

A Relation Algebraic Semantics for a Lazy Functional Logic Language

Bernd Braßel and Jan Christiansen

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

Abstract. We propose a special relation algebraic model as semantics for lazy functional logic languages. The resulting semantics provides several interesting advantages over former approaches for this class of languages. For once, the equational reasoning supported by an *algebraic* approach implies a great increase in the economy and readability of proofs. Second, the richly developed field of relation algebra provides a pool of theorems and insights that can now be utilized in reasoning about functional logic programs. A further advantage is the strong and simple correspondence between functional logic programs and the semantics developed here. This correspondence enables us to go both ways, i.e. from program to semantics *and back*. The latter direction is essential for using relation algebraic transformations to optimize functional logic code. Last but not least, the development of the relation algebraic formalization gave us valuable insights, especially with respect to the desired properties of function inversion.

1 Introduction

In this paper we present a relation algebraic model as semantics for lazy functional logic languages. The functional logic community is very intent on not only developing powerful techniques with application to their languages, but also to have formal proofs of the soundness of these techniques. A recent example is the detailed consideration of a theoretic framework for debugging tools in [5]. Approximately every second publication in the field contains at least a portion of formal reasoning. Therefore, formalisms that provide proof economy, i.e., allow proofs in a compact and comprehensive way, are desirable. An *algebraic* approach seems ideal to meet this goal.

Having an algebraic formalism alone, however, is not enough for proof economy. The key quality to make proofs about programs comprehensive is a strong and simple correspondence between the program and the underlying algebra. We tried to meet this requirement by two related steps, first a transformation from general functional logic programs to a simpler subset and then a transformation of that subset to relation algebra. Each of these transformations was designed to bear as close a resemblance to the original as possible. The first step transforms arbitrary functional logic programs to a point-free subset of the

same language and was presented in [6, 7]. The resulting point-free programs are based on a small set of point-wise defined primitives. The term “point-free” refers to function declarations which do not contain an explicit access to the function’s arguments. The following example contains two point-free declarations and demonstrates how close the correspondence between point-free programs and their relation algebraic semantics is. Here and in the following we use the syntax of Curry [10] for the examples.

Example 1. The functions `not` and `iff` define negation and if-and-only-if for boolean values. These functions are the results of the transformation of the corresponding Curry functions to point-free style.

```
not :: Bool -> Bool
not = (invert true * false) ? (invert false * true)

iff :: (Bool, Bool) -> Bool
iff = (invert (true/id) * snd') ? (invert (true/id) * snd' * not)
```

The relation algebraic equations defining the semantics of these functions are:

$$\begin{aligned} \text{not} &= \text{true}^\top \circ \text{false} \cup \text{false}^\top \circ \text{true} \cup \mathbf{U} \\ \text{iff} &= (\text{true} \parallel \mathbf{1})^\top \circ \text{snd}' \cup (\text{false} \parallel \mathbf{1})^\top \circ \text{snd}' \circ \text{not} \cup \mathbf{U} \end{aligned}$$

The relation \mathbf{U} is part of correctly modeling laziness as explained in chapter 2.2. The operators $(*)$, $(/)$, $(?)$, `id` and `invert` are examples for the point-wise defined primitives. These primitives are directly translated to relation algebraic constructs like $(*)$ to relation multiplication \circ . The function `snd'` is not primitive and will be defined shortly.

In addition to obtain a close resemblance to the original a further aim of our approach was a small number of point-wise primitives. All in all there are only seven such primitives, in addition to those contained in the example, the remaining are called `fork` and `unit`. Additionally there is one external primitive `nf` to compute the normal form of an expression, cf. Section 2.3. A small set of primitives provides clear conceptual distinctions, e.g., all non-determinism is introduced by $(?)$, all laziness is due to `unit` and sharing is introduced by `fork`.

1.1 Point-wise and point-free Curry programs

In this section we give an introduction to the basic features of functional logic languages like Curry and provide examples for the subset of point-free Curry programs. We do not present the general definition of the transformation into this subset, cf. [7, 6] for details. Rather we present some illustrating examples in this section and define the syntax of the resulting programs in Section 2.3. Note that Curry is a statically typed language. Therefore it is ensured that well-typed programs with respect to the Curry type system are translated to expressions which are well-typed with respect to the requirements of relation algebra.

In comparison to other transformations to point-free style, we aimed at keeping the resulting programs as close to the original as possible. For example, we keep the original data constructors. Furthermore we preserve the original structure of pattern matching by employing a function inversion operator `invert :: (a -> b) -> (b -> a)`. In Curry functions are defined by a set of rules. We preserve the structure of these rules by combining their corresponding expressions by a non-deterministic choice operator `(?) :: a -> a -> a`. Both, function inversion and non-deterministic choice are part of the logic extensions of Curry. All function applications of the original programs are replaced by employing the operator `(*) :: (a -> b) -> (b -> c) -> (a -> c)`.

Example 2 (Point-Free Style 1: Values and Pattern Matching). The following Curry program introduces a boolean type by a `data` declaration and defines boolean negation by pattern matching.

```
data Bool = True | False

not :: Bool -> Bool
not True  = False
not False = True
```

The transformation preserves the declaration of `Bool` but introduces new constructing functions:

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

not :: Bool -> Bool
not = (invert true * false) ? (invert false * true)
```

In the point-free programs, values are mappings from the so called unit “`()`” to the original value, i.e., values become vectors. For example, we transform the expression `(not True)`, which evaluates to the value `False`, to `(true * not)`. This expression defines the mapping $\{() \mapsto \text{False}\}$. Therefore, evaluating the expression `(true * not) ()` in the transformed program yields `False`.

Functions with more than one argument are translated by introducing a parallel operator `(/) :: (a -> b, c -> d) -> (a, c) -> (b, d)`. Although function application in Curry may be written in a curried style, e.g., `(f x y)` instead of `f(x,y)` we have to stick to the uncurried version. This means that functions of type `a -> b -> c` become functions of type `(a,b) -> c`. Because of the isomorphism between the corresponding domains this is not a restriction. The introduction of uncurried functions only has an impact on higher order functions, which is not considered in this paper, cf. [7] for details.

Example 3 (Point-Free Style 2: Currying and Parallel Composition). The following function defines “if and only if” on boolean values.

```

iff :: Bool -> Bool -> Bool
iff True y = y
iff False y = not y

```

This declaration is transformed to:

```

iff :: (Bool,Bool) -> Bool
iff = (invert (true/id) * snd')
      ? (invert (true/id) * snd' * not)

```

The function `snd'` in Example 1 is part of what we call “interface adaption”. An interface adaption is an expression which rearranges, copies or discards the results of pattern matching in the way the translated body of a rule requires. For example, the body of both rules of function `iff` only requires the second argument, which amends the introduction of `snd'`. The challenge in a *lazy* language is to ensure that pattern matching in the original program is still demanded in the transformed program. For example, we have to ensure that the pattern matching on `True` in the original version of `iff` is still demanded in its point-free version. If we would use `snd` the pattern matching would not be demanded. In contrast, `snd'` is defined on the basis of the primitives `unit` and `fork`:

```

snd' :: ((),a) -> a
snd' = (invert unit / id) * invert fork

unit :: a -> ()
unit _ = ()

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

```

The function `snd'` is associated with the following relation algebraic expression:

$$snd' = (L^T \parallel I) \circ [I, I]^T \cup U$$

In the next section we present the representation of the seven primitives and the constructor functions in relation algebra. In the remainder of this section we want to highlight two of the main requirements our model needs to meet: lazy evaluation and call-time choice.

Example 4 (Lazy Evaluation). In a lazy functional logic language, the function `true`s declared in the following program defines a (potentially) infinite list of values `True`:

```

data [a] = [] | a : [a]

true :: [Bool]
true = True : true

head :: [a] -> a
head (x:_) = x

```

Using these definitions, the expression `(head trues)` is evaluated to `True`. The point-free version shows the same behavior:

```

nil :: () -> [a]
nil () = []

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

trues :: () -> [Bool]
trues = fork * (true/trues) * cons

head :: [a] -> a
head = invert cons * fst

fst :: (a,_) -> a
fst = (unit/id) * fst'

```

The function `fst'` is defined analog to `snd'` above. It is important that selecting the head of the potentially infinite list `trues` is well defined, i.e., the expression `(trues * head) ()` is evaluated to `True` if we use the above definitions.

Example 5 (Call-Time Choice). An important property of a semantics in connection with logic features in Curry is call-time choice. This property can be demonstrated by the following program.

```

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

shared, independent :: () -> (Bool,Bool)
shared      = coin * fork * iff
independent = fork * (coin/coin) * iff

```

The operations `shared` and `independent` have different semantics. The expression `shared ()` may only evaluate to `True`, whereas `independent ()`, in contrast, may evaluate to `True` or `False`. The intuitive meaning of call-time choice is that shared expressions share the same non-deterministic choices.

In Section 2 we present a relation algebraic model to capture the semantics of point-free programs including laziness and call-time choice. Extending the presented approach by the transformation proposed in [6, 7], we gain a semantics for first order functional logic languages in general. Section 3.1 presents the semantics of the examples concerning call-time choice.

2 The Relation Algebraic Model

We present both an abstract and a concrete semantics for lazy functional logic programs. For the concrete model, we construct terms over the signature of the original program. The semantics of a function declaration is the set of pairs of such terms that models the input output behavior of the function.

2.1 Constructors, Selectors and Values

Definition 1 (Polymorphic Signature). Let Θ be a set of type constructors and $TV = \{\alpha, \beta, \dots\}$ be a set of type variables. A Polymorphic Signature Σ is a set of operator symbols with an associated type $\Sigma = \{\mathbf{f} :: (\tau_1, \dots, \tau_n, \tau_0) \mid \tau_i \in T_\Theta(TV)\}$ where $T_\Theta(TV)$ denotes the set of terms over the alphabet Θ and the set of variables TV . For each operator symbol $f = \mathbf{f} :: (\tau_1, \dots, \tau_n, \tau_0) \in \Sigma$ we denote its arity by $ar(f) := n$ its argument type by $arg(f) = (\tau_1 \dots \tau_n)$ and its result type by $res(f) = \tau_0$. Furthermore, we use the notational convention that \mathbf{f} is the name of the symbol f .

For a type τ the signature Σ constraint to type τ is a subset of Σ defined as $\Sigma|_\tau = \{\mathbf{f} :: (\sigma(\tau_1) \dots \sigma(\tau_n), \tau) \in \Sigma \mid \sigma \text{ is a unifier of } \tau \text{ and } res(f)\}$.

Example 6 (Signatures). The signature of the program of Example 4 is

$$\Sigma ::= \{\mathbf{True} :: Bool, \mathbf{False} :: Bool, \mathbf{[]} :: [\alpha], (\cdot) :: (\alpha, [\alpha], [\alpha])\}$$

The signature Σ constraint to the type $[Bool]$ only contains the (monomorphic) list constructors $\Sigma|_{[Bool]} = \{\mathbf{[]} :: [Bool], (\cdot) :: (Bool, [Bool], [Bool])\}$.

In the following we assume without loss of generality Σ and Θ to be fixed and that there is no symbol with name \mathbf{u} in Σ .

Definition 2 (Sets of (Partial) Values). The set of values of type τ V_τ and the set of parital values of type τ PV_τ are defined as the terms of type τ over Σ and $\Sigma \cup \{\mathbf{u} :: \alpha\}$, respectively.

$$V_\tau := T_{\Sigma|_\tau}(\emptyset) \qquad PV_\tau := T_{\Sigma \cup \{\mathbf{u} :: \alpha\}|_\tau}(\emptyset)$$

We may omit τ when it is clear from the context. Note that the nullary constructor symbol \mathbf{u} is polymorphic.

Example 7 (Boolean and List Values). For the signature of lists of boolean values, cf. Example 6, we get the following sets of values.

$$\begin{aligned} V_{[Bool]} &:= \{\mathbf{[]}, \mathbf{True} : \mathbf{[]}, \mathbf{False} : \mathbf{[]}, \mathbf{True} : \mathbf{True} : \mathbf{[]}, \mathbf{True} : \mathbf{False} : \mathbf{[]}, \dots\} \\ PV_{[Bool]} &:= V_{[Bool]} \cup \{\mathbf{u}, \mathbf{u} : \mathbf{[]}, \mathbf{True} : \mathbf{u}, \mathbf{False} : \mathbf{u}, \mathbf{u} : \mathbf{True} : \mathbf{u}, \dots\} \end{aligned}$$

For the abstract semantics of constructor symbols, we employ the isomorphism between Σ and $\{+^2, *^2, ()^0\}$ which is often used in the context of generic programming. That is, we identify the sets PV_τ and $T_{\{+^2, *^2, ()^0\} \cup \{\mathbf{u} :: \alpha\}}(\emptyset)|_\tau$. Instead of using binary injections we use generalized injections that are defined by means of the binary injections ι_1 and ι_2 . A generalized injection $\iota_{n,k}$ injects a value to the k -th position into an n -ary sum. An n -ary sum is represented by a right parenthesised binary sum. Instead of stating k and n explicitly we use injections of the form $\iota_{c, \Sigma|_{res(c)}}$ where $c \in \Sigma$ determines k and n is the power of $\Sigma|_{res(c)}$.

Definition 3 (Semantics of V_τ). Let $c \in \Sigma$, $\tau = arg(c)$ and $\tau' = res(c)$. We define the semantics of c on the basis of the injection $inj_C :: \tau \leftrightarrow \tau'$.

$$\llbracket c \rrbracket := inj_c := \iota_{c, \Sigma \cup \{\mathbf{u} :: \alpha\}|_{\tau'}}$$

Let $(\tau_1, \dots, \tau_m) = \text{arg}(c)$. In the model of the concrete relation algebra the semantics of c is given by the following set.

$$\llbracket c \rrbracket = \{(x_1, \dots, (x_{m-1}, x_m)) \leftrightarrow c \ x_1 \dots x_m \mid x_i \in PV_{\tau_i}, 1 \leq i \leq m\}$$

We set $(x_1, \dots, (x_{m-1}, x_m)) = x_1$ if $m = 1$ and $(x_1, \dots, (x_{m-1}, x_m)) = ()$ if $m = 0$.

Example 8 (Value Semantics). According to Definition 3 the semantics of $(:)$ and \square of the signature of Boolean lists, cf. Example 6, are defined by:

Constructor	Abstract Semantics	Concrete Model
$(:)$	$\text{inj}_{(:)}$	$\{(x, y) \leftrightarrow x:y \mid x \in PV_{Bool}, y \in PV_{[Bool]}\}$
\square	inj_{\square}	$\{() \leftrightarrow \square\}$

As we have illustrated in Section 1.1, pattern matching is defined by a multiplication from the left with the inverse of the constructor semantics. The following lemma justifies this definition, stating that pattern matching with a pattern that corresponds to the outermost constructor peels off the constructor while pattern matching with all other patterns fails.

Lemma 1 (Pattern Matching). *Let $c, d \in \Sigma$.*

1. $\llbracket c \rrbracket \circ \llbracket c \rrbracket^\top = \mathbf{1}$
2. $c \neq d \Rightarrow \llbracket c \rrbracket \circ \llbracket d \rrbracket^\top = \mathbf{0}$

Proof. Induction over the structure of inj_c and the basic properties of injection.

Example 9 (Semantics Pattern Matching). Reconsider the definition of *not* from Example 1. For the application of *not* to the value *true* we get:

$$\begin{aligned} \text{true} \circ \text{not} &= \text{true} \circ (\text{true}^\top \circ \text{false} \cup \text{false}^\top \circ \text{true} \cup \mathbf{U}) \\ &= \text{true} \circ \text{true}^\top \circ \text{false} \cup \text{true} \circ \text{false}^\top \circ \text{true} \cup \text{true} \circ \mathbf{U} \\ &= \mathbf{1} \circ \text{false} \cup \mathbf{0} \circ \text{true} \cup \text{true} \circ \mathbf{U} \\ &= \text{false} \cup \text{true} \circ \mathbf{U} \end{aligned}$$

We have not yet defined what to do with the sub-term $\text{true} \circ \mathbf{U}$. We define \mathbf{U} in the next subsection.

2.2 Laziness and Demand

To model laziness in the relation algebraic semantics we define two special relations, namely \mathbf{U} and \mathbf{N} .

Definition 4 (Special Relations \mathbf{U} and \mathbf{N}). *The relation $\mathbf{U}_{\tau\tau'} :: \tau \leftrightarrow \tau'$ which introduces the value \mathbf{u} (unevaluated) and the relation $\mathbf{N}_\tau :: \tau \leftrightarrow \mathbf{1}$ which introduces the value $()$ (unit) are defined as:*

$$\mathbf{U}_{\tau\tau'} := \mathbf{L}_{\tau\mathbf{1}} \circ \text{inj}_{\mathbf{u}::\tau'} \qquad \mathbf{N}_\tau := \mathbf{L}_{\tau\mathbf{1}}$$

In the concrete relation algebra these relations are defined as follows.

$$\mathbf{U}_{\tau\tau'} = \{(x, u) \mid x \in PV_{\tau'}\} \quad \mathbf{N}_{\tau} = \{(x, ()) \mid x \in PV_{\tau}\}$$

We may omit τ, τ' when they are clear from the context.

Proposition 1 (Connection between U, N and L). *The following propositions hold for all relations $R :: \tau \leftrightarrow \tau'$ and the relations U and N.*

- $\mathbf{U}_{\tau\tau'} \circ \mathbf{U}_{\tau\tau'}^{\top} = \mathbf{L}_{\tau\tau'} = \mathbf{N}_{\tau} \circ \mathbf{N}_{\tau'}^{\top}$
- $\mathbf{U}_{\tau\tau'} \circ \mathbf{L}_{\tau'\tau''} = \mathbf{L}_{\tau\tau''}$ and $\mathbf{N}_{\tau} \circ \mathbf{L}_{\mathbf{1}\tau'} = \mathbf{L}_{\tau\tau'}$
- $R_{\tau\tau'} \neq \mathbf{O}_{\tau, \tau'} \Rightarrow R_{\tau\tau'} \circ \mathbf{U}_{\tau'\tau''} = \mathbf{U}_{\tau\tau'}$
- $R_{\tau\tau'} \neq \mathbf{O}_{\tau, \tau'} \Rightarrow R_{\tau\tau'} \circ \mathbf{N}_{\tau'} = \mathbf{N}_{\tau'}$

Proof. Immediate from Definition 4.

Example 10 (Properties of U). Employing Proposition 1 we can continue the reasoning from Example 9. $true \circ not = false \cup true \circ \mathbf{U} = false \cup \mathbf{U}$.

Combining the simple relations U and N is the center piece of our approach to model laziness. The key idea is to treat reducible expressions as non-deterministic choices. Either the expression is not evaluated and we associate the special value \mathbf{u} with it or the expression is evaluated yielding any value different from \mathbf{u} . If it is evaluated and there is no value different from \mathbf{u} we get the null relation.

We generally use expressions of the form $R \cup \mathbf{U}$, where R defines the value of the expression if it is evaluated. By \mathbf{U} we ensure that the expression can also be unevaluated. The relation \mathbf{N} is employed wherever the evaluation of an expression is not needed. Therefore, the key equation for modeling laziness is that for all R we have $(R \cup \mathbf{U}) \circ \mathbf{N} = \mathbf{N}$ which follows from Proposition 1.

Example 11 (Laziness). Reconsider the declarations of **fst**, **head** and **true**s from Example 4. In the next subsection we define the relation algebraic semantics of these declarations to be the smallest fixpoint of the following equations.

$$\begin{aligned} trues &= [\mathbf{I}, \mathbf{I}] \circ (true \parallel trues) \circ cons \cup \mathbf{U} \\ head &= cons^{\top} \circ fst \cup \mathbf{U} \\ fst &= (\mathbf{N} \parallel \mathbf{I}) \circ (\mathbf{N}^{\top} \parallel \mathbf{I}) \circ [\mathbf{I}, \mathbf{I}]^{\top} \end{aligned}$$

First we observe that we require the existence of suitable products because $fst = [\mathbf{N}^{\top} \circ \mathbf{N}, \mathbf{I}]^{\top} = [\mathbf{L}, \mathbf{I}]^{\top} = \pi_1$. Therefore, for the application $trues \circ head$ we get:

$$\begin{aligned} trues \circ head &= ([\mathbf{I}, \mathbf{I}] \circ (true \parallel trues) \circ cons \cup \mathbf{U}) \circ (cons^{\top} \circ \pi_1 \cup \mathbf{U}) \\ &= ([true, trues] \circ cons \cup \mathbf{U}) \circ cons^{\top} \circ \pi_1 \\ &\quad \cup ([true, trues] \circ cons \cup \mathbf{U}) \circ \mathbf{U} \\ \{\{\text{Proposition 1}\}\} &= ([true, trues] \circ cons \cup \mathbf{U}) \circ cons^{\top} \circ \pi_1 \cup \mathbf{U} \\ &= ([true, trues] \circ cons \circ cons^{\top} \cup \mathbf{U} \circ cons^{\top}) \circ \pi_1 \cup \mathbf{U} \\ \{\{\text{Lemma 1}\}\} &= ([true, trues] \circ \mathbf{I} \cup \mathbf{O}) \circ \pi_1 \cup \mathbf{U} \\ &= true \cup \mathbf{U} \end{aligned}$$

We have shown how unevaluated expressions can be discarded. The next definition provides the means to evaluate an expression to its normal form.

Definition 5 (Normal Form). *Let τ be a type and $c :: (\tau_1 \dots \tau_n, \tau_0) \in \Sigma$. The relation $\text{NF}_\tau :: \tau \leftrightarrow \tau$ is a partial identity which is defined by the following axioms.*

$$\text{NF}_\tau \subseteq \text{I}_\tau \quad \text{U}_{\tau\tau'} \circ \text{NF}_{\tau'} = \text{O}_{\tau\tau'} \quad \llbracket c \rrbracket \circ \text{NF}_{\tau_0} = (\text{NF}_{\tau_1} \parallel \dots \parallel \text{NF}_{\tau_n}) \circ \llbracket c \rrbracket$$

In the concrete relation algebra this relation is defined as follows.

$$\text{NF}_\tau = \{(x, x) \mid x \in V_\tau\}$$

On top level, e.g., on the prompt of an interactive environment for Curry like PAKCS [9], an expression e is evaluated non-deterministically to some or all of its possible normal forms. The set of possible normal forms is given in the concrete model of our semantics as the set $\{x \mid ((), x) \in \llbracket e \rrbracket \circ \text{NF}\}$.

Example 12 (Normal Form). The set of normal forms of expression `true * not` is `{False}`, cf. Examples 9, 10. The set of normal forms of `coin`, Example 5, is `{True, False}`. The expression `trues`, Examples 4 and 11, however, does not have any normal form, i.e., $\text{trues} \circ \text{NF} = \text{O}$.

In lazy functional logic languages normal forms are also required for other purposes. Let `Success` be the trivial type that is declared by `data Success = Success`. We can define the unification operator `(=:=) :: a -> a -> Success` as follows and discuss the implications in Section 3.3.

```
(=:=) = invert fork * nf * unit * success
```

2.3 Expressions and Programs

The transformation that was presented in [6, 7] transforms arbitrary Curry programs to point-free programs. Such (point-free) programs are expressions over a signature Σ which is a disjoint union of a set of symbols \mathcal{C} , called constructor symbols, and \mathcal{F} , called function symbols.

Definition 6 (Syntax). A program over signature Σ is a one to one mapping of the symbols $f \in \mathcal{F}$ to expressions E of the following form.

Expressions	$E ::= E * E$	{sequential composition}
	E / E	{parallel composition}
	$E ? E$	{non-deterministic choice}
	<code>invert</code> P	{inversion}
	<code>id</code>	{identity}
	<code>fork</code>	{sharing}
	<code>invert fork</code>	{unification}
	<code>unit</code>	{discarding}
	<code>invert unit</code>	{free variable}
	$o \in \Sigma$	{operator symbol}
Patterns	$P ::= P * P$	{sequential composition}
	P / P	{parallel composition}
	<code>id</code>	{identity}
	$c \in \mathcal{C}$	{constructor symbol}

We constrain the argument of `invert` to a subset of the expressions. Arguments of `invert` are composed of two groups of expressions: a) the primitives `*`, `/`, `id` along with the constructor symbols and b) the primitives `fork` and `unit`. On the Curry level, the first group is employed to translate patterns, i.e., the linear constructor expressions which are allowed on the left hand side of function declarations, cf. the examples in Section 1. The second group is used to define logic features that are provided by Curry in addition to non-determinism. These features are unification and free variables. In the previous section we have already stated the definition of the unification operator (`:=`). The next section discusses the modeling of logic features in detail.

In Definition 3 we have defined the semantics associated with constructor symbols. Next we define the semantics for the primitive operations.

Definition 7 (Semantics of Expressions). We define semantics for the point-free primitives as follows:

$$\begin{aligned}
\llbracket f * g \rrbracket &= \llbracket f \rrbracket \circ \llbracket g \rrbracket & \llbracket f / g \rrbracket &= \llbracket f \rrbracket \parallel \llbracket g \rrbracket \\
\llbracket f ? g \rrbracket &= \llbracket f \rrbracket \cup \llbracket g \rrbracket & \llbracket \text{invert } f \rrbracket &= \llbracket f \rrbracket^\top \\
\llbracket \text{id} \rrbracket &= \text{I} \\
\llbracket \text{fork} \rrbracket &= [\text{I}, \text{I}] & \llbracket \text{unit} \rrbracket &= \text{N}
\end{aligned}$$

Example 13 (Semantics of Expressions). For the expressions on the right hand sides of Example 5 we get:

$$\begin{aligned}
\llbracket \text{true} ? \text{false} \rrbracket &= \text{inj}_{\text{True}} \cup \text{inj}_{\text{False}} \\
\llbracket \text{coin} * \text{fork} * \text{iff} \rrbracket &= \llbracket \text{coin} \rrbracket \circ [\text{I}, \text{I}] \circ \llbracket \text{iff} \rrbracket \\
\llbracket \text{fork} * (\text{coin}/\text{coin}) * \text{iff} \rrbracket &= [\text{I}, \text{I}] \circ (\llbracket \text{coin} \rrbracket \parallel \llbracket \text{coin} \rrbracket) \circ \llbracket \text{iff} \rrbracket
\end{aligned}$$

In the concrete model the first expression denotes the set $\{(\text{(), True}), (\text{(), False})\}$. The sets denoted by the other two expressions are not yet defined because they contain the semantics of function symbols.

The last missing step to complete the semantics for Curry is associating the function symbols $f \in \mathcal{F}$ with a semantics. A program is a set of pairs of elements of \mathcal{F} and expressions.

Definition 8 (Semantics of Programs). *Let n be a natural number and $\mathcal{P} = \{f_i \mapsto e_i \mid 1 \leq i \leq n\}$ be a program. The semantics of \mathcal{P} is the smallest solution of the set of equations $\{\llbracket f_i \rrbracket = \llbracket e_i \rrbracket \cup \mathbf{U} \mid 1 \leq i \leq n\}$.*

Note, that we use the terms $\llbracket \mathbf{f} \rrbracket$ where $f \in \mathcal{F}$ as variables of the equation system. This motivates to write, e.g., *not* instead of $\llbracket \mathbf{not} \rrbracket$, clarifying that *not* is essentially a place holder.

Since none of our translations uses any form of negation the framework of relation algebra ensures the existence of the fixpoints required to solve the equation system.

Example 14 (Program Semantics). Recall the declarations of **true**s and **head** in Example 4. In the concrete model the semantics of this program is:

$$\begin{aligned} \mathit{trues} &= \{((\mathbf{u}), \mathbf{u}), ((\mathbf{True}), \mathbf{u}), ((\mathbf{True}), \mathbf{True}), \mathbf{u} \dots\} \\ \mathit{head} &= \{(x : y, x) \mid x \in PV_\alpha, y \in PV_{[\alpha]}\} \cup \{(y, \mathbf{u}) \mid y \in PV_{[\alpha]}\} \end{aligned}$$

The semantics associated with *trues* is identical to the standard approaches to model laziness, i.e., employ ideals in complete partial orders (CPO) for functional programming, e.g., [12], and cones for functional logic programs respectively, cf. [8]. We think that the beauty of the presented approach is that no additional concepts like a CPO to relation algebra is needed. In this way a uniform and high level framework is available for semantics, program analysis, partial evaluation, etc.

3 Logic Features

This section contains some considerations on how the presented semantics models the logic features of languages like Curry.

3.1 Call-Time Choice

In Section 1 we emphasised that our semantics has to correctly model call-time choice. This means in essence, that shared expressions share non-deterministic choices. The reason why the presented semantics correctly reflects call-time choice can be subsumed as follows. In general all sharing is introduced by the primitive **fork**. The semantics would be run-time choice iff the expressions $\mathbf{f} * \mathbf{fork}$ and $\mathbf{fork} * (\mathbf{f} / \mathbf{f})$ are equal regardless of \mathbf{f} . In contrast, in relation algebra the following two properties hold.

$$\begin{aligned} R \circ [I, I] &\subseteq [I, I] \circ (R \parallel R) \\ R \text{ univalent} &\iff R \circ [I, I] = [I, I] \circ (R \parallel R) \end{aligned}$$

Example 15 (Call-Time Choice Revisited). We can reconsider Examples 1 and 5. In the concrete model, *iff* and *coin* are associated with the sets:

$$\begin{aligned} \text{iff} &= \{((\mathbf{True}, \mathbf{True}), \mathbf{True}), ((\mathbf{True}, \mathbf{False}), \mathbf{False}), \\ &\quad ((\mathbf{False}, \mathbf{False}), \mathbf{True}), ((\mathbf{False}, \mathbf{True}), \mathbf{False})\} \cup \mathbf{U} \\ \text{coin} &= \{((\), \mathbf{True}), ((\), \mathbf{False}), ((\), \mathbf{u})\} \end{aligned}$$

The equations for *shared* and *independent* are:

$$\begin{aligned} \text{shared} &= \text{coin} \circ [1, 1] \circ \text{iff} \cup \mathbf{U} \\ \text{independent} &= [1, 1] \circ (\text{coin} \parallel \text{coin}) \circ \text{iff} \cup \mathbf{U} = [\text{coin}, \text{coin}] \circ \text{iff} \cup \mathbf{U} \end{aligned}$$

The important point is that the definition of the tupling ensures that $\text{coin} \circ [1, 1] = \{((\), (\mathbf{True}, \mathbf{True})), ((\), (\mathbf{False}, \mathbf{False})), ((\), (\mathbf{u}, \mathbf{u}))\}$ whereas $[\text{coin}, \text{coin}]$ associates all possible pairs of the set $\{\mathbf{True}, \mathbf{False}, \mathbf{u}\}$ with $(\)$. Therefore, we get, as intended:

$$\begin{aligned} \text{shared} \circ \mathbf{NF} &= \{((\), \mathbf{True})\} \\ \text{independent} \circ \mathbf{NF} &= \{((\), \mathbf{True}), ((\), \mathbf{False})\} \end{aligned}$$

3.2 Free Variables

Curry allows declarations of the form `let x free in e`, where e is an expression. The intended meaning is that free variables are substituted with constructor terms as needed to compute the normal form of a given expression. It is important that the substituted constructor terms can contain other free variables, but a mutual dependency of such bindings is not allowed.

The transformation employs the expression `(invert unit)` to introduce free variables. By Definition 7, this expression is associated with the relation algebraic expression $L_{\tau \mathbf{1}}^\top = L_{\mathbf{1}\tau}$ for any type τ . By definition, this expression is a vector of all partial values of type τ . This indeed captures the intended semantics of free variables, because the partial values model the case that a variable has been substituted with a term containing other variables. The notion of an identity on free variables needed in other frameworks is not necessary here. A variable can only appear at different positions of a constructor term if it was shared. Therefore, the call-time choice mechanism considered in the previous section correctly takes care of this case.

Example 16 (Free Variables). Applying the function `not` from Example 2 to a free variable, i.e., evaluating `(let x free in not x)`, yields non-deterministically `True` or `False`, as does the result of its transformation `invert unit * not`. The semantics associated with `not` is:

$$\text{not} = \{(\mathbf{True}, \mathbf{False}), (\mathbf{False}, \mathbf{True})\} \cup \mathbf{U}$$

The evaluation of `invert unit * not` in the context of this program, yields, as intended:

$$L_{\mathbf{1}B\text{ool}} \circ \text{not} \circ \mathbf{NF} = \{((\), \mathbf{False}), ((\), \mathbf{True})\}$$

Likewise, using the free variable twice, e.g., `(let x free in iff (not x) x)` yields `False` as does the transformed expression `invert unit * fork * (not/id) * iff`. Accordingly, the associated relation algebraic expression yields the intended semantics for the same reasons discussed above in Example 15.

$$\mathbf{L}_{1\text{Bool}} \circ [1, 1] \circ (\text{not} \parallel 1) \circ \text{iff} \circ \mathbf{NF} = \{(\text{()}, \text{False})\}$$

3.3 Unification

Closely related to the topic of free variables is unification. In Curry, for given expressions e_1, e_2 the expression $e_1 := e_2$ iff e_1 and e_2 can be evaluated to the same ground constructor term. This evaluation may include the substitution of free variables, which in the most basic version may only be bound to ground constructor terms, i.e., terms without free variables. This basic version is, what we have realized in this approach.

We have already presented the definition of the unification operator in point-free style. From that definition we get the relation algebraic definition for the unification relation, which we denote by *unify*:

$$\text{unify} = [1, 1]^\top \circ \mathbf{NF} \circ \mathbf{N} \circ \text{inj}_{\text{Success}}$$

The reason why this expression does indeed capture the intended behavior is as follows. Since \mathbf{NF} is injective, $\mathbf{NF} \subseteq 1$, *unify* is equivalent to the expression $[\mathbf{NF}, \mathbf{NF}] \circ \mathbf{N} \circ \text{inj}_{\text{Success}}$. Also, by definition of tupling, whenever we have a multiplication $[R, S] \circ \text{unify}$, we can instead write $(R \circ \mathbf{NF} \cap S \circ \mathbf{NF}) \circ \mathbf{N} \circ \text{inj}_{\text{Success}}$. Considering the definition of \mathbf{NF} , this term elegantly captures the basic definition, that the arguments of *unify* have to be evaluable to the same normal form.

4 Related Work

There are several semantics for functional logic languages, capturing various levels of abstraction. The standard operational big steps semantics is presented in [1]. A model theoretic semantics is proposed in [8] and there are several extension for this approach. We believe that the approach presented here provides a more uniform framework based on which further development, e.g., towards program analysis and partial evaluation is more direct. Nevertheless, a close comparison to the semantics of [8] is desirable for future work, ideally a proof of equivalence.

The way we present laziness in our concrete model is very similar, if not identical to the standard way of theoretical treatments of lazy semantics of programming languages, e.g., [3, 12]. However, we achieve the same effects by much simpler means. This is, of course, mainly because the framework of relation algebra already provides a rich setting including the properties of a complete lattice.

Finally, this work is related to other approaches to capture the semantics of programming languages employing relation algebra, e.g., [13]. Also with comparison to [13], our approach to capture laziness is simpler. However, the approach

in [13] is much more developed, covering for example higher order. Here also, a closer comparison is desirable future work.

5 Conclusion

We have presented a relational semantics for the first order part of the functional logic programming language Curry. The presented semantics provides a one-to-one correspondence between relation algebra and (point-free) Curry programs. This property opens a wide field of applications for this semantics. One possible application is the proof of correctness of transformations of Curry programs. This is an even more appealing application because the algebraic structure allows economic proofs, i.e., proofs that are compact and comprehensive.

Another direction for application is program inversion as presented for functional programs in [4] and [11]. Whereas in the functional paradigm every resulting non-deterministic definition *must* be eliminated, the framework of functional logic languages allows much less restricted use of algebraic methods.

There is a close relation between function inversion and the so called function patterns. Function patterns are an extension of functional logic languages introduced in [2]. For future work we aim at defining a semantics for function patterns for the first time. Although in a strict version of the presented semantics function patterns are a straightforward extension, their integration into the lazy approach is not trivial. Already, the presented semantics has provided valuable new insights for the possibilities of providing a semantics of functions patterns. The overhead to evaluate function patterns can be considerable [2]. Therefore, one application of a relation algebraic semantics of such patterns would be to derive equivalent programs without them. Such derivations could be directly employed to implement optimization techniques.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the International Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.
3. Rudolf Berghammer. *Semantik von Programmiersprachen*. Logos Verlag Berlin, 2001.
4. R. Bird and O. de Moor. *Algebra of programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
5. B. Braßel, S. Fischer, M. Hanus, F. Huch, and G. Vidal. Lazy call-by-value evaluation. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, 2007. To be published.
6. Bernd Brassel and Jan Christiansen. Denotation by transformation - towards obtaining a denotational semantics by transformation to point-free style. Technical report, August 2007.

7. Bernd Brassel and Jan Christiansen. Towards a new denotational semantics for curry and the algebra of curry. Technical report, August 2007.
8. J. C. González-Moreno, Maria Teresa Hortalá-González, Francisco Javier López-Fraguas, and Mario Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *J. Log. Program.*, 40(1):47–87, 1999.
9. M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. Pakcs: The portland aachen kiel curry system. Available at <http://www.informatik.uni-kiel.de/~pakcs/>, 2003.
10. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at <http://www.informatik.uni-kiel.de/~curry>, 2006.
11. S.-C. Mu. *A Calculational Approach to Program Inversion*. PhD thesis, Oxford University Computing Laboratory, 2003.
12. Peter Thiemann. *Grundlagen der funktionalen Programmierung*. B. G. Teubner Stuttgart, 1994.
13. Hans Zierer. *Programmierung mit Funktionsobjekten: Konstruktive Erzeugung semantischer Bereiche und Anwendung auf die partielle Auswertung*. PhD thesis, Technische Universität München, Fakultät für Informatik, 1988.