A Look at Program Galculation

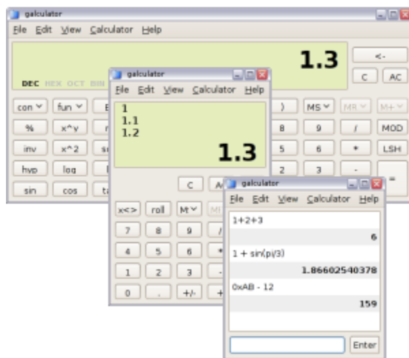
J.N. Oliveira
(joint work with Paulo Silva)

Dept. Informática,
Universidade do Minho
Braga, Portugal

IFIP WG2.1 meeting #65
January 2010
Braga, Portugal

The 'G'alculator Project

The 'G'alculator is **not** the GTK 2 based calculator which Google offers you in the first place...



Context

- **'G'alculator Project** — design of a **proof assistant** solely based on **Galois connections** (GCs), eg.

$$\langle \forall\ x, y\ \ ::\ \ f\ x \leq y\ \ \Leftrightarrow\ \ x \leq g\ y \rangle$$

  and indirect equality (IE)

$$n = m\ \Leftrightarrow\ \langle \forall\ x\ \ ::\ \ x \leq n \Leftrightarrow x \leq m \rangle$$

  See PhD thesis by Paulo Silva on his implementation of a prototype of the 'G'alculator in Haskell and our PPDP'08 paper (Silva and Oliveira, 2008).

- **MathIS project** — are GCs a good recipe for teaching MPC culture at (late) middle-school level?

Why Galois connections?

- GCs in "natural language":

  *Software requirements normally full of superlatives such as ... **best** solution .... **smallest** such number ... **longest** such list ...*

- **Superlatives** always mean **suprema** or **infima** in some ordered domain.
- Very often, such limits can be identified with **adjoints** of a GC.
- Conclusion: "CGcs are **specifications**"

How about:

- calculating **properties** from them?
- calculating **algorithms** from them?

CGcs are specifications

Example — requirements for **whole division** $x \div y$:

- write a program which computes number z which, multiplied by y, approximates x.
- check your program with the following test data:
  $x, y, z = 7, 2, 1$
  $x, y, z = 7, 2, 2$
- ups! Forgot to tell that one wants the largest such number (sorry!):
  $x, y, z = 7, 2, 3$

Deriving the algorithm: ok, but...

... *where is the formal specification of* $x \div y$*?*

CGcs are specifications

Example — requirements for **whole division** $x \div y$:

- write a program which computes number z which, multiplied by y, approximates x.
- check your program with the following test data:
  $x, y, z = 7, 2, 1$
  $x, y, z = 7, 2, 2$
- ups! Forgot to tell that one wants the largest such number (sorry!):
  $x, y, z = 7, 2, 3$

Deriving the algorithm: ok, but...

... *where is the formal specification of* $x \div y$*?*

CGcs are specifications

First version (literal):

$$x \div y = \langle \bigvee\ z\ ::\ z \times y \leq x \rangle \tag{1}$$

Second version (involved):

$$z = x \div y\ \Leftrightarrow\ \langle \exists\ r\ :\ 0 \leq r < y :\ x = z \times y + r \rangle \tag{2}$$

Third version (clever!):

$$z \times y \leq x\ \Leftrightarrow\ z \leq x \div y \qquad (y > 0) \tag{3}$$

— a Galois connection.

Why (3) is better than (1,2)

It captures the requirements:

- It is a solution: $x \div y$ multiplied by y approximates x

$$(x \div y) \times y \leq x$$

(ie., upper cancellation of (3) — let $z := x \div y$ and simplify)

- It is the best solution (provides **largest** such number):

$$z \times y \leq x \;\Rightarrow\; z \leq x \div y \qquad (y > 0)$$

(the $\Rightarrow$ part of $\Leftrightarrow$).

Advantages:

Highly calculational!

Anticipating properties of algorithm

Easy ones, of the kind *instantiate & simplify* — from

$$z \times y \leq x \;\Leftrightarrow\; z \leq x \div y \qquad (y > 0)$$

infer:

$$0 \leq x \div y \qquad (z := 0)$$

$$y \leq x \;\Leftrightarrow\; 1 \leq x \div y \qquad (z := 1)$$

However,

$$x \div 1 = x$$

not so immediate. How do we go about this?

Indirect equality (IE) principle

Well-known in set theory to define set equality:

$$A = B \quad \Leftrightarrow \quad \langle \forall\, x \;::\; x \in A \Leftrightarrow x \in B \rangle$$

Another form is:

$$A = B \quad \Leftrightarrow \quad \langle \forall\, X \;::\; X \subseteq A \Leftrightarrow X \subseteq B \rangle$$

In general: any **partial order** can be used to establish equality by indirection. In case of numbers, for instance:

$$n = m \quad \Leftrightarrow \quad \langle \forall\, x \;::\; x \leq n \Leftrightarrow x \leq m \rangle \tag{4}$$

GC + IE = 'G'alculator

Calculation of $x \div 1 = x$:

$$
\begin{array}{cl}
 & z \times 1 \leq x \;\Leftrightarrow\; z \leq x \div 1 \\
\Leftrightarrow & \quad \{ \;\; 1 \text{ unit of } \times \;\; \} \\
 & z \leq x \;\Leftrightarrow\; z \leq x \div 1 \\
:: & \quad \{ \;\; \text{indirect equality} \;\; \} \\
 & x \div 1 = x
\end{array}
$$

Easy — any child will do it! (Recall MathIS project context.)
Another example of proof in the same vein:

$$
(n \div m) \div d \;=\; n \div (d \times m) \tag{5}
$$

'G'alculator proof

(For $m, d > 0$):

$$
\begin{array}{cl}
 & x \leq (n \div m) \div d \\
\Leftrightarrow & \{ \text{ GC (3) } \} \\
 & x \times d \leq n \div m \\
\Leftrightarrow & \{ \text{ GC (3) } \} \\
 & (x \times d) \times m \leq n \\
\Leftrightarrow & \{ \times \text{ is associative } \} \\
 & x \times (d \times m) \leq n \\
\Leftrightarrow & \{ \text{ GC (3) } \} \\
 & x \leq n \div (d \times m)
\end{array}
$$

Thus,

$$(n \div m) \div d \quad \Leftrightarrow \quad n \div (d \times m)$$

Algebra $\preceq$ "al-jabr"

The **tradition**: numeric GCs of school algebra, such as eg.

$$x + \overbrace{z \leq y \quad \Leftrightarrow \quad x \leq y -}^{\text{"al-jabr"}} z$$

$$x \times \overbrace{z \leq y \quad \Leftrightarrow \quad x \leq y \times}^{\text{"al-hatt"}} z^{-1} \qquad (z > 0)$$

are known since 9c AD, cf. Al-Khwarizmi's *Compendious Book on Calculation by Completion and Balancing*.

Galculation of $x \div y$ algorithm

$$
\begin{array}{ll}
 & z \leq x \div y \\
\Leftrightarrow & \quad \{ \text{ GC (3) assuming } y > 0 \ \} \\
 & z \times y \leq x \\
\Leftrightarrow & \quad \{ \text{ cancellation } \} \\
 & z \times y - y \leq x - y \\
\Leftrightarrow & \quad \{ \text{ distribution } \} \\
 & (z-1) \times y \leq x - y \\
\Leftrightarrow & \quad \{ \text{ GC (3) } \} \\
 & z - 1 \leq (x - y) \div y \\
\Leftrightarrow & \quad \{ \text{ GC } (-1) \vdash (+1) \ \} \\
 & z \leq (x - y) \div y + 1
\end{array}
$$

Galculation of $x \div y$ algorithm

Thus

$$x \div y = (x - y) \div y + 1 \tag{6}$$

— the inductive step of the algorithm.

The base step will be galculated in a similar way, leading to the well-known algorithm:

```
x -:- y |  x <  y     = 0
        |  x >= y     = 1 + (x - y) -:- y
```

Note the intuition shift: $x \div y$ means (after all) counting the number of times y "fits" in x.

PF-transformed GCs

$$z \underbrace{(\times y)}_{f} \leq x \;\Leftrightarrow\; z \leq x \underbrace{(\div y)}_{g}$$

Equivalence becomes relational equality, thus GC

$$f\ x \leq y \quad \Leftrightarrow \quad x \sqsubseteq g\ y$$

becomes

$$f^{\circ} \cdot \leq \quad = \quad \sqsubseteq \cdot g \qquad (7)$$

Abbreviated notation

$$f \vdash g \qquad (8)$$

for (7) wherever the orderings are implicit in the context.

PF-transformed IE

Indirect equality principle

$$f = g \quad \Leftrightarrow \quad \langle \forall\, y, x \;::\; y \leq (f\ x) \Leftrightarrow y \leq (g\ x) \rangle$$

becomes, once PF-transformed

$$f = g \quad \Leftrightarrow \quad \leq \cdot f = \leq \cdot g \tag{9}$$

Thus the only inference rule needed is substitution of equals for equals.

The 'G'alculator (proof-of-concept prototype implemented in Haskell) is a proof assistant which works at this level.

Program Galculation

The approach is more general than first thought because

- GCs abound in mathematics and modeling
- There is an **algebra of GCs** enabling one to build new connections from old (composition, converse, forks, relators, etc)
- Using relational products, adjoints can have an arbitrary number of parameters
- See Paulo's thesis for a comprehensive account.

GC Algebra

Assume $f \vdash g$ and $f' \vdash g'$ hold in:

Identity

$$id \vdash id$$

Composition

$$f \cdot f' \vdash g' \cdot g$$

Converse (symmetry)

$$f \vdash g \;\Leftrightarrow\; g \vdash f$$

Functors (preorders)

$$\mathsf{F}f \vdash \mathsf{F}g$$

Splitting (lattices)

$$\langle f, f' \rangle \vdash \sqcap \cdot (g \times g')$$

In particular, for $f, f' := id$, $g, g' := id$:

$$\triangle \vdash \sqcap \tag{10}$$

for $\triangle x = (x, x)$.

Example of GC derivation — splits of adjoints

Given $f \vdash g$ and $f' \vdash g'$, calculate GC whose lower adjoint is $\langle f, f' \rangle$:

$$
\begin{array}{ll}
& \langle f, f' \rangle^{\circ} \cdot (\leq \times \leq') = \sqsubseteq \cdot ? \\
\Leftrightarrow & \qquad \{ \text{ add variables } \} \\
& f\ x \leq y \wedge f'\ x \leq' y' \Leftrightarrow x \sqsubseteq ?(y, y') \\
\Leftrightarrow & \qquad \{ \text{ shunt on } f \vdash g,\ f' \vdash g' \ \} \\
& x \sqsubseteq g\ y \wedge x \sqsubseteq g'\ y' \Leftrightarrow x \sqsubseteq ?(y, y') \\
\Leftrightarrow & \qquad \{ \text{ lub } \sqcap \text{ exists if lattice } \} \\
& x \sqsubseteq (g\ y \sqcap g'\ y') \Leftrightarrow x \sqsubseteq ?(y, y') \\
:: & \qquad \{ \text{ IE } \} \\
& ?(y, y') = (g\ y \sqcap g'\ y')
\end{array}
$$

GCs over lists

AoP reasoning about list processing algorithms (eg. fusion, tupling etc) assumes that the functions of interest have been *understood* as **folds** (catamorphisms), **unfolds** (anamorphisms) or hylomorphisms.

Type information can be of help in guiding one in such a constructive approach to programming, but it doesn't help in many contexts.

How does one justify that a particular ana, or hylo, **meets its specification**?

Below we will see that our life is easier if GCs are used as specs, in particular if based on well-known orderings such as **prefix**.

GCs over lists

In fact, GCs combine with inductive (relational) definitions of such orderings in a nice way, enabling the galculation of algorithmic solutions to the adjoints involved.

Example: longest common prefix (lcp) of two lists specified by GC

$$ys \preceq xs \wedge ys \preceq xs' \quad \Leftrightarrow \quad ys \preceq lcp(xs, xs') \tag{11}$$

— that is, lcp is the **glb** of the prefix ordering $\preceq$, cf.

$$\triangle \;\; \vdash \;\; lcp \tag{12}$$

using compact notation (8) — recall (10).

Dissecting the prefix ordering

Galculation of lcp calls for knowledge about the behaviour of the prefix ($\preceq$) ordering.

Let us take (Bird and de Moor, 1997)'s definition of *prefix* as a relational fold on lists,

$$\preceq \; = \; (\![[nil, nil \cup cons]]\!) \tag{13}$$

where $nil \;_ = [\,]$ and $cons(h, t) = h : t$ — cf. diagram

$$\begin{array}{ccc} A^\star & \overset{[nil,cons]^\circ}{\underset{[nil,cons]}{\cong}} & 1 + A \times A^\star \\ \preceq \downarrow & & \downarrow id+id\times\preceq \\ A^\star & \xleftarrow[{[nil,nil\cup cons]}]{} & 1 + A \times A^\star \end{array}$$

Dissecting the prefix ordering

PF (fixpoint) equation

$$\preceq \cdot [nil, cons] = [nil, nil \cup cons] \cdot (id + id \times \preceq) \tag{14}$$

means the following properties:

$$[\,] \preceq [\,] \tag{15}$$

$$[\,] \preceq (h : t) \tag{16}$$

$$s \preceq (h : t) \;\Leftrightarrow\; s = [\,] \;\vee\; \langle \exists\, s' \;:\; s = (h : s') :\; s' \preceq t \rangle \tag{17}$$

From (17) we infer:

$$(h : k) \preceq (h : t) \quad \Leftrightarrow \quad k \preceq t \tag{18}$$

(substitution $s := (h : k)$ + simplification)

Calculating *lcp*

Base case $xs := [\,]$ ($xs' := [\,]$ the same):

$$
\begin{array}{cl}
 & ys \preceq lcp([\,], xs') \\
\Leftrightarrow & \qquad \{ \ \text{GC (11)} \ \} \\
 & ys \preceq [\,] \wedge ys \preceq xs' \\
\Leftrightarrow & \qquad \{ \ \text{prefix (16), thus } ys = [\,]\text{; (15)} \ \} \\
 & ys \preceq [\,] \\
:: & \qquad \{ \ \text{IE} \ \} \\
 & lcp([\,], xs') \ = \ [\,]
\end{array}
$$

Calculating *lcp*

Inductive case:

$$
\begin{array}{ll}
 & ys \preceq lcp(a : as, b : bs) \\
\Leftrightarrow & \quad \{ \text{ GC (11) } \} \\
 & ys \preceq (a : as) \wedge ys \preceq (b : bs) \\
\Leftrightarrow & \quad \{ \text{ prefix (17) twice; } \wedge\text{-}\vee \text{ distribution } \} \\
 & ys = [\,] \vee ys = (a : ys') \wedge ys' \preceq as \wedge ys = (b : ys'') \wedge ys'' \preceq bs \\
\Leftrightarrow & \quad \{ \text{ list reflection ; one-point } (ys'' := yx' \ \} \\
 & ys = [\,] \vee ys = (a : ys') \wedge a = b \wedge ys' \preceq as \wedge ys' \preceq bs \\
\Leftrightarrow & \quad \{ \text{ GC (11) } \}
\end{array}
$$

Calculating *lcp*

$$
\begin{array}{ll}
& ys = [\,] \vee ys = (a : ys') \wedge a = b \wedge ys' \preceq lcp(as, bs) \\
\Leftrightarrow & \quad \{ \ (18) \ ; \text{substitution} \ \} \\
& ys = [\,] \vee ys = (a : ys') \wedge a = b \wedge ys \preceq (a : lcp(as, bs))
\end{array}
$$

Two cases: for $a \neq b$,

$$
\begin{array}{ll}
& ys = [\,] \vee \textsc{False} \\
\Leftrightarrow & \quad \{ \ ys \preceq [\,] \Leftrightarrow ys = [\,] \ \} \\
& ys \preceq [\,]
\end{array}
$$

Thus, by IE, $lcp(a : as, b : bs) = [\,]$ for this case.

Calculating *lcp*

Otherwise ($a = b$):

$$
\begin{array}{ll}
& ys = [\,] \vee ys = (a : ys') \wedge ys \preceq a : lcp(as, bs) \\
\Leftrightarrow & \{ \ [\,] \preceq a : lcp(as, bs) \ \} \\
& ys \preceq a : lcp(as, bs)
\end{array}
$$

Thus, $lcp(a : as, b : bs) = a : lcp(as, bs)$ in this case. Altogether, we've calculated an algorithm for *lcp* — in Haskell, below:

```
lcp :: (Eq a) => [a] -> [a] -> [a]
lcp [] _ = []
lcp _ [] = []
lcp (a:l) (b:m) | a == b = a:(lcp l m)
                | a /= b = []
```

Another example — take

The specification of take is GC (identified by Roland Backhouse)

$$(\mathsf{len} \times id) \cdot \triangle \quad \vdash \quad take \tag{19}$$

which has a slightly more elaborate lower adjoint when compared to that of lcp,

$$(\mathsf{len}\, ys, ys)\, (\leq \times \preceq)\, (n, xs) \quad \Leftrightarrow \quad ys \preceq take(n, xs) \tag{20}$$

that is,

$$\mathsf{len}\, ys \leq n \ \wedge \ ys \preceq xs \quad \Leftrightarrow \quad ys \preceq take(n, xs) \tag{21}$$

Another example — take

Before implementing this function, let us exploit its *property space* — standard procedure thanks to spec being GC

$$\mathsf{len}\, ys \leq n \;\wedge\; ys \preceq xs \;\Leftrightarrow\; ys \preceq take(n, xs)$$

Upper cancellation ($ys := take(n, xs)$):

$$\mathsf{len}\,(take(n, xs)) \leq n \quad\wedge\quad take(n, xs) \preceq xs \tag{22}$$

Lower cancellation ($ys := xs$, then simplify; upper cancellation (22)):

$$xs = take(n, xs) \quad\Leftrightarrow\quad \mathsf{len}\, xs \leq n \tag{23}$$

Another example — take

Composition:

$$take(n, take(m, xs)) \quad = \quad take(min(n, m), xs) \tag{24}$$

cf.

$$
\begin{array}{ll}
 & ys \preceq take(n, take(m, xs)) \\
\Leftrightarrow & \qquad \{ \ \text{GC (20)} \ \} \\
 & \text{len}\ ys \leq n \wedge ys \preceq take(m, xs) \\
\Leftrightarrow & \qquad \{ \ \text{GC (20) again} \ \} \\
 & \text{len}\ ys \leq n \wedge \text{len}\ ys \leq m \wedge ys \preceq xs \\
\Leftrightarrow & \qquad \{ \ \text{GC of } min \text{ of two numbers (25)} \ \} \\
 & \text{len}\ ys \leq min(n, m) \wedge ys \preceq xs \\
\Leftrightarrow & \qquad \{ \ \text{GC (20) again, now folding} \ \} \\
 & ys \preceq take(min(n, m), xs)
\end{array}
$$

By the way — *min*

Calculation of *min* is trivial but illustrates strategy useful wherever the ordering underlying calculation is to be tested in the algorithm:

$$x \leq n \wedge x \leq m \;\Leftrightarrow\; x \leq min(n, m) \tag{25}$$

Strategy consists in finding conditions for rendering one of the conjuncts of the lower adjoint redundant, thus enabling IE over the other: since

$$x \leq n \wedge n < m \;\Rightarrow\; x \leq m$$

in case $n < m$, GC shrinks to

$$x \leq n \;\Leftrightarrow\; x \leq min(n, m) \tag{26}$$

Therefore:

```
min(n,m) | n < m  = n
         | n >= m = m
```

Another example — take

Finally galculating the algorithm. Base cases $take(0, xs)$ and $take(n, [\,])$ are immediate. Case $take(n+1, h : xs)$ follows:

$$
\begin{array}{ll}
 & ys \preceq take(n+1, h : xs) \\
\Leftrightarrow & \{ \text{ GC (20) ; len } (h : t) = 1 + \text{len } t \text{ ; prefix (17) } \} \\
 & \text{len } ys \leq n+1 \wedge (ys = [\,] \ \vee \ ys = (h : ys') \ \wedge \ ys' \preceq xs) \\
\Leftrightarrow & \{ \text{ distribution ; len } (h : t) = 1 + \text{len } t \text{ ; simplification } \} \\
 & ys = [\,] \ \vee \ \text{len } ys' \leq n \wedge ys = (h : ys') \ \wedge \ ys' \preceq xs) \\
\Leftrightarrow & \{ \text{ GC (20) } \} \\
 & ys = [\,] \ \vee \ ys' \preceq take(n, xs) \wedge ys = (h : ys') \\
\Leftrightarrow & \{ \text{ prefix (18) ; substitution } \} \\
 & ys = [\,] \ \vee \ ys \preceq h : take(n, xs) \wedge ys = (h : ys') \\
\Leftrightarrow & \{ \text{ list reflection ; further simplification } \} \\
 & ys \preceq h : take(n, xs))
\end{array}
$$

Another example — take

Putting everything together:

```
take :: (Integral t) => t -> [a] -> [a]
take 0 _  = []
take _ [] = []
take (n+1) (h:xs) = h:(take n xs)
```

Fold / Unfold

From the given galculations, note how base GC is used in basically two ways, one in the beginning **unfolding** the target adjoint and another, at some stage, **folding** back again.

Compared to classical **fold/unfold** transformation (Burstall and Darlington, 1977), such steps are always **safe** since we stay with equivalences all the time.

Algorithmic structure of the calculated (functional) program follows the inductive structure implicit in the definition of the ordering.

Conclusions & current work

- GC+IE reasoning is very systematic and easy to teach
- For simple (eg. numeric) GCs this is teachable to kids at school because they are familiar with "shunting" in linear algebra
- Things become less easy when the underlying ordered structures become less tractable
- Currently experimenting with mapping lists to relations "a la Alloy" and using GC+IE at relational level (eg. functions such as `nub`)
- For a comprehensive account of the role of GCs in computing see eg. (Backhouse, 2004)

Postscriptum

After Zhenjiang's and Jeremy's talks, and answering to Michel's "exercise" proposal, here is my (tentative!) GC approach to bidirectional programming:

$$(get \times id) \cdot \triangle \quad \vdash \quad put \tag{27}$$

Deja vu? Yes, just make $get, put := len, take$ and you get the GC specification of $take$ (19).

Details: for $\leq_v$ and $\leq_s$ preorders on *views* and *states*, respectively, (27) expands to

$$\langle get, id \rangle^{\circ} \cdot (\leq_v \times \leq_s) \quad = \quad \leq_s \cdot put$$

and to the pointwise:

$$get\ s' \leq_v v \ \wedge \ s' \leq_s s \quad \Leftrightarrow \quad s' \leq_s put(v, s) \tag{28}$$

for all views v and sources s, s'.

Postscriptum

After Zhenjiang's and Jeremy's talks, and answering to Michel's "exercise" proposal, here is my (tentative!) GC approach to bidirectional programming:

$$(get \times id) \cdot \triangle \quad \vdash \quad put \tag{27}$$

Deja vu? Yes, just make $get, put := len, take$ and you get the GC specification of $take$ (19).

Details: for $\leq_v$ and $\leq_s$ preorders on *views* and *states*, respectively, (27) expands to

$$\langle get, id \rangle^{\circ} \cdot (\leq_v \times \leq_s) \quad = \quad \leq_s \cdot put$$

and to the pointwise:

$$get\ s' \leq_v v \ \wedge \ s' \leq_s s \quad \Leftrightarrow \quad s' \leq_s put(v, s) \tag{28}$$

for all views v and sources s, s'.

Postscriptum

After Zhenjiang's and Jeremy's talks, and answering to Michel's "exercise" proposal, here is my (tentative!) GC approach to bidirectional programming:

$$(get \times id) \cdot \triangle \quad \vdash \quad put \tag{27}$$

Deja vu? Yes, just make $get, put := len, take$ and you get the GC specification of $take$ (19).

Details: for $\leq_v$ and $\leq_s$ preorders on *views* and *states*, respectively, (27) expands to

$$\langle get, id \rangle^{\circ} \cdot (\leq_v \times \leq_s) \quad = \quad \leq_s \cdot put$$

and to the pointwise:

$$get\ s' \leq_v v \ \wedge \ s' \leq_s s \quad \Leftrightarrow \quad s' \leq_s put(v, s) \tag{28}$$

for all views v and sources s, s'.

Postscriptum

After Zhenjiang's and Jeremy's talks, and answering to Michel's "exercise" proposal, here is my (tentative!) GC approach to bidirectional programming:

$$(get \times id) \cdot \triangle \quad \vdash \quad put \tag{27}$$

Deja vu? Yes, just make $get, put := len, take$ and you get the GC specification of $take$ (19).

Details: for $\leq_v$ and $\leq_s$ preorders on *views* and *states*, respectively, (27) expands to

$$\langle get, id \rangle^{\circ} \cdot (\leq_v \times \leq_s) \quad = \quad \leq_s \cdot put$$

and to the pointwise:

$$get\ s' \leq_v v \ \wedge \ s' \leq_s s \quad \Leftrightarrow \quad s' \leq_s put(v, s) \tag{28}$$

for all views v and sources s, s'.

Postscriptum

Upper-cancellation:

$$get(put(v, s)) \leq_v v \ \wedge \ put(v, s) \leq_s s \tag{29}$$

Lower-cancellation:

$$s \leq_s put(get\ s, s)$$

In fact, it turns up (by monotonicity of get) that

$$s = put(get\ s, s) \tag{30}$$

is ensured (next slide), provided $\leq_s$ is a partial order.

Postscriptum

$$
\begin{array}{ll}
 & s' \leq_s put(get\ s, s) \\
\Leftrightarrow & \quad \{\ (28)\ \} \\
 & get\ s' \leq_v get\ s\ \wedge\ s' \leq_s s \\
\Leftrightarrow & \quad \{\ get \text{ monotonic }\} \\
 & s' \leq_s s \\
:: & \quad \{\ \text{IE}\ \} \\
 & put(get\ s, s) = s
\end{array}
$$

Annex

Going pointfree

Success of the 'G'alculator method crucially depends on the **tractability** of GC orderings.

In some situations, it may pay the effort to transform (inductive) data-structures into **non-inductive** ones so as to work with simpler orderings.

Lists, for instance, can be transformed into **simple relations** from positions to elements. This enables a relational, **pointfree** way of expressing properties about lists, for instance

- **ordered** list means **increasing** relation
- **no duplicates in** list means **injective** relation
- **infinite** list means a **function** (stream)

Particularly relevant when using declarative (abstract) notations such as Alloy's.

Example — *nub*

$a\ L\ i$ meaning L holds a in position i, we could say that $nub\ L$ is the largest sublist of L such that

$$\langle \forall\ a, i\ :\ a(nub\ L)i :\ \langle \forall\ j\ :\ aLj :\ j \geq i \rangle \rangle$$

In PF- notation, this corresponds to GC

$$L^{\circ} \cdot X \subseteq \geq\ \wedge\ X \subseteq L\ \Leftrightarrow\ X \subseteq nub\ L \tag{31}$$

(Note in passing that this GC shares the same "pattern" as that of *make*, *put* etc.)

Example — *nub*

Function *nub* found to be a particular case of "range thining" operator adopted to perform non-inductive proofs on a particular structure of a NAND flash model in Alloy (Ferreira and Oliveira, 2010):

$$X \subseteq R \uparrow S \quad \Leftrightarrow \quad X \subseteq R \ \wedge \ X \cdot R^{\circ} \subseteq S \tag{32}$$

which ensures that $R \uparrow S$ is the largest sub-relation X of R such that $\langle \forall\ b', b\ :\ \langle \exists\ a\ ::\ b' X a \wedge b R a \rangle :\ b' S b \rangle$.

NB: (32) can be found in the AoP book (Bird and de Moor, 1997) as the *universal property of unary min* S.

Example — *nub*

It follows that, for S antisymmetric, $R \upharpoonright S$ is always simple.

Properties:

$$R \upharpoonright \bot \;=\; \bot \tag{33}$$

$$R \upharpoonright \top \;=\; R \tag{34}$$

$$R \upharpoonright \Phi \;=\; \text{largest deterministic fragment of } \Phi \cdot R \tag{35}$$

$$(R \cup S) \upharpoonright U = (R \upharpoonright U) \cup (S \upharpoonright U) \;\Leftarrow\; R \cdot S^{\circ} \subseteq \bot \tag{36}$$

How does one derive the algorithm of *nub* from the given "indirect" GC? Working on this at the moment.

R.C. Backhouse. *Mathematics of Program Construction*. Univ. of Nottingham, 2004. Draft of book in preparation. 608 pages.

R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997. C.A.R. Hoare, series editor.

R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, January 1977.

M.A. Ferreira and J.N. Oliveira. Variations on an Alloy-centric tool-chain in verifying a journaled file system model, 2010. (To be submitted to Springer's Formal Aspects of Computing).

P.F. Silva and J.N. Oliveira. 'Galculator': functional prototype of a Galois-connection based proof assistant. In *PPDP '08: Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, pages 44–55, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-117-0. doi: http://doi.acm.org/10.1145/1389449.1389456.