Relational algebra: a Kleene algebra central to the mathematics of program construction

J.N. Oliveira

Dept. Informática,
Universidade do Minho
Braga, Portugal

II Jornadas Luso-Galegas de Álgebra e Computação
Braga, 10-10-2008
Interaction between maths and computing:

- computers helping maths: theorem proving, computational maths etc
- maths helping computing: many examples, among which the algebra of programming (AoP)

While the former are widely acknowledged, among the latter AoP is known only to the initiated.

This talk aims at framing AoP in its proper algebraic context while showing its relevance to program construction.

It all starts from semirings of computations [3]...
On maths and computing

Interaction between maths and computing:

• computers helping maths: theorem proving, computational maths etc

• maths helping computing: many examples, among which the algebra of programming (AoP)

While the former are widely acknowledged, among the latter AoP is known only to the initiated.

• This talk aims at framing AoP in its proper algebraic context while showing its relevance to program construction.

It all starts from semirings of computations [3]...
Semirings of computations

Abstract notion of a computation:

Semiring \((S, +, \cdot, 0, 1)\) inhabited by computations (eg. instructions, statements) where

- \(x \cdot y\) (usually abbreviated to \(xy\)) captures sequencing
- \(x + y\) captures choice (alternation)
- \(0\) means death
- \(1\) means skip (do nothing)

Technically:

- \((M, \cdot, 1)\) is a monoid
- \((M, +, 0)\) is a Abelian monoid
- \((\cdot)\) distributes over \((+)\)
- \(0\) annihilates \((\cdot)\)
Idempotency

• If \( x + x = x \) holds for all \( x \), then

\[
x \leq y \quad \text{def} \quad x + y = y
\]

is a partial order.

• Clearly, \( 0 \leq x \) for all \( x \) and \( (+) \) is the \textit{lub} with respect to \( \leq \):

\[
x + y \leq z \quad \iff \quad x \leq z \land y \leq z
\]

\textbf{NB}: \( z := x + y \) in (2) means \( x + y \) is upper bound; \( \iff \) means it is the \textbf{least} upper bound (\textit{lub}).
A Kleene algebra [5] adds to semiring \((S, +, \cdot, 0, 1)\) the *Kleene star* operator \((\ast)\) such that

\[
y + x(x^\ast y) \leq x^\ast y
\]

(3)

\[
y + (yx^\ast) x \leq yx^\ast
\]

(4)

and

\[
y + xz \leq z \Rightarrow x^\ast y \leq z
\]

(5)

\[
y + zx \leq z \Rightarrow yx^\ast \leq z
\]

(6)

These basically establish \(x^\ast y\) and \(yx^\ast\) as prefix points of (monotonic) functions \((y + x \cdot \underline{\_})\) and \((y + \underline{\_} \cdot x)\), respectively.
KATs (tests and domains)

KAT = Kleene algebra with tests

- every $p$ below 1 ($p \leq 1$) is a test and such that, for every such $p$ there is $\neg p$ (the complement of $p$) such that
  
  \[ p + \neg p = 1 \]
  
  \[ p \cdot \neg p = 0 = \neg p \cdot p \]

- Recent addition to semirings (inc. KATs) of a domain operator $d(x)$ capturing “enabledness” and satisfying axioms
  
  \[ d(x) \leq 1 \]
  
  \[ d(0) = 0 \]
  
  \[ d(x + y) = d(x) + d(y) \]
  
  \[ d(xy) = d(x \cdot d(y)) \]
  
  \[ x \leq d(x) x \]
Binary relations

The algebra of **binary relations** is a well known KAT:

<table>
<thead>
<tr>
<th>KAT</th>
<th>Binary relations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \cdot y$</td>
<td>$R \cdot S$</td>
<td>composition</td>
</tr>
<tr>
<td>$x + y$</td>
<td>$R \cup S$</td>
<td>union</td>
</tr>
<tr>
<td>0</td>
<td>$\perp$</td>
<td>empty relation</td>
</tr>
<tr>
<td>1</td>
<td>$\text{id}$</td>
<td>identity relation</td>
</tr>
<tr>
<td>$x \leq y$</td>
<td>$R \subseteq S$</td>
<td>inclusion</td>
</tr>
<tr>
<td>$p, \neg p$</td>
<td>$R \subseteq \text{id}, \neg R = \text{id} - R$</td>
<td>coreflexive relations</td>
</tr>
<tr>
<td>$d(x)$</td>
<td>$\delta R$</td>
<td>domain of $R$</td>
</tr>
</tbody>
</table>

Moreover, they form a complete, distributive lattice once glbs

$$X \subseteq R \cap S \iff (X \subseteq R) \land (X \subseteq S)$$  \hspace{1cm} (7)

and supremum $\top$ are added.
How useful are binary relations?

- Not much if regarded merely as “sets of pairs”
- Very useful indeed — as a device for the algebraization of logic — if regarded as “arrows” ie. morphisms of a particular allegory [4]
- Arrows bring about a type discipline which leads to good things such as parametric polymorphism, etc etc
Relations as morphisms

Binary relations are typed:

**Arrow notation**

Arrow $A \xrightarrow{R} B$ denotes a binary relation from $A$ (source) to $B$ (target).

$A, B$ are types. Writing $B \xleftarrow{R} A$ means the same as $A \xrightarrow{R} B$.

**Infix notation**

The usual infix notation used in natural language — eg. 

\[ \text{John IsFatherOf Mary} \]

— and in maths — eg. 

\[ 0 \leq \pi \]

— extends to arbitrary $B \xleftarrow{R} A$: we write

\[ b R a \]

to denote that $(b, a) \in R$. 
Relations as morphisms

Binary relations are typed:

**Arrow notation**

Arrow \( A \xrightarrow{R} B \) denotes a binary relation from \( A \) (source) to \( B \) (target).

\( A, B \) are types. Writing \( B \xleftarrow{R} A \) means the same as \( A \xrightarrow{R} B \).

**Infix notation**

The usual infix notation used in natural language — eg.

\( \text{John IsFatherOf Mary} \)

— and in maths — eg.

\( 0 \leq \pi \)

— extends to arbitrary \( B \xleftarrow{R} A \): we write

\( b \xrightarrow{R} a \)

to denote that \( (b, a) \in R \).
Functions are relations

- Lowercase letters (or identifiers starting by one such letter) will denote special relations known as **functions**, eg. \( f, g, \text{suc}, \) etc.

- We regard **function** \( f : A \longrightarrow B \) as the binary **relation** which relates \( b \) to \( a \) iff \( b = f(a) \). So,

\[
\text{\( b f a \) literally means} \quad b = f(a)
\]

- Therefore, we generalize

\[
\begin{array}{c}
\text{\( B \leftarrow^f A \)} \\
\text{\( b = f(a) \)}
\end{array}
\quad \text{to} \quad 
\begin{array}{c}
\text{\( B \leftarrow^R A \)} \\
\text{\( b R a \)}
\end{array}
\]

- So, **function** \( \text{id} \) is the equality (equivalence) **relation**:

\[
\text{\( b \text{id} a \) means the same as} \quad b = a
\]
Composition

Function composition

\[ B \xleftarrow{f} A \xleftarrow{g} C \]

\[ b = f(g \ c) \]

extends to \( R \cdot S \) in the obvious way:

\[ b(R \cdot S)c \iff \langle \exists a :: b R a \land a S c \rangle \]

Note how this rule removes quantifier \( \exists \) when applied from right to left.
Converses

Every relation \( B \leftarrow R \rightarrow A \) has a converse \( B \rightarrow R^\circ \rightarrow A \) which is such that, for all \( a, b \),

\[
a(R^\circ)b \iff b R a
\]  

(10)

Note that converse commutes with composition

\[
(R \cdot S)^\circ = S^\circ \cdot R^\circ
\]  

(11)

and cancels itself

\[
(R^\circ)^\circ = R
\]  

(12)
Function converses

Function converses $f^\circ, g^\circ$ etc. always exist (as relations) and enjoy the following (very useful) property:

$$(f \circ b)R(g \circ a) \iff b(f^\circ \cdot R \cdot g)a$$  \hspace{1cm} (13)

cf. diagram:
Why \( id \) (really) matters

Terminology:

- Say \( R \) is **reflexive** iff \( id \subseteq R \)
  
  pointwise:
  \[
  \langle \forall a :: a R a \rangle
  \]

- Say \( R \) is **coreflexive** iff \( R \subseteq id \)
  
  pointwise:
  \[
  \langle \forall b, a :: b R a :: b = a \rangle
  \]

Define, for \( B \xleftarrow{R} A \):

<table>
<thead>
<tr>
<th>Kernel of ( R )</th>
<th>Image of ( R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A \xleftarrow{\ker R} A )</td>
<td>( B \xleftarrow{\img R} B )</td>
</tr>
<tr>
<td>( \ker R \triangleq R^\circ \cdot R )</td>
<td>( \img R \triangleq R \cdot R^\circ )</td>
</tr>
</tbody>
</table>
Kernels of functions:

\[ a'(\ker f)a \]

\[ \Leftrightarrow \{ \text{substitution} \} \]

\[ a'(f^\circ \cdot f)a \]

\[ \Leftrightarrow \{ \text{PF-transform rule (13)} \} \]

\[ (f\ a') = (f\ a) \]

In words: \( a'(\ker f)a \) means \( a' \) and \( a \) “have the same \( f \)-image”
Binary relation taxonomy

Topmost criteria:

injective  entire  simple  surjective

Definitions:

<table>
<thead>
<tr>
<th></th>
<th>Reflexive</th>
<th>Coreflexive</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ker R$</td>
<td>entire $R$</td>
<td>injective $R$</td>
</tr>
<tr>
<td>$\text{img } R$</td>
<td>surjective $R$</td>
<td>simple $R$</td>
</tr>
</tbody>
</table>

Facts:

\[
\ker (R^\circ) = \text{img } R \quad (15)
\]
\[
\text{img } (R^\circ) = \ker R \quad (16)
\]
Binary relation taxonomy

The whole picture:

```
  injective  entire  simple  surjective
   ↖ ↖    ↖    ↖    ↖
   ↖    ↖    ↖    ↖
  representation  function  abstraction
  ↖    ↖    ↖    ↖
  injection  surjection
  ↖    ↖
  bijection
```

Clearly:

- converse of *injective* is *simple* (and vice-versa)
- converse of *entire* is *surjective* (and vice-versa)
- smaller than injective (simple) is injective (simple)
- larger than entire (surjective) is entire (surjective)
A function $f$ is a binary relation such that

<table>
<thead>
<tr>
<th>Pointwise</th>
<th>Pointfree</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Left” Uniqueness</td>
<td>img $f \subseteq id$</td>
</tr>
<tr>
<td>$b \in f \land b' \in f \land a \Rightarrow b = b'$</td>
<td></td>
</tr>
<tr>
<td>Leibniz principle</td>
<td>id $\subseteq \ker f$</td>
</tr>
<tr>
<td>$a = a' \Rightarrow f , a = f , a'$</td>
<td></td>
</tr>
</tbody>
</table>

which both together are equivalent to any of “al-gabr” rules

$$f \cdot R \subseteq S \Leftrightarrow R \subseteq f^\circ \cdot S$$  \hspace{1cm} (18)

$$R \cdot f^\circ \subseteq S \Leftrightarrow R \subseteq S \cdot f$$  \hspace{1cm} (19)
“Al-gabr” rules?

Recall *calculus of al-gabr and al-muqâbala*\(^1\):

**al-gabr**
\[
x - z \leq y \iff x \leq y + z
\]

**al-hatt**
\[
x \cdot z \leq y \iff x \leq y \cdot z^{-1} \quad (z > 0)
\]

**al-muqâbala**

Ex:
\[
4x^2 + 3 = 2x^2 + 2x + 6 \iff 2x^2 = 2x + 3
\]

---

\(^1\)Cf. *Kitâb al-muhtasar fi hisab al-gabr wa-almuqâbala* by Abû Al-Huwârizmî, the famous 9c Persian mathematician.
Example: function equality

Equating functions means comparing them in either way:

\[ f = g \iff f \subseteq g \iff g \subseteq f \]  \hspace{1cm} (20)

Calculation:

\[ f \subseteq g \]
\[ \iff \quad \{ \text{“al-gabr” (18) on } f \} \]
\[ id \subseteq f^\circ \cdot g \]
\[ \iff \quad \{ \text{“al-gabr” (19) on } g \} \]
\[ g^\circ \subseteq f^\circ \]
\[ \iff \quad \{ \text{converses} \} \]
\[ g \subseteq f \]
A “Laplace transform analog” for logical quantification

The pointfree (PF) transform

<table>
<thead>
<tr>
<th>φ</th>
<th>PF φ</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle \exists a :: b \mathrel{R} a \land a \mathrel{S} c \rangle$</td>
<td>$b(R \cdot S)c$</td>
</tr>
<tr>
<td>$\langle \forall a, b :: b \mathrel{R} a \Rightarrow b \mathrel{S} a \rangle$</td>
<td>$R \subseteq S$</td>
</tr>
<tr>
<td>$\langle \forall a :: a \mathrel{R} a \rangle$</td>
<td>$id \subseteq R$</td>
</tr>
<tr>
<td>$\langle \forall x :: x \mathrel{R} b \Rightarrow x \mathrel{S} a \rangle$</td>
<td>$b(R \setminus S)a$</td>
</tr>
<tr>
<td>$\langle \forall c :: b \mathrel{R} c \Rightarrow a \mathrel{S} c \rangle$</td>
<td>$a(S \setminus R)b$</td>
</tr>
<tr>
<td>$b \mathrel{R} a \land c \mathrel{S} a$</td>
<td>$(b, c) \langle R, S \rangle a$</td>
</tr>
<tr>
<td>$b \mathrel{R} a \land d \mathrel{S} c$</td>
<td>$(b, d)(R \times S)(a, c)$</td>
</tr>
<tr>
<td>$b \mathrel{R} a \land b \mathrel{S} a$</td>
<td>$b (R \cap S) a$</td>
</tr>
<tr>
<td>$b \mathrel{R} a \lor b \mathrel{S} a$</td>
<td>$b (R \cup S) a$</td>
</tr>
<tr>
<td>$(f \circ b) \mathrel{R} (g \circ a)$</td>
<td>$b(f^\circ \cdot R \cdot g)a$</td>
</tr>
<tr>
<td><strong>TRUE</strong></td>
<td>$b \top a$</td>
</tr>
<tr>
<td><strong>FALSE</strong></td>
<td>$b \bot a$</td>
</tr>
</tbody>
</table>

What do $\langle R, S \rangle$, $R \times S$ etc mean?
Forks for tupling

The fork ("split") combinator is essential for transforming predicates holding more than two quantified variables. From the definition,

\[(b, c)\langle R, S\rangle a \iff b R a \land c S a\]  \hspace{1cm} (21)

which PF-transforms to

\[\langle R, S \rangle = \pi_1 \cdot R \cap \pi_2 \cdot S\] \hspace{1cm} (22)

we infer diagram

\[\begin{array}{c}
A \leftarrow R \quad A \times B \quad B \\
\downarrow \langle R, S \rangle \quad \uparrow \pi_1 \quad \pi_2 \quad \downarrow S \\
C \end{array}\]

and "al-gabr" rule (Galois connection)

\[\pi_1 \cdot X \subseteq R \land \pi_2 \cdot X \subseteq S \iff X \subseteq \langle R, S \rangle\] \hspace{1cm} (23)
Coproducts for “if-then-else’ing”

Define dual (“either”) combinator as

\[ [R, S] = (R \cdot i_1^\circ) \cup (S \cdot i_2^\circ) \]

\[ A \xrightarrow{i_1} A + B \xleftarrow{i_2} B \]

\[ R \xrightarrow{[R, S]} C \xleftarrow{S} \]

From this and the lub rule (2) we infer another “al-gabr” rule (Galois connection)

\[ [R, S] \subseteq X \iff R \subseteq X \cdot i_1 \land S \subseteq X \cdot i_2 \]  \hspace{1cm} (24)

In fact, the stronger universal property holds:

\[ [R, S] = X \iff R = X \cdot i_1 \land S = X \cdot i_2 \]  \hspace{1cm} (25)
Multiplying and adding relations

From “fork” and “either” derive

\[ R \times S \triangleq \langle R \cdot \pi_1, S \cdot \pi_2 \rangle \]  \hspace{1cm} (26)

\[ R + S = [i_1 \cdot R, i_2 \cdot S] \]  \hspace{1cm} (27)

whose pointwise meaning is, as given earlier:

\[
\begin{array}{c|c}
\phi & PF \phi \\
\hline
a R c \land b S c & (a, b) \langle R, S \rangle c \\
b R a \land d S c & (b, d)(R \times S)(a, c)
\end{array}
\]

Absorption properties:

\[ \langle R \cdot X, S \cdot Y \rangle = (R \times S) \cdot \langle X, Y \rangle \]  \hspace{1cm} (28)

\[ [R, S] \cdot (X + Y) = [R \cdot X, S \cdot Y] \]  \hspace{1cm} (29)
From both (23) and (25) we easily infer the **exchange law**, 

\[
[\langle R, S \rangle, \langle T, V \rangle] = \langle [R, T], [S, V] \rangle
\]  

(30)

holding for all relations as in diagram

![Diagram](image-url)
Inductive relations

Example — inductive definition of $\geq$ over the natural numbers: for all $y, x \in \mathbb{N}_0$, define $\mathbb{N}_0 \leftarrow \geq \mathbb{N}_0$ as the least relation satisfying

$$
\begin{align*}
  y &\geq 0 \\
  y \geq x &\Rightarrow (y + 1) \geq (x + 1)
\end{align*}
$$

Thanks to (13), these clauses PF-transform to

$$
\begin{align*}
  \top &\subseteq \geq \cdot 0 \\
  \geq &\subseteq \text{suc}^\circ \cdot \geq \cdot \text{suc}
\end{align*}
$$

where 0 denotes the everywhere 0 constant function.
Least prefix points

We reason:

\[
\begin{cases}
\top \subseteq \geq \cdot 0 \\
\geq \subseteq \text{suc} \circ \geq \cdot \text{suc}
\end{cases}
\]

\[\Leftrightarrow\] \{ al-gabr (18) ; coproducts \}

\[[\top, \text{suc} \cdot \geq] \subseteq \geq \cdot [0, \text{suc}]\]

\[\Leftrightarrow\] \{ “al-gabr” (19) \}

\[[\top, \text{suc} \cdot \geq] \cdot [0, \text{suc}]^\circ \subseteq \geq\]

\[\Leftrightarrow\] \{ absorption property (29) \}

\[[\top, \text{suc}] \cdot (\text{id} + \geq) \cdot [0, \text{suc}]^\circ \subseteq \geq\]

In summary: \(\geq\) is the least **prefix** point of monotonic function

\[f_X \triangleq [\top, \text{suc}] \cdot (\text{id} + X) \cdot [0, \text{suc}]^\circ\]
Recognizing $[0, suc] = in$ as initial $(1 + \_\_)$-algebra with carrier $N_0$ (Peano isomorphism) we draw

\[
\begin{array}{c}
\begin{array}{c}
\quad \xymatrix{\mathbb{N}_0 \ar[r]^{in^\circ} \ar[d]_{\geq} & 1 + \mathbb{N}_0 \ar[d]_{id + \geq} \\
\quad \mathbb{N}_0 \ar[r]_{[\top, suc]} & 1 + \mathbb{N}_0}
\end{array}
\end{array}
\]

\[\quad [\top, suc] \cdot (id + \geq) \subseteq \geq \cdot in\]

Since $[\top, suc]$ uniquely determines $\geq$ (least prefix points are unique, etc), we resort to the popular notation

\[\geq = ([\top, suc]) \quad (31)\]

to express this fact. (See summary of general theory in the sequel.)
Introducing the $\kappa\alpha\tau\alpha$ combinator

In general, for $F$ a polynomial functor (relator) and $\mu F \xleftarrow{in} F(\mu F)$ initial:

$$
\begin{array}{c}
\mu F \\
\downarrow (\langle R \rangle) \\
A
\end{array} \cong \begin{array}{c}
\mu F \\
\downarrow \circ \Rightarrow \\
F(\mu F) \\
\downarrow in \\
F(\langle R \rangle)
\end{array} \cong \begin{array}{c}
F \circ A \\
R \\
F(\mu F)
\end{array}
$$

there is a unique solution to equation $X = R \cdot F X \cdot in^\circ$
characterized by universal property:

$$
X = (\langle R \rangle) \iff X = R \cdot F X \cdot in^\circ \quad (32)
$$

(Read $\langle R \rangle$ as “$\kappa\alpha\tau\alpha \ R$”.)
Introducing the $\kappa\alpha\tau\alpha$ combinator

Therefore (cf. Knaster-Tarski) $(|R|)$ is both the least prefix point

$$ (|R|) \subseteq X \iff R \cdot F X \cdot in^o \subseteq X \quad (33) $$

and the greatest postfix point:

$$ X \subseteq (|R|) \iff X \subseteq R \cdot F X \cdot in^o \quad (34) $$

Corollaries include reflexion,

$$ (|in|) = id \quad (35) $$

$\kappa\alpha\tau\alpha$-fusion,

$$ S \cdot (|R|) \subseteq (|X|) \iff S \cdot R \subseteq X \cdot F S \quad (36) $$

monotonicity,

$$ (|R|) \subseteq (|X|) \iff R \subseteq X \quad (37) $$

e tc.
Why $\kappa\alpha\tau\alpha$s?

- What’s the advantage of writing $\geq = ([\top, \text{suc}])$? Is it just a matter of style or economy of notation?
- No: think of proving that $\geq$ is transitive:

$$\langle \forall x, y, z :: x \geq y \land y \geq z \Rightarrow x \geq z \rangle$$

Instead of providing an explicit (inductive) proof, we go pointfree and write:

$$\geq \cdot \geq \subseteq \geq$$

which instantiates $\kappa\alpha\tau\alpha$-fusion (36), for $R, X := [\top, \text{suc}]$. 
Thank you, κατα-fusion

We reason:

\[ \geq \cdot \geq \subseteq \geq \]
\[ \Leftrightarrow \{ \text{definition (31)} \} \]
\[ \geq \cdot ([\top, \text{suc}]) \subseteq ([\top, \text{suc}]) \]
\[ \Leftarrow \{ \kappa\alpha\tau\alpha\text{-fusion (36)} \} \]
\[ \geq \cdot [\top, \text{suc}] \subseteq [\top, \text{suc}] \cdot (id + \geq) \]
\[ \Leftrightarrow \{ \text{coproducts (29, etc)} \} \]
\[ \geq \cdot \top \subseteq \top \land \geq \cdot \text{suc} \subseteq \text{suc} \cdot \geq \]
\[ \Leftrightarrow \{ \text{everything is at most } \top \} \]
\[ \geq \cdot \text{suc} \subseteq \text{suc} \cdot \geq \]
\[ \Leftarrow \{ \geq \cdot \text{suc} = \text{suc} \cdot \geq \text{ (32)} \} \]

True
By the way

Direct use of universal property (32) would lead to

\[ \geq = ([\top, \text{suc}]) \]

\[ \Leftrightarrow \{ (32) \} \]

\[ \geq \cdot [0, \text{suc}] = [\top, \text{suc}] \cdot (\text{id} + \geq) \]

\[ \Leftrightarrow \{ \text{expand, go pointwise, simplify} \} \]

\[ \begin{cases} y \geq 0 \\ y \geq (x + 1) \Leftrightarrow y > 0 \land (y - 1) \geq x \end{cases} \]

So, the above and our starting (co-inductively flavored) definition

\[ y \geq 0 \]

\[ y \geq x \Rightarrow (y + 1) \geq (x + 1) \]

are equivalent (by construction).
What about $\kappa\alpha\tau\alpha$s which are forks? We reason:

\[
(|\langle R, S \rangle|) \subseteq \langle X, Y \rangle
\]

$\iff$ \{ least prefix point (33) \}

\[
\langle R, S \rangle \cdot F\langle X, Y \rangle \cdot in^\circ \subseteq \langle X, Y \rangle
\]

$\iff$ \{ “al-gabr” rule (23) \}

\[
\begin{cases}
\pi_1 \cdot \langle R, S \rangle \cdot F\langle X, Y \rangle \cdot in^\circ \subseteq X \\
\pi_2 \cdot \langle R, S \rangle \cdot F\langle X, Y \rangle \cdot in^\circ \subseteq Y
\end{cases}
\]

$\iff$ \{ $X := \langle R, S \rangle$ in (23); monotonicity \}

\[
\begin{cases}
R \cdot F\langle X, Y \rangle \cdot in^\circ \subseteq X \\
S \cdot F\langle X, Y \rangle \cdot in^\circ \subseteq Y
\end{cases}
\]
Handling mutually recursive relations

• Rule

\[
(|\langle R, S \rangle|) \subseteq \langle X, Y \rangle \iff \left\{ \begin{array}{l}
R \cdot F\langle X, Y \rangle \cdot \text{in}^\circ \subseteq X \\
S \cdot F\langle X, Y \rangle \cdot \text{in}^\circ \subseteq Y
\end{array} \right.
\] (38)

tells us how to combine two mutually recursive relations into a single one.

• In the case of functions (20) we get equivalence

\[
\left\{ \begin{array}{l}
x \cdot \text{in} = r \cdot F\langle x, y \rangle \\
y \cdot \text{in} = s \cdot F\langle x, y \rangle
\end{array} \right. \iff \langle x, y \rangle = (|\langle r, s \rangle|)
\] (39)

known as “Fokkinga’s mutual recursion theorem” [2].

• Both (38,39) generalize to \( n > 2 \) mutually recursive relations (functions) and can be used for program optimization.
Handling mutually recursive relations

• Notice that $in^\circ$ plays no special role in the calculation of (38); so it can be replaced by arbitrary (suitably typed) $D$.

• This generalizes rule (38) to divide-and-conquer algorithms described by recursive relations which are fixpoints of $f X \triangle R \cdot (F X) \cdot D$, where $R$ describes the conquer step and $D$ the divide step.
  (Btw, these are known as hylomorphisms [2].)

• For economy of presentation, the example which follows is a direct application of the special case where all relations are functions (39).
Example — exponential function

Taylor series:

\[ e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \]  \hspace{1cm} (40)

Computing finite approximation \((n\) terms\):

\[ e^x \cdot n = \sum_{i=0}^{n} \frac{x^i}{i!} \]  \hspace{1cm} (41)

takes quadratic time. Wishing to calculate a linear-time algorithm from this mathematical definition, we first head for an inductive definition:

\[ e^x \cdot 0 = 1 \]

\[ e^x \cdot (n+1) = \frac{x^{n+1}}{(n+1)!} + \sum_{i=0}^{n} \frac{x^i}{i!} \]
Example — exponential function

We thus get primitive recursive definition

\[ e^x 0 = 1 \]
\[ e^x (n + 1) = h_x n + e^x n \]

where \( h_x n \) unfolds to \( \frac{x^{n+1}}{(n+1)!} = \frac{x}{n+1} \frac{x^n}{n!} \). Therefore:

\[ h_x 0 = x \]
\[ h_x(n + 1) = \frac{x}{n+2} (h_x n) \]

Introducing \( s2 \ n = n + 2 \), we derive:

\[ s2 \ 0 = 2 \]
\[ s2(n + 1) = 1 + s2 \ n \]
Example — exponential function

We can thus put $e^x$, $s2$ and $h_x$ together in a system of three mutually recursive functions $e^x$, $s2_x$ and $h_x$ over the naturals, which PF-transform to

$$
e^x \cdot in = \left[ 1 , (+) \cdot \langle \pi_1, \pi_2 \cdot \pi_2 \rangle \right] \cdot F\langle e^x, \langle s2, h \rangle \rangle$$

$$s2_x \cdot in = \left[ 2 , suc \cdot \pi_1 \cdot \pi_2 \right] \cdot F\langle e^x, \langle s2, h \rangle \rangle$$

$$h_x \cdot in = \left[ x , (\ast) \cdot ((x/) \times id) \cdot \pi_2 \right] \cdot F\langle e^x, \langle s2, h \rangle \rangle$$

respectively, for

$$in = [0 , suc]$$

$$FX = id + X$$
Example — exponential function

From this system we obtain, thanks to the mutual recursion law (39)

\[
\text{aux}_x \triangleq \langle e^x, \langle s_2^x, h_x \rangle \rangle \\
= \{ (39) \} \\
\text{(|}\langle r, \langle s, t \rangle \rangle\text{|)}
\]

for

\[
\begin{align*}
  r &= [1, (+) \cdot \langle \pi_1, \pi_2 \cdot \pi_2 \rangle] \\
  s &= [2, \text{suc} \cdot \pi_1 \cdot \pi_2] \\
  t &= \underbrace{[x, (\ast) \cdot ((x/) \times \text{id}) \cdot \pi_2]}_{u}
\end{align*}
\]
Example — exponential function

Next we apply the exchange law (30) to $\langle r, \langle s, t \rangle \rangle$ (twice):

$$\langle r, \langle s, t \rangle \rangle = \left[ \langle 1, \langle 2, x \rangle \rangle, \langle (+) \cdot \langle \pi_1, \pi_2 \cdot \pi_2 \rangle, \langle \text{suc} \cdot \pi_1 \cdot \pi_2, u \rangle \rangle \right]$$

Thanks to universal properties (32) and (23) \(^2\) we obtain

$$\begin{align*}
\text{aux}_x \cdot 0 &= \langle 1, \langle 2, x \rangle \rangle \\
\text{aux}_x \cdot \text{suc} &= \langle (+) \cdot \langle \pi_1, \pi_2 \cdot \pi_2 \rangle, \langle \text{suc} \cdot \pi_1 \cdot \pi_2, u \rangle \rangle \cdot \text{aux}_x \\
e^x &= \pi_1 \cdot \text{aux}_x
\end{align*}$$

that is, we have calculated linear implementation

\(^2\)For functions.
Example — exponential function

\[
\text{exp } x \ n = \text{let } (e, b, c) = \text{aux } x \ n \\
\text{in } e \text{ where} \\
\text{aux } x \ 0 = (1, 2, x) \\
\text{aux } x \ (i+1) = \text{let } (e, s, h) = \text{aux } x \ i \\
\text{in } (e+h, s+1, (x/s)*h)
\]

which can be identified as the denotational semantics of a while loop, encoded below in the C programming language:

```c
float exp(float x, int n)
{
    float e=1; int s=2; float h=x; int i;
    for (i=0; i<n+1; i++) {e=e+h; h=(x/s)*h; s++;}
    return e;
}
```
Summing up

- Algebra of Programming (**AoP**): calculating ("correct by construction") programs from specifications
- Pointfree notation: Tarski’s *set theory without variables* [7]
- Kleene algebra of (typed) relations: arrows (not points) provide further structure while ensuring type checking
- *Ut faciant opus signa*:
  
  [Symbolisms] “*have invariably been introduced to make things easy. [...] by the aid of symbolism, we can make transitions in reasoning almost mechanically by the eye, which otherwise would call into play the higher faculties of the brain. [...] Civilisation advances by extending the number of important operations which can be performed without thinking about them.*”

  *(Alfred Whitehead, 1911)*
Despite textbooks such as [2], *Algebra of Programming* is still land of nobody. Why?

- Software theorists: too busy with their pre-scientific theories (if any)
- Algebraists: not sufficiently aware of program construction as a mathematical discipline
- Both: the required background (categories, allegories, etc) is most often found missing from undergrad curricula.
Selected topic of interest

- Pointfree notations are emerging elsewhere in the context of eg. digital signal processing (SPIRAL project, CMU [6]) which abstract linear signal transforms in terms of (index-free) matrix operators.
- Kleene algebras scale up to the corresponding matrix Kleene algebras [1]
- Parallel with relational algebra is obvious.
- Following a similar path, we want to investigate the “matrices as arrows” approach purported by categories of matrices (PhD project).
- We believe a better (typed!) calculus of (Kleene) matrix algebras will emerge which will improve reasoning about linear transforms in DSP, divide-and-conquer algorithms, etc.
R.C. Backhouse.

*Mathematics of Program Construction.*
Univ. of Nottingham, 2004.

R. Bird and O. de Moor.

*Algebra of Programming.*
C.A.R. Hoare, series editor.

J. Desharnais and G. Struth.

Domain axioms for a family of near-semirings.

P.J. Freyd and A. Ščedrov.

*Categories, Allegories*, volume 39 of *Mathematical Library.*

Dexter Kozen.

Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel.
Formal datapath representation and manipulation for implementing DSP transforms.

A. Tarski and S. Givant.
A Formalization of Set Theory without Variables.
AMS Colloquium Publications, volume 41, Providence, Rhode Island.