \dots \times free \times id)\ i_1\theta)] \\ \{ def\ \times, *, / \} \mapsto^* \mathcal{C}[adapt(i'_1, i_2)\ ((free \times \dots \times free)\ i_1, id\ i_1\theta)] \\ \{ def\ id \} \mapsto \mathcal{C}[adapt(i'_1, i_2)\ ((free \times \dots \times free)\ i_1, i_1\theta)] \\ \{ Lemma\ 4 \} \mapsto^* \mathcal{C}[adapt(i'_1, i_2)\ ((y_1, \dots, y_n), i_1\theta)] \\ = \mathcal{C}[adapt(i'_1, i_2)\ ((x_1, \dots, x_n), i_1)\rho\theta] \\ = \mathcal{C}[adapt(i'_1, i_2)\ i'_1\rho\theta] \\ \{ Lemma\ 3 \} \mapsto^* \mathcal{C}[i_2\rho\theta] = \mathcal{C}[i_2\theta] \end{array}$$

where  $y_1, \dots, y_n \notin var(\mathcal{C}[adapt(i'_1, i_2)\ (i_1, id\ i_1\theta)])$  and  $\rho = \{x_i \mapsto y_i\}$ .

*Proof (Lemma 5).* By structural induction on  $e$ .

$$\begin{array}{l} e = x : \\ \mathcal{C}[exp(x)\ int(x)\theta] \quad \{ def\ exp(\cdot) \} = \mathcal{C}[id\ int(x)\theta] \\ \{ def\ int(\cdot) \} = \mathcal{C}[id\ x\theta] \quad \{ def\ id \} \mapsto \mathcal{C}[x\theta] \\ e = s : \\ \mathcal{C}[exp(s)\ int(s)\theta] \quad \{ def\ exp(\cdot) \} = \mathcal{C}[(val\ s)\ int(s)\theta] \\ \{ def\ int(\cdot) \} = \mathcal{C}[(val\ s)\ ()] \quad \{ def\ val \} \mapsto \mathcal{C}[s] = \mathcal{C}[s\theta] \\ e = s\ (e_1, \dots, e_n) : \\ \mathcal{C}[exp(e)\ int(e)\theta] \\ \{ def\ exp(\cdot) \} = \mathcal{C}[(exp(e_1)/ \dots / exp(e_n)) * s]\ int(e)\theta \\ \{ def\ int(\cdot) \} = \mathcal{C}[(exp(e_1)/ \dots / exp(e_n)) * s]\ (int(e_1), \dots, int(e_n))\theta \\ \{ def * \} \mapsto \mathcal{C}[s\ ((exp(e_1)/ \dots / exp(e_n))\ (int(e_1)\theta, \dots, int(e_n)\theta))] \\ \{ def\ / \} \mapsto^* \mathcal{C}[s\ (exp(e_1)\ int(e_1)\theta, \dots, exp(e_n)\ int(e_n)\theta)] \\ \{ ind.\ hypothesis \} \mapsto^* \mathcal{C}[s\ (e_1\theta, \dots, e_n\theta)] = \mathcal{C}[e\theta] \end{array}$$

*Proof (Lemma 7).*

$$\begin{aligned}
& \mathcal{C}[(rule(f\ p = r)\ e)] \\
\{ \text{def } rule(\cdot) \} &= \mathcal{C}[(((invert(p_1)/\dots/invert(p_n)) * adp * exp(r))\ e)] \\
\{ \text{def } * \} &\mapsto \mathcal{C}[((adp * exp(r))\ ((invert(p_1)/\dots/invert(p_n))\ e))] \\
\text{case 1 there exists a } \theta \text{ with } e_1 = p_1\theta \wedge \dots \wedge e_n = p_n\theta & \\
\{ \text{def } / \} &\mapsto^* \mathcal{C}[((adp * exp(r))\ (invert(p_1)\ p_1\theta, \dots, invert(p_n)\ p_n\theta))] \\
\{ \text{Lem 6, } \theta \text{ subst.} \} &\mapsto^* \mathcal{C}[((adp * exp(r))\ (int(p_1), \dots, int(p_n))\theta)] \\
\{ \text{def } int(\cdot) \} &= \mathcal{C}[((adp * exp(r))\ int(p)\theta)] \\
\{ \text{def } *, \text{ def } adp \} &\mapsto \mathcal{C}[(exp(r)\ (addfree(int(p), int(r))\ int(p)\theta))] \\
\{ \text{Lem 4} \} &\mapsto^* \mathcal{C}[(exp(r)\ int(r)\theta)] \\
\{ \text{Lem 5} \} &\mapsto^* \mathcal{C}[r\theta] \\
\text{case 2 } e_i \text{ and } p_i \text{ not unifiable for one } i \in \{1, \dots, n\} & \\
\text{By Lemma 6 } \llbracket invert(p_i)\ e_i \rrbracket = \emptyset. \text{ As all applications are strict, this implies that} & \\
\text{the whole expression cannot be reduced to a normal form.} &
\end{aligned}$$