"Fractal" Types: an Attempt to Generalize Hash Table Calculation

J.N. Oliveira
Dept. de Informática
Universidade do Minho,
Campus de Gualtar
4700 Braga
Portugal

May 14, 1998

1 Introduction

This paper addresses a class of generic program calculations which aim at introducing concurrency into programs by changing data-structure representations. The approach will be restricted to the fairly common situation in which the calculated program (implementation) runs at a level of abstraction lower than that of the starting program (specification). This conforms to the following quotation from [12]:

> *The laws of Section 17.4 dealt with a very general case of data refinement, in which the coupling invariant linking the abstract and concrete states could be anything whatever.*
>
> *This is actually a fairly rare occurrence in everyday program development however: it is much more common for the coupling invariant to be functional from concrete to abstract.* [...]
>
> *The general form for such coupling invariants, called* functional abstractions *is*
>
> $$a = \mathsf{af}\ c \wedge \mathsf{dti}\ c\,, \qquad (17.1)$$
>
> *where* af *is a function, called the* abstraction function *and* dti *is a predicate, in which* a *does not appear, called the* data-type invariant. [...]

We will force abstraction function af to be surjective (epic) so that every abstract value has a valid representation. However, instead of choosing (inventing) particular af and dti for each particular situation, these will follow from the generic laws for data structure calculation which are at the core of the SETS calculus [13, 14, 17, 18].

These laws, which will be introduced in the following section, allow for "polytypic data refinement" in the sense that they are applicable independently of each particular data-structuring functor F [3], for a broad class of such functors.

So, we depart from the "invent & verify" techniques typical of the formal software development method tradition in two ways: first, we want to calculate rather than prove; second, we would like to develop generic, datatype-parametric paradigms about generic components by generalizing particular programming solutions.

This paper describes our efforts to generalize a well-known programming device — the hash-table data manager [6]. The main idea is to show that this device has some potential for concurrency and might be converted into a broader programming paradigm. Some difficulties we have encountered will be presented.

2 Abstraction Invariant Generic Calculation

SETS stands for "**Set**-based specification", where $\mathbf{Set}$ is the category of functions on (countable) sets. It is a calculus of *abstraction invariants* as understood in the above quotation [12]. But instead of writing and reason about predicates such as *(17.1)*, in predicate logic, inequations of the form

$$A \trianglelefteq_{\mathsf{af}} B_{\mathsf{dti}} \tag{1}$$

are recorded meaning that arrow $A \xleftarrow{\mathsf{af}} B_{\mathsf{dti}}$ exists and is epic. But, what does it mean, in this notation, to subscript a set B by an invariant dti?

In set-theoretical terms, $B_{\mathsf{dti}} = \{b \in B \mid \ \mathsf{dti}\, b\}$, *e.g.* $I\!N_{\lambda n.2 \leq n \leq 4} = \{2, 3, 4\}$. But, for generality, one should replace subsets by sub-objects [10]. Therefore, given a predicate $B \xrightarrow{\phi} 2$, we will define B_ϕ as the "greatest" object which (up to isomorphism) makes the following diagram to commute, where i is monic [1]:

$$\begin{tikzcd} B_\phi \arrow[r, "!_{B_\phi}"] \arrow[d, "i"'] & 1 \arrow[d, "\underline{\mathrm{TRUE}}"] \\ B \arrow[r, "\phi"'] & 2 \end{tikzcd} \qquad \phi(i\, b) = \mathrm{TRUE} \quad , \quad b \in B_\phi \tag{2}$$

Therefore, by B_ϕ we understand the *pullback* of the "corner" of arrows ϕ and $\underline{\mathrm{TRUE}}$. Clearly, $B_{\underline{\mathrm{TRUE}}} \cong B$ holds. It follows that, regarded as an ordering on $\mathbf{Set}$-objects, $\trianglelefteq$ is reflexive (up to isomorphism):

$$B \trianglelefteq_{id_B} B_{\underline{\mathrm{TRUE}}} \tag{3}$$

Sticking an invariant ϕ to the domain of an arrow $B \xrightarrow{f} A$ has the effect of restricting f to ϕ, that is, of enforcing ϕ as a precondition:

$$\begin{tikzcd} B_\phi \arrow[r, "f|\phi"] \arrow[d, "i"'] & A \arrow[d, "id_A"] \\ B \arrow[r, "f"'] & A \end{tikzcd} \tag{4}$$

The converse effect of forcing some $C \xrightarrow{g} B$ to produce its output "inside" B_ϕ, that is, of satisfying invariant ϕ on its output, is expressed also by a pullback, that of corner $\phi \circ g$ and $\underline{\mathrm{TRUE}}$,

$$\begin{tikzcd} C_{\phi \circ g} \arrow[r, "g'"] \arrow[d, "i'"'] & B_\phi \arrow[r, "!_{B_\phi}"] \arrow[d, "i"] & 1 \arrow[d, "\underline{\mathrm{TRUE}}"] \\ C \arrow[r, "g"'] & B \arrow[r, "\phi"'] & 2 \end{tikzcd} \qquad \text{where } g' = g|(\phi \circ g) \tag{5}$$

Then the effect of chaining two facts $A \trianglelefteq_f B_\phi$ and $B \trianglelefteq_g C_\gamma$ will be expressed by

$$\begin{tikzcd} (C_\gamma)_{\phi \circ g} \arrow[r, "g'"] \arrow[d, "i'"'] & B_\phi \arrow[r, "f"] \arrow[d, "i"] & A \arrow[r, "!_A"] & 1 \arrow[d, "\underline{\mathrm{TRUE}}"] \\ C_\gamma \arrow[r, "g"'] & B \arrow[rr, "\phi"'] & & 2 \end{tikzcd} \qquad \text{where } g' = g|(\phi \circ g) \tag{6}$$

[1] i performs the "inclusion" of B_ϕ into B and 2 is $\mathbf{Set}$'s sub-object classifier, that is, the Booleans. $\underline{\mathrm{TRUE}}$ means the *everywhere true* Boolean morphism. In general, given some $b \in B$, $\underline{b}_A : A \longrightarrow B$ is natural transformation $\underline{b}_A a = b$, for all $a \in A$ ("the everywhere b" constant morphism). Subscript A is dropped wherever implicit in the context.

for g and f both epic. Therefore, $\trianglelefteq$ is a preorder on $\mathbf{Set}$-objects: it is reflexive (3), it is transitive [2],

$$A \trianglelefteq_f B_\phi \wedge B \trianglelefteq_g C_\gamma \quad \Rightarrow \quad A \trianglelefteq_{f \circ g} C_{\gamma \wedge \phi \circ g} \tag{7}$$

and $\trianglelefteq$-antisymmetry induces set-theoretical isomorphism [3]:

$$A \trianglelefteq_f B \wedge B \trianglelefteq_g A \quad \Rightarrow \quad A \cong B$$

Following [9], it has become standard in the program calculation community to regard a (parametric) "data sort" as an (endo) functor in some category — $\mathbf{Set}$, in our case. In this context, the following theorems are central to abstraction invariant structural calculation.

Theorem 1 (Functorial $\trianglelefteq$-monotonicity in $\mathbf{Set}$). *Let* F *be a* $\mathbf{Set}$ *endofunctor. If*

$$A \trianglelefteq_f B_\phi \tag{8}$$

holds, then

$$\mathsf{F}\,A \trianglelefteq_{\mathsf{F}\,f} \mathsf{F}\,B_{(\mathsf{F}\,\phi)} \tag{9}$$

Proof. (8) means the existence of epic $A \xleftarrow{f} B_\phi$. In $\mathbf{Set}$, all functors preserve epics [4], that is, $\mathsf{F}A \xleftarrow{\mathsf{F}f} \mathsf{F}(B_\phi)$ is epic [5]. Finally, the fact that $\mathsf{F}(B_\phi) \cong (\mathsf{F}\,B)_{(\mathsf{F}\,\phi)}$ holds is shown in theorem 3. $\square$

Theorem 2 (Invariant "pulling"). *Wherever* $B \trianglelefteq_g C_\gamma$ *holds and* $B \xrightarrow{\phi} 2$ *is an invariant on* B*, then* ϕ *is "pulled" to* C *as follows:*

$$B_\phi \trianglelefteq_g C_{\gamma \wedge \phi \circ g}$$

Proof. Check diagram (6). Invariant $\phi \circ g$ is the "least" impact of ϕ at C-level. $\square$

Theorem 3 (Functorial sub-objects). *Let* F *be a* $\mathbf{Set}$ *regular endofunctor and consider* B_ϕ *from diagram (2). Then,* $\mathsf{F}(B_\phi) \cong (\mathsf{F}\,B)_{(\mathsf{F}\,\phi)}$.

Proof. (From [1].) Apply F to diagram (2) and introduce $(2,\ 2 \xleftarrow{\square_{\mathsf{F}}} \mathsf{F}\,2\ )$, the F-algebra which reduces an F-structure of Booleans to a single Boolean via the $\wedge$ logical connective:

$$\begin{tikzcd}
\mathsf{F}(B_\phi) \arrow[r, "\mathsf{F}(!_{B_\phi})"] \arrow[d, "\mathsf{F}\,i"'] & \mathsf{F}\,1 \arrow[d, "\mathsf{F}\,\underline{\mathrm{TRUE}}"] & \\
\mathsf{F}\,B \arrow[r, "\mathsf{F}\,\phi"'] & \mathsf{F}\,2 \arrow[r, "\square_{\mathsf{F}}"'] & 2
\end{tikzcd}$$

This raises pullback square

$$\begin{tikzcd}
\mathsf{F}(B_\phi) \cong (\mathsf{F}\,B)_{\square_{\mathsf{F}} \circ (\mathsf{F}\,\phi)} \arrow[rr, "!_{\mathsf{F}(B_\phi)}"] \arrow[d, "\mathsf{F}\,i"'] & & 1 \arrow[d, "\underline{\mathrm{TRUE}}"] \\
\mathsf{F}\,B \arrow[r, "\mathsf{F}\,\phi"'] & \mathsf{F}\,2 \arrow[r, "\square_{\mathsf{F}}"'] & 2
\end{tikzcd}$$

So, $\mathsf{F}(B_\phi) \cong (\mathsf{F}\,B)_{\square_{\mathsf{F}} \circ (\mathsf{F}\,\phi)}$. To avoid notation cluttering we abbreviate $(\mathsf{F}\,B)_{\square_{\mathsf{F}} \circ (\mathsf{F}\,\phi)}$ to $(\mathsf{F}\,B)_{(\mathsf{F}\,\phi)}$. $\square$

---

[2] Note that a double subscript, such as in $(C_\gamma)_{\phi \circ g}$, has the effect of a conjunction of subscripts, *cf.* $C_{\wedge \circ \langle \gamma, \phi \circ g \rangle}$ [10]. Conjunctions are abbreviated in the same way as in [10], *i.e.* $\wedge \circ \langle h, k \rangle$ to $h \wedge k$. Also note that, strictly speaking, in (7) we should write $f \circ g'$ instead of $f \circ g$.

[3] As a rule, we will drop the $\underline{\mathrm{TRUE}}$ invariant.

[4] This is not true in every category, see *e.g.* [10].

[5] This result is due to [1, 20]. It was introduced earlier by [14], but in a restricted version.

Table 1: Summary of Functorial Calculus.

$\mathsf{F}\,X$	$\mathsf{F}\,f$	$\mathsf{F}\,\phi$
X	f	ϕ
C	id_C	$\lambda c.\text{TRUE}$
$\mathcal{P}\,X$	$\mathcal{P}\,f = \lambda s.\{f\,x \mid\ x \in s\}$	$\mathcal{P}\,\phi = \lambda s.\forall b \in s : \phi\, b$
$X^\star$	$f^\star = \lambda l.[f\,x \mid x \leftarrow l]$	$\phi^\star = \lambda l.\forall 1 \leq i \leq length\, l : \phi(l\, i)$
X^C	$\begin{array}{rcl} f^C & = & \lambda\sigma.f \circ \sigma \\ & = & \lambda\sigma.\begin{pmatrix} c \\ f(\sigma\, c) \end{pmatrix}_{c \in C} \end{array}$	$\phi^C = \lambda\sigma.\forall c \in C : \phi(\sigma\, c)$
$C \rightharpoonup X$	$\begin{array}{rcl} C \rightharpoonup f & = & \lambda\sigma.f \circ \sigma \\ & = & \lambda\sigma.\begin{pmatrix} c \\ f(\sigma\, c) \end{pmatrix}_{c \in dom\, \sigma} \end{array}$	$C \rightharpoonup \phi = \lambda\sigma.\forall x \in rng(\sigma) : \phi\, x$
$\mathsf{G}\,X \times \mathsf{H}\,X$	$\mathsf{G}\,f \times \mathsf{H}\,f$	$\lambda(x,y).(\mathsf{G}\,\phi)x \wedge (\mathsf{H}\,\phi)y$
$\mathsf{G}\,X + \mathsf{H}\,X$	$\mathsf{G}\,f + \mathsf{H}\,f$	$\lambda x.\begin{cases} x = i_1\, a & \Rightarrow\quad (\mathsf{G}\,\phi)a \\ x = i_2\, b & \Rightarrow\quad (\mathsf{H}\,\phi)b \end{cases}$
$\mathsf{G}\,X + 1$	$\lambda x.\begin{cases} x = i_2\, NIL & \Rightarrow\quad x \\ x = i_1\, a & \Rightarrow\quad i_1((\mathsf{G}\,f)a) \end{cases}$	$\lambda x.\begin{cases} x = i_2\, NIL & \Rightarrow\quad \text{TRUE} \\ x = i_1\, a & \Rightarrow\quad (\mathsf{G}\,\phi)a \end{cases}$

The scope of application of this result is wide and extends to bifunctors [14]. Table 1 — where $\mathsf{F}\,\phi$ abbreviates $\Box_\mathsf{F} \circ (\mathsf{F}\,\phi)$ — provides an account of structural abstraction invariant synthesis for the most common functors in SETS specification. By $A^\star$ (and $f^\star$) we mean the type functor associated with base bifunctor $\mathsf{F}(A,X) \stackrel{\text{def}}{=} 1 + A \times X$, that is, we are talking about the carrier $A^\star = \bigcup_{n \geq 0} A^n$ of the initial algebra solution to equation $X \cong 1 + A \times X$. By $\mathcal{P}\,A$ we mean $\{s \mid\ s \subseteq A\}$, for a finite A, which is the carrier of another (non-initial) F-algebra. By $A \rightharpoonup B$ we mean, for a finite A, the set of all partial finite maps from A to B,

$$A \rightharpoonup B = \bigcup_{K \subseteq A} B^K$$

which is the carrier of a non-initial solution to the equation of base functor $\mathsf{G}(A,B,X) \stackrel{\text{def}}{=} 1 + A \times B \times X$. Both $\mathcal{P}\,f$ and $g \rightharpoonup f$ (for monic g) are formalized by appropriate hylomorphisms (see [18] for details) which we abbreviate to

$$\lambda s.\{f\,a \mid\ a \in s\}$$

and

$$\lambda\sigma.\begin{pmatrix} g\,a \\ f(\sigma\,a) \end{pmatrix}_{a \in dom\,\sigma}$$

respectively. Functors $\mathcal{P}\,(\_)$ and $(\_) \rightharpoonup (\_)$ will be central to the hash-table calculations to be presented in the sequel.

3 Local Invariants

As a rule, data refinement entails the introduction of datatype invariants required for an implementation to be faithful to a specification. As described above, the SETS calculus is based on a "redundancy" preorder ($\trianglelefteq$) which guarantees data representation while saving as many datatype invariants as technically possible [6].

This strategy has the advantage of preventing from implementations "stronger" than necessary, but ignores other aspects of data refinement which are relevant in practice. On the one hand, the proper starting specification may involve datatypes affected by 'local' invariants which record real-life conventions, laws, rules, norms or natural constraints [15]. On the other hand, local invariants may be introduced throughout refinement which have to do not with representation faithfulness but rather with other facets of refinement such as, for instance, *efficiency*.

[6]This is guaranteed by pullback constructions (5) and (6).

We will focus on the latter case. What is the scope of local invariant introduction for efficiency purposes? Let us see an example: a basic reification fact is

$$\mathcal{P}\, A \quad \trianglelefteq_{elems} \quad A^\star \tag{10}$$

meaning that finite lists implement finite sets [7]. However, for efficiency we may be interested in implementing $\mathcal{P}\, A$ by lists in $A^\star$ which are ordered by some total order on A, which is irrelevant *wrt.* the abstraction function $elems$.

Recall f in inequation (8). It may happen that f is "too much" non-injective and there is room for some economy in data redundancy. For instance, the $elems$ function establishes law (10) in a fairly redundant way: sequences with no repeated elements will still be able to represent finite sets.

In general, given inequation (8), it may be the case that $A \trianglelefteq_{f|\beta} (B_\phi)_\beta$ holds, that is, not only f but also $f|\beta$ are epic. Predicate β can be as strong as we wish provided that the cardinality of $B_{\phi\wedge\beta}$ is greater or equal to the cardinality of A, since $f|\beta$ should be epic. In the "limit" one has

$$A \cong_{f|\phi} B_{\phi\wedge\beta} \tag{11}$$

that is, the sub-object at implementation-level is an isomorphic copy of the specification itself.

This situation (in which there exists a unique inverse representation function) is fairly common in data refinement and embodies the hash-table calculation sketched in the following section.

4 From (Power)sets to Hash Tables

This section presents a summary of a former exercise [15] on SETS-based derivation of the *hash table* implementation of a finite collection of data.

Let A be an unbounded but finite domain of data which we want to record and maintain in a database file. We will regard $\mathcal{P}\, A$ as a convenient (however primitive!) model for the database file itself. That is, our notion of an abstract database file "re-uses" the algebra of finite sets:

$$\begin{array}{|c|c|}\hline \begin{array}{rcl} empty & : & \longrightarrow \mathcal{P}\, A \\ empty & \stackrel{\mathrm{def}}{=} & \emptyset \end{array} & \begin{array}{rcl} insert & : & A \times \mathcal{P}\, A \longrightarrow \mathcal{P}\, A \\ insert(a,\sigma) & \stackrel{\mathrm{def}}{=} & \sigma \cup \{a\} \end{array} \\ \hline \begin{array}{rcl} find & : & A \times \mathcal{P}\, A \longrightarrow 2 \\ find(a,\sigma) & \stackrel{\mathrm{def}}{=} & (a \in \sigma) \end{array} & \begin{array}{rcl} remove & : & A \times \mathcal{P}\, A \longrightarrow \mathcal{P}\, A \\ remove(a,\sigma) & \stackrel{\mathrm{def}}{=} & \sigma - \{a\} \end{array} \\ \hline \end{array} \tag{12}$$

First, we concentrate on the hash-table implementation of carrier set $\mathcal{P}\, A$. Let $n \in I\!N$ be the number of entries of our intended hash table, and $A \xrightarrow{H} n$ be some fixed hash function [8]. Usually, n is much smaller than the cardinal of A, that is, H is clearly non-monic. Our target is to use $\trianglelefteq$-reasoning in order to convert $\mathcal{P}\, A$ into something we can identify as an acceptable model of a hash table. Our reasoning will start from fact $A \trianglelefteq_{\pi_2} n \times A$, which instantiates $A \trianglelefteq_{\pi_2} B \times A$ — a SETS-inequation expressing the fact that a data domain can always be refined by adding extra information to it. We proceed by imposing the following local invariant on $n \times A$,

$$\phi(i,a) \stackrel{\mathrm{def}}{=} i = H\, a \tag{13}$$

forcing every datum $a \in A$ to be coupled with its hash-index $H\, a$. Since $\pi_2|\phi$ is epic on A, we have, by (11):

$$A \trianglelefteq_{\pi_2|\phi} (n \times A)_\phi \tag{14}$$

[7] $elems$ is the (epic) catamorphism $(\!|\alpha|\!)_{\mathsf{F}}$ [3], for $\mathsf{F}\, X = 1 + A \times X$ and $\alpha = [\,\emptyset, \lambda(a,s).\{a\} \cup s\,]$.

[8] In SETS notation we write n to denote the initial segment of $I\!N$ of size n, that is, the set $\{1, 2, \ldots, n\}$. For other idiosyncrasies of the notation see *e.g.* [18].

By $\trianglelefteq$-monotonicity (theorem 1) of the powerset functor $\mathcal{P}\,(\_)$, we obtain, from (14),

$$\mathcal{P}\,A \trianglelefteq_{f_1} (\mathcal{P}\,(n \times A))_{\Phi_1} \tag{15}$$

where $f_1 = \mathcal{P}\,(\pi_2|\phi) = \lambda s.\{a \in A \mid (i,a) \in s\}$ and $\Phi_1 = \mathcal{P}\,\phi = \lambda s.(\forall (i,a) \in s : i = H\,a)$.

The remainder of the calculation requires two basic $\mathbf{Set}$ isomorphisms: $\mathcal{P}\,A \cong 2^A$ and the "currying" law of exponentiation:

$$C^{B\times A} \quad \cong \quad (C^A)^B \tag{16}$$

Putting these together, we infer (for $C = 2$),

$$\mathcal{P}\,(B \times A) \quad \cong_{uncurry} \quad (\mathcal{P}\,A)^B$$

where $uncurry\,t \stackrel{\mathrm{def}}{=} \{(b,c) \mid b \in B \wedge c \in t\,b\}$, see [15] for details. Therefore:

$$\mathcal{P}\,(n \times A) \cong (\mathcal{P}\,A)^n \tag{17}$$

Pulling Φ_1 through $uncurry$ we obtain, according to theorem 2

$$(\mathcal{P}\,(n \times A))_{\Phi_1} \cong_{f_2} ((\mathcal{P}\,A)^n)_{\Phi_2} \tag{18}$$

where $f_2 = uncurry = \lambda t.\{(i,a) \mid i \in n \wedge a \in t\,i\}$ and

$$\begin{aligned} \Phi_2 &= \Phi_1 \circ uncurry \\ &= \lambda t.\forall i \in n : (\forall a \in t\,i : i = H\,a) \end{aligned} \tag{19}$$

It is easy to show that $((\mathcal{P}\,A)^n)_{\Phi_2}$ above — *i.e.* the "array" structure [9] $(\texttt{array}\ n\ \texttt{of}\ \mathcal{P}\,A)_{\Phi_2}$ — is in fact a model for hash tables. According to Φ_2 (19), every set in the range of such "arrays" contains *synonyms* [6]. As is easy to prove [15], all such sets are mutually disjoint. Therefore, every $db \in \mathcal{P}\,A$ is partitioned into n-many, disjoint "collision segments", each one addressed by the relevant hash index computed by H. We have thus achieved the intended "hashing effect" [10].

However, the task of reifying $\mathcal{P}\,A$ "recurs" in the co-domain of $\texttt{array}\ n\ \texttt{of}\ \mathcal{P}\,A$ — a kind of "fractal" effect on this kind of data structuring. This problem is identified in the literature under the headings *overflow handling* or *collision handling* [6]. Several ways out are known, including re-hashing [15]. Should $\mathcal{P}\,A$ be realized by $A^\star$ (10), the expected target data structure in *e.g.* PASCAL,

```
type  Collision   =  record
                      Synonym:  A;
                      Next:  Bucket
                     end;
      Bucket      =  ^ Collision;
      HashTable   =  array[1..n] of Bucket;
```
(20)

will be calculated thanks to a generic result in SETS [14, 18] which represents recursive data structures by introducing pointers.

Space limitations preclude the presentation of the calculation of the operations at hash-table level (see [15] for details):

[9] In the SETS folklore it is common to regard exponentials A^B as (mathematical models of) "à la PASCAL" $\texttt{array}\ B\ \texttt{of}\ A$ data structures, for finite B [14].

[10] We claim the outcome of this synthesis of hash-tables from sets to be more constructive and easier to grasp than the development found in, *e.g.*, [19].

$$\begin{array}{rcl} empty_2 & : & \longrightarrow ((\mathcal{P}\,A)^n)_{\Phi_2} \\ empty_2 & = & \left(\begin{array}{c} i \\ \emptyset \end{array}\right)_{i\in n} \end{array}$$	$$\begin{array}{rcl} insert_2 & : & A \times ((\mathcal{P}\,A)^n)_{\Phi_2} \longrightarrow ((\mathcal{P}\,A)^n)_{\Phi_2} \\ insert_2(a,t) & = & t \dagger \left(\begin{array}{c} H\,a \\ insert(a, t(H\,a)) \end{array}\right) \end{array}$$
$$\begin{array}{rcl} find_2 & : & A \times ((\mathcal{P}\,A)^n)_{\Phi_2} \longrightarrow 2 \\ find_2(a,t) & = & find(a, t(H\,a)) \end{array}$$	$$\begin{array}{rcl} remove_2 & : & A \times ((\mathcal{P}\,A)^n)_{\Phi_2} \longrightarrow ((\mathcal{P}\,A)^n)_{\Phi_2} \\ remove_2(a,t) & = & t \dagger \left(\begin{array}{c} H\,a \\ remove(a, t(H\,a)) \end{array}\right) \end{array}$$

It is clear, however, that each low-level operation resorts to its specification "tuned" to the relevant collision bucket — something to be expected from the "fractal" structure of the calculated data structure itself: $\mathcal{P}\,A \trianglelefteq (\mathcal{P}\,A)^n$.

5 Going Generic

There is no *a priori* reason to regard the *hash effect* as a privilege of "powersets". So, it may be worthwhile to extend the above constructive calculation to a wider range of data structures. In what follows we seek to characterize the class of $\mathbf{Set}$-endofunctors T (parameterized types) which can be refined via the *hashing paradigm*,

$$\mathsf{T}A \quad \trianglelefteq \quad (\mathsf{T}A)^n \tag{21}$$

that is to say, which are, in some way, "hashable". The particular case $\mathsf{T}X = \mathcal{P}\,X$ is used as a guideline.

A general criterion for deciding which data-structures can be refined by "hashing them" should arise from a suitable generalization of the "hashing step". In the first place, we have seen that a condition necessary for this hashing-step to be applicable to generic T is

$$\mathsf{T}(n \times A) \quad \cong \quad (\mathsf{T}A)^n \tag{22}$$

— *cf.* (17) — for $n > 0$, where n is the codomain of some fixed hash function $H : A \longrightarrow n$. Secondly, H is the basis for a strategic local invariant introduction

$$\mathsf{T}(\lambda(i,a).i = H\,a)$$

structurally forcing every datum $a \in A$ to be coupled with its hash-index $H\,a$.

From "currying" law (16) we find exponentials readily available for hashing, for $\mathsf{T}\,X = B^X$. So, the "array" data-structure B^A (`array` A `of` B) should be hashable, for a finite A. However, we shall need to investigate the implications of the contravariance of this functor in the calculations. Before we do this, we present another candidate for T: since we know that the following isomorphism holds,

$$(C \times A) \rightharpoonup B \quad \cong_f \quad (A \rightharpoonup B)^C \tag{23}$$

for

$$f \quad = \quad \lambda\sigma. \bigcup_{c\in C} \left(\begin{array}{c} (c,a) \\ (\sigma\,c)a \end{array}\right)_{a\in dom\,(\sigma\,c)} \tag{24}$$

as a consequence of

$$A \rightharpoonup B \quad \cong \quad (B+1)^A \tag{25}$$

then

$$(n \times A) \rightharpoonup B \quad \cong \quad (A \rightharpoonup B)^n \tag{26}$$

holds and so $\mathsf{T}\,X = X \rightharpoonup B$ is hashable.

Both candidates for T above are contravariant. Do we need to recalculate everything? No, because the only instance of a morphism $\mathsf{T}\,f$ happens for $f = \pi_2|\phi$ and, thanks to (13), $\pi_2|\phi$ is not only epic but also monic. Since the covariant counterpart of T is a well-defined functor for monics, we rely on this, as can be checked for instance $\mathsf{T}\,X = X \rightharpoonup B$ [11]: theorem 3 holds,

$$\begin{array}{rcl} (A_\phi) \rightharpoonup B & = & \{f \in A \rightharpoonup B \mid \mathit{dom}\,f \subseteq A_\phi\} \\ & = & \{f \in A \rightharpoonup B \mid \forall a \in \mathit{dom}\,f : \phi\,a\} \\ & = & (A \rightharpoonup B)_{\phi \rightharpoonup B} \end{array}$$

and so, for f as defined in (24),

$$\begin{array}{rcl} A \rightharpoonup B & \trianglelefteq_{(\pi_2|\phi) \rightharpoonup B} & ((n \times A) \rightharpoonup B)_{\phi \rightharpoonup B} \\ & \cong_f & (A \rightharpoonup B)^n_{\Phi_2} \end{array}$$

where

$$\begin{array}{rcl} \Phi_2 & = & (\phi \rightharpoonup B) \circ f \\ & = & \lambda\sigma.\forall x \in \mathit{dom}\,(f\,\sigma) : \phi\,x \\ & = & \lambda\sigma.\forall i \in n : (\forall a \in \mathit{dom}\,(\sigma\,i) : i = H\,a) \end{array}$$

etc. etc.

6 Discussion

The "fractal" qualifier in the title arises from (22), the property which enables some parametric data-structure T to become factored into a collection of n-similar (sub) structures — a kind of "fractal" effect on data structuring. This effect is mirrored at functional level: the calculation of an abstract operation *op* defined at $\mathsf{T}A$-level — *e.g.* $find(a,t)$ — leads to an operation *op′* which, at $(\mathsf{T}A)^n$-level, is "tuned" to the fractal (collision) substructure selected by hash-index $H\ a \in n$, — *e.g.* $find_2(a,t) = find(a, t(H\ a))$.

Our interest in such-called fractal types arises from their potential for concurrency. Since

$$(\mathsf{T}A)^n \quad \cong \quad \underbrace{\mathsf{T}A \times \ldots \times \mathsf{T}A}_{n}$$

hashing can be viewed as a step in the *horizontal refinement* [5] of a single state-based software component into a network of n "smaller" components (each behaving like its ancestor), which can be regarded as truly concurrent processes.

How much "concurrency" is gained with this kind of *horizontal refinement*? Some, in particular due to the local invariant we have imposed relative to a fixed hash function H. Extra behaviour arises from the fact that some operations on disjoint collision sub-structures are independent and can overlap in time (*e.g.* the operation of adding a synonym of hash index i can proceed in parallel with another one which is removing a synonym of hash index j, for $i \neq j$). Clearly, regarding the hash-effect as mere data optimization is a limited view of hashing. Instead of "data hashing" one should talk about "process hashing" [16].

Hashing (*i.e.*, fractal data-refinement) is, in fact, a common device in service or process management. Fractal data-structures can be identified in the formal specification of a railway system presented in [4] [12]. But the trivial illustration which follows will present a problem: suppose that service in a small post-office is accomplished by a single clerk. One queue is formed which guarantees first-in-first-out service. Suppose now that, due to too much

[11] See *e.g.* [11] for the implications of contravariance in program calculation.
[12] Cf. section 4.1 (*Distributing a map*).

service, this post-office "refines itself" into a "hashed" configuration: n clerks become available, each one at a desk labelled with a special service, *e.g.* stamps, parcels, telephone *etc.*. Now n concurrent queues will be formed, one per clerk. When entering the post-office, one just has to "apply the hash-function" to one's target service, thus joining the relevant (collision) queue.

In this example, the fractal type is $\mathsf{T}A = A^\star$ (ie, $list\ A$) and, in fact, we apparently have

$$(n \times A)^* \quad \cong \quad (A^*)^n \tag{27}$$

since both sides of (27) have the same cardinality ($\aleph_0$). Yet intuition may tell us that A^* is a "non-hashable" data-structure, for it seems impossible to reconstruct, from the n sequences on the right-hand side of (27), the original interleaving of such sequences [13]. This calls for further research and discussion.

7 Topics for the Workshop

Besides gathering some input concerning the problem raised above by (27), we would like to present a truly generic characterization of what we mean by a fractal-type $\mathsf{T}A$ and, in particular, a generic specification and (hash-table like) refinement of its kernel functionality (insert, remove, update, find), suitably parameterized by T.

Furthermore, we would like to discuss in what proportion data structures interfere with behaviour, distribution and concurrency calculation [2], in a kind of follow up of [8]. It should be possible to calculate the behaviour of the *horizontal refinement* of an algebra induced by transformation of its carrier. Many laws of SETS work in this direction, particularly the ones which "push the $\times$ construct outwards" [14], *e.g.*

$$\begin{array}{rcl} (\Pi_{i \in n} B_i)^\star & \trianglelefteq_{zip} & \Pi_{i \in n}({B_i}^\star) \\ A \rightharpoonup (\Pi_{i \in n} B_i) & \trianglelefteq_{\bowtie} & \Pi_{i \in n}(A \rightharpoonup B_i) \end{array}$$

because they help in factorizing complex state structures into collections of simpler structures which may have (some) independent behaviour. This is what we might call "column-wise" concurrency, since the data-structure is split in the vertical dimension.

The hash-table paradigm leads to an orthogonal way of calculating concurrent programs, which works "row-wise", *cf.*

$$\begin{array}{rcl} (n \times A) \rightharpoonup B & \cong & (\sum_{i \in n} A) \rightharpoonup B \\ & \trianglelefteq & \Pi_{i \in n}(A \rightharpoonup B) \end{array}$$

something we might call "co-concurrency", due to the presence of the coproduct. This kind of refinement is akin to vector processing and divide and conquer algorithms.

In the first case, we have $\mathsf{T}(\Pi_{i \in n} B_i) \trianglelefteq \Pi_{i \in n}(\mathsf{T}\, B_i)$ and many functors work in this direction, *cf. e.g.* the polytypic function zip in [7]. It is the second class of functors, those such as $\mathsf{T}(\sum_{i \in n} A_i) \trianglelefteq \Pi_{i \in n}(\mathsf{T}\, A_i)$, which has been — perhaps — less studied. Fractal types belong to this class.

References

[1] L. S. Barbosa. A note on SETS functorial foundations, March 1997. DI/UM Internal Research Note.

[2] L. S. Barbosa. *Um Cálculo para Reificação de Processos*. University of Minho, 1998. Ph. D. thesis (in preparation).

[13] A surjection can be easily found from the left-hand side of (27) to the right-hand side,

$$\lambda l. \left(\begin{array}{c} i \\ [(i, \pi_2\, x) \mid x \leftarrow l \wedge (\pi_1\, x) = i] \end{array} \right)_{i \in n}$$

It is the opposite surjection which is difficult to define.

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

[4] C. George. Distributed train rescheduling. Technical Report 42, UNU/IIST, April 1995. (Draft).

[5] J. A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16–28, 1986.

[6] E. Horowitz and S. Sahni. *Fundamentals of Data Structures*. Computer Software Engineering Series. Pitman, 1978. E. Horowitz (Ed.).

[7] P. Jansson and J. Jeuring. Polylib — a library of polytypic functions. In *Workshop on Generic Programming (WGP'98), Marstrand, Sweden*, 1998.

[8] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[9] E. G. Manes and M. A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1986. D. Gries, series editor.

[10] C. McLarty. *Elementary Categories, Elementary Toposes*. Oxford Logic Guides n.º 21. Calendron Press, Oxford, 1^{st} edition, 1995.

[11] E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In S. Peyton Jones, editor, *Proceedings of Functional Programming Languages and Computer Architecture (FPCA95)*, 1995.

[12] C. Morgan. *Programming from Specification*. Series in Computer Science. Prentice-Hall International, 1990. C. A. R. Hoare, series editor.

[13] J. N. Oliveira. A reification calculus for model-oriented software specification. *Formal Aspects of Computing*, 2(1):1–23, April 1990.

[14] J. N. Oliveira. Software reification using the SETS calculus. In *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. Springer-Verlag, 8–10 January 1992. (Invited paper).

[15] J. N. Oliveira. Hash tables — a case study in $\trianglelefteq$-calculation. Technical Report DI/INESC 94-12-1, DI/INESC, U. Minho, December 1994.

[16] J. N. Oliveira. Can distribution be (statically) calculated?, 1997. Seminar presented at UNU/IIST, Macau, May 16th, 1997.

[17] J. N. Oliveira. *Métodos Formais de Programação*. University of Minho, 4^{th} edition, 1997. Textbook (489 p. in Portuguese [14]). English version under preparation at the time of writing.

[18] J. N. Oliveira. A data structuring calculus and its application to program development, May 1998. Lecture Notes of M.Sc. Course (150 p.[15]). Maestria em Ingeneria del Software, Departamento de Informatica, Facultad de Ciencias Fisico-Matematicas y Naturales, Universidad de San Luis, Argentina.

[19] Peter Pepper, Ralph Betschko, Sabine Dick, and Klaus Didrich. Realizing sets by hash tables. In Manfred Broy and Stefan Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, number 1009 in LNCS. Springer, 1995.

[20] C. J. Rodrigues. A very general monotonicity result in SETS, July 1997. DI/UM Internal Research Note.

[14] 846K gzipped PS file available from `http://www.di.uminho.pt/~jno/ps/mfp.ps.gz`.

[15] 390K gzipped PS file available from `http://www.di.uminho.pt/~jno/ps/sanl98.ps.gz`.