Extended Static Checking by Calculation using the Pointfree Transform

José N. Oliveira

CCTC, Universidade do Minho, 4700-320 Braga, Portugal,
jno@di.uminho.pt

Abstract. The pointfree transform offers to the predicate calculus what the Laplace transform offers to the differential/integral calculus: the possibility of changing the underlying mathematical space so as to enable agile algebraic calculation. This paper addresses the foundations of the transform and its application to a calculational approach to extended static checking (ESC) in the context of abstract modeling. In particular, a calculus is given whose rules help in breaking the complexity of the proof obligations involved in static checking arguments. The close connection between such calculus and that of weakest pre-conditions makes it possible to use the latter in ESC proof obligation discharge, where pointfree notation is again used, this time to calculate with invariant properties to be maintained.

A connection with the *"everything is a relation"* lemma of Alloy is established, showing how close to each other the pointfree and Alloy notations are. The main advantage of this connection is that of complementing pen-and-paper pointfree calculations with model checking support wherever validating sizable abstract models.

Keywords: Theoretical foundations; formal methods; proof obligations; extended static checking.

> *"Certaines personnes ont [l'affectation] d'éviter en apparence toute espèce de calcul, en traduisant par des phrases fort longues ce qui s'exprime très brièvement par l'algèbre, et ajoutant ainsi à la longueur des opérations, les longueurs d'un langage qui n'est pas fait pour les exprimer. Ces personnes-là sont en arrière de cent ans."*
>
> Evariste Galois (1831)

1 Introduction

Much of our programming effort goes into making sure that a number of *"good" relationships* hold among the software artifacts we build. There are two main ways of ensuring that such good things happen. According to the first, the intended relationship is first *postulated* as a logic statement and then *verified*. We shall refer to this as the "invent & verify" way. Alternatively, one may try and *calculate* the intended relationship out of other valid relationships using an algebra, or theory of relationships. This will be referred to as the "correct by construction" approach.

Let us illustrate the contrast between these two approaches with examples. Whenever Haskell programmers declare function types, eg.

$$\underbrace{f}_{function} :: \underbrace{a \rightarrow b}_{type} \tag{1}$$

they are postulating a *"is of type"* relationship involving two kinds of artifact: functions (λ-expressions) and types (τ-expressions). It is quite common to declare $f :: a \rightarrow b$ first, write the body of f afterwards and then wait for the interpreter's reaction (type checker) when verifying the consistency of both declarations.

Clearly, this is an *invent & verify* approach to writing type correct functional code. What about the *correct by construction* alternative? It goes the other way round: one writes the body of f first and lets the interpreter *calculate* (by polymorphic type inference) its principal type, which can be instantiated later, if convenient.

Note that absence of type errors in the *invent & verify* approach does not ensure associating a function to its most generic type: the programmer's guess (invention) may happen to be stronger, implicitly reducing the scope of application of the function being declared. This is also a danger of *invent & verify* applied to *extended typed checking* as in, for instance, typing code using Hoare triples:

$$\{p\}P\{q\}$$

This postulate about piece of code P captures relationship *"is such that pre-condition p ensures post-condition q"*. So it could be alternatively written as

$$\underbrace{P}_{program} :: \underbrace{p \rightarrow q}_{predicative\ type} \tag{2}$$

involving, as artifacts, programs (imperative code) and pre/post conditions (predicates). The *invent & verify* way of handling Hoare triples consists of writing P, inventing p and q and finally proving that $\{p\}P\{q\}$ holds. The *correct by construction* equivalent consists of writing two of the ingredients q, P and p and calculating the third. Typically, one will calculate the weakest pre-condition (*wp*) for q to hold upon execution of P. Again, the calculated pre-condition p may be strengthened at a later stage, if convenient.

Our third and last example, in the area of discrete maths, is perhaps the most eloquent in contrasting verification against calculation. Think of how to postulate that a given function f is a bijection: one may prove f injective, total and surjective or, in typical *invent & verify* mode, guess its converse f° and then prove the two cancellations $\langle \forall\ x\ ::\ f^\circ(f\ x) = x \rangle$ and $\langle \forall\ y\ ::\ f(f^\circ y) = y \rangle$. By contrast, a constructive, calculational alternative will go as follows: using relation algebra, one calculates f°, which in general is a relation, not a function; both f and f° will be bijective iff a function f° is obtained. The approach is constructive (f° is calculated, not guessed) and simpler.

From the examples above it can be observed that "traditional thinking" in maths and software design tends to follow the *invent & verify* reasoning style. This paper is devoted to the alternative, constructive approach to building correct code. In particular, it focuses on a calculational approach to discharging proof obligations involved in writing type correct software, a discipline which can be framed into the wider topic of *extended static type checking* (ESC) [24].

Our starting point is the observation that thinking constructively requires a “turn of mind”. And this raises the question: are the logics and calculi we traditionally rely upon up-to-date for such a turn of mind? In an excellent essay on the history of scientific technology, Russo [55] writes:

> *The immense usefulness of exact science consists in providing models of the real world within which there is a guaranteed method for telling false statements from true. (...) Such models, of course, allow one to describe and predict natural phenomena, by translating them to the theoretical level via correspondence rules, then solving the “exercises” thus obtained and translating the solutions obtained back to the real world.*

The verdict is that disciplines unable to build themselves around *exercises* should be regarded as *pre-scientific*.

This fits neatly into the current paper’s overall message. Our idea is to invest in a scientific theory for software development whereby code is obtained by solving exercises whose solutions are the artifacts one wants to produce. So, the formulæ and equations involved in such exercises should range over programs and properties of programs (assertions, specifications, etc) and not over the particular data values handled (stored, retrieved etc) by such programs. This identifies a first challenge: to devise a way of abstracting from program control/data structures. A second challenge consists in finding a single notation unifying properties of programs, program data, the programs themselves (or models thereof) and their “desirable” relationships.

Fortunately, such a unified notation exists already and does not need to be (re)invented: it is the notation of the pointfree relation calculus [60, 13, 7]. The link between conventional point-level logic and such a relation calculus is a transformation which abstracts from quantifiers and bound variables (points) found in predicates and converts these to formulæ involving binary relations only. In this *pointfree transform* (PF-transform for short) [60, 50] variables are removed from program descriptions in the same way Backus develops his algebra of programs [10]. The main difference stays in the fact that one is transforming logical formulæ while Backus does so for functional terms only [1].

Structure of the paper. The remainder of this paper is organized as follows: sections 2 to 5 are concerned with motivation, background and related work. Extended static checking (ESC) is addressed in sections 6 and 7. The PF-transform and relational calculus, which are central to the whole paper, are given in sections 3, 9 and 12. PF-transformed ESC reasoning leads to the ESC/PF calculus for typing functions which is the subject of sections 8, 10 and 11. The generalization of this to relations is given in sections 14, 15 and 17. Sections 13 and 18 are concerned with case studies illustrating the use of the ESC/PF calculus. The second of these case studies, introduced in section16, is a real-life problem tackled in the context of the Verified Software Initiative. The connection with Alloy is addressed in sections 4 and 19. Conclusions and future work are given in sections 20 and 21, respectively. Annex A lists a number of laws of the Eindhoven quantifier calculus which are relevant to the PF-transform.

[1] See section 5 for more details on the pointfree notation and the origins of relational methods in computer science.

2 Motivation

Consider the following fragment of requirements put by a hypothetical telecom company:

> *(...) For each* list of calls *stored in the mobile phone (eg. numbers dialed, SMS messages, lost calls), the* store *operation should work in a way such that (a) the more recently a* call *is made the more accessible it should be; (b) no number appears twice in a list; (c) only the last 10 entries are stored in each list.*

It is not difficult to write a functional model for the required *store* operation on finite lists of calls,

$$store\ c\ l \quad \triangleq \quad take\ 10\ (c : [\, x \mid x \leftarrow l, x \neq c\,]) \tag{3}$$

where $c : l$ denotes list l prefixed by call number c and $take\ n\ l$ returns the prefix of l of length n, or l itself if $n > length\ l$, as in the Haskell notation and standard libraries [33]. However, how can one be sure that all requirements are properly met by (3)? Think of clause (b), for instance. Intuitively, missing $x \neq c$ in the list comprehension would compromise this property. But, is this enough? too strong?

Following the standard practice in formal methods, one first of all needs to formalize requirement (b) in the form of a predicate on lists of calls [2]:

$$noDuplicates\ l \quad \triangleq \quad \langle \forall\, i, j\ :\ 1 \leq i, j \leq length\ l :\ (l\ i) = (l\ j) \Rightarrow i = j \rangle \tag{4}$$

Next, we need to formulate and discharge the proof obligation which ensures that the *store* operation on lists of calls maintains property $noDuplicates$:

$$\langle \forall\, c, l\ :\ noDuplicates\ l :\ noDuplicates(store\ c\ l) \rangle \tag{5}$$

Desirable properties such as (4) which should be maintained by all operations of a given software application are known as *invariant* properties [32, 31]. Our toy requirements include other such properties, for instance that corresponding to clause (c):

$$leq10\ l \quad \triangleq \quad length\ l \leq 10 \tag{6}$$

Ensuring that invariants are preserved by software operations entails the need for formal proofs. The complexity of such proofs grows dramatically with the complexity of the formal models of both invariant properties and operations. So, any effort to modularize such models and proofs is welcome. In the case of (3), for instance, it can be observed that $store$ is the "pipeline" of three sub-operations: filtering c first, *cons*'ing it afterwards and finally taking 10 elements at most. This is nicely expressed by writing

$$store\ c \triangleq (take\ 10) \cdot (c\ :) \cdot filter(c\ \neq) \tag{7}$$

where $filter$ is the obvious list processing function and combinator "$\cdot$ " denotes function composition:

$$(f \cdot g)\, a \quad \triangleq \quad f\,(g\, a) \tag{8}$$

[2] We use notation $\langle \forall\ x\ :\ R :\ T \rangle$ to mean *for all x in range R it is the case that T holds.* Properties of this notation, known as the *Eindhoven quantifier notation* [7, 4], are given in appendix A.

Note that (7) abstracts from input variable l (of type *list of calls*) thanks to (8) and to extensional functional equality:

$$f = g \quad \Leftrightarrow \quad \langle \forall\, a \ :: \ f\, a = g\, a \rangle \tag{9}$$

Also note the use of sections $(take\ 10)$ and $(c\ :)$ in converting curried binary operators $take$ and $(:)$ into unary ones by "freezing" the first argument.

The main advantage of (7) when compared to (3) is that different invariants may happen to be maintained by different stages of the pipeline, reducing the overall complexity of proof obligations. For instance, $leq10$ has to do with lists not going beyond 10 elements: clearly, this is ensured by the outermost stage alone,

$$\langle \forall\, l \ :: \ length(take\ 10\ l) \leq 10 \rangle \tag{10}$$

independently of how argument list l is built. Property (10) can in fact be shown to hold for function

$$\begin{aligned} take\ 0\ \_ &= [\,] \\ take\ \_\ [\,] &= [\,] \\ take\ (n+1)\ (x : xs) &= x : take\ n\ xs \end{aligned}$$

Clearly, proving (10) requires less effort that proving that $leq10$ is preserved by the whole function $store$:

$$\langle \forall\, c, l \ : \ length\, l \leq 10 : \ length(take\ 10\ (c : [\, x \mid x \leftarrow l, x \neq c\,])) \leq 10 \rangle \tag{11}$$

The use of notation (7) instead of (3) above is an example of PF-transformation: instead of writing $f(g\ a)$ such as in the right hand side of (8), one writes $f \cdot g$ and drops variable (point) a. This kind of transformation, which is not a privilege of functions, is introduced in the section which follows.

3 Overview of the PF-transform

Composing relations. Functional composition (8) is a special case of *relational composition*,

$$b(R \cdot S)c \quad \Leftrightarrow \quad \langle \exists\, a \ :: \ bRa \ \wedge \ aSc \rangle \tag{12}$$

where R, S are binary relations and notation yRx means "*y is related to x by R*".

No other concept traverses human knowledge more ubiquitously than that of a *relation*, from philosophy to mathematics, to information systems (think eg. of relational databases [38]), etc. Symbol R in yRx can stand for virtually any relationship we may think of: not only those expressed by the "::" symbol in type assertions (1,2) but also those expressing facts as simple as eg. `"a"` *prefix_of* `"ab"` among strings, $n \leq n + 1$ among natural numbers, $\text{TRUE} \in \{\text{TRUE}, \text{FALSE}\}$ in the Booleans, etc. In particular, R can be a function f, in which case $y\ f\ x$ means that y is the output of f for input x.

Before going further, note the notation convention of writing outputs on the left hand side and inputs on the right hand side, as suggested by the usual way of declaring functions in ordinary mathematics, $y = f\ x$, where y ranges over outputs (cf. the vertical axis of the Cartesian plane) and x over inputs (cf. the other, horizontal axis). This convention is adopted consistently throughout this text and is extended to relations, as already seen above.

Comparing relations. The main advantage of relational thinking lies in its powerful combinators and associated laws, of which composition (12) is among the most useful: it expresses data flow in maths formulæ in a natural, implicit way while dropping existential quantifiers. Removing quantifiers from formulæ makes these more amenable to calculation. For instance, the rule which introduces *relational inclusion*

$$R \subseteq S \quad \Leftrightarrow \quad \langle \forall\, b, a \;:\; b\, R\, a :\; b\, S\, a \rangle \tag{13}$$

can be regarded (if read from right to left) as a way of dropping a very common pattern of universal quantification. (Read $R \subseteq S$ as "R is at most S", meaning that S is either more defined or less deterministic than R.)

Relational equality is usually established by circular inclusion:

$$R = S \quad \Leftrightarrow \quad R \subseteq S \;\wedge\; S \subseteq R \tag{14}$$

A less obvious, but very useful way of calculating the equality of two relations is the method of *indirect equality* [1, 13]:

$$R = S \quad \Leftrightarrow \quad \langle \forall\, X \;::\; (X \subseteq R \Leftrightarrow X \subseteq S) \rangle \tag{15}$$

The reader unaware of this way of indirectly setting algebraic equalities will recognize that the same pattern of indirection is used when establishing set equality via the membership relation, cf. $A = B \;\Leftrightarrow\; \langle \forall\, x \;::\; x \in A \Leftrightarrow x \in B \rangle$.

Dividing relations. It is easy to check that $R \cdot S$ (12) has a *multiplicative* flavour: it is associative (albeit not commutative), it distributes over the union of two relations $R \cup S$, defined by

$$b(R \cup S)a \;\triangleq\; bRa \vee bSa$$

and it has a unit element, the identity relation id defined in the obvious way: $b\; id\; a$ iff $b = a$. Given such a multiplicative flavour, one may question: is there any reasonable notion of *relation division*? It turns out that the following property holds, for all binary relations R, S and T

$$X \cdot R \subseteq S \;\Leftrightarrow\; X \subseteq S/R \tag{16}$$

where S/R is the relation whose pointwise meaning is

$$a(S/R)b \quad \Leftrightarrow \quad \langle \forall\, c \;:\; b\, R\, c :\; a\, S\, c \rangle \tag{17}$$

Again note the economy of notation S/R when compared to its pointwise expansion as a universal quantification. Expanding the whole of (16) will lead to formula

$$\langle \forall\, b, a \;:\; \langle \exists\, c \;:\; bXc :\; cRa \rangle :\; bSa \rangle \quad \Leftrightarrow \quad \langle \forall\, b, c \;:\; bXc :\; \langle \forall\, a \;:\; cRa :\; bSa \rangle \rangle$$

which expresses a trading rule between existential and universal quantification harder to parse and memorize.

Phrase *pointfree transform* (or *PF-transform* for short) will denote, throughout this paper, this process of transforming predicate calculus expressions into their equivalent relational combinator based representations.

Coreflexives. Given a binary predicate p, we will denote by R_p the binary relation such that $b\ R_p\ a \ \Leftrightarrow\ p(b, a)$ holds, for all suitably typed a and b. How does one transform a unary predicate $u\ a$ into a *binary* relation? We will see that this can be done in more than one way, for instance by building the relation Φ_u such that $b\ \Phi_u\ a$ means $(b = a)\ \wedge\ (u\ a)$. That is, Φ_u is the relation that maps every a which satisfies u (and only such a) onto itself. Clearly, such relation is a fragment of the identity relation: $\Phi_u \subseteq id$.

Relations at most id are referred to as *coreflexive* relations and those larger than id as *reflexive* relations. Coreflexives will be denoted by uppercase Greek letters (Φ, Ψ) as in the case of Φ_u.

Composition with coreflexives expresses pre-conditioning and post-conditioning in a natural way, cf. $R \cdot \Phi$ and $\Psi \cdot R$, respectively. Coreflexives also act as data *filters*. For instance, suppose we need to transform the following variant of the right hand side of (12), $\langle \exists\ a\ :\ u\ a :\ b\ R\ a\ \wedge\ a\ Sc \rangle$, where u shrinks the range of the quantification. It can be easily checked that $R \cdot \Phi_u \cdot S$ is the corresponding extension to (12) [3].

Arrow notation and diagrams. We will use arrows to depict relations. In general, arrow $B \xleftarrow{R} A$ denotes a binary relation with source type A and target type B. We will say that $B \longleftarrow A$ is the *type* of R and write $b\ R\ a$ to mean that pair (b, a) is in R. Type declarations $B \xleftarrow{R} A$ and $A \xrightarrow{R} B$ mean the same. Arrow notation makes it possible to explain relational formulæ in terms of diagrams. For instance,

$$\begin{tikzcd} C \arrow[d, "X"'] & & A \arrow[ll, "R"'] \arrow[dll, "S"] \\ B & \subseteq & \end{tikzcd} \quad \text{equivalent to} \quad \begin{tikzcd} C \arrow[d, "X"'] \arrow[dr, phantom, "\subseteq"] & C \arrow[l, "id"'] \arrow[d, "S/R"] \\ B & B \arrow[l, "id"] \end{tikzcd}$$

helps in understanding (16).

Galois connections. Properties such as (16) are known as *Galois connections* (GCs) [51] and prove very useful in problem understanding and reasoning, while bearing particular resemblance with school algebra: compare, for instance, (16) with a similar property defining integer division, for all $d, n, q \in I\!N$ $(d > 0)$ [4]:

$$q \times d \leq n\ \Leftrightarrow\ q \leq n/d \qquad \begin{array}{r|l} n & d \\ \cline{2-2} r & q \end{array} \tag{18}$$

By substituting $X := S/R$ in (16) we obtain $(S/R) \cdot R \subseteq S$ meaning that S/R approximates S once composed with R; by reading (16) from left to right, we obtain implication $X \cdot R \subseteq S\ \Rightarrow\ X \subseteq S/R$, which means that S/R is largest among all such approximations. So S/R is a supremum (as is quotient n/d).

[3] See section 9 in the sequel for more about this important class of relations.

[4] See [56] for a derivation of the algorithm of integer division from Galois connection (18) as an example of PF-calculation performed by the *Galculator*, the prototype of a proof assistant solely based on the algebra of Galois connections and PF-reasoning.

Table 1. Sample of PF-transform rules.

Pointwise	Pointfree
$\langle \exists\, a\ ::\ b\, R\, a\ \wedge\ a\, S\, c \rangle$	$b(R \cdot S)c$
$\langle \forall\, x\ :\ x\, R\, b :\ x\, S\, a \rangle$	$b(R \setminus S)a$
$\langle \forall\, c\ :\ b\, R\, c :\ a\, S\, c \rangle$	$a(S\, /\, R)b$
$b\, R\, a\ \wedge\ c\, S\, a$	$(b,c)\langle R, S\rangle a$
$b\, R\, a\ \wedge\ d\, S\, c$	$(b,d)(R \times S)(a,c)$
$b\, R\, a\ \wedge\ b\, S\, a$	$b\ (R \cap S)\ a$
$b\, R\, a\ \wedge\ \neg b\, S\, a$	$b\ (R - S)\ a$
$b\, R\, a \vee b\, S\, a$	$b\ (R \cup S)\ a$
$(f\ b)\ R\ (g\ a)$	$b(f^\circ \cdot R \cdot g)a$
TRUE	$b\ \top\ a$
FALSE	$b\ \bot\ a$
$\langle \forall\, b, a\ :\ b\, R\, a :\ b\, S\, a \rangle$	$R \subseteq S$
$\langle \forall\, a\ ::\ a\, R\, a \rangle$	$id \subseteq R$

As example of other Galois connections bearing relationship with school algebra consider the following, which captures the operation which “subtracts” relations,

$$X - R \subseteq Y \quad \Leftrightarrow \quad X \subseteq Y \cup R \tag{19}$$

and is analogue of number subtraction:

$$x - n \leq y \quad \Leftrightarrow \quad x \leq y + n \tag{20}$$

Table 1 lists the most common relational operators associated to the PF-transform. $R \cap S$ denotes the intersection (or *meet*) of two relations R and S. $\top$ is the largest relation of its type. Its dual is $\bot$, the smallest such relation (the empty one). The following universal properties of relational *meet* and *join* are also Galois connections:

$$X \subseteq R \cap S \quad \Leftrightarrow \quad X \subseteq R \wedge X \subseteq S \tag{21}$$

$$R \cup S \subseteq X \quad \Leftrightarrow \quad R \subseteq X \wedge S \subseteq X \tag{22}$$

The two variants of division in table 1 arise from the fact that relation composition is not commutative, the Galois connection for $R \setminus S$ being similar to (16):

$$R \cdot X \subseteq S \Leftrightarrow X \subseteq R \setminus S \tag{23}$$

Converses. Every relation $A \xrightarrow{R} B$ has a converse, which is relation $A \xleftarrow{R^\circ} B$ such that

$$a(R^\circ)b \Leftrightarrow b\ R\ a \tag{24}$$

holds. Two important properties of converse follow: it is an involution

$$(R^\circ)^\circ = R \tag{25}$$

and it commutes with composition in a contravariant way:

$$(R \cdot S)^\circ = S^\circ \cdot R^\circ \tag{26}$$

Functions. Lowercase symbols (f, g) stand for relations which are *functions*. The interplay between functions and relations is a rich part of the binary relation calculus [13]. From table 1 we single out rule

$$b(f^\circ \cdot R \cdot g)a \;\Leftrightarrow\; (f\ b)R(g\ a) \qquad \text{cf. diagram} \qquad \begin{tikzcd} C & D \arrow[l, "R"'] \\ B \arrow[u, "f"] & A \arrow[l, "f^\circ \cdot R \cdot g"] \arrow[u, "g"'] \end{tikzcd} \tag{27}$$

which involves two functions f, g and relation R and plays a special role in pushing variables out of relational expressions.

The exact characterization of functions as special cases of relations is achieved in terms of converse, which is in fact of paramount importance in establishing the whole taxonomy of binary relations depicted in figure 1. First, we define two important notions: the *kernel* of a relation R, $\mathsf{ker}\, R \;\triangleq\; R^\circ \cdot R$ and its dual, $\mathsf{img}\, R \;\triangleq\; R \cdot R^\circ$, the *image* of R [5].

From (25, 26) one immediately draws

$$\mathsf{ker}\,(R^\circ) = \mathsf{img}\, R \tag{28}$$

$$\mathsf{img}\,(R^\circ) = \mathsf{ker}\, R \tag{29}$$

Kernel and image lead to the four top criteria of the taxonomy of figure 1:

	Reflexive	*Coreflexive*
$\mathsf{ker}\, R$	entire R	injective R
$\mathsf{img}\, R$	surjective R	simple R

(30)

In words: a relation R is said to be *entire* (or total) iff its kernel is reflexive and to be *simple* (or functional) iff its image is coreflexive. Dually, R is *surjective* iff R° is entire, and R is *injective* iff R° is simple.

Let us check (30) with examples. First, we PF-transform the pointwise formula which captures function injectivity:

$$\begin{array}{cl} & f \text{ is injective} \\ \Leftrightarrow & \quad \{ \text{ recall definition from school maths } \} \\ & \langle \forall\, y, x \;:\; (f\ y) = (f\ x) :\; y = x \rangle \\ \Leftrightarrow & \quad \{ \text{ introduce } id \text{ (twice) } \} \\ & \langle \forall\, y, x \;:\; (f\ y)id(f\ x) :\; y(id)x \rangle \\ \Leftrightarrow & \quad \{ \text{ (27) } \} \\ & \langle \forall\, y, x \;:\; y(f^\circ \cdot id \cdot f)x :\; y(id)x \rangle \end{array} \tag{31}$$

[5] These operators are relational extensions of two concepts familiar from set theory: the image of a function f, which corresponds to the set of all y such that $\langle \exists\ x \;::\; y = f\ x \rangle$, and the kernel of f, which is the equivalence relation $b(\mathsf{ker}\, f)a \;\Leftrightarrow\; f\ b = f\ a$. (See exercise 3 later on.)

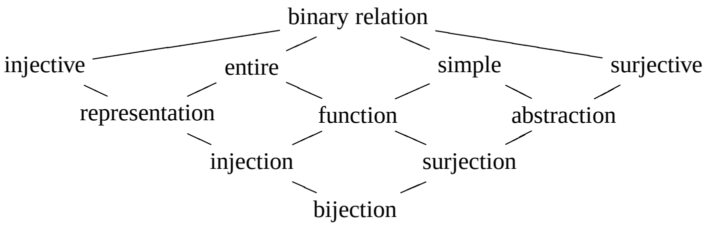


Fig. 1. Binary relation taxonomy

$$
\begin{array}{ll}
\Leftrightarrow & \{ \; id \text{ is the unit of composition; then go pointfree via (13)} \; \} \\
 & f^{\circ} \cdot f \subseteq id \\
\Leftrightarrow & \{ \; \text{definition} \; \} \\
 & \mathsf{ker}\, f \subseteq id
\end{array}
$$

Going the other way round, let us now see what $id \subseteq \mathsf{img}\, f$ means:

$$
\begin{array}{ll}
 & id \subseteq \mathsf{img}\, f \\
\Leftrightarrow & \{ \; \text{definition} \; \} \\
 & id \subseteq f \cdot f^{\circ} \\
\Leftrightarrow & \{ \; \text{relational inclusion (13)} \; \} \\
 & \langle \forall\, y, x \; : \; y(id)x : \; y(f \cdot f^{\circ})x \rangle \\
\Leftrightarrow & \{ \; \text{identity relation ; composition (12)} \; \} \\
 & \langle \forall\, y, x \; : \; y = x : \; \langle \exists\, z \; :: \; y \; f \; z \; \wedge \; z \; f^{\circ} x \rangle \rangle \\
\Leftrightarrow & \{ \; \text{converse (24)} \; \} \\
 & \langle \forall\, y, x \; : \; y = x : \; \langle \exists\, z \; :: \; y \; f \; z \; \wedge \; x \; f \; z \rangle \rangle \\
\Leftrightarrow & \{ \; \forall\text{-one point rule (175) ; trivia ; function } f \; \} \\
 & \langle \forall\, x \; :: \; \langle \exists\, z \; :: \; x = f \; z \rangle \rangle \\
\Leftrightarrow & \{ \; \text{recalling definition from school maths} \; \} \\
 & f \text{ is surjective}
\end{array}
$$

The interested reader is welcome to convert the two remaining entries of (30) to pointwise notation.

Exercise 1. Resort to (28,29) and (30) to prove the following four rules of thumb:

- 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)

□

Exercise 2. Show that

$$R \cup S \text{ is injective} \;\Leftrightarrow\; R \text{ is injective} \wedge S \text{ is injective} \wedge R^\circ \cdot S \subseteq id \tag{32}$$

$$R \cup S \text{ is simple} \;\Leftrightarrow\; R \text{ is simple} \wedge S \text{ is simple} \wedge R \cdot S^\circ \subseteq id \tag{33}$$

Suggestion: resort to universal property (22).
□

Exercise 3. Given a function $B \xleftarrow{f} A$, use (27) in the calculation of

$$b(\mathsf{ker}\, f)a \;\Leftrightarrow\; f\, b = f\, a \tag{34}$$

□

Constant functions. Quite often one needs to internalize particular constant values in PF-expressions. For instance, we may want to say that, given some x, there exists some z such that $x > z$ and $f\ z = c$, for some fixed value c. This requires the *"everywhere* c*"* constant function. In general, given a nonempty datatype C and $c \in C$, notation $\underline{c}$ denotes such a function:

$$\begin{aligned} \underline{c} &: A \longrightarrow C \\ \underline{c}\, a &\triangleq c \end{aligned} \tag{35}$$

Thanks to (35) and (27) it can be easily checked that PF-term $> \cdot f^\circ \cdot \underline{c}$ asserts the requirement above.

Constant functions are also useful in PF-transforming particular relation pairs. For instance, it is easy to check that $\underline{b} \cdot \underline{c}^\circ$ is the singleton relation $\{(b, c)\}$. Then $\mathsf{img}\, \underline{c}$ is the singleton coreflexive $\{(c, c)\}$ which PF-transforms predicate $\lambda x.x = c$:

$$\Phi_{\lambda x.x=c} = \mathsf{img}\, \underline{c} \tag{36}$$

Thanks to (34), it is easy to show that $\top$ is the kernel of every constant function, $1 \xleftarrow{!} A$ included, where function ! — read "!" as "bang" — is the unique function of its type, where 1 denotes the singleton data domain.

Exercise 4. Check the meaning of relation $\underline{b}^\circ \cdot \underline{c}$.
□

The Reynolds-Backhouse relation on functions. Consider two functions f and g related in the following way: if $y = f\ x$ holds then $y \leq g\ x$ holds, for a given ordering $\leq$ on the outputs of both f and g [6]. It is easy to see that this relationship between f and g PF-transforms to $f \subseteq \leq \cdot g$. Now suppose that g is such that $g \cdot \preceq \subseteq \leq \cdot g$, for $\preceq$ another ordering, this time on the input side. Back to points, this re-writes to $\langle \forall\, x, x' : x \preceq x' : g\, x \leq g\, x' \rangle$, meaning that g is monotonic.

[6] It is common to record this fact by writing $f \stackrel{.}{\leq} g$, the so-called pointwise ordering on functions.

$$\begin{tikzcd} B \arrow[d, "f"'] \arrow[dr, phantom, "\subseteq"] & A \arrow[l, "S"'] \arrow[d, "g"] \\ C & D \arrow[l, "R"] \end{tikzcd}$$

Once PF-transformed, the two situations just above are instances of the diagram aside, for suitable versions of relations R, S and functions f, g (these two, in particular, can be the same, as in the monotonicity condition). The diagram captures a very useful way of relating functions and relations (note the higher-order flavour) which was identified first by John Reynolds [54] and later treated in the pointfree style by Roland Backhouse [5, 3].

We will refer to this as the relational "arrow combinator", to be written $R \leftarrow S$. Given R and S, $R \leftarrow S$ is a relation on functions f and g defined as follows:

$$f(R \leftarrow S)g \quad \Leftrightarrow \quad f \cdot S \subseteq R \cdot g \tag{37}$$

With points, $f(R \leftarrow S)g$ means $\langle \forall\, b, a \ :\ b\ S\ a :\ (f\ b) R (g\ a)\rangle$, that is, f and g produce R-related outputs $f\ b$ and $g\ a$ provided their inputs are S-related ($b\ S\ a$).

Properties and applications of this (PF) relational combinator can be found in eg. [3, 12]. The special case $f(R \leftarrow S)f$ will suit our needs later on, and it will prove useful to write $R \xleftarrow{f} S$ to mean $f(R \leftarrow S)f$. Therefore, we will rely on equivalence

$$R \xleftarrow{f} S \quad \Leftrightarrow \quad f \cdot S \subseteq R \cdot f \tag{38}$$

The notation just introduced captures the view that types of functions can be regarded as relations. This is indeed the essence of the abstraction theorem [54] on type polymorphism which, as we shall see in section 11, plays its role in what is to come.

This important combinator closes our introduction to the relational combinators involved in the PF-transform. Before proceeding to the application of this transform to our topics of interest, let us frame it into a wider context.

4 Haskell and Alloy: two PF-flavoured languages

As will become apparent throughout this paper, PF-notation will be regarded as a single, abstract (ie. technology free) unifying notation encompassing program specifications, implementations, program data and program properties. How far is such a notation from programming languages and notations available from the community?

Most commercially available programming languages are pointwise. But there are notations and languages which embody a pointfree subset. Functional programming languages with higher order functions have the power to define functional combinators and therefore make it possible to program in the pointfree style. Among these, some actually have pointfree constructs in their core syntax. Haskell [33] is one of these, as we have already seen in the motivating example of section 2. However, the artifacts one can build in Haskell do not go beyond partial functions, that is, simple relations.

Alloy [30] — a notation and associated model-checking tool which has been successful in *alloying* a number of disparate approaches to software modeling, namely model-orientation, object-orientation, etc. — is a rare example of a language where relations and their combinators are the standard way of doing things. In fact, the "everything is a relation" motto of Alloy matches perfectly with the view purported in the current paper. Quoting [30]:

> (...) “ *All structures are represented as relations, and structural properties are expressed with a few simple but powerful operators.* (...) *Sets are represented as relations with a single column, and scalars as singleton sets.* (...) *In the Alloy logic, all values are relations* [and] *the unification of sets and relations makes the syntax simpler, since there is no need to convert between sets and relations, or between scalars and sets.*” (...) *In Alloy, everything’s a relation.*

It is interesting to note that Haskell and Alloy complement each other in a nice way: Haskell provides for models closer to implementations in the sense that they are reactive by construction: the idea is to evaluate typed λ-expressions which express the reaction of a system to input stimuli. However, there is no native syntax to express datatype invariants, pre and post-conditions and assertions. As a checking tool, Haskell invites the software designer to invent test cases and check for their behaviour [7].

Alloy is not functional, therefore it is a passive language. One writes uninterpreted data models and predicates about such models, as well as assertions about such predicates. The system runs checks for such assertions trying and finding counter-examples able to falsify such assertions. If no counter-example is found then the formula *may be* valid. Purists often regard model-checking as the poor relative to theorem proving. Experience tells, however, that many subtleties and design flaws can be unveiled by model checking. In other words: the checker does not prove things for certain but is of great help in improving what one wants to prove.

To catch a glimpse of the proximity between Alloy and the PF-notation adopted in the current paper, consider the Alloy pointwise definition of an injective relation R [8],

```
pred Injective {
all x, y : A, z : B | z in x.R && z in y.R => x=y
}
```

and its PF-equivalent,

```
pred Injective’ {
R.˜R in iden :> A
}
```

— recall $R^\circ \cdot R \subseteq id$ (30), for R° denoted by `˜R` and id denoted by `iden :> A`. Also note that composition is written in reverse order.

5 Related Work

The idea of encoding predicates in terms of relations was initiated by De Morgan in the 1860s and followed by Peirce who, in the 1870s, found interesting equational laws of the calculus of binary relations [53]. The pointfree nature of the notation which emerged from this embryonic work was later further exploited by Tarski and his students [60]. In the 1980’s, Freyd and Ščedrov [25] developed the notion of an *allegory* (a category whose homsets are partially ordered) which eventually accommodates the

[7] Tools such as QuickCheck [15] help in this respect.

[8] Note the transposed notation `x.R` meaning set $\{y \mid y\ R\ x\}$.

binary relation calculus as special case. In this context, a relation R is viewed as an arrow (morphism) $B \xleftarrow{R} A$ between objects B and A, respectively referred to as the target and source of R. Composition of such arrows corresponds to relational composition (12), identity is id, and relational expressions can be "type-checked" by drawing diagrams such as in category theory.

Such advances in mathematics were meanwhile captured by the Eindhoven computer science school in their development of program construction as a mathematical discipline [1, 8, 20, 13, 7] enhanced by judicious use of Galois connections, as already illustrated above.

Our view of this approach as a kind of *Laplace transform* [37] for logic was first expressed in [42]. Such a transform (the PF-transform) has henceforth been applied to several areas of the software sciences, namely relational database schema design [44, 2, 18], *hashing* [49], software components [11], coalgebraic reasoning [12], algorithmic refinement [50], data refinement [18, 48] and separation logic [65].

The remainder of this paper will be devoted to yet another example of application of the PF-transform which we regard as a particularly expressive illustration of its power: extended static type checking (ESC) [24]. If performed at abstract model level, ESC includes what is commonly known as invariant preservation and satisfiability proof obligations in specification languages such as VDM [32, 22] and Z [57]. Hoare triples [27] and weakest pre-condition calculus [19] are also related to ESC, as will be shown later. With notable exceptions (eg. [9, 6]) these theories are available in the *pointwise* style, as most theories in computing are. Evidence will be provided not only of the unifying effect of the PF-transform in putting together different (but related) theories in programming but also of how it can be used and applied to real-sized (non trivial) case studies in connection with mechanical support provided by model checking [30] and theorem proving [26].

6 Extended static checking and datatype invariants

Type theory [52] is unanimously regarded as one of the most solid and relevant branches of computer science. Thanks to the concept of a *type*, the quality of code can be checked statically, ie. before execution. In programming languages such as Haskell, for instance, ill-typed programs simply don't compile, meaning that types are an effective way of controlling software robustness.

The ESC acronym for "extended static checking" was coined at Compaq SRC in their development of a tool for Java (ESC/Java) able to detect as many programming errors as possible at compile-time [24]:

> *Our group at the Systems Research Center has built and experimented with two realizations of a new program checking technology that we call extended static checking (ESC): "static" because the checking is performed without running the program, and "extended" because ESC catches more errors than are caught by conventional static checkers such as type checkers.*

If we look at the particular kinds of error which such a tool is able to catch — null dereferencing, array bounds errors, negative array indices, etc — we realize that these can be abstractly characterized by properties of particular datatypes which are violated by the running program, and/or a pre-condition of a given operation which is not ensured in some program trace. These two are related: the standard way of ensuring that a particular property of a datatype is maintained consists of adding pre-conditions to operations which may put such properties at risk.

However, adding arbitrary run-time checks for every property (a style often referred to as *defensive programming* [39]) may be counterproductive: one may write too many or much too strong checks. In the limit, the context may happen to ensure the properties one wants to maintain, thus rendering such checks useless and redundant.

Properties statically associated to datatypes are known as *invariants* [32] and as *state invariants* in case the particular datatypes embody the state of some state-based machine or system, often handled coalgebraically [31, 12]. For instance, in a system for monitoring aircraft flight paths of in a controlled airspace [22], altitude, latitude and longitude cannot be specified simply as

$$Alt \;=\; Lat \;=\; Lon \;=\; \mathbb{R}$$

because altitudes cannot be negative, latitudes must range between $-90°$ and $90°$ and longitudes between $-180°$ and $180°$. Using traditional maths notation, one would write:

$$\begin{aligned} Alt &= \{a \in \mathbb{R} \mid a \geq 0\} \\ Lat &= \{x \in \mathbb{R} \mid -90 \leq x \leq 90\} \\ Lon &= \{y \in \mathbb{R} \mid -180 \leq y \leq 180\} \end{aligned} \tag{39}$$

Formal modeling notations such as VDM and Z cater specially for invariants. In the case of languages of the VDM family (eg. VDM-SL [22], VDM++ [23]) the standard notation is

$$\begin{aligned} &Alt \;=\; \mathbb{R} \\ &\textbf{inv}\;\; a \triangleq\; a \geq 0 \end{aligned}$$

for Alt (39) (and similarly for Lat and Lon), which implicitly defines a predicate

$$\begin{aligned} &\text{inv-}Alt : \mathbb{R} \to \mathbb{B} \\ &\text{inv-}Alt\; a \triangleq\; a \geq 0 \end{aligned}$$

known as the *invariant* of Alt. In general, given A and a predicate $p : A \to \mathbb{B}$, data type declaration

$$\begin{aligned} &T \;=\; A \\ &\textbf{inv}\;\; a \triangleq\; p\, a \end{aligned}$$

means the type whose extension is

$$T = \{x \in A \mid p\, x\}$$

Therefore, writing $a \in T$ means $a \in A \;\wedge\; p\; a$. Note that A itself can have its own invariant, so the process of finding which properties hold about a given datatype is inductive on the structure of types. (See more about this in section 17.)

7 Invariants entail proof obligations

Static checking of formal models involving invariants is a complex process relying on generation and discharge of proof obligations, as pointed out more than two decades ago by Jones [32]:

> *The valid objects of $Datec$ are those which (...) satisfy inv-$Datec$. This has a profound consequence for the type mechanism of the notation. (...) The inclusion of a sub-typing mechanism which allows truth-valued functions forces the type checking here to rely on proofs.*

The required proofs, which are known under the headings *invariant preservation* or *satisfiability* [32] [9] belong clearly to the ESC family. Recalling the mobile phone toy requirements of section 2, it should be clear by now that predicates $noDuplicates$ (4) and $leq10$ (6) are components of the invariant of the list of calls datatype handled by $store$, say

$$ListOfCalls = Call^{\star}$$
$$\mathbf{inv}\ \ l \triangleq noDuplicates\, l \ \wedge\ leq10\, l$$

and that (5) and (11) express two proof obligations entailed by such an invariant, concerning the $store$ operation.

In general, given a function $A \xrightarrow{f} B$ where both A and B have invariants, extended static checking (ESC) of f means discharging proof obligation (PO)

$$\langle \forall\, a \ :\ \text{inv-}A\ a :\ \text{inv-}B(f\ a)\rangle \tag{40}$$

which ensures that f is invariant-preserving. The fact that invariants are intrinsic to datatypes is better captured by the following version of the above,

$$\langle \forall\, a \ :\ a \in A :\ (f\ a) \in B\rangle \tag{41}$$

where membership ($\in$) should be understood in the broad sense of encompassing all invariants. (Again we anticipate that this will be handled in precise terms later on in section 17.) Also note the following variant of (41),

$$\langle \forall\, a, b \ :\ a \in A\ \wedge\ b = f\ a :\ b \in B\rangle \tag{42}$$

which is granted by the $\forall$-one-point rule (175).

How does one handle ESC POs? The sheer complexity of such proofs in real-size problems calls for mechanical support and this can be essentially of three kinds: PO-generation, model-checking and theorem-proving.

Generating all proof obligations (POs) needed for checking a particular formal model is a mechanical process available from tool-sets such as eg. the VDMTools [17]. In practice, the number of generated POs is larger than expected because of the adoption of “rich types” such as sequences and finite mappings, which can be regarded as *simple* relations (30), as we shall see. Such types, in a sense, hide particularly common invariants which “turn up” at PO-level.

The following situations can take place:

[9] This nuance will be explained in section 14.

1. Independently of satisfying (42) or not, f is "semantically wrong" because it does not behave according to the requirements. This calls for manual tests, which may include running the model as a prototype, should an interpreter be available.
2. f survives all tests compiled in the previous step (including dynamic type checks) and yet testers are not aware that it does not satisfy (42). In this case, a model checker able to automatically generate counter-examples to (42) which could suggest how to improve f is welcome.
3. The model checker of the step just above finds no counter-examples. In this case a theorem prover is welcome to mechanically check (42).
4. Proof obligation (42) is too complex for the available theorem prover. In this situation, our ultimate hope is a pen-and-paper manual proof, or some kind of exercise able to decompose too complex POs into smaller sub-proofs.

The main purpose of this paper is to show the suitability of the PF-transform and relation calculus to carry out the pen-and-paper proofs (as exercises in the sense of [55]) mentioned in the last step. The idea is to regard such POs as "first class citizens" which are represented by arrows which, in turn, can be put together or decomposed in simpler ones using a suitable PO-calculus supported by the relational calculus.

8 PF-transformed ESC

In [46] it is argued that the complexity of POs mentioned above is partly due to the pointwise notation itself, which does not scale up very well to complex models, leading to long, unreadable POs full of nested quantifications. Experience in PF-transforming such formulæ invariably leads to much shorter, sharp relation-level formulæ which (albeit more cryptic) convey the essence of the proof, which quite often has to do with particular relationships between data flows.

In this section we set ourselves the task of investigating PF-transformed ESC proof obligations. As we shall see, these include invariant preservation, satisfiability and Hoare triples. We begin with a very simple example: checking a function which doubles even numbers,

$$\begin{aligned} &twice : Even \to Even \\ &twice\, n \triangleq 2n \end{aligned}$$

where

$$\begin{aligned} &Even \;=\; \mathbb{N}_0 \\ &\textbf{inv}\;\; n \triangleq \underbrace{\langle \exists\, k \;:\; k \in \mathbb{N}_0 :\; n = 2k \rangle}_{even\ n} \end{aligned} \tag{43}$$

Is $twice$ properly typed? To be so, the following instance of (42) telling that function $twice$ preserves even numbers

$$\langle \forall\, x, y \;:\; even\, x \;\wedge\; y = twice\, x :\; even\, y \rangle \tag{44}$$

should be discharged. According to our strategy, the first step consists in PF-transforming (44). We tackle the range of quantification (44) first,

$$
\begin{array}{cl}
 & y = \mathit{twice}\ x \ \wedge\ \mathit{even}\ x \\
\Leftrightarrow & \quad \{ \ \exists\text{-one-point (176)} \ \} \\
 & \langle \exists\ z\ :\ z = x :\ y = \mathit{twice}\ z\ \wedge\ \mathit{even}\ z \rangle \\
\Leftrightarrow & \quad \{ \ \exists\text{-trading (174) ; introduce coreflexive } \Phi_{even} \ \} \\
 & \langle \exists\ z\ ::\ y = \mathit{twice}\ z\ \wedge\ \underbrace{z = x\ \wedge\ \mathit{even}\ z}_{z\ \Phi_{even}\ x} \rangle \\
\Leftrightarrow & \quad \{ \ \text{composition (12)} \ \} \\
 & y(\mathit{twice} \cdot \Phi_{even})x
\end{array}
$$

cf. diagram

$$\begin{tikzcd}
\mathbb{N}_0 \arrow[d, "twice"'] & \mathbb{N}_0 \arrow[l, "\Phi_{even}"'] \\
\mathbb{N}_0 &
\end{tikzcd}$$

which expresses twice pre-conditioned by even. Next, we proceed to the whole thing:

$$
\begin{array}{cl}
 & \langle \forall\ x, y\ :\ y = \mathit{twice}\ x\ \wedge\ \mathit{even}\ x :\ \mathit{even}\ y \rangle \\
\Leftrightarrow & \quad \{ \ \text{just above} \ \} \\
 & \langle \forall\ x, y\ :\ y(\mathit{twice} \cdot \Phi_{even})x :\ \mathit{even}\ y \rangle \\
\Leftrightarrow & \quad \{ \ \exists\text{-one-point (176)} \ \} \\
 & \langle \forall\ x, y\ :\ y(\mathit{twice} \cdot \Phi_{even})x :\ \langle \exists\ z\ :\ z = y :\ \mathit{even}\ z \rangle \rangle \\
\Leftrightarrow & \quad \{ \ \text{predicate calculus: } p\ \wedge\ \textsc{True} = p \ \} \\
 & \langle \forall\ x, y\ :\ y(\mathit{twice} \cdot \Phi_{even})x :\ \langle \exists\ z\ ::\ y = z\ \wedge\ \mathit{even}\ z\ \wedge\ \textsc{True} \rangle \rangle \\
\Leftrightarrow & \quad \{ \ \top \text{ is the topmost relation, cf. table 1} \ \} \\
 & \langle \forall\ x, y\ :\ y(\mathit{twice} \cdot \Phi_{even})x :\ \langle \exists\ z\ ::\ y\ \Phi_{even}\ z\ \wedge\ z \top x \rangle \rangle \\
\Leftrightarrow & \quad \{ \ \text{composition (12)} \ \} \\
 & \langle \forall\ x, y\ :\ y(\mathit{twice} \cdot \Phi_{even})x :\ y(\Phi_{even} \cdot \top)x \rangle \\
\Leftrightarrow & \quad \{ \ \text{go pointfree (13)} \ \} \\
 & \mathit{twice} \cdot \Phi_{even}\ \subseteq\ \Phi_{even} \cdot \top
\end{array}
\qquad (45)
$$

Note that the two occurrences of unary predicate even in (44) are PF-transformed in two different but related ways: via coreflexive Φ_{even} on the lower side of (45) and via $\Phi_{even} \cdot \top$ on the upper side — a so-called (left) *condition* [10]. Coreflexives relate to

[10] For a detailed account of this duality see the *monotype-condition isomorphism* formalised in [20].

conditions in a number of ways, namely in what concerns pre/post restrictions:

$$R \cdot \Phi = R \cap \top \cdot \Phi \qquad (46)$$

$$\Psi \cdot R = R \cap \Psi \cdot \top \qquad (47)$$

This makes it possible to transform (45) even further:

$$\begin{array}{ll}
 & (45) \\
\Leftrightarrow & \quad \{ \text{ (21), since } twice \cdot \Phi_{even} \subseteq twice \ \} \\
 & twice \cdot \Phi_{even} \ \subseteq \ twice \cap \Phi_{even} \cdot \top \\
\Leftrightarrow & \quad \{ \text{ (47) } \} \\
 & twice \cdot \Phi_{even} \ \subseteq \ \Phi_{even} \cdot twice \\
\Leftrightarrow & \quad \{ \text{ (38) } \} \\
 & \Phi_{even} \xleftarrow{twice} \Phi_{even}
\end{array} \qquad (48)$$

cf. diagram

$$\begin{tikzcd}
\mathbb{N}_0 \arrow[d, "twice"'] & & \mathbb{N}_0 \arrow[ll, "\Phi_{even}"'] \arrow[d, "twice"] \\
\arrow[rr, phantom, "\subseteq"] & & \\
\mathbb{N}_0 & & \mathbb{N}_0 \arrow[ll, "\Phi_{even}"]
\end{tikzcd}$$

In retrospect, PF-statement (48) of proof obligation (44) is interesting from a number of viewpoints: notationally, it is of great economy; conceptually, it really purports the idea that ESC has to do with types, which are now regarded as predicates (encoded by coreflexives); last of not least, it is of great calculational value, as we shall soon see.

Let us generalize what we have obtained thus far:

Definition 1 (Predicative types of functions). *Let function* $B \xleftarrow{f} A$ *and predicates* $\mathbb{B} \xleftarrow{p} A$ *and* $\mathbb{B} \xleftarrow{q} B$ *be given. We say that* f *has* predicative type

$$\Phi_q \xleftarrow{f} \Phi_p \qquad (49)$$

wherever

$$f \cdot \Phi_p \ \subseteq \ \Phi_q \cdot f \qquad (50)$$

holds, cf. diagram

$$\begin{tikzcd}
A \arrow[d, "f"'] & & A \arrow[ll, "\Phi_p"'] \arrow[d, "f"] \\
\arrow[rr, phantom, "\subseteq"] & & \\
B & & B \arrow[ll, "\Phi_q"]
\end{tikzcd}$$

Condition (50) — which is equivalent to

$$f \cdot \Phi_p \ \subseteq \ \Phi_q \cdot \top \qquad (51)$$

as we have seen above — is the PF-transform of ESC proof obligation $\langle \forall\ x\ :\ p\ x\ :\ q\ (f\ x)\rangle$ *stating that function* f *ensures property* q *on the output once property* p *is granted on the input.*
□

Stating that a given function is of a particular predicative type is an assertion which needs to be checked. Predicative types obey to a number of interesting and useful properties which can be proved using the PF-calculus alone. Such properties, together with the relational calculus itself, make proof obligation discharge more structured and easier, as we shall soon see. Prior to this, we need to present a little more of the relational calculus itself.

9 More about the relational calculus

Coreflexives. Recall from section 3 that unary predicates PF-transform to fragments of id (coreflexives) as captured by the following universal property:

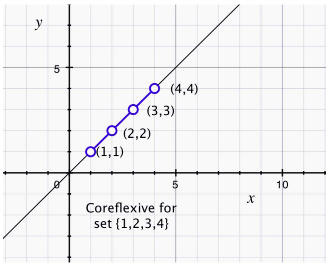
$$\Psi = \Phi_p \quad \Leftrightarrow \quad \langle \forall\ y, x\ ::\ y\, \Psi\, x \Leftrightarrow y = x\ \wedge\ p\ y\rangle \tag{52}$$

Via cancellation $\Psi := \Phi_p$, (52) yields

$$y\, \Phi_p\, x\ \Leftrightarrow\ y = x\ \wedge\ p\ y \tag{53}$$

A set S can also be PF-transformed into a coreflexive by calculating $\Phi_{(\in S)}$, cf. eg. the following graphic display of the transform of set $\{1, 2, 3, 4\}$:

$$\Phi_{(\in\{1,2,3,4\})} \quad =$$



Thanks to the isomorphism between predicates and coreflexives implicit in (52), it is easy to show that predicate algebra can be expressed in terms of coreflexives alone [7]. In particular, given predicates p, q, we have

$$\Phi_{p\ \wedge\ q} = \Phi_p \cdot \Phi_q \tag{54}$$

$$\Phi_{p \vee q} = \Phi_p \cup \Phi_q \tag{55}$$

$$\Phi_{\neg p} = id - \Phi_p \tag{56}$$

$$\Phi_{false} = \perp \tag{57}$$

$$\Phi_{true} = id \tag{58}$$

where $false$ (resp. $true$) denote the everywhere FALSE (resp. everywhere TRUE) predicates and $R - S$ denotes relational difference (19).

Coreflexives are symmetric and transitive relations, that is,

$$\Phi^\circ = \Phi = \Phi \cdot \Phi \tag{59}$$

hold for Φ coreflexive. The fact that meet of coreflexives is composition

$$\Phi \cap \Psi = \Phi \cdot \Psi \tag{60}$$

is of great calculational advantage since it enables one to pipeline restrictions (or conditions) while taking advantage of the central role played by composition in the whole calculus.

Exercise 5. Given a function $B \xleftarrow{f} A$, show that $\mathsf{img}\, f$ is the coreflexive Φ_p of predicate $p\, b \triangleq \langle \exists\, a \ ::\ b = f\ a \rangle$.
□

Domain and range. The coreflexive fragments of kernel and image are named *domain* (δ) and *range* (ρ)

$$\delta\, R \ \triangleq\ \mathsf{ker}\, R \cap id \tag{61}$$

$$\rho\, R \ \triangleq\ \mathsf{img}\, R \cap id \tag{62}$$

Therefore:

$$R \cdot \delta\, R \ =\ R \ =\ \rho\, R \cdot R \tag{63}$$

Clearly:

$$\delta\, R = \mathsf{ker}\, R \Leftrightarrow R \text{ is injective} \tag{64}$$

$$\rho\, R = \mathsf{img}\, R \Leftrightarrow R \text{ is simple} \tag{65}$$

$$\delta\, R = R = \rho\, R \Leftrightarrow R \text{ is coreflexive} \tag{66}$$

In particular, range and image of functions coincide.

From the definitions above we easily recover their pointwise equivalents. For instance, let us calculate $\rho\, twice$:

$$\begin{array}{ll}
& y(\rho\, twice)x \\
\Leftrightarrow & \quad \{ \text{ (65) } \} \\
& y(twice \cdot twice^\circ)x \\
\Leftrightarrow & \quad \{ \text{ exercise 5 ; coreflexives } \} \\
& y = x \ \wedge\ \langle \exists\, k \ ::\ y = twice\ k \rangle \\
\Leftrightarrow & \quad \{ \text{ definition of } twice \ \} \\
& y = x \ \wedge\ \langle \exists\, k \ ::\ y = 2k \rangle \\
\Leftrightarrow & \quad \{ \text{ (53) ; definition of } even \text{ (43) } \} \\
& y\ \Phi_{even}\ x
\end{array}$$

So, the range of $twice$ is the same relation as Φ_{even}. Taking advantage of this, we check predicative type assertion (48) of the previous section:

$$
\begin{array}{ll}
& \Phi_{even} \xleftarrow{twice} \Phi_{even} \\
\Leftrightarrow & \qquad \{ \text{ (49,50) } \} \\
& twice \cdot \Phi_{even} \subseteq \Phi_{even} \cdot twice \\
\Leftrightarrow & \qquad \{ \ \Phi_{even} = \rho\, twice \text{ (above) } \} \\
& twice \cdot \Phi_{even} \subseteq \rho\, twice \cdot twice \\
\Leftrightarrow & \qquad \{ \text{ (63) } \} \\
& twice \cdot \Phi_{even} \subseteq twice \\
\Leftarrow & \qquad \{ \text{ composition is monotonic } \} \\
& \Phi_{even} \subseteq id \\
\Leftrightarrow & \qquad \{ \ \Phi_{even} \text{ is coreflexive } \} \\
& \text{TRUE}
\end{array}
$$

This first ESC/PF exercise gives an idea of the flavour of discharging proof obligations by calculation. The example is very simple and so the distance between this and the equivalent pointwise proof stemming directly from (44) is not much. Non-trivial examples to be given later in sections 13 and 18 will provide a better idea of the advantages of doing things in the pointfree style.

It should be noted that the closed formulæ given above (61,62) do not provide the best way to infer properties such as the above. It is much simpler to rely on universal properties which domain and range enjoy and which are (once again) Galois connections, as explained below.

Structuring the calculus. As anticipated in section 3, Galois connections provide a convenient way to structure the relational calculus in the sense that they offer (universal) properties which implicitly capture the meaning of the two relational combinators (termed *adjoints*) involved in each connection [11].

A paradigmatic example is that of capturing the meaning of functions: it can be shown that functions are *exactly* those relations h which obey the following Galois connection, for all other (suitably typed) relations R, S:

$$h \cdot R \subseteq S \Leftrightarrow R \subseteq h^\circ \cdot S \qquad (67)$$

Taking converses, this is equivalent to [12]

$$R \cdot h^\circ \subseteq S \Leftrightarrow R \subseteq S \cdot h \qquad (68)$$

[11] This approach to the relational calculus was pioneered in the 1990s by the Mathematics of Program Construction (MPC) school, see eg. references [1, 29, 20, 62, 7].

[12] These Galois connections are often referred to as *shunting rules* [13].

Again we stress on the resemblance with school algebra: like number n in (20), function h in (67,68) can be shifted back and forth in relational expressions by "swapping sign" (which in the relational context means taking converses).

The fact that *at most* and equality coincide in the case of functions

$$f \subseteq g \;\Leftrightarrow\; f = g \;\Leftrightarrow\; f \supseteq g \tag{69}$$

is among several other beneficial consequences of these rules (see eg. [13]).

Exercise 6. Use the shunting rules (67,68) to show that $\underline{c} \cdot R$ is always simple and $S \cdot \underline{c}^{\circ}$ is always injective, for all suitably typed R, S.
□

Domain and range are characterized by Galois connections

$$\delta\, R \subseteq \Phi \Leftrightarrow R \subseteq \top \cdot \Phi \tag{70}$$

$$\rho\, R \subseteq \Phi \Leftrightarrow R \subseteq \Phi \cdot \top \tag{71}$$

where Φ ranges over coreflexives, from which a number of properties arise, namely:

$$\top \cdot \delta\, R \;=\; \top \cdot R \tag{72}$$

$$\rho\, R \cdot \top \;=\; R \cdot \top \tag{73}$$

$$\Phi \subseteq \Psi \Leftrightarrow \Phi \subseteq \top \cdot \Psi \tag{74}$$

$$\delta\, R \subseteq \delta\, S \Leftrightarrow R \subseteq \top \cdot S \tag{75}$$

$$\delta\,(R \cdot S) \;=\; \delta\,(\delta\, R \cdot S) \tag{76}$$

$$\rho\,(R \cdot S) \;=\; \rho\,(R \cdot \rho\, S) \tag{77}$$

In general, all such Galois connections instantiate the equivalence at the top of table 2. It should be mentioned that some rules in this table appear in the literature under different guises and usually not identified as GCs [13]. For a thorough presentation of the relational calculus in terms of GCs see [1, 7]. There are *many* advantages in such an approach: further to the systematic tabulation of operators (of which table 2 is just a sample), GCs have a rich algebra of properties, namely:

- both adjoints f and g in a GC are monotonic;
- lower adjoint f commutes with join and upper-adjoint g commutes with meet;
- two cancellation laws hold, $R \subseteq g(f\ R)$ and $f(g\ S) \subseteq S$, respectively known as *left-cancellation* and *right-cancellation*.

In summary, all relational combinators involved in table 2 are monotonic. The ones in the f-column distribute over $\cup$, eg.

$$(R \cup S)^{\circ} = R^{\circ} \cup S^{\circ} \tag{78}$$

$$f \cdot (R \cup S) = f \cdot R \cup f \cdot S \tag{79}$$

and the ones in the g-column distribute over $\cap$, eg.:

$$(R \cap S)^{\circ} = R^{\circ} \cap S^{\circ} \tag{80}$$

$$(R \cap S) \cdot f = R \cdot f \cap S \cdot f \tag{81}$$

[13] For instance, *shunting* rule (67) is called *cancellation law* in [66].

Table 2. *Tabulation of Galois connections in the relational calculus (sample). The general formula given on top is a logical equivalence universally quantified on S and R. It has a left part involving lower adjoint f and a right part involving upper adjoint g. These are expressed using sections of binary operators. So, each line in the table corresponds in fact to a* family *of adjoints indexed by the argument frozen in each section, eg. h in $(h\cdot)$, $(h^\circ\cdot)$ in the line marked* shunting rule.

$$(f\ R) \subseteq S \Leftrightarrow R \subseteq (g\ S)$$

Description	f	g	**Comment**
converse	$(\_)^\circ$	$(\_)^\circ$	
shunting rule	$(h\cdot)$	$(h^\circ\cdot)$	h is a function
“converse” *shunting* rule	$(\cdot h^\circ)$	$(\cdot h)$	h is a function
difference	$(\_ - R)$	$(R \cup\ )$	
Left-division	$(R\cdot)$	$(R \setminus\ )$	read “R under ...”
Right-division	$(\cdot R)$	$(\ / R)$	read “... over R”
domain	δ	$(\top\cdot)$	left $\subseteq$ restricted to coreflexives
range	ρ	$(\cdot\top)$	left $\subseteq$ restricted to coreflexives

Simplicity. Simple relations (also known as partial functions) will be particularly relevant in the sequel because of their ubiquity in software modeling. In particular, they can be used to model data structures “embodying a functional dependency” such as eg. mappings from object identifiers to object attribute values [47, 48].

In the same way simple relations generalize functions as shown in figure 1, *shunting* rules (67, 68) generalize to

$$S \cdot R \subseteq T \Leftrightarrow (\delta\, S) \cdot R \subseteq S^\circ \cdot T \tag{82}$$

$$R \cdot S^\circ \subseteq T \Leftrightarrow R \cdot \delta\, S \subseteq T \cdot S \tag{83}$$

for S simple. In the case of coreflexives (which are special cases of simple relations), rules (82,83) instantiate to

$$\Phi \cdot R \subseteq S \Leftrightarrow \Phi \cdot R \subseteq \Phi \cdot S \tag{84}$$

$$R \cdot \Phi \subseteq S \Leftrightarrow R \cdot \Phi \subseteq S \cdot \Phi \tag{85}$$

Harpoon arrows $B \overset{R}{\leftharpoondown} A$ or $A \overset{R}{\rightharpoondown} B$ in diagrams indicate that R is simple. Later on we will need to describe simple relations at pointwise level. The notation we shall adopt for this purpose is borrowed from VDM [32], where it is known as *mapping comprehension*. This notation exploits the applicative nature of a simple relation S by writing $b\ S\ a$ as

$$a \in dom\ S\ \wedge\ b = S\ a \tag{86}$$

where $\wedge$ should be understood non-strict on the right argument [14] and $dom\ S$ is the set-theoretic version of coreflexive $\delta\, S$, that is,

$$\delta\, S = \Phi_{(dom\ S)} \tag{87}$$

holds (cf. the isomorphism between sets and coreflexives). In this way, relation S itself can be written as $\{a \mapsto S\ a \mid a \in dom\ S\}$ and projection $f \cdot S \cdot g^{\circ}$ as

$$\{g\ a \mapsto f(S\ a) \mid a \in dom\ S\} \tag{88}$$

provided S satisfies functional dependency $g \rightarrow f$, to ensure simplicity (see exercise 8).

Exercise 7. Further to exercise 2 show that condition

$$M \cdot N^{\circ} \subseteq id \tag{89}$$

(which ensures that the union of two simple relations M and N is simple) converts to pointwise notation as follows,

$$\langle \forall\ a\ :\ a \in (dom\ M \cap dom\ N) :\ (M\ a) = (N\ a)\rangle$$

— a condition known as (map) *compatibility* in the VDM terminology [22].
□

Exercise 8. A relation S is said to satisfy functional dependency $g \rightarrow f$ wherever projection $f \cdot S \cdot g^{\circ}$ is simple, that is, iff

$$\mathsf{ker}\,(g \cdot S^{\circ}) \subseteq \mathsf{ker}\, f \tag{90}$$

holds [45].

1. Show that (90) trivially holds wherever g is injective and S is simple, for all (suitably typed) f.
2. Resort to (86), (90) and to the rules of both the PF-transform and the Eindhoven quantifier calculus (appendix A) to show that the healthiness condition (90) imposed on mapping comprehension (88) is equivalent to

$$\langle \forall\ a, b\ :\ a, b \in dom\ S\ \wedge\ (g\ a) = (g\ b) :\ f(S\ a) = f(S\ b)\rangle$$

□

10 Building up the ESC/PF calculus

What we have seen so far about the PF relational calculus is enough to start developing our own calculus of ESC predicative type assertions, stemming from definition 1. Let us see, for instance, what happens wherever the input predicate in (49) is a disjunction:

$$\Phi_q \xleftarrow{\ f\ } \Phi_{p_1} \cup \Phi_{p_2}$$

$$\Leftrightarrow \qquad \{\ (50)\ \}$$

[14] VDM embodies a logic of partial functions (LPF) which takes this into account [32].

$$
\begin{array}{ll}
& f \cdot (\Phi_{p_1} \cup \Phi_{p_2}) \subseteq \Phi_q \cdot f \\
\Leftrightarrow & \quad \{ \text{ distribution (79) } \} \\
& f \cdot \Phi_{p_1} \;\cup\; f \cdot \Phi_{p_2} \;\subseteq\; \Phi_q \cdot f \\
\Leftrightarrow & \quad \{ \text{ $\cup$-universal (22) } \} \\
& f \cdot \Phi_{p_1} \subseteq \Phi_q \cdot f \;\;\wedge\;\; f \cdot \Phi_{p_2} \subseteq \Phi_q \cdot f \\
\Leftrightarrow & \quad \{ \text{ (50) twice } \} \\
& \Phi_q \xleftarrow{\;f\;} \Phi_{p_1} \;\wedge\; \Phi_q \xleftarrow{\;f\;} \Phi_{p_2}
\end{array}
$$

Thus distributive law

$$\Phi_q \xleftarrow{\;f\;} \Phi_{p_1} \cup \Phi_{p_2} \;\;\Leftrightarrow\;\; \Phi_q \xleftarrow{\;f\;} \Phi_{p_1} \;\wedge\; \Phi_q \xleftarrow{\;f\;} \Phi_{p_2} \tag{91}$$

holds. The dual rule,

$$\Phi_{q_1} \cdot \Phi_{q_2} \xleftarrow{\;f\;} \Phi_p \;\;\Leftrightarrow\;\; \Phi_{q_1} \xleftarrow{\;f\;} \Phi_p \;\wedge\; \Phi_{q_2} \xleftarrow{\;f\;} \Phi_p \tag{92}$$

is calculated in the same way.

The fact that predicative arrows compose,

$$\Psi \xleftarrow{\;g \cdot h\;} \Phi \Leftarrow \Psi \xleftarrow{\;g\;} \Upsilon \;\wedge\; \Upsilon \xleftarrow{\;h\;} \Phi \tag{93}$$

follows straight from (49, 50), as does the obvious rule concerning identity

$$\Psi \xleftarrow{\;id\;} \Phi \;\;\Leftrightarrow\;\; \Phi \subseteq \Psi \tag{94}$$

whereby $\Phi \xleftarrow{\;id\;} \Phi$ always holds. Thus it makes sense to draw predicative diagrams such as, for instance,

$$\begin{tikzcd}
\Psi & \Psi \times \Upsilon \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & \Upsilon \\
& \Phi \arrow[ul, "f"] \arrow[u, "\langle f,g \rangle"] \arrow[ur, "g"'] &
\end{tikzcd} \qquad (95)$$

where predicates (coreflexives) are promoted to objects (nodes in diagrams). In this case, the diagram explains the ESC behaviour of the combinator which pairs the results of two functions,

$$\langle f, g \rangle c \;\triangleq\; (f\ c, g\ c) \tag{96}$$

recall table 1. In the literature, this is often referred to as the *split* or *fork* combinator. The two projections π_1, π_2 are such that

$$\pi_1(a,b) = a \;\wedge\; \pi_2(a,b) = b \tag{97}$$

and $\Psi \times \Upsilon$ instantiates relational product $R \times S$ of table 1. The diagram expresses the two cancellation properties

$$\pi_1 \cdot \langle f, g\rangle = f \quad \wedge \quad \pi_2 \cdot \langle f, g\rangle = g \tag{98}$$

The question is: is diagram (95) properly typed?

We defer to section 11 the discussion about the arrows labelled with π_1, π_2 (which are instances of a more general result) and focus on arrow $\langle f, g\rangle$. We need to recall the universal property of relational *splits*

$$X \subseteq \langle R, S\rangle \;\;\Leftrightarrow\;\; \pi_1 \cdot X \subseteq R \,\wedge\, \pi_2 \cdot X \subseteq S \tag{99}$$

(another Galois connection, see exercise 9) and that $\times$-absorption holds [13]:

$$\langle R \cdot T, S \cdot U\rangle = (R \times S) \cdot \langle T, U\rangle \tag{100}$$

Then we reason:

$$\begin{array}{cl}
 & \Psi \times \Upsilon \xleftarrow{\langle f,g\rangle} \Phi \\
\Leftrightarrow & \qquad \{ \text{ (49,50) } \} \\
 & \langle f, g\rangle \cdot \Phi \subseteq (\Psi \times \Upsilon) \cdot \langle f, g\rangle \\
\Leftrightarrow & \qquad \{ \text{ absorption (100) } \} \\
 & \langle f, g\rangle \cdot \Phi \subseteq \langle \Psi \cdot f, \Upsilon \cdot g\rangle \\
\Leftrightarrow & \qquad \{ \text{ universal property (99) } \} \\
 & \pi_1 \cdot \langle f, g\rangle \cdot \Phi \subseteq \Psi \cdot f \quad \wedge \quad \pi_2 \cdot \langle f, g\rangle \cdot \Phi \subseteq \Upsilon \cdot g \\
\Leftrightarrow & \qquad \{ \text{ cancellations (98) } \} \\
 & f \cdot \Phi \subseteq \Psi \cdot f \quad \wedge \quad g \cdot \Phi \subseteq \Upsilon \cdot g \\
\Leftrightarrow & \qquad \{ \text{ (49,50) twice } \} \\
 & \Psi \xleftarrow{f} \Phi \quad \wedge \quad \Upsilon \xleftarrow{g} \Phi
\end{array}$$

In summary, we have calculated ESC/PF rule

$$\Psi \times \Upsilon \xleftarrow{\langle f,g\rangle} \Phi \quad \Leftrightarrow \quad \Psi \xleftarrow{f} \Phi \,\wedge\, \Upsilon \xleftarrow{g} \Phi \tag{101}$$

which justifies the existence of arrow $\langle f, g\rangle$ in diagram (95).

Exercise 9. Show that

$$\langle R, S\rangle = (\pi_1^\circ \cdot R) \cap (\pi_2^\circ \cdot S) \tag{102}$$

is the PF-transform of the clause given for this combinator in table 1. Furthermore infer (99) from (102) and universal property (21).

□

Let us finally see how to handle conditional expressions of the form $if\ (c\ x)\ then\ (f\ x)\ else\ (g\ x)$, which PF-transform into the following version of McCarthy's conditional combinator:

$$c \rightarrow f\ , g\ =\ f \cdot \Phi_c \cup g \cdot \Phi_{\neg c} \tag{103}$$

In this case, (51) offers a better standpoint for calculation than (50), as the reader may check in calculating the following rule for conditionals:

$$\Phi_q \xleftarrow{c \rightarrow f,g} \Phi_p \quad \Leftrightarrow \quad \Phi_q \xleftarrow{f} \Phi_p \cdot \Phi_c\ \wedge\ \Phi_q \xleftarrow{g} \Phi_p \cdot \Phi_{\neg c} \tag{104}$$

Further ESC/PF rules can calculated on the same basis, either elaborating on the predicate structure or on the combinator structure. However, all the cases above involve functions only and the semantics of computations are, in general, relations. So our strategy is to generalize definition 1 to relations and develop the calculus on such a generic basis. Before this, let us present a generic result which still has to do with functions and is of great interest to type checking.

11 ESC "for free"

In his well-known paper *Theorems for free!* [64], Philip Wadler writes:

> *From the type of a polymorphic function we can derive a theorem that it satisfies.* (...) *How useful are the theorems so generated? Only time and experience will tell.*

The generosity of this quotation stems from John Reynolds *abstraction theorem* on parametric polymorphism [54] of which several applications have been found in the meantime, namely in program transformation [59], abstract interpretation and safety analysis [3], relation calculus [49], program correctness [63], etc. [15]

In this section we identify a class of ESC/PF rules which are corollaries of this theorem and which, as such, do not need to be discharged. We follow the pointfree styled presentation of this theorem given in [3], which is remarkably elegant: let f be a polymorphic *function* $f : t$, whose type t can be written according to the following <<grammar>> of types:

$t := t' \leftarrow t''$

$t := \mathsf{F}(t_1, \ldots, t_n)$ for n-ary parametric type F

$t := v$ for v a type variable (= polymorphism <<dimension>>)

Let V be the set of type variables involved in type t, $\{R_v\}_{v \in V}$ be a V-indexed family of relations (f_v in case all such R_v are functions) and R_t be a relation defined inductively as follows:

$$R_{t:=t' \leftarrow t''} = R_{t'} \leftarrow R_{t''} \tag{105}$$

$$R_{t:=\mathsf{F}(t_1,\ldots,t_n)} = \mathsf{F}(R_{t_1}, \ldots, R_{t_n}) \tag{106}$$

$$R_{t:=v} = R_v \tag{107}$$

[15] For the automatic generation of free theorems (in Haskell syntax) see Janis Voigtlaender's home page: `http://linux.tcs.inf.tu-dresden.de/˜voigt/ft` .

where $R_{t'} \leftarrow R_{t''}$ is defined by (37) and symbol F is overloaded in (106): it denotes a parametric type on the left hand side and the n-ary *relator* [35, 8] which captures its semantics on the right hand side. (More details to follow.)

The *free theorem of type* t then reads as follows: *given any function* $f : t$ *and* V *as above,* $f\ R_t\ f$ *holds for any relational instantiation of type variables in* V. Note that this theorem is a result about t and holds for *any* polymorphic function of type t *independently* of its actual definition [16].

$$\begin{tikzcd}
A \arrow[r, dash, dotted] \arrow[d, "R"'] & \mathsf{F}\,A \arrow[d, "\mathsf{F}\,R"] \\
B \arrow[r, dash, dotted] & \mathsf{F}\,B
\end{tikzcd}$$

Before proceeding to the application of this theorem, we need to explain the meaning of F in (106). Technically, the parametricity of F is captured by regarding it as a *relator* [8], a concept which extends *functors* to relations: $\mathsf{F}\,R$ is a relation from $\mathsf{F}\,A$ to $\mathsf{F}\,B$ wherever R is a relation from A to B (see diagram aside).

By definition, relators are monotonic

$$R \subseteq S \Rightarrow \mathsf{F}\,R \subseteq \mathsf{F}\,S \tag{108}$$

and commute with composition, converse and the identity:

$$\mathsf{F}\,(R \cdot S) = (\mathsf{F}\,R) \cdot (\mathsf{F}\,S) \tag{109}$$

$$\mathsf{F}\,(R^{\circ}) = (\mathsf{F}\,R)^{\circ} \tag{110}$$

$$\mathsf{F}\,id = id \tag{111}$$

The most simple relators are the *identity* relator Id, which is such that $\mathsf{Id}\ A = A$ and $\mathsf{Id}\ R = R$, and the *constant* relator K which, for a particular concrete data type K, is such that $\mathsf{K}\ A = K$ and $\mathsf{K}\ R = id_K$.

Relators can also be multi-parametric. Two well-known examples of binary relators are product and sum,

$$R \times S = \langle R \cdot \pi_1, S \cdot \pi_2 \rangle \tag{112}$$

$$R + S = [i_1 \cdot R\ , i_2 \cdot S] \tag{113}$$

where π_1, π_2 are the projections of a Cartesian product (97), i_1, i_2 are the injections of a disjoint union, and the *split/either* relational combinators are defined by (102) and

$$[R\ , S] = (R \cdot i_1^{\circ}) \cup (S \cdot i_2^{\circ}) \tag{114}$$

respectively. By putting product, sum, identity and constant relators together with fixpoint definitions one is able to specify a large class of parametric structures — referred to as *polynomial* — such as those implementable in Haskell, for instance.

Let us see how the free theorem of projections π_1 and π_2 justifies (for free) arrows $\Psi \xleftarrow{\pi_1} \Psi \times \Upsilon$ and $\Psi \times \Upsilon \xrightarrow{\pi_2} \Upsilon$ in diagram (95). The polymorphic type of π_1 being $t = a \leftarrow a \times b$, one has $R_t = R_a \leftarrow R_a \times R_b$. We reason:

$$\pi_1 (R_t) \pi_1$$

[16] See [3] for comprehensive evidence on the power of this theorem when combined with Galois connections.

$$
\begin{array}{ll}
\Leftrightarrow & \quad \{ \text{ abbreviating } R_a, R_b := R, S \ \} \\
& \pi_1(R \leftarrow R \times S)\pi_1 \\
\Leftrightarrow & \quad \{ \ (37) \ \} \\
& \pi_1 \cdot (R \times S) \subseteq R \cdot \pi_1 \\
\Leftrightarrow & \quad \{ \ (38) \ \} \\
& R \xleftarrow{\pi_1} (R \times S)
\end{array}
$$

Thus, $R \xleftarrow{\pi_1} R \times S$ holds for *all* (suitably typed) R, S, thus covering coreflexives Ψ and Υ as special cases. (The calculation of $R \times S \xrightarrow{\pi_2} S$ is identical.)

The free theorem of a polymorphic type conveys the idea that types too "are relations". Its wide scope is better appreciated once dealing with higher-order combinators. Let us see the case of functional composition $(\cdot)$, which is of type $t = (c \leftarrow a) \leftarrow (b \leftarrow a) \leftarrow (c \leftarrow b)$:

$$
\begin{array}{ll}
& (\cdot) R_t (\cdot) \\
\Leftrightarrow & \quad \{ \ (109) \text{ to } (111) \ \} \\
& (\cdot)((R_c \leftarrow R_a) \leftarrow (R_b \leftarrow R_a) \leftarrow (R_c \leftarrow R_b))(\cdot) \\
\Leftrightarrow & \quad \{ \text{ introducing abbreviations such as in the previous calculation } \} \\
& (\cdot)((U \leftarrow R) \leftarrow (S \leftarrow R) \leftarrow (U \leftarrow S))(\cdot) \\
\Leftrightarrow & \quad \{ \ (37) \ \} \\
& (\cdot) \cdot (U \leftarrow S) \subseteq ((U \leftarrow R) \leftarrow (S \leftarrow R)) \cdot (\cdot) \\
\Leftrightarrow & \quad \{ \ (67) \ \} \\
& (U \leftarrow S) \subseteq (\cdot)^\circ \cdot ((U \leftarrow R) \leftarrow (S \leftarrow R)) \cdot (\cdot) \\
\Leftrightarrow & \quad \{ \ (13) \text{ assuming } \forall\text{-quantification implicit} ; (27) \ \} \\
& f(U \leftarrow S)g \Rightarrow (f\cdot)((U \leftarrow R) \leftarrow (S \leftarrow R))(g\cdot) \\
\Leftrightarrow & \quad \{ \ (37) \text{ twice } \} \\
& f \cdot S \subseteq U \cdot g \Rightarrow (f\cdot) \cdot (S \leftarrow R) \subseteq (U \leftarrow R) \cdot (g\cdot) \\
\Leftrightarrow & \quad \{ \ (67) \text{ again } \} \\
& f \cdot S \subseteq U \cdot g \Rightarrow (S \leftarrow R) \subseteq (f\cdot)^\circ \cdot (U \leftarrow R) \cdot (g\cdot) \\
\Leftrightarrow & \quad \{ \ (13) ; (27) \text{ again } \} \\
& f \cdot S \subseteq U \cdot g \Rightarrow h(S \leftarrow R)j \Rightarrow (f \cdot h)(U \leftarrow R)(g \cdot j) \\
\Leftrightarrow & \quad \{ \ (37) \ \} \\
& f \cdot S \subseteq U \cdot g \ \wedge \ h \cdot R \subseteq S \cdot j \Rightarrow f \cdot h \cdot R \subseteq U \cdot g \cdot j
\end{array}
$$

Substituting $f, j, R, S, U := g, h, \Phi, \Upsilon, \Psi$ we obtain

$$g \cdot \Upsilon \subseteq \Psi \cdot g \ \wedge \ h \cdot \Phi \subseteq \Upsilon \cdot h \ \Rightarrow \ g \cdot h \cdot \Phi \subseteq \Psi \cdot g \cdot h$$

which is nothing but composition rule (93) already presented. So, (93) is an example of "ESC for free", as is the $\Rightarrow$ part of equivalence (101) [17].

Reference [12] elaborates on these corollaries of the free theorem of functional combinators in building a category Pred of "predicates as objects" proposed as a suitable universe for describing coalgebraic systems subject to invariants. Pred's objects are predicates, represented by coreflexives. An arrow $\Psi \xleftarrow{f} \Phi$ in Pred means a function which ensures property Ψ on its output whenever property Φ holds on its input. Arrows in Pred can therefore be seen as ESC proof-obligations concerning the functions involved.

Exercise 10. From the free theorem of $1 \xleftarrow{!} A$ and fact $\mathsf{ker}\,! = \top$ infer

$$f \cdot R \subseteq \top \cdot S \Leftrightarrow R \subseteq \top \cdot S \tag{115}$$

□

12 Calculating pre-conditions for ESC

Wherever a function f does not ensure preservation of a given invariant inv, that is, $\Phi_{inv} \xleftarrow{f} \Phi_{inv}$ does not hold, there is always a pre-condition pre which enforces this at the cost of *partializing* f. In the limit, pre is the everywhere false predicate. Programmers often become aware of the need for such pre-conditions at runtime, in the testing phase. One can do better and find it much earlier, at specification (modeling) time, when trying to discharge the standard proof obligation

$$\langle \forall\, a \ : \ inv\, a : \ inv(f\ a) \rangle \tag{116}$$

which then extends to

$$\langle \forall\, a \ : \ inv\, a \ \wedge \ pre\, a : \ inv(f\ a) \rangle \tag{117}$$

Bound to *invent* pre, one will hope to guess the *weakest* such pre-condition. Otherwise, future use of f will be spuriously constrained. However, how can one be sure of having hit such weakest pre-condition?

As it will be explained below, predicate $inv(f\ a)$ in (117) is itself the weakest pre-condition for inv to hold upon execution of f. In our ESC/PF approach we will proceed as follows: we take the PF-transform of $inv(f\ a)$ — at data level — as starting point and attempt to rewrite it into the conjunction of predicate $inv\ a$ (or weaker) and possibly "something else" — the *calculated* pre-condition pre. So we strengthen (117) to equivalence

$$inv\ a \ \wedge \ pre\ a \Leftrightarrow inv(f\ a) \tag{118}$$

thus meaning that pre will be not only sufficient but also necessary for inv to be maintained by f. This method works provided all calculation steps are equivalences. Let us start by detailing this strategy.

[17] See eg. [49] for the derivation of the free theorem of the functional *split* combinator.

Weakest pre-conditions. Back to definition 1, let us transform (49) according to the PF-calculus studied so far:

$$\begin{array}{ll}
& \Phi_q \xleftarrow{f} \Phi_p \\
\Leftrightarrow & \quad \{ \text{ (51) } \} \\
& f \cdot \Phi_p \subseteq \Phi_q \cdot \top \\
\Leftrightarrow & \quad \{ \text{ (71) } \} \\
& \rho\,(f \cdot \Phi_p) \subseteq \Phi_q
\end{array}$$

On the other hand,

$$\begin{array}{ll}
& f \cdot \Phi_p \subseteq \Phi_q \cdot \top \\
\Leftrightarrow & \quad \{ \text{ (67) } \} \\
& \Phi_p \subseteq f^\circ \cdot \Phi_q \cdot \top
\end{array}$$

Putting everything together, we obtain GC

$$\rho\,(f \cdot \Phi_p) \subseteq \Phi_q \quad \Leftrightarrow \quad \Phi_p \subseteq f^\circ \cdot \Phi_q \cdot \top \tag{119}$$

which is the expected composition of GCs (71) and (67). The left hand side of (119) tells that Φ_p is *sufficient* as a pre-condition for f to ensure Φ_q on the output. Its right hand side tells that $f^\circ \cdot \Phi_q \cdot \top$ is the largest (weakest) such condition [18]. In other words, $f^\circ \cdot \Phi_q \cdot \top$ is *necessary* for q to hold on f's output.

Weakest pre-conditions have been studied extensively in the literature, both in the pointwise and pointfree style [19, 6]. As we shall soon see, they have a calculus of their own which is closely related to that of predicative types. This connection between the two calculi will be given in section 15, where it will be presented in its full generality, that is, concerning relations in general instead of functions.

Let us, for the moment, refrain from going into such foundational work and see two examples of ESC ensured by weakest pre-condition PF-calculation.

13 ESC/PF calculus at work — case study 1

Recalling the mobile phone case study of section 2, we want to ensure that $store$ maintains invariant $inv = noDuplicates \wedge leq10$, that is, to check $\Phi_{inv} \xleftarrow{store} \Phi_{inv}$. Thanks to ESC/PF rule (92), we know we can split this into $\Phi_{noDuplicates} \xleftarrow{store} \Phi_{inv} \;\wedge\; \Phi_{leq10} \xleftarrow{store} \Phi_{inv}$. We address the first of these arrows in this section.

Thanks to the pipelined structure of $store$ (7), we can split the problem in two. First we address $\Phi_{noDuplicates} \xleftarrow{(c:)} \Phi_{inv}$ and then we promote this result to $store$.

When compared to the definition of injective function (31), the pointwise definition of $noDuplicates$ (4) is suggestive of what needs to be done towards a calculational,

[18] Back to points, this is predicate $\langle \lambda\, x \;::\; q(f\; x)\rangle$.

PF-argument: a list has no duplicates if and only if, regarded as a (partial) function from indices to elements, it is injective. Thus we can represent a list l of elements in C by a *simple* relation in $\mathbb{N} \rightharpoonup C$ telling which elements take which positions in list, and define

$$noDuplicates\ L \quad \triangleq \quad L^\circ \cdot L \subseteq id \tag{120}$$

In this context, appending c at the front of list L becomes relational operator

$$c : L \quad \triangleq \quad \underline{c} \cdot \underline{1}^\circ \cup L \cdot succ^\circ \tag{121}$$

where $succ\ n \triangleq n+1$ is the successor function in $\mathbb{N}_0$. (Mind that L indices exclude 0.) Back to points and using mapping notation for simple relations (88), the body of (121) becomes the expected $\{1 \mapsto c\} \cup \{i+1 \mapsto (L\ i) \mid i \leftarrow dom\ L\}$.

13.1 ESC calculations for $(c\,:)$

First of all, we need to show that $(c\,:)$ preserves simplicity:

$$
\begin{array}{ll}
& c : L \ \text{ is simple} \\
\Leftrightarrow & \quad \{\ \text{(121) followed by (33)}\ \} \\
& L \cdot succ^\circ \ \text{ is simple } \wedge \ \underline{c} \cdot \underline{1}^\circ \ \text{ is simple } \wedge \ L \cdot succ^\circ \cdot \underline{1} \cdot \underline{c}^\circ \subseteq id \\
\Leftrightarrow & \quad \{\ \text{exercise 6; } succ^\circ \cdot \underline{1} = \underline{0}\ \} \\
& L \cdot succ^\circ \ \text{ is simple } \wedge \ L \cdot \underline{0} \cdot \underline{c}^\circ \subseteq id \\
\Leftrightarrow & \quad \{\ succ \text{ is an injection, thus } succ^\circ \cdot succ = id\ \} \\
& L \text{ is simple } \wedge \ L \cdot \underline{0} \cdot \underline{c}^\circ \subseteq id \\
\Leftrightarrow & \quad \{\ 0 \text{ is not in the domain of } L\ \} \\
& L \text{ is simple } \wedge \ \bot \cdot \underline{c}^\circ \subseteq id \\
\Leftrightarrow & \quad \{\ \bot \text{ is at most anything}\ \} \\
& L \text{ is simple}
\end{array}
$$

Since all steps in the calculation are equivalences, L being simple is the *weakest* pre-condition for $c : L$ being simple.

Next we calculate with $noDuplicates(c : L)$ aiming at splitting this into invariant $noDuplicates\ L$ plus "something else" — the calculated weakest pre-condition for $noDuplicates$ preservation:

$$
\begin{array}{ll}
& noDuplicates(c : L) \\
\Leftrightarrow & \quad \{\ \text{(121, 120)}\ \} \\
& \underline{c} \cdot \underline{1}^\circ \cup L \cdot succ^\circ \ \text{ is injective} \\
\Leftrightarrow & \quad \{\ \text{(32)}\ \} \\
& \underline{c} \cdot \underline{1}^\circ \text{ is injective } \ \wedge \ L \cdot succ^\circ \text{ is injective } \ \wedge \ (\underline{c} \cdot \underline{1}^\circ)^\circ \cdot L \cdot succ^\circ \subseteq id
\end{array}
$$

$$\begin{array}{ll}
\Leftrightarrow & \{ \text{ exercise 6; definition of injective ; shunting (67,68) } \} \\
& succ \cdot L^\circ \cdot L \cdot succ^\circ \subseteq id \ \wedge\ \underline{c}^\circ \cdot L \subseteq \underline{1}^\circ \cdot succ \\
\Leftrightarrow & \{ \text{ shunting again (67,68) } \} \\
& L^\circ \cdot L \subseteq succ^\circ \cdot succ \ \wedge\ \underline{c}^\circ \cdot L \subseteq \underline{1}^\circ \cdot succ \\
\Leftrightarrow & \{ \ \mathsf{ker}\, succ = id \ \} \\
& L^\circ \cdot L \subseteq id \ \wedge\ \underline{c}^\circ \cdot L \subseteq \underline{1}^\circ \cdot succ \\
\Leftrightarrow & \{ \text{ definition } \} \\
& L \text{ is injective } \wedge\ \underline{c}^\circ \cdot L \subseteq \underline{1}^\circ \cdot succ
\end{array} \qquad (122)$$

In summary, we have calculated:

$$c : L \text{ has no duplicates} \Leftrightarrow \underbrace{L \text{ is injective}}_{\text{no duplicates in } L} \ \wedge\ \underbrace{\underline{c}^\circ \cdot L \subseteq \underline{1}^\circ \cdot succ}_{wp} \qquad (123)$$

We finish the exercise by calculating the pointwise-expansion of wp:

$$\begin{array}{ll}
& \underline{c}^\circ \cdot L \subseteq \underline{1}^\circ \cdot succ \\
\Leftrightarrow & \{ \text{ go pointwise: (27) twice } \} \\
& \langle \forall\, n \ :\ c\, L\, n :\ 1 = 1 + n \rangle \\
\Leftrightarrow & \{ \text{ (86) } \} \\
& \langle \forall\, n \ :\ n \in\ dom\, L \ \wedge\ \ c = L\, n :\ 1 = 1 + n \rangle \\
\Leftrightarrow & \{ \ 1 = 1 + n \text{ always false } (n \notin dom\, L) \text{ ; } \forall\text{-trading (173) } \} \\
& \langle \forall\, n \ :\ n \in\ dom\, L :\ c = L\, n \Rightarrow \textsc{False} \rangle \\
\Leftrightarrow & \{ \text{ predicate calculus } \} \\
& \langle \forall\, n \ :\ n \in\ dom\, L :\ c \neq L\, n \rangle
\end{array}$$

We obtain the expected pre-condition preventing c from being in the list already. In summary:

$$\Phi_{noDuplicates} \xleftarrow{(c:)} \Phi_{noDuplicates\ \wedge\ wp\ c}$$

holds, for $wp\ c\ L \triangleq \langle \forall\, n \ :\ n \in\ dom\, L :\ c \neq L\, n \rangle$.

13.2 ESC calculation for $(c :) \cdot filter(c \neq)$

Next we address arrow $\Phi_{noDuplicates} \xleftarrow{store} \Phi_{inv}$. Note that, looking at (7), it is sufficient to show that $(c :) \cdot filter(c \neq)$ preserves $noDuplicates$, since $take\ n\ L$ is at most L, for all n, and *smaller than injective is injective* (exercise 1). Also note that, defined over PF-transformed lists, $filter$ becomes

$$filter\ p\ L \quad \triangleq \quad \Phi_p \cdot L \qquad (124)$$

Thus

$$filter(c \neq)L = (\neg\rho\,\underline{c}) \cdot L \tag{125}$$

where the negated range operator $(\neg\rho)$ is defined by $\neg\rho\,R \;\triangleq\; id - \rho\,R$ and satisfies property

$$\Phi \subseteq \neg\rho\,R \Leftrightarrow \Phi \cdot R \subseteq \bot \tag{126}$$

That filtering preserves simplicity follows immediately from exercise 1 (*smaller than simple is simple*). Concerning the injectivity of $c : (filter(c \neq)L)$, we reason:

$$\begin{array}{cl}
& c : (filter(c \neq)L) \text{ is injective} \\
\Leftrightarrow & \quad \{ \text{ (123) ; (125) } \} \\
& (\neg\rho\,\underline{c}) \cdot L \text{ is injective } \wedge\ \underline{c}^\circ \cdot (\neg\rho\,\underline{c}) \cdot L \subseteq \underline{1}^\circ \cdot succ \\
\Leftrightarrow & \quad \{ \text{ converses } \} \\
& (\neg\rho\,\underline{c}) \cdot L \text{ is injective } \wedge\ L^\circ \cdot (\neg\rho\,\underline{c}) \cdot \underline{c} \subseteq succ^\circ \cdot \underline{1} \\
\Leftrightarrow & \quad \{ (\neg\rho\,\underline{c}) \cdot \underline{c} = \bot \text{ by left-cancellation of (126) } \} \\
& (\neg\rho\,\underline{c}) \cdot L \text{ is injective } \wedge\ L^\circ \cdot \bot \subseteq succ^\circ \cdot \underline{1} \\
\Leftrightarrow & \quad \{ \bot \text{ is below anything } \} \\
& (\neg\rho\,\underline{c}) \cdot L \text{ is injective}
\end{array}$$

In this case, the calculated (weakest) pre-condition is even weaker than the invariant to maintain, since L injective implies $(\neg\rho\,\underline{c}) \cdot L$ injective. (*Smaller than injective is injective*, recall exercise 1 once again.) In summary, we have checked:

$$\Phi_{noDuplicates} \xleftarrow{(c:)\cdot filter(c\neq)} \Phi_{noDuplicates}$$

In retrospect, note that not having PF-transformed lists into simple relations would lead to defining $noDuplicates$ inductively on lists, in turn leading to an inductive proof. The PF-transform has, in a sense, "converted induction into deduction" (calculation).

Exercise 11. Show that (126) stems from Galois connection [1]

$$\Phi \subseteq \neg\delta\,R \Leftrightarrow R \subseteq \bot/\Phi \tag{127}$$

(among others) where $\neg\delta\,R = id - \delta\,R$.
□

14 From functional to relational ESC

Computer programs have, in general, a relational semantics, as they can be partial (eg. non-terminating) and non-deterministic. What is the impact of moving from function

$A \xrightarrow{f} B$ to relation $A \xrightarrow{R} B$ in definition 1? We reason:

$$
\begin{aligned}
& R \cdot \Phi_p \ \subseteq \ \Phi_q \cdot R \\
\Leftrightarrow \quad & \quad \{ \text{ (47, 21) } \} \\
& R \cdot \Phi_p \ \subseteq \ \Phi_q \cdot \top \\
\Leftrightarrow \quad & \quad \{ \text{ (23) } \} \\
& \Phi_p \ \subseteq \ R \setminus (\Phi_q \cdot \top) \\
\Leftrightarrow \quad & \quad \{ \text{ (13) ; (53) } \} \\
& \langle \forall\, a \ : \ p\, a : \ a(R \setminus (\Phi_q \cdot \top))a \rangle \\
\Leftrightarrow \quad & \quad \{ \text{ table 1 } \} \\
& \langle \forall\, a \ : \ p\, a : \ \langle \forall\, b \ : \ bRa : \ b(\Phi_q \cdot \top)a \rangle \rangle \\
\Leftrightarrow \quad & \quad \{ \text{ (53) ; table 1 } \} \\
& \langle \forall\, a \ : \ p\, a : \ \langle \forall\, b \ : \ bRa : \ q\, b \rangle \rangle
\end{aligned}
$$

This means that, for all inputs to R satisfying p, all outputs (if any) will satisfy q [19]. So p is sufficient for ensuring q on the output. What is the weakest such p? It is easy to repeat the reasoning which lead to (119), this time for relation R instead of function f, and for GC (23) instead of (67):

$$
\rho\,(R \cdot \Phi_p) \subseteq \Phi_q \Leftrightarrow \Phi_p \subseteq \underbrace{R \setminus (\Phi_q \cdot \top)}_{R \multimap \Phi_q} \tag{128}
$$

Notation $R \multimap \Phi$ for the weakest (liberal) pre-conditions is taken from [6]. Adjective *liberal* stresses the fact that the condition encompasses all input values for which R is undefined. Thus the definition which follows:

Definition 2 (Relational predicative types). *Let $B \xleftarrow{R} A$ be a relation and $\mathbb{B} \xleftarrow{p} A$ and $\mathbb{B} \xleftarrow{q} B$ be predicates. We shall say that R has* predicative type

$$
\Phi_q \xleftarrow{R} \Phi_p \tag{129}
$$

wherever

$$
R \cdot \Phi_p \ \subseteq \ \Phi_q \cdot R \tag{130}
$$

holds. The following are equivalent ways of stating (130):

$$
\begin{aligned}
\Phi_q \xleftarrow{R} \Phi_p \quad & \Leftrightarrow \quad R \cdot \Phi_p \ \subseteq \ \Phi_q \cdot \top && (131) \\
& \Leftrightarrow \quad \Phi_p \ \subseteq \ R \multimap \Phi_q && (132)
\end{aligned}
$$

□

[19] This will be related to the concept of *satisfiability* [32] in the sequel.

Relationship with Hoare Logic. Suppose $R = [\![P]\!]$ is the semantics of a given program P running over state space S, that is, $S \xrightarrow{R=[\![P]\!]} S$. Writing $s \overset{P}{\mapsto} s'$ for $s'Rs$, meaning that program P *may* reach state s' once executing over starting state s, fact $\Phi_q \xleftarrow{[\![P]\!]} \Phi_p$ PF-transforms to

$$\langle \forall\, s \;:\; p\, s :\; \langle \forall\, s' \;:\; s \overset{P}{\mapsto} s' :\; q\, s' \rangle\rangle \tag{133}$$

which is nothing but the meaning of Hoare triple

$$\{p\}P\{q\} \tag{134}$$

Hoare triples are thus special cases of predicative types, as suggested in the introduction by writing (2). In summary, "declaration"

$$\Psi \xleftarrow{R} \Phi \tag{135}$$

can be regarded as the type assertion that, if fed with values (or starting on states) "of type Φ" computation R yields results (or moves to states) "of type Ψ" (if it terminates). So ESC proof obligations and Hoare triples are one and the same device: a way to type computations, be them specified as (always terminating, deterministic) functions or encoded into (possibly non-terminating, non-deterministic) programs. This means that all relational ESC/PF calculation rules to follow apply to Hoare triples.

Satisfiability. Definition 2 is related to another notion of predicative typing known as *satisfiability* [32]: given R, p and q as in definition 2, R is said to be *satisfiable* with respect to (p, q) iff

$$\langle \forall\, a \;:\; p\, a :\; \langle \exists\, b \;:\; q\, b :\; bRa \rangle\rangle$$

holds, that is

$$\Phi_p \;\subseteq\; R^\circ \cdot \Phi_q \cdot R \tag{136}$$

in PF-notation. (As expected, shifting R from the left to the right hand side of (130) turns universal into existential quantification.) Usually, p and q are the invariants associated to (respectively) the input and output types of operations whose semantics are captured by pre/post-condition pairs ”à la VDM”, that is, post-conditions relating outputs to inputs, of pattern

$$\begin{array}{l} R : (b : B) \leftarrow (a : A) \\ \textbf{pre} \;\ldots a \ldots \\ \textbf{post} \;\;\ldots b \ldots a \ldots \end{array}$$

In this case, the satisfiability condition becomes, using a VDM-like syntax

$$\langle \forall\, a \;:\; a \in A :\; \text{pre-}R\, a \Rightarrow \langle \exists\, b \;:\; b \in B :\; \text{post-}R(b, a) \rangle\rangle \tag{137}$$

Clearly, (137) has to do with a particular semantic interpretation of R's non-determinism: *vagueness*. For instance, post-condition $b = 2a \vee b = a + 1$ for $A, B := Even$ (43) specifies an operation which is satisfiable but fails to maintain inv-$Even$.

The relationship between invariant preservation (130) and satisfiability (136) depends on the kind of relation R involved. For R simple, satisfiability is stronger than invariant preservation. For R entire, the former is weaker than the latter (see exercise 12 below). Therefore, both notions coincide in the case of functions.

Exercise 12. Show that

- for R entire, (130) entails (136) and, for R simple, (136) entails (130). Hint: resort to the shunting rules of simple relations.
- (136) can be written alternatively as

  $$\Phi_p \subseteq \delta\,(\Phi_q \cdot R) \tag{138}$$

  and therefore as

  $$\Phi_p \subseteq \top \cdot \Phi_q \cdot R \tag{139}$$

  Hint: resort to the properties of δ and of coreflexive relations in general (section 9).

□

15 Relational ESC/PF calculus

We are now in position to list rules of the relational ESC/PF calculus stemming from definition 2. Some of these rules actually extend those already given in section 10. In general, they help in breaking complexity of ESC/PF obligations. Note that most rules are *equivalences*, not just implications, as they tend to be written in eg. Hoare logic. So they contribute to ensuring ESC/PF predicative types *by construction*.

Relational ESC/PF rules. We begin by presenting and justifying the rule which extends relators to predicative types:

- **Relators**: rule

  $$\mathsf{F}\,\Psi \xleftarrow{\mathsf{F}\,R} \mathsf{F}\,\Phi \quad \Leftarrow \quad \Psi \xleftarrow{R} \Phi \tag{140}$$

  holds for every relator F.

This is easy to justify:

$$\begin{array}{ll}
 & \mathsf{F}\,\Psi \xleftarrow{\mathsf{F}\,R} \mathsf{F}\,\Phi \\
\Leftrightarrow & \quad \{\ (130)\ \} \\
 & \mathsf{F}\,R \cdot \mathsf{F}\,\Phi \subseteq \mathsf{F}\,\Psi \cdot \mathsf{F}\,R \\
\Leftrightarrow & \quad \{\ (109)\ \} \\
 & \mathsf{F}(R \cdot \Phi) \subseteq \mathsf{F}(\Psi \cdot R) \\
\Leftarrow & \quad \{\ (108)\ \} \\
 & R \cdot \Phi \subseteq \Psi \cdot R \\
\Leftrightarrow & \quad \{\ (130)\ \} \\
 & \Psi \xleftarrow{R} \Phi
\end{array}$$

Further to (92,94,104) and (140), the following rules hold:

- **Trivia**:

$$id \xleftarrow{R} \Phi \quad \Leftrightarrow \quad \textsc{True} \quad \Leftrightarrow \quad \Phi \xleftarrow{R} \bot \tag{141}$$

- **Trading**:

$$\Upsilon \xleftarrow{R} \Phi \cdot \Psi \quad \Leftrightarrow \quad \Upsilon \xleftarrow{R \cdot \Phi} \Psi \tag{142}$$

  As we shall see soon, (142) is useful for trading coreflexives between the input type of a given ESC arrow and the relation typed by the arrow.

- **Composition** (**Fusion**):

$$\Psi \xleftarrow{R \cdot S} \Phi \quad \Leftarrow \quad \Psi \xleftarrow{R} \Upsilon \ \wedge \ \Upsilon \xleftarrow{S} \Phi \tag{143}$$

  This rule extends (93) to relations.

- **Split by conjunction**:

$$\Psi_1 \cdot \Psi_2 \xleftarrow{R} \Phi \quad \Leftrightarrow \quad \Psi_1 \xleftarrow{R} \Phi \ \wedge \ \Psi_2 \xleftarrow{R} \Phi \tag{144}$$

  This equivalence generalizes (92).

- **Weakening/strengthening**:

$$\Psi \xleftarrow{R} \Phi \quad \Leftarrow \quad \Psi \supseteq \Theta \ \wedge \ \Theta \xleftarrow{R} \Upsilon \ \wedge \ \Upsilon \supseteq \Phi \tag{145}$$

- **Separation**:

$$\Upsilon \cdot \Theta \xleftarrow{R} \Phi \cdot \Psi \quad \Leftarrow \quad \Upsilon \xleftarrow{R} \Phi \ \wedge \ \Theta \xleftarrow{R} \Psi \tag{146}$$

  This rule follows from (144,145).

- **Splitting**:

$$\Psi \times \Upsilon \xleftarrow{\langle R,S \rangle} \Phi \quad \Leftrightarrow \quad \Psi \xleftarrow{R} \Phi \cdot \delta\, S \ \wedge \ \Upsilon \xleftarrow{S} \Phi \cdot \delta\, R \tag{147}$$

  This generalizes (101) from functions to arbitrary relations.

- **Product**:

$$\Phi' \times \Psi' \xleftarrow{R \times S} \Phi \times \Psi \quad \Leftrightarrow \quad \Phi' \xleftarrow{R} \Phi \ \wedge \ \Psi' \xleftarrow{S} \Psi \tag{148}$$

  Note that rule (140) already ensures part ⇐ of equivalence (148).

- **Conditional**: equivalence

$$\Psi \xleftarrow{\quad c \to R, S \quad} \Phi \quad \Leftrightarrow \quad \Psi \xleftarrow{R} \Phi \cdot \Phi_c \ \wedge \ \Psi \xleftarrow{S} \Phi \cdot \Phi_{\neg c} \tag{149}$$

  where

$$c \to R\,,S \ \triangleq \ R \cdot \Phi_c \cup S \cdot \Phi_{\neg c} \tag{150}$$

  generalizes (103) to relations [20].

The interested reader is welcome to provide PF-calculations for all rules listed above.

Exercise 13. The Hoare logic rule corresponding to (143) is

$$\frac{\{p\}P_1\{q\}\ ,\ \{q\}P_2\{s\}}{\{p\}P_1;P_2\{s\}}$$

for $\Phi = \Phi_p$, $\Psi = \Phi_s$, $\Upsilon = \Phi_q$, $S = [\![P_1]\!]$, $R = [\![P_2]\!]$ and $[\![P;Q]\!] = [\![Q]\!] \cdot [\![P]\!]$. Check which other Hoare logic rules correspond to which ESC/PF rules, bearing in mind that some of latter may split into two of the former because they are equivalences, not implications.

□

[20] For a wider generalization of conditionals to relations see eg. [1].

Formal correspondence with WLP calculus. Recall from (128) that the weakest (liberal) pre-condition operator $(R\mathbin{\backslash\!\!\bullet})$ is the upper adjoint of a GC which combines two adjoints already seen — range (71) and left division (23). The pointwise version $wlp\ R\ q$ of $R \mathbin{\backslash\!\!\bullet} \Phi_q$ is

$$wlp\,R\,q \triangleq \langle \bigvee\ p\ :\ \langle \forall\, b, a\ :\ b\,R\,a\ \wedge\ p\,a :\ q\,b\rangle :\ p\rangle$$

Also recall (132), which tells that checking $\Phi_q \xleftarrow{R} \Phi_p$ is the same as first calculating $R \mathbin{\backslash\!\!\bullet} \Phi_q$ and then showing that this is weaker than Φ_p. This leads to the constructive method for ESC which has already been adopted in the case study of section 13.

Besides this practical application, (132) is central to the close relationship between the ESC/PF calculus and the WLP-calculus. In fact, the two approaches are related by indirect equality (15). Let us take as example a rule of the latter calculus

$$R \mathbin{\backslash\!\!\bullet} (\Upsilon \cdot \Psi) = (R \mathbin{\backslash\!\!\bullet} \Upsilon) \cdot (R \mathbin{\backslash\!\!\bullet} \Psi)$$

which holds since $(R\mathbin{\backslash\!\!\bullet})$ is an upper-adjoint and therefore distributes over meet, ie. composition in the case of coreflexives [9]. We reason:

$$
\begin{array}{ll}
& R \mathbin{\backslash\!\!\bullet} (\Upsilon \cdot \Psi) = (R \mathbin{\backslash\!\!\bullet} \Upsilon) \cdot (R \mathbin{\backslash\!\!\bullet} \Psi) \\
\Leftrightarrow & \quad \{ \text{ indirect equality (15) } \} \\
& \langle \forall\, \Phi\ ::\ \Phi \subseteq R \mathbin{\backslash\!\!\bullet} (\Upsilon \cdot \Psi)\ \Leftrightarrow\ \Phi \subseteq (R \mathbin{\backslash\!\!\bullet} \Upsilon) \cdot (R \mathbin{\backslash\!\!\bullet} \Psi) \rangle \\
\Leftrightarrow & \quad \{ \text{ (60) ; (21) } \} \\
& \langle \forall\, \Phi\ ::\ \Phi \subseteq R \mathbin{\backslash\!\!\bullet} (\Upsilon \cdot \Psi)\ \Leftrightarrow\ \Phi \subseteq R \mathbin{\backslash\!\!\bullet} \Upsilon\ \wedge\ \Phi \subseteq R \mathbin{\backslash\!\!\bullet} \Psi \rangle \\
\Leftrightarrow & \quad \{ \text{ (132) three times, omitting universal quantification } \} \\
& \Upsilon \cdot \Psi \xleftarrow{R} \Phi\ \ \Leftrightarrow\ \ \Upsilon \xleftarrow{R} \Phi\ \wedge\ \Psi \xleftarrow{R} \Phi
\end{array}
$$

We thus obtain (144), the equivalent ESC/PF-rule. A more interesting example is the transformation of WLP-rule

$$(S \cdot R) \mathbin{\backslash\!\!\bullet} \Phi = R \mathbin{\backslash\!\!\bullet} (S \mathbin{\backslash\!\!\bullet} \Phi)$$

into ESC/PF format:

$$
\begin{array}{ll}
& R \mathbin{\backslash\!\!\bullet} (S \mathbin{\backslash\!\!\bullet} \phi) = (S \cdot R) \mathbin{\backslash\!\!\bullet} \phi \\
\Leftrightarrow & \quad \{ \text{ indirect equality (15) } \} \\
& \psi \subseteq R \mathbin{\backslash\!\!\bullet} (S \mathbin{\backslash\!\!\bullet} \phi) \Leftrightarrow \psi \subseteq (S \cdot R) \mathbin{\backslash\!\!\bullet} \phi \\
\Leftrightarrow & \quad \{ \text{ (132) twice } \} \\
& (S \mathbin{\backslash\!\!\bullet} \phi) \xleftarrow{R} \psi\ \ \Leftrightarrow\ \ \phi \xleftarrow{(S \cdot R)} \psi
\end{array}
$$

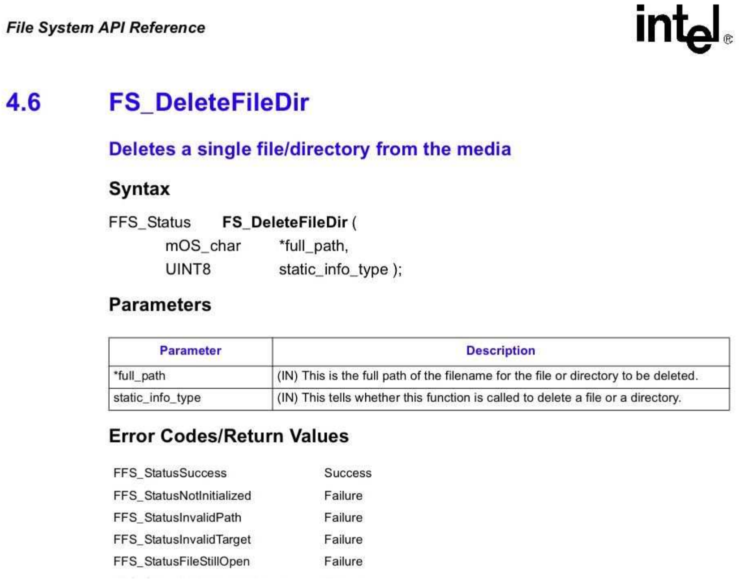
The outcome is a ESC/PF rule which, still involving the $\mathbin{\backslash\!\!\bullet}$ operator, is an advantageous replacement for (143), since it is an equivalence.

16 Case study 2 — Verified File System

Our second group of experiments with the ESC/PF calculus has to do with a real-life project. In the context of the Verified Software Initiative [28], we want to validate a formal model of a file system as part of a broader exercise on providing a verified file system (VFS) on flash memory — a challenge put forward by Rajeev Joshi and Gerard Holzmann of NASA JPL [34].

An explanation of the overall approach to the problem, involving not only formal modeling but also model checking in Alloy and theorem proving in HOL [26] can be found in [21]. Below we shall be concerned only with showing the role of the PF-transform in statically checking the model by pen-and-paper calculation.

As explained in [21], the problem has two levels — the POSIX level and the NAND flash level. The work so far has focussed on the top level, taking as working document Intel's *Flash File System Core Reference Guide* [16]. This is a layered collection of APIs, of which we are considering FS (file system), the top one. Figure 2 gives an idea of what is to be modeled for each file system operation, in this case the one which enables file/directory deletion.



File System API Reference

4.6 FS_DeleteFileDir

Deletes a single file/directory from the media

Syntax

```
FFS_Status     FS_DeleteFileDir (
        mOS_char       *full_path,
        UINT8          static_info_type );
```

Parameters

Parameter	Description
*full_path	(IN) This is the full path of the filename for the file or directory to be deleted.
static_info_type	(IN) This tells whether this function is called to delete a file or a directory.

Error Codes/Return Values

FFS_StatusSuccess	Success
FFS_StatusNotInitialized	Failure
FFS_StatusInvalidPath	Failure
FFS_StatusInvalidTarget	Failure
FFS_StatusFileStillOpen	Failure

Fig. 2. Example of API specification in [16]. (Permission to reproduce this excerpt is kindly granted by Intel Corporation.)

Data model. By inspecting reference guide [16] we have arrived at a formal, relational model of the file system structure which, stripped of details irrelevant for the operation of figure 2, can be depicted in the relational diagram which follows:

$$\begin{tikzcd}
\mathit{FileHandler} \arrow[r, rightharpoonup, "M"] & \mathit{OpenFileDescriptor} \arrow[dl, "path"] \\
\mathit{Path} \arrow[r, rightharpoonup, "N"'] & \mathit{File}
\end{tikzcd} \tag{151}$$

This tells that there are two simple relations in the model, one (N in the diagram) relating paths to file contents and another (M in the diagram) giving details of each opened file identified by a file handler. These two data structures are linked by function path which selects paths from the information recorded in $\mathit{OpenFileDescriptor}$s and is central to the main invariant of the model — the referential integrity condition which ensures that non-existing files cannot be opened:

$$\begin{array}{l}
\mathit{System} = \{\mathit{table} : \mathit{OpenFileDescriptorTable}, \mathit{fs} : \mathit{FStore}\} \\
\textbf{inv}\ \ \mathit{sys} \triangleq \langle \forall\, d\ :\ d \in \mathit{rng}\ (\mathit{table}\ \mathit{sys}) :\ \mathit{path}\ d \in \mathit{dom}\ (\mathit{fs}\ \mathit{sys})\rangle
\end{array} \tag{152}$$

In this “linguistic version” of diagram (151) the choice of long identifiers is justified by practical reasons, due to the overall complexity of the whole model. While datatypes

$$\begin{array}{l}
\mathit{OpenFileDescriptorTable} = \mathit{FileHandler} \rightharpoonup \mathit{OpenFileDescriptor} \\
\mathit{OpenFileDescriptor} = \{\mathit{path} : \mathit{Path}, ...\}
\end{array}$$

are subject to no invariant (in this simplified version), file stores should be such that father directories always exist and are indeed directories:

$$\begin{array}{l}
\mathit{FStore} = \mathit{Path} \rightharpoonup \mathit{File} \\
\textbf{inv}\ \ \mathit{store} \triangleq \langle \forall\, p\ :\ p \in \mathit{dom}\ \mathit{store} :\ \mathit{dirName}(p) \in \mathit{dom}\ \mathit{store}\ \wedge \\
\qquad\qquad \mathit{fileType}(\mathit{attributes}(\mathit{store}(\mathit{dirName}\ p))) = \mathit{Directory}\rangle
\end{array} \tag{153}$$

The function $\mathit{dirName} : \mathit{Path} \rightarrow \mathit{Path}$ tells the father path of a given path. There exists a topmost path Root in the path hierarchy which, according to the requirements [16], is such that $\mathit{dirName}\ \mathit{Root} = \mathit{Root}$. Files have attributes and $\mathit{fileType}$ is one such attribute. For space economy, we omit all other details of the model’s data types.

Modeling the operations. Let us focus on the API operation which enables file deletion (figure 2) modeled after the requirements in [16] as follows:

$$\begin{array}{l}
\mathit{FS\_DeleteFileDir} : \mathit{Path} \rightarrow \mathit{System} \rightarrow (\mathit{System} \times \mathit{FFS\_Status}) \\
\mathit{FS\_DeleteFileDir}\ p\ \mathit{sys} \triangleq \\
\qquad \mathit{if}\ p \neq \mathit{Root}\ \wedge\ p \in \mathit{dom}\ (\mathit{fs}\ \mathit{sys})\ \wedge\ \text{pre-}\mathit{FS\_DeleteFileDir\_System}\ p\ \mathit{sys} \\
\qquad \mathit{then}\ (\mathit{FS\_DeleteFileDir\_System}\ p\ \mathit{sys}, \mathit{FFS\_StatusSuccess}) \\
\qquad \mathit{else}\ (\mathit{sys}, \mathit{FS\_DeleteFileDir\_Exception}\ p\ \mathit{sys})
\end{array}$$

This is a function that either deletes the FStore entry whose path p is given or raises an exception, leaving the state unchanged and returning the appropriate error code (of type $\mathit{FFS\_Status}$). Function $\mathit{FS\_DeleteFileDir\_Exception}$ returning error codes does not interfere with ESC and is therefore omitted. By contrast, the core of the success trace is the (partial) function which updates the system once the specified entry can indeed be deleted:

$$
\begin{array}{l}
FS\_DeleteFileDir\_System : Path \rightarrow System \rightarrow System \\
FS\_DeleteFileDir\_System\; p\; (h, t) \triangleq \\
\qquad (h, FS\_DeleteFileDir\_FStore\; \{p\}\; t) \\
\textbf{pre}\; \langle \forall\, d\; :\; d \in rng\; h :\; path\; d \neq p \rangle\; \wedge\; \text{pre-}FS\_DeleteFileDir\_FStore\; \{p\}\; t
\end{array}
$$

This, in turn, calls a function whose scope is the $FStore$ component of $System$. This is where things actually happen:

$$
\begin{array}{l}
FS\_DeleteFileDir\_FStore : \mathcal{P}Path \rightarrow FStore \rightarrow FStore \\
FS\_DeleteFileDir\_FStore\; s\; store \triangleq\; store \setminus s \\
\textbf{pre}\; \langle \forall\, p\; :\; p \in dom\; store\; \wedge\; dirName\; p \in s :\; p \in s \rangle
\end{array}
\qquad (154)
$$

This function actually deletes sets of entries (and not individual ones) using the *domain restricted by* operator $M \setminus S$ typical of model-oriented specification languages such as VDM or Z, whose meaning is *select the largest sub-relation of M whose keys are not in S*. Formally, the PF-transform of this operator is

$$
[\![M \setminus S]\!] = M \cdot \Phi_{(\notin S)} \qquad (155)
$$

Note that $FS\_DeleteFileDir\_System$ and $FS\_DeleteFileDir\_FStore$ are subject to pre-conditions *invented* by the software analyst who wrote the model. Such pre-conditions are the main target of our reasoning below. Several questions arise: how *"good"* are these? are they *sufficient* for the invariants to be maintained? are they too *strong*? which are concerned with ESC alone and which are restrictions posed by the API specifier derived from POSIX recommendations or constraints?

Before answering these questions, we should say that real problems such as this have the merit of showing *where the complexity actually is*, and part of it has to do with the (often intricate) structure of datatypes involving nested invariants. This calls for an effective way of calculating which invariants hold at which levels of a given data model in terms of the associated coreflexives, as shown next.

17 Invariant structural synthesis

Let us denote by F_p the fact that data type constructor F is constrained by invariant p. Of course, F itself can be defined in terms of other type constructors constrained by their own invariants. We write $\in\!\!\!\in_{\mathsf{F}_p}$ to denote the coreflexive which captures *all* constraints involved in declaring type F_p. This is defined by induction on the structure of type constructors (relators) [21]:

$$
\in\!\!\!\in_{\mathsf{F}_p} = (\in\!\!\!\in_{\mathsf{F}}) \cdot \Phi_p \qquad (156)
$$

[21] The choice of symbol "$\in\!\!\!\in$" instead of "$\in$", which would be more natural regarding its use in eg. (42), is due to the fact that notation $\in_{\mathsf{F}}$ is already taken by *structural membership* [29], a related but different concept.

$$\text{€}_{\mathsf{K}} = id \tag{157}$$

$$\text{€}_{\mathsf{Id}} = id \tag{158}$$

$$\text{€}_{\mathsf{F}\times\mathsf{G}} = \text{€}_{\mathsf{F}} \times \text{€}_{\mathsf{G}} \tag{159}$$

$$\text{€}_{\mathsf{F}+\mathsf{G}} = \text{€}_{\mathsf{F}} + \text{€}_{\mathsf{G}} \tag{160}$$

$$\text{€}_{\mathsf{F}\cdot\mathsf{G}} = \mathsf{F}(\text{€}_{\mathsf{G}}) \tag{161}$$

For instance, $Even = (I\!N_0)_{even}$, recall (43). Thus $\text{€}_{Even} = \Phi_{even}$ by direct application of (156) and (157). In the calculation of €_{System} which follows we abbreviate its invariant declared in (152) by predicate ri (for "referential integrity") and $FStore$'s invariant (153) by pc (for "paths closed"):

$$
\begin{array}{ll}
 & \text{€}_{System} \\
= & \quad \{ \text{ definition of } System \text{ (152) } \} \\
 & \text{€}_{(OpenFileDescriptorTable \times FStore)_{ri}} \\
= & \quad \{ \text{ (156) and datatype definitions } \} \\
 & (\text{€}_{FileHandler \rightharpoonup OpenFileDescriptor} \times \text{€}_{(Path \rightharpoonup File)_{pc}}) \cdot \Phi_{ri} \\
= & \quad \{ \text{ (157) and (156) } \} \\
 & (id \times \text{€}_{Path \rightharpoonup File} \cdot \Phi_{pc}) \cdot \Phi_{ri} \\
= & \quad \{ \text{ (157) } \} \\
 & (id \times \Phi_{pc}) \cdot \Phi_{ri}
\end{array}
\tag{162}
$$

18 ESC/PF calculus at work — case study 2

Now that we know the pointfree structure $(id \times \Phi_{pc}) \cdot \Phi_{ri}$ of the overall invariant which we have to ESC for, let us investigate the structure of the operation we want to check — $FS\_DeleteFileDir$. Our main goal is to discharge proof obligation

$$\text{€}_{System \times FFS\_Status} \xleftarrow{FS\_DeleteFileDir\ p} \text{€}_{System} \tag{163}$$

We start by using the PF-transform to "find structure" in the specification text. By freezing parameter p (which is not active in the specification of the operation) and PF-transforming $FS\_DeleteFileDir\ p$ we obtain a PF-expression which has the "shape" of a McCarthy conditional (150)

$$c \to \langle f, \underline{k} \rangle\ ,\ \langle id, g \rangle \tag{164}$$

where

- c abbreviates section $(c\, p)$ of the condition of the main if-then-else, that is $c\, p\, sys \triangleq p \neq Root\ \wedge\ p \in dom\ (fs\ sys)\ \wedge$ pre-$FS\_DeleteFileDir\_System\ p\ sys$
- f abbreviates $FS\_DeleteFileDir\_System\ p$
- k abbreviates $FFS\_StatusSuccess$, the success output code
- g abbreviates $FS\_DeleteFileDir\_Exception\ p$.

Facing complexity. What's the advantage of PF-pattern (164)? Below we show how to apply the ESC/PF calculus of section 15 to (164) in a "divide and conquer" manner, thus breaking the complexity of the target proof obligation (163):

$$€_{System \times FFS\_Status} \xleftarrow{FS\_DeleteFileDir\ p} €_{System}$$

$$\Leftrightarrow \quad \{ \text{ (164), (159) and } €_{FFS_{S}tatus} = id \text{ (157) } \}$$

$$€_{System} \times id \xleftarrow{c \rightarrow \langle f, \underline{k} \rangle, \langle id, g \rangle} €_{System}$$

$$\Leftrightarrow \quad \{ \text{ conditional (149) } \}$$

$$€_{System} \times id \xleftarrow{\langle f, \underline{k} \rangle} €_{System} \cdot \Phi_{c} \quad \wedge \quad €_{System} \times id \xleftarrow{\langle id, g \rangle} €_{System} \cdot \Phi_{\neg c}$$

$$\Leftrightarrow \quad \{ \text{ splitting (101) } \}$$

$$\begin{array}{c} €_{System} \xleftarrow{f} €_{System} \cdot \Phi_{c} \ \wedge \ id \xleftarrow{\underline{k}} €_{System} \cdot \Phi_{c} \\ \wedge \\ €_{System} \xleftarrow{id} €_{System} \cdot \Phi_{\neg c} \ \wedge \ id \xleftarrow{g} €_{System} \cdot \Phi_{\neg c} \end{array}$$

$$\Leftrightarrow \quad \{ \text{ (141), (94) } \}$$

$$€_{System} \xleftarrow{f} €_{System} \cdot \Phi_{c}$$

$$\Leftrightarrow \quad \{ \text{ trading (142) and unfolding } €_{System} \text{ (162) } \}$$

$$(id \times \Phi_{pc}) \cdot \Phi_{ri} \xleftarrow{f \cdot \Phi_{c}} (id \times \Phi_{pc}) \cdot \Phi_{ri}$$

$$\Leftarrow \quad \{ \text{ separating (146) } \}$$

$$\Phi_{ri} \xleftarrow{f \cdot \Phi_{c}} \Phi_{ri} \quad \wedge \quad id \times \Phi_{pc} \xleftarrow{f \cdot \Phi_{c}} id \times \Phi_{pc}$$

Clearly, the focus has moved from the main function to $FS\_DeleteFileDir\_System\, p$ (abbreviated to f above) with respect to two (now *separate*) proofs: one concerning path referential integrity (ri) and the other concerning path closure (pc).

It can be further observed that condition c splits in two independent parts, that is, $\Phi_{c} = \Phi_{c_1} \times \Phi_{c_2}$ where [22]

$$c_1\, p\, h \triangleq \langle \forall\, d\ :\ d \in rng\, h :\ path\, d \neq p \rangle$$
$$c_2\, p\, t \triangleq p \neq Root \ \wedge \ p \in dom\, t \ \wedge \ \text{pre-}FS\_DeleteFileDir\_FStore\, \{p\}\, t$$

Moreover,

$$f = FS\_DeleteFileDir\_System\, p$$

[22] Note the somewhat arbitrary decision of adding condition $p \neq Root$ to c_2. We shall have more to say about this. Also note the notation convention of abbreviating sections $(c_1\ p)$ and $(c_2\ p)$ by c_1 and c_2 in coreflexives' subscripts.

$$= id \times FS\_DeleteFileDir\_FStore\ \{p\} \tag{165}$$
$$= id \times f_2 \tag{166}$$

introducing abbreviation f_2 to save space. So we can calculate further:

$$\Phi_{ri} \xleftarrow{f \cdot \Phi_c} \Phi_{ri} \quad \wedge \quad id \times \Phi_{pc} \xleftarrow{f \cdot \Phi_c} id \times \Phi_{pc}$$

$\Leftrightarrow \qquad \{$ $\Phi_c = \Phi_{c_1} \times \Phi_{c_2}$; $f = id \times f_2$; $\times$-relator (109) $\}$

$$\Phi_{ri} \xleftarrow{f \cdot \Phi_c} \Phi_{ri} \quad \wedge \quad id \times \Phi_{pc} \xleftarrow{\Phi_{c_1} \times f_2 \cdot \Phi_{c_2}} id \times \Phi_{pc}$$

$\Leftrightarrow \qquad \{$ (148) ; (141) $\}$

$$\Phi_{ri} \xleftarrow{f \cdot \Phi_c} \Phi_{ri} \quad \wedge \quad \Phi_{pc} \xleftarrow{f_2 \cdot \Phi_{c_2}} \Phi_{pc}$$

$\Leftrightarrow \qquad \{$ trading (142) $\}$

$$\Phi_{ri} \xleftarrow{f} \Phi_{ri} \cdot \Phi_c \quad \wedge \quad \Phi_{pc} \xleftarrow{f_2} \Phi_{pc} \cdot \Phi_{c_2} \tag{167}$$

Going "in-the-small". So much for ESC/PF calculation *in-the-large*. Going *in-the-small* means spelling out invariants, functions and pre-conditions and reason as in the previous case study.

Let us pick the first proof obligation in (167), $\Phi_{ri} \xleftarrow{f} \Phi_{ri} \cdot \Phi_c$. Following (132) as earlier on, we go pointwise and try to rewrite weakest pre-condition $ri(f(M, N))$ — where M handles open file descriptors and N file contents, recall diagram (151) — into $ri(M, N)$ and a pre-condition, which will be the weakest for maintaining ri provided all steps in the calculation are equivalences. Then we compare the outcome with what the designer wrote (Φ_c).

Taking advantage of the fact that both data structures M and N are relations, we choose to start by PF-transforming ri

$$ri(M, N) \ \triangleq \ \rho\,(path \cdot M) \subseteq \delta\, N$$

according to diagram (151) and thus investing on PF-notation again. This expression for ri, which clearly spells out the referential integrity constraint relating paths in opened file descriptors and paths in the file store N, further transforms to

$$ri(M, N) \ \triangleq \ path \cdot M \subseteq N^\circ \cdot \top \tag{168}$$

cf. diagram

$$\begin{tikzcd} OpenFileDescriptor \arrow[d, "path"'] \arrow[dr, phantom, "\subseteq"] & FileHandler \arrow[l, "M"'] \arrow[d, "\top"] \\ Path & File \arrow[l, "N^\circ"] \end{tikzcd}$$

On the other hand, (165, 166) and (155) lead to

$$\begin{aligned} ri(f(M,N)) &= ri(FS\_DeleteFileDir\_System\, p\, (M,N)) \\ &= ri(M, N \cdot \Phi_{(\notin \{p\})}) \end{aligned}$$

In the calculation below we generalize $\{p\}$ to any set S of paths:

$$\begin{aligned}
& ri(M, N \cdot \Phi_{(\notin S)}) \\
\Leftrightarrow \quad & \{ \text{ (168) } \} \\
& path \cdot M \subseteq (N \cdot \Phi_{(\notin S)})^{\circ} \cdot \top \\
\Leftrightarrow \quad & \{ \text{ converses (26,59) } \} \\
& path \cdot M \subseteq \Phi_{(\notin S)} \cdot N^{\circ} \cdot \top \\
\Leftrightarrow \quad & \{ \text{ (47) } \} \\
& path \cdot M \subseteq N^{\circ} \cdot \top \cap \Phi_{(\notin S)} \cdot \top \\
\Leftrightarrow \quad & \{ \text{ $\cap$-universal (21) } \} \\
& path \cdot M \subseteq N^{\circ} \cdot \top \ \wedge \ path \cdot M \subseteq \Phi_{(\notin S)} \cdot \top \\
\Leftrightarrow \quad & \{ \text{ (168) ; shunting (67) } \} \\
& ri(M,N) \ \wedge \ \underbrace{M \subseteq path^{\circ} \cdot \Phi_{(\notin S)} \cdot \top}_{wp}
\end{aligned}$$

The obtained weakest pre-condition wp converts back to the pointwise $\langle \forall\, b \ : \ b \in rng\ M \ : \ path\ b \notin S \rangle$ which instantiates to $\langle \forall\, b \ : \ b \in rng\ M \ : \ path\ b \neq p \rangle$ for $S := \{p\}$. This is in fact a conjunct of pre-$FS\_DeleteFileDir\_System$, itself a conjunct of $c\ p$, the condition of $FS\_DeleteFileDir$'s if-then-else. So we are done as far invariant ri is concerned.

Before moving to invariant pc, note two levels of reasoning in ESC/PF calculations: the *in-the-large* level using the ESC/PF arrow calculus and the *in-the-small* level, where PF-notation describes data and properties of data, typically invariants.

Checking for paths-closed invariant preservation. Our last ESC/PF exercise has to do with the remaining proof obligation

$$\Phi_{pc} \xleftarrow{FS\_DeleteFileDir\_FStore\ \{p\}} \Phi_{pc} \cdot \Phi_{c_2} \tag{169}$$

where $FS\_DeleteFileDir\_FStore\ \{p\}$ PF-transforms to

$$(FS\_DeleteFileDir\_FStore\ S)\ N = N \cdot \Phi_{(\notin S)} \tag{170}$$

generalizing $\{p\}$ to an arbitrary set of paths S, as we have seen. The PF-transform of invariant pc,

$$pc\ N \triangleq \underline{Directory} \cdot N \subseteq fileType \cdot attributes \cdot N \cdot dirName \tag{171}$$

is explained by the rectangle added below to diagram (151):

$$\begin{tikzcd}
FileHandler \arrow[r, "M"] & OpenFileDescriptor \arrow[dl, "path"'] & \\
Path \arrow[r, "N"] & File \arrow[r, "attributes"] \arrow[d, phantom, "\rotatebox{90}{$\subseteq$}"] & Attributes \arrow[d, "fileType"] \\
Path \arrow[u, "dirName"] \arrow[r, "N"'] & File \arrow[r, "\underline{Directory}"'] & FileType
\end{tikzcd}$$

Again, our strategy will be to ignore Φ_{c_2} in (169) for a moment and calculate the weakest pre-condition for $FS\_DeleteFileDir\_FStore\ S$ to preserve pc; then we compare Φ_{c_2} with the pre-condition thus obtained. For improved readability, we introduce abbreviations $ft := fileType \cdot attributes$ and $d := \underline{Directory}$:

$$\begin{array}{ll}
& pc(FS\_DeleteFileDir\_FStore\ S\ N) \\
\Leftrightarrow & \quad \{ \text{ (170) and (171) } \} \\
& d \cdot (N \cdot \Phi_{(\notin S)}) \subseteq ft \cdot (N \cdot \Phi_{(\notin S)}) \cdot dirName \\
\Leftrightarrow & \quad \{ \text{ shunting (69) } \} \\
& d \cdot N \cdot \Phi_{(\notin S)} \cdot dirName^{\circ} \subseteq ft \cdot N \cdot \Phi_{(\notin S)} \\
\Leftrightarrow & \quad \{ \text{ (46) } \} \\
& d \cdot N \cdot \Phi_{(\notin S)} \cdot dirName^{\circ} \subseteq ft \cdot N \cap \top \cdot \Phi_{(\notin S)} \\
\Leftrightarrow & \quad \{ \text{ $\cap$-universal ; shunting } \} \\
& d \cdot N \cdot \Phi_{(\notin S)} \subseteq ft \cdot N \cdot dirName \ \wedge \ d \cdot N \cdot \Phi_{(\notin S)} \subseteq \top \cdot \Phi_{(\notin S)} \cdot dirName \\
\Leftrightarrow & \quad \{ \text{ $\top$ absorbs d (115) } \} \\
& \underbrace{d \cdot N \cdot \Phi_{(\notin S)} \subseteq ft \cdot N \cdot dirName}_{\text{weaker than } pc(N)} \ \wedge \ \underbrace{N \cdot \Phi_{(\notin S)} \subseteq \top \cdot \Phi_{(\notin S)} \cdot dirName}_{wp}
\end{array}$$

This ends the PF-calculation of this ESC proof obligation. It remains to compare c_2 with wp just above which, back to points, re-writes to:

$$\begin{array}{ll}
& \langle \forall\, q\ :\ q \in dom\ N\ \wedge\ q \notin S :\ dirName\ q \notin S \rangle \\
\Leftrightarrow & \quad \{ \text{ predicate logic } \} \\
& \langle \forall\, q\ :\ q \in dom\ N\ \wedge\ (dirName\ q) \in S :\ q \in S \rangle
\end{array}$$

This is pre-condition (154) which, in words, means: *if parent directory of existing path q is marked for deletion than so must be q*. Condition c_2 involves this pre-condition, for $S := \{p\}$,

$$\langle \forall\, q\ :\ q \in dom\ N\ \wedge\ (dirName\ q) = p :\ q = p \rangle$$

but adds further constraints. So c_2 is stronger than the calculated weakest pre-condition and, thanks to (132), we are done. In particular, c_2 doesn't allow for $Root$ deletion. Condition wp enables so (since $dirName\ Root = Root$) provided no other files exist in the file system.

The interest of these observations, which we have reached by calculation, lies in the fact that the POSIX standard itself [58] is ambiguous in this matter. Whether the minimal $FStore$ is the empty relation or whether it must be the root directory's singleton is a bit of a philosophical question. In the POSIX System Interface [58] one reads, concerning the `rmdir()` system call:

> *The rmdir() function shall remove a directory whose name is given by path. The directory shall be removed only if it is an empty directory. If the directory is the root directory or the current working directory of any process, it is unspecified whether the function succeeds, or whether it shall fail and set errno to [EBUSY].*

Another aspect of the starting specification is clause $p \in dom\ (fs\ sys)$. From the calculations above we infer that no harm arises from trying to delete a non-existing file, as nothing happens to the system. So, the corresponding error code should be interpreted more as a warning than as an exception.

19 Alloy friendship

The "everything is a relation" lemma of Alloy and the PF-flavour of its notation turn the Alloy Analyzer into a very helpful tool supporting the PF-transform on practical grounds. This tool has been developed by the Software Design Group at MIT for analyzing models written in a simple structural modeling language based on first-order logic. Being a model checker, it does not discharge proofs as such but is very useful in finding (via counter-examples) design flaws, as reported in [21] concerning the VFS project.

Space constraints prevent us from giving the Alloy model in detail. We focus on invariant pc which PF-transforms to (171). Note the similarity between (171) and the corresponding code in Alloy syntax,

```
pred pcInvariant[t: FStore]{
  RelCalc/Simple[t.map, Path]
  (t.map).(File->Directory)
          in dirName.(t.map).attributes.fileType
}
```

where predicate `Simple` is available from library `RelCalc` (it checks for relation simplicity), composition is written in reverse order and `map` has to do with the declaration of $FStore$ as an Alloy signature [30]:

```
sig FStore {
  map: Path -> File,
}
```

Note how `File->Directory` elegantly represents the constant function $\underline{Directory}$ in (171). The alternative, pointwise version of pc is written in Alloy as follows:

```
pred pcInvariantPW[t: FStore]{
   RelCalc/Simple[t.map, Path]
   all p: Path |
       p in RelCalc/dom[t.map] =>
            p.dirName in RelCalc/dom[t.map] &&
                 t.map[p.dirName].attributes.fileType =
                                                  Directory
}
```

Checking (not proving!) the equivalence of these two alternative predicates can be expressed in Alloy by running assertion

```
assert equivPWPF {
  all t: FStore | pcInvariant[t] <=> pcInvariantPW[t]
}
```

See [21] for more about the role of Alloy in the VFS case study.

Exercise 14. Alloy will find counter-examples to the assertion above once the simplicity requirement `RelCalc/Simple[t.map, Path]` is dropped from both predicates. Resort to the PF-calculus and show why the calculations which lead to (171) are not valid for arbitrary N. □

20 Conclusions

In full-fledged formal software development one is obliged to provide mathematical proofs that desirable properties of software systems hold. An important class of such properties has to do with (extended) type checking and includes those which ensure that datatype invariants are not violated by some trace of the system at runtime. A way to prevent this consists of abstractly modeling the intended system using a formal language, formulating such proof obligations and proving them. Because this is not done at run-time, this class of proof belongs to the *static* world of software quality checking and is known under the ESC (*extended static checking*) acronym.

ESC proofs can either be performed as paper-and-pencil exercises or, in case of sizeable models, be supported by theorem provers and model checkers. Real-life case studies show that *all* such approaches to adding quality to a formal model are useful in their own way and have a proper place in software engineering using formal methods.

The main novelty of the approach put forward in the current paper resides in the chosen method of proof construction: first-order formulæ in proof obligations are subject to the PF-transform before they are reasoned about. This “Laplace flavoured” transformation eliminates quantifiers and bound variables and reduces complex formulas to algebraic relational expressions which are more agile to calculate with. Suitable relational encoding of recursive structures often makes it possible to perform non-inductive proofs over such structures.

The overall approach is structured in two layers: one is a formal set of rules (the ESC/PF calculus) which enable one to break complex proof obligations into smaller ones, by exploiting both the structure of the predicates involved (expressed as coreflexive relations) and the PF structure of the software operations being checked. This is referred to as the *in-the-large* ESC/PF level, which uses arrow notation clearly reminding the user that one is doing (extended) type checking.

One moves into the *in-the-small* level wherever discharging elementary proofs, that is, ESC-arrows which cannot be further decomposed by *in-the-large* calculation. In spite of the reasoning going pointwise at this level, the PF-calculus turns up again wherever the particular data structures are encoded as relations and invariants as PF-formulæ involving such relations. As already stressed in [48], this is a novel ingredient in PF-calculation, since most work on the pointfree relation calculus has so far been focused on reasoning about programs (ie. algorithms) [13]. Advantages of our proposal to *uniformly* PF-transform both programs and data are already apparent at practical level, see eg. the work reported in [41]. The approach contrasts with the VDM tradition where universal quantifications over finite lists and finite mappings are carried out by induction on such structures [32]. It should be noted, however, that not *every* proof obligation leads to such calculations. The encoding of lists into simple relations, for instance, does not take *finiteness* aspects (eg. counting elements, etc) into account.

Last but not least, this paper helps in better characterizing the notion of *type* of an arbitrary piece of code, since Hoare logic is shown to be under the umbrella of ESC/PF, as is the weakest pre-condition calculus.

21 Future work

This paper finds its roots in the excellent background for computer science research developed by the MPC (Mathematics of Program Construction) group [1, 29, 13, 7]. Surely there is still much to explore. For instance, Voermans's PhD thesis [62] investigates the use of PERs (partial equivalence relations) to model datatypes subject to axioms, as in the classic abstract data type (ADT) tradition. Coreflexives are minimal PERs, so the view of *coreflexives as types* implicit in the current paper can surely be extended to that of *PERs as types*. How much is gained in this generalization needs to be balanced against what is likely to be lost.

The idea that the proposed ESC/PF calculus bridges Hoare logic and type theory needs to be better exploited, in particular concerning the work by Kozen [36] on subsuming propositional Hoare logic under Kleene algebra with tests (of which the relational calculus is a well known instance [7]) and the work emerging on *Hoare type theory* (HTT) [40], which should be carefully studied. Still on the type theory track, the alternative use of dependent types to model types subject to invariants and the way in which ESC proofs are carried out by systems such as Agda [14] should be compared to the current paper's approach.

The arrow notation adopted in the ESC/PF calculus not only is adequate to express proof obligation discharge as a type-system kind of problem, but also triggers synergies with similar notation used in other branches of computing. Pick functional dependence (FD) theory [38], for instance, where one writes $f \xrightarrow{R} g$ to mean that in database relation R (set of tuples), attribute g is functionally dependent on attribute f:

$$\langle \forall\, t, t' \;:\; t, t' \in R :\; f\,t = f\,t' \Rightarrow g\,t = g\,t' \rangle$$

Compare, for instance, (145) with the *decomposition* axiom of FDs

$$h \xrightarrow{R} k \Leftarrow h \geq f \ \wedge \ f \xrightarrow{R} g \ \wedge \ g \geq k$$

where $\leq$ compares (sets of) attributes. In the PF-approach to FD-theory developed in [45, 47] [23], R in $f \xrightarrow{R} g$ is modeled by a coreflexive relation and attributes f, g by functions. (So functions and coreflexives swap places when compared with ESC arrows.) Checking how much structure is shared among these two (so far apart) theories is something the PF-transform has potential for.

As far as tool support is concerned, reference [41] already presents visible progress in the automation of the relational calculus applied to ESC-like situations. Calculations are performed using a Haskell term rewriting system written in the strategic programming style. Another related line of research is the design of the *Galculator* [56], a prototype of a proof assistant of a special brand: it is solely based on the algebra of Galois connections. When combined with the PF-transform and tactics such as indirect equality (15), it offers a powerful, generic device to tackle the complexity of proofs in program verification. Moreover, we think the ESC/PF calculus could be of help in designing a *generic* proof obligation generator which could be instantiated to particular tool-sets such as, for instance, the one developed by Vermolen [61] for VDM.

The ESC/PF calculus can be further developed taking into account other aspects of model-based reasoning such as, for instance, refinement [32, 66]. The reader is left with an exercise which provides a foretaste of ESC rules entailed by operation refinement.

Exercise 15. The refinement ordering on pre/post-specification pairs viewed as binary relations can be defined by

$$S \vdash R \quad \triangleq \quad \delta\, S \subseteq (R \setminus S) \cap \delta\, R \tag{172}$$

meaning that S (the specification) is smaller domain-wise and vaguer range-wise than R (the implementation) [50]. That is, implementations can only be more defined and more deterministic than specifications.

From $\Psi \xleftarrow{S} \Phi$ and $S \vdash R$ infer $\Psi \xleftarrow{R} \Phi \cdot \delta\, S$.
□

Acknowledgments

Part of this research was carried out in the context of the PURe Project *(Program Understanding and Re-engineering: Calculi and Applications)* funded by FCT contract POSI/ICHS/44304/2002.

The author wishes to thank the organizers of the PUReCafe meetings for the opportunity to present his first ideas about ESC/PF at one of the seminars, back to 2004 [43]. He also thanks Simão Sousa for pointing him to HTT.

The work of Claudia Necco and Joost Visser on putting the ESC/PF into practice is also gratefully acknowledged. Thanks are also due to Miguel Ferreira and Samuel Silva for their interest in the VFS project at Minho.

[23] Also recall exercise 8 from section 9 of the current paper.

A Background — Eindhoven quantifier calculus

When writing $\forall, \exists$-quantified expressions is useful to know a number of rules which help in reasoning about them. Throughout this paper we adopt the Eindhoven quantifier notation and calculus [7, 4] whereby

$$\langle \forall\, x \;:\; R : \; T \rangle$$
$$\langle \exists\, x \;:\; R : \; T \rangle$$

mean, respectively

- *"for all x in range R it is the case that T"*
- *"there exists x in range R such that T".*

Some useful rules about $\forall, \exists$ follow, taken from [7] [24]:

- Trading:

$$\langle \forall\, i \;:\; R \wedge S : \; T \rangle = \langle \forall\, i \;:\; R : \; S \Rightarrow T \rangle \tag{173}$$
$$\langle \exists\, i \;:\; R \wedge S : \; T \rangle = \langle \exists\, i \;:\; R : \; S \wedge T \rangle \tag{174}$$

- One-point:

$$\langle \forall\, k \;:\; k = e : \; T \rangle = T[k := e] \tag{175}$$
$$\langle \exists\, k \;:\; k = e : \; T \rangle = T[k := e] \tag{176}$$

- de Morgan:

$$\neg\langle \forall\, i \;:\; R : \; T \rangle = \langle \exists\, i \;:\; R : \; \neg T \rangle \tag{177}$$
$$\neg\langle \exists\, i \;:\; R : \; T \rangle = \langle \forall\, i \;:\; R : \; \neg T \rangle \tag{178}$$

Nesting:

$$\langle \forall\, a,b \;:\; R \wedge S : \; T \rangle = \langle \forall\, a \;:\; R : \; \langle \forall\, b \;:\; S : \; T \rangle\rangle \tag{179}$$
$$\langle \exists\, a,b \;:\; R \wedge S : \; T \rangle = \langle \exists\, a \;:\; R : \; \langle \exists\, b \;:\; S : \; T \rangle\rangle \tag{180}$$

- Empty range:

$$\langle \forall\, k \;:\; \textsc{False} : \; T \rangle = \textsc{True} \tag{181}$$
$$\langle \exists\, k \;:\; \textsc{False} : \; T \rangle = \textsc{False} \tag{182}$$

- Splitting:

$$\langle \forall\, j \;:\; R : \; \langle \forall\, k \;:\; S : \; T \rangle\rangle = \langle \forall\, k \;:\; \langle \exists\, j \;:\; R : \; S \rangle : \; T \rangle \tag{183}$$
$$\langle \exists\, j \;:\; R : \; \langle \exists\, k \;:\; S : \; T \rangle\rangle = \langle \exists\, k \;:\; \langle \exists\, j \;:\; R : \; S \rangle : \; T \rangle \tag{184}$$

[24] As forewarned in [7], the application of a rule is invalid if *(a)* it results in the capture of free variables or release of bound variables; *(b)* a variable ends up occurring more than once in a list of dummies.

References

1. C. Aarts, R.C. Backhouse, P. Hoogendijk, E.Voermans, and J. van der Woude. A relational theory of datatypes, December 1992. Available from `www.cs.nott.ac.uk/˜rcb`.
2. T.L. Alves, P.F. Silva, J. Visser, and J.N. Oliveira. Strategic term rewriting and its application to a *VDM-SL* to SQL conversion. In *FM'05*, volume 3582 of *LNCS*, pages 399–414. Springer-Verlag, 2005.
3. K. Backhouse and R.C. Backhouse. Safety of abstract interpretations for free, via logical relations and Galois connections. *SCP*, 15(1–2):153–196, 2004.
4. R. Backhouse and D. Michaelis. Exercises in quantifier manipulation. In Tarmo Uustalu, editor, *MPC'06, Mathematics of Program Construction 2006*, pages 70–81. Springer LNCS (4014), 2006.
5. R.C. Backhouse. On a relation on functions. In *Beauty is our business: a birthday salute to Edsger W. Dijkstra*, pages 7–18, New York, NY, USA, 1990. Springer-Verlag.
6. R.C. Backhouse. Fixed point calculus, 2000. Summer School and Workshop on *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, Lincoln College, Oxford, UK 10th to 14th April 2000.
7. R.C. Backhouse. *Mathematics of Program Construction*. Univ. of Nottingham, 2004. Draft of book in preparation. 608 pages.
8. R.C. Backhouse, P. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J. van der Woude. Polynomial relators. In *AMAST'91*, pages 303–362. Springer, 1992.
9. R.C. Backhouse and J. Woude. Demonic operators and monotype factors. *Mathematical Structures in Computer Science*, 3(4):417–433, 1993.
10. J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *CACM*, 21(8):613–639, August 1978.
11. L.S. Barbosa and J.N. Oliveira. Transposing partial components — an exercise on coalgebraic refinement. *Theoretical Computer Science*, 365(1):2–22, 2006.
12. L.S. Barbosa, J.N. Oliveira, and A.M. Silva. Calculating invariants as coreflexive bisimulations. In AMAST'08, volume 5140 of *LNCS*, pages 83–99. Springer-Verlag, 2008.
13. R. Bird and O. de Moor. Algebra of Programming. Series in Computer Science. Prentice-Hall International, 1997. C.A.R. Hoare, series editor.
14. A. Bove and P. Dybjer. Dependent types at work, Feb. 2008. Lecture Notes (47p.) for the LerNet Summer School, Piriapolis, Uruguay.
15. K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.
16. Intel Corporation. *Intel Flash File System Core Reference Guide*, October 2004. Doc. Ref. 304436-001.
17. CSK. *The Integrity Checking: Using Proof Obligations*, 2007.
18. A. Cunha, J.N. Oliveira, and J. Visser. Type-safe two-level data transformation. In FM'06 , volume 4085 of *LNCS*, pages 284–289. Springer-Verlag, Aug. 2006.
19. E.W. Dijkstra and C.S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag, New York, NY, USA, 1990.
20. H. Doornbos, R. Backhouse, and J. van der Woude. A calculational approach to mathematical induction. *Theoretical Computer Science*, 179(1–2):103–135, 1997.
21. M.A. Ferreira, S.S. Silva, and J.N. Oliveira. Verifying Intel FLASH file system core specification. In *Modelling and Analysis in VDM: Proceedings of the Fourth* Overture/VDM++ *Workshop at FM'08, 26th of May 2008, Turku, Finland*. University of Newcastle, Computer Science. Technical Report CS-TR-1099, 2008.
22. J. Fitzgerald and P.G. Larsen. Modelling Systems: Practical Tools and Techniques for Software Development . Cambridge University Press, 1st edition, 1998.

23. J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. Validated Designs for Object–oriented Systems. Springer, New York, 2005.
24. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.
25. P.J. Freyd and A. Ščedrov. *Categories, Allegories*, volume 39 of *Mathematical Library*. North-Holland, 1990.
26. M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
27. C.A.R. Hoare. An axiomatic basis for computer programming. *CACM*, 12,10:576–580, 583, October 1969.
28. C.A.R. Hoare and J. Misra. Verified software: theories, tools, experiments — vision of a Grand Challenge project. *Proceedings of IFIP working conference on Verified Software: theories, tools, experiments*, 2005.
29. P. Hoogendijk. *A Generic Theory of Data Types*. PhD thesis, University of Eindhoven, The Netherlands, 1997.
30. D. Jackson. *Software abstractions: logic, language, and analysis*. The MIT Press, Cambridge Mass., 2006. ISBN 0-262-10114-9.
31. B. Jacobs. Introduction to Coalgebra. Towards Mathematics of States and Observations. Draft Copy. Institute for Computing and Information Sciences, Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
32. C.B. Jones. *Systematic Software Development Using VDM*. Series in Computer Science. Prentice-Hall Int., 1990. 1st edition (1986).
33. S.L. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, Cambridge, UK, 2003. Also published as a Special Issue of the Journal of Functional Programming, 13(1) Jan. 2003.
34. R. Joshi and G.J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Asp. Comput.*, 19(2):269–272, 2007.
35. Y. Kawahara. Notes on the universality of relational functors. *Mem. Fac. Sci. Kyushu Univ. (Series A, Mathematics)*, 27(2):275–289, 1973.
36. D. Kozen. On Hoare logic and Kleene algebra with tests. *Trans. Computational Logic*, 1(1):60–76, July 2000.
37. E. Kreyszig. *Advanced Engineering Mathematics*. J. Wiley & Sons, 6th edition, 1988.
38. D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.
39. B. Meyer. Applying ”design by contract”. *IEEE Computer*, 25(10):40–51, 1992.
40. A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In *ICFP ’06*, pages 62–73, New York, NY, USA, 2006. ACM.
41. C. Necco, J.N. Oliveira, and J. Visser. ESC/PF: Static checking of relational models by calculation, 2008. (Submitted).
42. J.N. Oliveira. <<Bagatelle in C arranged for *VDM* SoLo>>. JUCS, 7(8):754–781, 2001.
43. J.N. Oliveira. Constrained datatypes, invariants and business rules: a relational approach, 2004. PUReCafé talk, DI-UM, 2004.5.20, PURE PROJECT (POSI/CHS/44304/2002).
44. J.N. Oliveira. *Calculate databases with ‘simplicity’*, September 2004. Presentation at the *IFIP WG 2.1 #59 Meeting*, Nottingham, UK. (Slides available from the author’s website.).
45. J.N. Oliveira. *Data dependency theory made generic — by calculation*, December 2006. Presentation at the *IFIP WG 2.1 #62 Meeting*, Namur, Belgium.
46. J.N. Oliveira. *Reinvigorating pen-and-paper proofs in VDM: the pointfree approach*, 2006. Presented at the Third OVERTURE Workshop: Newcastle, UK, 27-28 November 2006.
47. J.N. Oliveira. Pointfree foundations for (generic) lossless decomposition, 2008. (Submitted).
48. J.N. Oliveira. *Transforming Data by Calculation*. In *GTTSE 2007*, volume 5235 of *LNCS*, pages 134–192, 2008.

49. J.N. Oliveira and C.J. Rodrigues. Transposing relations: from *Maybe* functions to hash tables. In *MPC'04*, volume 3125 of *LNCS*, pages 334–356. Springer, 2004.
50. J.N. Oliveira and C.J. Rodrigues. Pointfree factorization of operation refinement. In FM'06, volume 4085 of *LNCS*, pages 236–251. Springer-Verlag, 2006.
51. O. Ore. Galois connexions, 1944. Trans. Amer. Math. Soc., 55:493-513.
52. B.C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.
53. V. Pratt. Origins of the calculus of binary relations. In *Proc. of the 7th Annual IEEE Symp. on Logic in Computer Science*, pages 248–254, Santa Cruz, CA, 1992. IEEE Comp. Soc.
54. J.C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing 83*, pages 513–523, 1983.
55. L. Russo. *The Forgotten Revolution: How Science Was Born in 300BC and Why It Had to Be Reborn*. Springer-Verlag, September 2003.
56. 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.
57. J.M. Spivey. *The Z Notation — A Reference Manual*. Series in Computer Science. Prentice-Hall International, 1989. C.A.R. Hoare (series editor).
58. Open Group Technical Standard. Standard for information technology - Portable operating system interface (POSIX). System interfaces. *IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. System Interfaces*, 2004.
59. A. Takano and E. Meijer. Shortcut to deforestation in calculational form. In *Proc. FPCA'95*, 1995.
60. A. Tarski and S. Givant. *A Formalization of Set Theory without Variables*. American Mathematical Society, 1987. AMS Colloquium Publications, volume 41, Providence, Rhode Island.
61. S.D. Vermolen. Automatically discharging VDM proof obligations using HOL. Master's thesis, Radboud University Nijmegen, Computing Science Department, June-August 2007.
62. T.S. Voermans. *Inductive Datatypes with Laws and Subtyping — A Relational Model*. PhD thesis, University of Eindhoven, The Netherlands, 1999.
63. J. Voigtländer. Proving correctness via free theorems: The case of the destroy/build-rule. In Robert Glück and Oege de Moor, editors, *Symposium on Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, Proceedings*, pages 13–20. ACM Press, January 2008.
64. P.L. Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*, pages 347–359, London, Sep. 1989. ACM.
65. Shuling Wang, L.S. Barbosa, and J.N. Oliveira. *A Relational Model for Confined Separation Logic*. In TASE 2008, pages 263–270, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
66. J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.