Formal Aspects of Computing (1997) 3: 1–000
© 1997 BCS

Hash Tables: A Case Study in SETS-Calculation

J.N. Oliveira

DI/INESC
Universidade do Minho,
Gualtar,
4700 Braga
Portugal

Keywords: Program calculi, abstraction invariants, reification cook-books

Abstract. This paper describes the reasoning carried out in the SETS reification calculus in order to formally calculate the *hash table* implementation of a finite collection of data.
This exercise is a suggestive illustration on handling so-called "ad hoc" data type invariants, that is, constraints not implied by the reification process which are enforced into an implementation to obtain extra advantages (typically, efficiency).
The paper shows, in a formal way, that hash tables are an instance of a class of implementation where concern for efficiency precedes that of data representativeness. The reasoning carried out in the paper includes a final generalization of the "hashing construction" which provides some insight into calculating distribution and concurrency in SETS.

1. Introduction

The purpose of this paper is to describe the process of "calculating" the *hash table* [Wir76, HS78] implementation of an abstract database file, as an example of handling so-called *"ad hoc" data type invariants* in the SETS program calculus [Oli90, Oli92]. For economy of presentation, some acquaintance with these references is assumed and only a synopsis of SETS is provided.

This work is part of a research plan aimed at building a "reification cook-book", *i.e.* a repository of (re-usable) standard software solutions taken from the literature

Correspondence and offprint requests to: J.N. Oliveira

but formally analysed, classified [1] and justified by using the SETS calculus, eventually available in machine readable form.

1.1. Hashing

Hash tables are well known data structures [Wir76, HS78] whose purpose is to efficiently combine the advantages of both static and dynamic storage of data. Static structures such as *arrays* provide random access to data but have the disadvantage of filling too much primary storage. Dynamic, *pointer*-based structures (*e.g.* search lists, search trees *etc.*) are more versatile with respect to storage requirements but access to data is not as immediate.

The idea of *hashing* is suggested by the informal meaning of the term itself: a large database file is "hashed" into as many "pieces" as possible, each of which is randomly accessed. Since each sub-database is smaller than the original, the time spent on accessing data is shortened by some order of magnitude. Random access is normally achieved by a so-called *hash function*,

$$H : Data \longrightarrow Location$$

which computes, for each data item, its location in the *hash table*. Standard terminology regards as *synonyms* all data competing for the same location. A set of synonyms is called a *bucket*.

There are several ways in which data collision is handled, *e.g. linear probing* [Wir76] or *overflow handling* [HS78]. Because the former is not a totally correct implementation of a database file manager, only the latter will be addressed in this paper.

2. A Synopsis of SETS

This section summarizes the SETS notation for constructive set-theory-based data-type specification, adopted in this paper.

SETS should not be regarded as a competitor to specification languages such as META-IV (vulg. VDM) [Jon86] or Z [Spi89], but rather as a framework to write and reason about specifications in a simple, "pure" mathematical notation, the one arising from the constructs of the cartesian closed category of sets — *cf.* product ($A \times B$) and exponentiation (A^B), plus co-product ($A + B$) [Oli90]. Although SETS has a categorial foundation (see *e.g.* [OB97, Rod97] for details), the use of category theory at practitioner level is rather mild.

2.1. The Notation

Let A and B be data sets. Product and exponentiation are defined in the usual manner,

$$\begin{aligned} A \times B &\stackrel{\text{def}}{=} \{\langle a, b\rangle \mid a \in A \wedge b \in B\} \\ A^B &\stackrel{\text{def}}{=} \{f \mid f : B \to A\} \end{aligned}$$

[1] For an account of how to formally classify software components resorting to the SETS notation and calculus see *e.g.* [Oli91, OC93, OB97].

Table 1. SETS versus Programming Language Data-structuring Notation.

SETS	PASCAL	C	**Informal Description**

$A \times B$	`record`
	`  P: A;`
	`  S: B`
	`end;`
		`struct {`
		`  A first;`
		`  B second;`
		`};`
			Records

$A + B$	`record`
	`  case`
	`  tag: integer`
	`  of x =`
	`  1: (P:A);`
	`  2: (S:B)`
	`end;`
		`struct {`
		`  int tag; /* 1,2 */`
		`  union {`
		`  A ifA;`
		`  B ifB;`
		`  } data;`
		`};`
			Variant records

B^A	`array[A] of B`	`B ...[A]`	Arrays
$A + 1$	`^A`	`A *...`	Pointers

Co-product (disjoint union) is handled explicitly by using integer tags, *i.e.*

$$\begin{aligned} A + B &= (\{1\} \times A) \cup (\{2\} \times B) \\ &= \{\langle 1, a\rangle \mid a \in A\} \cup \{\langle 2, b\rangle \mid b \in B\} \end{aligned}$$

instead of using is-A or is-B predicates hiding such tags [2]. Table 1 presents an analogy between conventional programming language data-structuring notation and SETS basic constructs' notation.

On top of product, co-product and exponentiation, in SETS one can build data models using the following *derived* constructs:

- subsets of a finite set A (*cf.* set of A in VDM):

  $$2^A$$

- binary relations over finite sets A and B (*cf.* set of $(A \times B)$ in VDM):

  $$2^{A \times B}$$

- partial maps from finite A to B (*cf.* map A to B in VDM):

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

- finite sequences on a set A (*cf.* seq of A in VDM):

  $$A^\star = \bigcup_{n \geq 0} A^{\overline{n}}$$

  where each exponent $\overline{n}$ denotes the initial segment of $\mathbb{N}$ whose cardinality is n. For simplicity, we will write n instead of $\overline{n}$, and therefore 0 instead of the empty set $\{\}$ [3].

[2] As in the VDM style notation [Jon80], for instance. Integer tag introduction can be abbreviated via co-product injections, *cf. e.g.* i_1 and i_2 in appendix C.

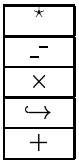
[3] In SETS, all sets with the same cardinality are isomorphic and therefore indistinguishable from a specification point of view. Initial segments of $\mathbb{N}$ provide an abstract mechanism for specifying enumerated types, of which only cardinality is retained. For instance, instead of enumerating

- union types [4] (*cf.* $[A]$ in VDM):

  $$A + 1$$

  where 1 stands for any singleton set.

In order to save SETS-expressions from too many parenteses, the following operator priorities are assumed:



For instance, expression

$$X \hookrightarrow A^2 \times B^\star + C$$

will abbreviate

$$(X \hookrightarrow ((A^2) \times (B^\star))) + C$$

etc.

Properties of both primitive and derived SETS constructs are listed in the appendices. A last ("meta") construct is recursive definition,

$$X \cong \mathcal{F}(X)$$

where X is the name of a data sort and $\mathcal{F}$ is a SETS-expression ("functor") involving the above primitive or derived constructs [5].

Recursive definitions are a powerful mechanism for specifying arbitrarily elaborate data structures. For example, the following concrete syntax in C for fairly unsophisticated *decision trees* where decisions are modelled by tree nodes with no answers,

```
typedef struct DecTree{
        char *What; /* Question or Decision */
        struct subTrees *R;
        };

typedef struct subTrees {
        struct node *first;
```

$$Weekday = \{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday\}$$

one simply writes

$$Weekday \cong \overline{7}$$

(note the isomorphism sign $\cong$) or even

$$Weekday \cong 7$$

Particularly common enumerated data types are 0 (the empty set), 1 (any singleton set) and 2 (the Booleans). Should the specification style become too terse, the specifier may adopt "cannonical" standards among each cardinality class, *e.g.* $2 = \{TRUE, FALSE\}$ and $1 = \{NIL\}$ (NIL is a particular good choice concerning union types and the analogy of table 1).

[4] Or "pointers", see analogy of table 1 below.

[5] Not every $\mathcal{F}$ can participate in the definition of a recursive data domain. See [Oli90] for a discussion on this topic.

```
        struct subTrees *next;
        };

typedef struct node {
        char *Answer;
        struct DecTree *SubTree;
        };
```

could have been specified in the SETS notation simply as follows:

$$DecTree \cong What \times (Answer \hookrightarrow DecTree)$$

that is, $DecTree \cong \mathcal{F}(DecTree)$ for $\mathcal{F}(X) \cong What \times (Answer \hookrightarrow X)$.

A more direct illustration is the following C syntax fragment for *genealogical diagrams* (the "pedigree view" of family relationship):

```
typedef struct GenDia {
        Ind individual;        /* data about an individual */
        struct GenDia *father; /* genealogy of his/her father
                                  (if known) */
        struct GenDia *mother; /* genealogy of his/her mother
                                  (if known) */
        };
```

which — on the basis of the analogy of table 1 — is easily "reversed" into SETS notation as follows,

$$GenDia \cong Ind \times (GenDia + 1) \times (GenDia + 1)$$

and even shortened to

$$GenDia \cong Ind \times (GenDia + 1)^2$$

In the opposite direction, table 1 can also be used for coding purposes. Consider, for instance, the following SETS specification of a linked list of objects A,

$$\begin{aligned} L &\cong 1 + Node \\ Node &\cong A \times L \end{aligned}$$

that is, the following "linear equation":

$$L \cong 1 + A \times L$$

From table 1 one draws that, in the C programming language, every $x \in L$ will be declared as

```
        Node *x;
```

since $L = 1 + Node$. Thus a datatype definition is required for `Node`, which is immediate from table 1 again:

```
typedef struct Node {
        A first;
        struct Node *next;
        };
```

Based on the above notation for specifying data domain, SETS adopts a notation for specifying operations on such domains (*i.e.* recursive functions) which is classical in mathematics, *e.g.*

$$x! = \left\{ \begin{array}{lcl} x = 0 & \Rightarrow & 1 \\ \neg(x = 0) & \Rightarrow & x \times (x-1)! \end{array} \right.$$

instead of usual *if-then-else* notation, *e.g.*

$$\begin{array}{l} x! = \ if \ \ x = 0 \\ \qquad\ then \ 1 \\ \qquad\ else \ x \times (x-1)! \end{array}$$

In general, conditional expressions such as

$$if \ \ p(x) \ \ then \ \ g(x) \ \ else \ \ h(x)$$

are written thus:

$$\left\{ \begin{array}{lcl} p(x) & \Rightarrow & g(x) \\ \neg p(x) & \Rightarrow & h(x) \end{array} \right.$$

Partial expressions are special cases of conditional expressions,

$$\{ \ p(x) \quad \Rightarrow \quad g(x)$$

meaning

$$\left\{ \begin{array}{lcl} p(x) & \Rightarrow & g(x) \\ \neg p(x) & \Rightarrow & \bot \end{array} \right.$$

i.e., $p(x)$ is a pre-condition for the computation of $g(x)$.

SETS's kernel functional notation includes all set-theoretical operators typical of constructive specification languages (such as *e.g.* VDM [Jon86] or Z [Spi89]) dealing with lists (A^*), sets (2^A), mappings ($A \hookrightarrow B$) and so on. Finite mappings are enumerated by writing *e.g.*

$$\left(\begin{array}{ccc} 1 & 2 & 3 \\ a & b & c \end{array} \right)$$

instead of (*cf.* VDM)

$$[1 \mapsto a, 2 \mapsto b, 3 \mapsto c]$$

or

$$\{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}.$$

This notation extends to mapping-comprehension, *e.g.*:

$$\left(\begin{array}{c} i \\ i+1 \end{array} \right)_{i \in 3}$$

SETS's functional notation also includes *functorial* constructs, which provide for much textual and reasoning economy. Functorial constructs arise from regarding SETS's data constructs as functors in the category-theoretical sense [Oli92], there being a functional construct per each SETS data construct. The most common are, for A, B, C given SETS data domains and $f : A \longrightarrow B$ a SETS morphism: 2^f, f^*, $C \times f$, $C + f$ and $C \hookrightarrow f$, defined as follows [6]:

$$2^f \quad : \quad 2^A \longrightarrow 2^B$$

[6] Because of their functorial nature, these constructs obey to "natural properties" which are very useful in calculation. For a thorough presentation of SETS functorial foundations see *e.g.* [OB97].

$$
\begin{array}{rcl}
2^f(S) & \stackrel{\text{def}}{=} & \{f(a) \mid a \in S\} \\
\\
f^* & : & A^* \longrightarrow B^* \\
f^*(s) & \stackrel{\text{def}}{=} & <f(a) \mid a \leftarrow s> \\
\\
C \times f & : & (C \times A) \longrightarrow (C \times B) \\
(C \times f)(\langle c, a\rangle) & \stackrel{\text{def}}{=} & \langle c, f(a)\rangle \\
\\
C + f & : & (C + A) \longrightarrow (C + B) \\
(C + f)(x) & \stackrel{\text{def}}{=} & \left\{ \begin{array}{lcl} x = \langle 1, c\rangle & \Rightarrow & x \\ x = \langle 2, a\rangle & \Rightarrow & \langle 2, f(a)\rangle \end{array} \right. \\
\\
C \hookrightarrow f & : & (C \hookrightarrow A) \longrightarrow (C \hookrightarrow B) \\
(C \hookrightarrow f)(\sigma) & \stackrel{\text{def}}{=} & \left(\begin{array}{c} c \\ f(\sigma(c)) \end{array} \right)_{c \in dom(\sigma)}
\end{array}
$$

Finally, the following standard operations involving products and co-products are also provided: *construction* [Bac78],

$$
\begin{array}{rcl}
[f, g] & : & A \longrightarrow (B \times C) \\
[f, g](a) & \stackrel{\text{def}}{=} & \langle f(a), g(a)\rangle
\end{array}
\tag{1}
$$

(for $g : A \longrightarrow C$), and *alternation* (for $h : B \longrightarrow C$):

$$
\begin{array}{rcl}
g \mid h & : & (A + B) \longrightarrow C \\
g \mid h(x) & \stackrel{\text{def}}{=} & \left\{ \begin{array}{lcl} x = \langle 1, a\rangle & \Rightarrow & g(a) \\ x = \langle 2, b\rangle & \Rightarrow & h(b) \end{array} \right.
\end{array}
\tag{2}
$$

2.2. The Calculus

SETS [Oli90, Oli92] is an (inequational) calculus of (functional) *abstraction invariants* as understood by *e.g.* [MG90]. Each law of the calculus is an inequation of the form

$$A \trianglelefteq_f^{\phi} B \tag{3}$$

meaning:

- there is an *abstraction surjection* f from B to A;
- wherever f is a partial function, ϕ is the characteristic predicate defining its domain — the so-called *concrete invariant* induced in B;
- in the terminology of [MG90], equation (3) states that the following abstraction invariant holds between the abstract variables $a \in A$ and the concrete variables $b \in B$:

$$(a = f(b)) \wedge \phi(b)$$

As in [Oli90, Oli92], the notation

$$A \trianglelefteq_f B \tag{4}$$

is accepted as a shorthand of (3) wherever f is a total function, that is, wherever $\phi(b) = TRUE$, for all $b \in B$.

It can be shown that $\trianglelefteq$ is a pre-ordering on SETS data structures and that set-theoretical isomorphism is the symmetric closure of $\trianglelefteq$:

$$A \trianglelefteq_f B \wedge B \trianglelefteq_{f^{-1}} A \quad \Rightarrow \quad A \cong B$$

for f a bijection (again $\phi = \lambda b.TRUE$ may be omitted).

The transitivity of $\trianglelefteq$ means that one can chain $\trianglelefteq$-steps and synthesize overall abstraction maps and invariants. For a chain of n $\trianglelefteq$-steps,

$$\trianglelefteq_i \left\{ \begin{array}{l} f_i \\ \phi_i \end{array} \right.$$

the overall abstraction map and invariant are given by:

$$\trianglelefteq \left\{ \begin{array}{l} f \stackrel{\text{def}}{=} \bigcirc_{i=1}^{n} f_i \\ \phi \stackrel{\text{def}}{=} \lambda x.\ \bigwedge_{i=n}^{1} \phi_i((\bigcirc_{j=i+1}^{n} f_j)(x)) \end{array} \right.$$

cf. [Oli92].

The appendices contain a reasonably comprehensive account of the SETS calculus, organized as follows:

- the equational ($\cong$) sub-calculus, chiefly involving primitive constructs, is presented first, in appendix A;
- the inequational sub-calculus, chiefly involving derived constructs, follows in appendix B; shorthand (4) will indicate laws which do not induce concrete invariants;
- in many reasoning situations, both f and ϕ in (3) can be left out until explicitly demanded by algorithm synthesis [Oli92]; all laws used along the paper are provided with the relevant abstraction map f and invariant ϕ, listed respectively in appendix C.1 and appendix C.2 [7]. Equation numbers are used consistently in the identification of such functions and predicates: $f_{(n)}$ and $\phi_{(n)}$ denote respectively the abstraction map and invariant associated with law (n).

3. Handling 'Ad Hoc' Invariants in the SETS Calculus

The process of data refinement normally entails the introduction of datatype invariants required for the implementation to be faithful to the specification. As described above, the SETS calculus is based on a so-called *redundancy ordering* which guarantees data representation whilst saving as many datatype invariants as technically possible.

This strategy has the advantage of preventing from implementations "stronger" than necessary, but ignores other aspects of data refinement which are relevant in practice, namely,

- the proper starting specification of a design normally involves datatypes affected by 'ad hoc' invariants [8] which formalize real-life conventions, laws, rules, norms or natural constraints. For instance, even a primitive datatype such as

$$Date \cong 31 \times 12 \times \mathbb{N}$$

[7] See [Oli96] for a more comprehensive account of the SETS calculus.

[8] The term "ad hoc" (as opposed to "formal") will be made precise in the sequel.

requires (in the Gregorian calendar) a nontrivial 'ad hoc' invariant:

$$
\begin{array}{l}
dateOk : Date \longrightarrow 2 \\
dateOk(d,m,y) \stackrel{\text{def}}{=} \\
\left\{ \begin{array}{lcl}
m \in \{1,3,5,7,8,10,12\} & \Rightarrow & d \leq 31 \wedge \\
 & & (\neg(y = 1582 \wedge m = 10) \\
 & & \vee (d < 5) \vee (14 < d)) \\
m \in \{4,6,9,11\} & \Rightarrow & d \leq 30 \\
m = 2 \wedge leapYear(y) & \Rightarrow & d \leq 29 \\
m = 2 \wedge \neg leapYear(y) & \Rightarrow & d \leq 28
\end{array} \right.
\end{array}
\tag{5}
$$

where

$$
\begin{array}{rcl}
leapYear & : & \mathbf{N} \longrightarrow 2 \\
leapYear(y) & \stackrel{\text{def}}{=} & rem(y, \left\{ \begin{array}{lcll} 1700 \leq y \wedge rem(y,100) = 0 & \Rightarrow & 400 \\ 1700 > y \vee rem(y,100) \neq 0 & \Rightarrow & 4 \end{array} \right.) = 0
\end{array}
$$

This invariant is a good illustration of the *ad hoc* character of most constraints formally imposed to specifications. It arises from a natural, cosmological fact — the number of days of the year is not a natural number — and from several historical attempts to devise a norm able to finitely approximate such a number [9].

- some rules of SETS are "context-sensitive" in the sense that they can be applied to a datatype only if this is affected by some invariant [Oli90];
- 'ad hoc' invariants may be introduced throughout refinement which have not to do with representation faithfulness (*representativeness*), but rather with other facets of refinement such as *e.g.* seeking *efficiency*.
  For instance, a basic reification fact in SETS is (65), for A a finite set, meaning that finite lists implement finite sets. However, for efficiency we may be interested in implementing 2^A by lists in $A^\star$ which are ordered by some total order on A, which is irrelevant *wrt.* the abstraction function $elems$ (76).

How can such informal design constraints be tackled in the SETS's calculus? In general, for A, B two finite sets such that $A \trianglelefteq_f B$, it may be the case that

$$A \trianglelefteq_{f|\phi} B_\phi$$

holds, where $\phi : B \longrightarrow 2$ is an invariant on B defining the subset $B_\phi = \{b \in B \mid \phi(b)\}$ and $f|\phi$ means the restriction of f to B_ϕ:

$$
\begin{array}{rcl}
f|\phi & : & B_\phi \longrightarrow A \\
b & \rightsquigarrow & f(b)
\end{array}
$$

Predicate ϕ can be as strong as we wish provided that the cardinality of B_ϕ is greater or equal to the cardinality of A, since $f|\phi$ should still be surjective. In the "limit" one has:

$$A \cong B_\phi \tag{6}$$

Such is the case of

$$2^A \quad \cong_{elems|linear} \quad (A^\star)_{linear} \tag{7}$$

where $linear$ forces every sequence $s \in A^\star$ to be strictly ordered by a given linear order $\sqsubset \in A \times A$:

[9] The situation is actually more complex, because this invariant is "geography" dependent. In fact, the 1582 discontinuity (10 days omitted in October of that year) of the Gregorian calendar was adopted much later in several countries — *e.g.* Poland (1586), Hungary (1587), Germany (1700), England (1752) *etc.* [Gof84].

Table 2. Summary of Functorial Calculus.

$\mathcal{F}(X)$	$\mathcal{F}(f)$	$\mathcal{F}(\phi)$
X	f	ϕ
C	1_C	$\lambda c.TRUE$
2^X	$2^f = \lambda s.\{f(x) \mid x \in s\}$	$2^\phi = \lambda s.\forall b \in s : \phi(b)$
$X^\star$	$f^\star = \lambda l.<f(x) \mid x \leftarrow l>$	$\phi^\star = \lambda l.\forall 1 \leq i \leq length(l) : \phi(l(i))$
X^C	$f^C = \lambda\sigma.f \circ \sigma = \lambda\sigma.\left(\begin{array}{c} c \\ f(\sigma(c)) \end{array}\right)_{c \in C}$	$\phi^C = \lambda\sigma.\forall c \in C : \phi(\sigma(c))$
$C \hookrightarrow X$	$C \hookrightarrow f = \lambda\sigma.f \circ \sigma = \lambda\sigma.\left(\begin{array}{c} c \\ f(\sigma(c)) \end{array}\right)_{c \in dom(\sigma)}$	$C \hookrightarrow \phi = \lambda\sigma.\forall x \in rng(\sigma) : \phi(x)$
$\mathcal{G}(X) \times \mathcal{H}(X)$	$\mathcal{G}(f) \times \mathcal{H}(f)$	$\lambda(x,y).\mathcal{G}(\phi)(x) \wedge \mathcal{H}(\phi)(y)$
$\mathcal{G}(X) + \mathcal{H}(X)$	$\mathcal{G}(f) + \mathcal{H}(f)$	$\lambda x.\left\{\begin{array}{lcl} x = \langle 1,a\rangle & \Rightarrow & \mathcal{G}(\phi)(a) \\ x = \langle 2,b\rangle & \Rightarrow & \mathcal{H}(\phi)(b) \end{array}\right.$
$\mathcal{G}(X) + 1$	$\lambda x.\left\{\begin{array}{lcl} x = \langle 2,NIL\rangle & \Rightarrow & x \\ x = \langle 1,a\rangle & \Rightarrow & \langle 1,\mathcal{G}(f)(a)\rangle \end{array}\right.$	$\lambda x.\left\{\begin{array}{lcl} x = \langle 2,NIL\rangle & \Rightarrow & TRUE \\ x = \langle 1,a\rangle & \Rightarrow & \mathcal{G}(\phi)(a) \end{array}\right.$

$$linear(s) \stackrel{\text{def}}{=} \forall 1 \leq i < j \leq length(s) : s(i) \sqsubset s(j)$$

We recall from [Oli92] that, following [MA86], datatype constructors in the SETS calculus are regarded as co-continuous (endo)functors in the SETS category of finite sets. A result central to the calculus is then a *functorial approach* to the calculation of abstraction invariants expressed by the following theorem:

Theorem 1. Let $\mathcal{F}$ be a polynomial (endo)functor in SETS. If

$$A \trianglelefteq_f^\phi B \tag{8}$$

then

$$\mathcal{F}(A) \trianglelefteq_{\mathcal{F}(f)}^{\mathcal{F}(\phi)} \mathcal{F}(B) \tag{9}$$

Proof. See [Oli92]. □

This theorem provides an elegant strategy for computing complex abstraction invariants throughout reification. Although $\mathcal{F}(X) = 2^X$ is not polynomial [MA86], (9) still holds for this functor, for finite A and B [Jou92]. Table 2 presents a summary of this "functorial calculus" for the most common SETS constructs.

One of the aims of this paper is to consider the impact of 'ad hoc' invariants on these results. First of all, we need to decorate every data domain A with its invariant ϕ,

$$A_\phi$$

because the invariant which affects A at some stage in refinement may then be stronger than entailed by the application of the involved $\trianglelefteq$-rules. In fact, extra ad hoc invariants may have been arbitrarily introduced by the implementor. Subscript ϕ in A_ϕ will be omitted only if it is the "everywhere true" predicate $\lambda a.TRUE$.

Jourdan's work on the foundations of SETS [Jou92] puts forward the following rule

for "pushing" an arbitrary invariant α downto lower refinement levels: should (3) hold, then

$$A_\alpha \trianglelefteq_f^\phi B_{\phi \wedge \alpha \circ f}$$

holds, where $\phi \wedge \alpha \circ f$ is a shorthand of predicate $\lambda b.\phi(b) \wedge \alpha(f(b))$.

Putting everything together, we state the following theorems concerning arbitrary datatype invariant handling:

Theorem 2. Let $A \trianglelefteq_f B$ hold and $\phi : B \longrightarrow 2$ be a predicate such that the restriction of f by ϕ,

$$f|\phi : B_\phi \longrightarrow A$$

is still a surjection. Then

$$A \trianglelefteq_{f|\phi} B_\phi$$

Proof. Obvious, from the definition of the $\trianglelefteq$-order [Oli90]. □

Theorem 3. If $A \trianglelefteq_f^\phi B_\phi$ holds and $\alpha : A \longrightarrow 2$ is an invariant on A, then α is "pushed" into B as follows:

$$A_\alpha \trianglelefteq_f^\phi B_{\phi \wedge \alpha \circ f}$$

Proof. See [Jou92]. □

Finally, the following theorem will help in reasoning about subscripted data domains.

Theorem 4. For $\mathcal{F}$ a polynomial (endo)functor in SETS or the powerset functor, A a finite set and $\phi : A \longrightarrow 2$ an invariant on A, one has

$$\mathcal{F}(A_\phi) \cong \mathcal{F}(A)_{\mathcal{F}(\phi)}$$

Proof. Concerning products we have:

$$\begin{array}{rcl} A_\phi \times B_\varphi & = & \{\langle a,b\rangle \mid\ a \in A_\phi \wedge b \in B_\varphi\} \\ & = & \{\langle a,b\rangle \mid\ a \in A \wedge b \in B \wedge \phi(a) \wedge \varphi(b)\} \\ & = & \{\langle a,b\rangle \mid\ \langle a,b\rangle \in A \times B \wedge (\phi \times \varphi)(a,b)\} \\ & = & (A \times B)_{\phi \times \varphi} \end{array}$$

Concerning co-products we have:

$$\begin{array}{rcl} A_\phi + B_\varphi & = & \{\langle 1,a\rangle \mid\ a \in A \wedge \phi(a)\} \cup \{\langle 2,b\rangle \mid\ b \in B \wedge \phi(b)\} \\ & = & \{x \mid\ x = \langle 1,a\rangle \wedge a \in A \wedge \phi(a) \vee x = \langle 2,b\rangle \wedge b \in B \wedge \phi(b)\} \\ & = & \{x \mid \left\{ \begin{array}{lcl} x = \langle 1,a\rangle \wedge a \in A & \Rightarrow & \phi(a) \\ x = \langle 2,b\rangle \wedge b \in B & \Rightarrow & \phi(b) \end{array} \right. \} \\ & = & (A + B)_{\phi + \varphi} \end{array}$$

Concerning finite exponentiation we have:

$$\begin{array}{rcl} (A_\phi)^C & = & \{f : C \to A \mid\ \forall c \in C : \phi(f(a))\} \\ & = & \{f : C \to A \mid\ \phi^C(f)\} \\ & = & (A^C)_{\phi^C} \end{array}$$

Finaly, concerning powersets we have:

$$\begin{array}{lcl} 2^{A_\phi} & = & \{S \mid S \subseteq A_\phi\} \\ & = & \{S \subseteq A \mid \forall a \in S : \phi(a)\} \\ & = & (2^A)_{2^\phi} \end{array}$$

□

At model functionality level, we have the following theorem guaranteeing that if an operation preserves an ad hoc invariant then any of its valid implementations will preserve the same invariant too, at low level.

Theorem 5. Let $\sigma : A \longrightarrow A$ be the specification of a function transforming data of sort A while preserving some ad hoc invariant α on A, that is [10],

$$\forall a \in A : \alpha(a) \Rightarrow \alpha(\sigma(a)) \tag{10}$$

Then the reification σ' of σ implied by reification step (3),

$$\begin{array}{ccc} A & \xrightarrow{\sigma} & A \\ {\scriptstyle f}\uparrow & & \uparrow{\scriptstyle f} \\ B_\phi & \xrightarrow[\sigma']{} & B_\phi \end{array}$$

will still preserve α at B-level.

Proof. σ' is any function satisfying

$$\forall b \in B_\phi : f(\sigma'(b)) = \sigma(f(b)) \tag{11}$$

σ' will preserve α at B-level iff

$$\forall b \in B_\phi : \alpha(f(b)) \Rightarrow \alpha(f(\sigma'(b)))$$

By contradiction, α will not be preserved if

$$\exists b \in B_\phi : \alpha(f(b)) \wedge \neg\alpha(f(\sigma'(b)))$$

that is,

$$\exists b \in B_\phi : \alpha(f(b)) \wedge \neg\alpha(\sigma(f(b)))$$

via (11). Because f is surjective, $f(b) = a$ for some $a \in A$. Thus we will have

$$\exists a \in A : \alpha(a) \wedge \neg\alpha(\sigma(a))$$

thus contradicting our hypothesis (10). □

[10] This theorem is stated for unary σ with no loss in generality: the result will be valid still for any extension of the arity of σ:

$$\sigma : \ldots \times A \times \ldots \to A$$

4. From a Simple Abstract Database Model to a Hash Table

Let A be an unbounded but finite domain of *data* which we want to record and maintain in a *database* file. We will regard

$$Database \quad \cong \quad 2^A \tag{12}$$

as a convenient (however primitive!) model for the database file itself [11]. That is, our notion of an abstract database file "re-uses" the algebra of finite sets:

$$\begin{array}{|rcl|}
\hline
initialDb & : & \longrightarrow Database \\
initialDb & \stackrel{\text{def}}{=} & \{\} \\
\hline
insert & : & A \times Database \longrightarrow Database \\
insert(a,\sigma) & \stackrel{\text{def}}{=} & \sigma \cup \{a\} \\
\hline
find & : & A \times Database \longrightarrow 2 \\
find(a,\sigma) & \stackrel{\text{def}}{=} & (a \in \sigma) \\
\hline
remove & : & A \times Database \longrightarrow Database \\
remove(a,\sigma) & \stackrel{\text{def}}{=} & \sigma - \{a\} \\
\hline
\end{array} \tag{13}$$

We start by calculating the hash-table implementation of (12). Then we will proceed to the deduction of the corresponding operators as specified above.

4.1. Data-level Calculation

Let $n \in \mathbb{N}$ be the number of entries of our intended hash table, and

$$H : A \longrightarrow n$$

be the selected hash function. Recall from section 2.1 the use of n to denote the initial segment of $\mathbb{N}$ of size n, that is, the set $\{1, 2, \ldots, n\}$. Usually, n is much smaller than the cardinal of A, that is, H is clearly non-injective.

Our target is to use $\trianglelefteq$-reasoning in order to convert

$$2^A \trianglelefteq \ldots$$

into something we can identify as an acceptable model of a hash table. Our reasoning will start from a basic result — fact (64) in appendix B — expressing the fact that a data domain can always be refined by adding extra information to it. From an instance of fact (64),

$$A \trianglelefteq_{\pi_2} n \times A \tag{14}$$

[11] See section 5 for a standard elaboration of this model.

we proceed by imposing the following ad hoc invariant on $n \times A$:

$$\phi(i,a) \stackrel{\text{def}}{=} i = H(a) \tag{15}$$

forcing every datum $a \in A$ to be coupled with its hash-index $H(a)$. Since the restriction of π_2 to $(n \times A)_\phi$ is surjective on A, that is,

$$2^{\pi_2}(\{(i,a) \in n \times A \mid\ i = H(a)\}) = A$$

we have

$$A \trianglelefteq_{\pi_2|\phi} (n \times A)_\phi \tag{16}$$

by theorem 2. By applying the powerset functor $\lambda X.2^X$ to (16) we obtain (*cf.* table 2 above)

$$2^A \trianglelefteq_{f_1} (2^{n \times A})_{\Phi_1} \tag{17}$$

where

$$\begin{array}{lcl} \Phi_1 & = & 2^\phi \\ & = & \lambda s.(\forall x \in s : \phi(x)) \\ & = & \lambda s.(\forall (i,a) \in s : i = H(a)) \end{array}$$

and

$$\begin{array}{lcl} f_1 & = & 2^{(\pi_2|\phi)} \\ & = & \lambda s.\{\pi_2(x) \mid\ x \in s\} \\ & = & \lambda s.\{a \in A \mid\ (i,a) \in s\} \end{array}$$

We now resort to the "currying" law of exponentiation, (56), established by the $uncurry$ bijection (86). For $A = 2$ we obtain the following version of $uncurry$,

$$uncurry(t) \stackrel{\text{def}}{=} \{\langle b,c\rangle \mid\ b \in B \wedge c \in t(b)\}$$

The instance of (56) which interests us — *cf.* the right-hand side of (17) — is

$$2^{n \times A} \cong (2^A)^n \tag{18}$$

By handling datatype invariants according to theorem 3 we obtain

$$(2^{n \times A})_{\Phi_1} \cong_{f_2} ((2^A)^n)_{\Phi_2} \tag{19}$$

where

$$\begin{array}{lcl} f_2 & = & \lambda t.uncurry(t) \\ & = & \lambda t.\{\langle i,a\rangle \mid\ i \in n \wedge a \in t(i)\} \end{array}$$

and

$$\begin{array}{lcl} \Phi_2 & = & \Phi_1 \circ uncurry \\ & = & \lambda t.\Phi_1(\{(i,a) \mid\ i \in n \wedge a \in t(i)\}) \\ & = & \lambda t.\forall i \in n : (\forall a \in t(i) : i = H(a)) \end{array} \tag{20}$$

It is easy to show that $((2^A)^n)_{\Phi_2}$ above — *i.e.* the "array" structure [12] $(n \to 2^A)_{\Phi_2}$ — is in fact a model for hash tables. According to Φ_2 (20), every set in the range of such "arrays" contains *synonyms* [HS78]. All such sets are mutually disjoint, *cf.* the following lemma:

[12] $A \to B$ may be used interchangeably with B^A.

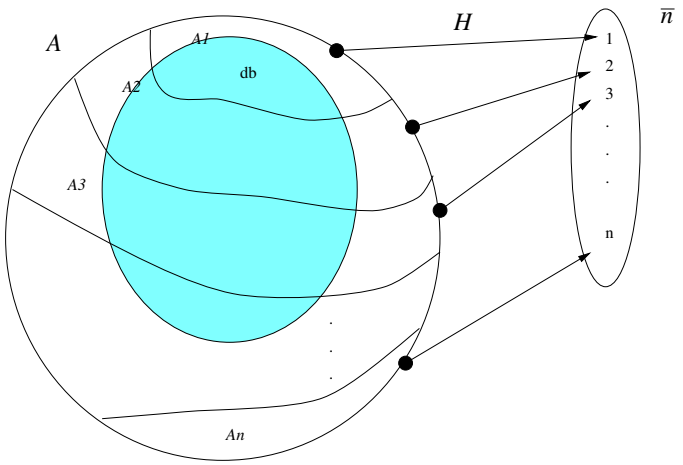


Fig. 1. The "hashing effect".

Lemma 1. For every hash table $t \in (n \to 2^A)_{\Phi_2}$ we have

$$\forall i \neq j \in n : t(i) \cap t(j) = \{\}$$

Proof. By contradiction, suppose that, for particular $i \neq j$ and $a \in A$, we have

$$a \in t(i) \wedge a \in t(j)$$

Then, by Φ_2 (20) one has

$$i = H(a) \wedge j = H(a)$$

that is, $i = j$ thus contradicting the starting assumption $i \neq j$. □

Therefore, every $db \in 2^A$ is partitioned into n-many, disjoint "collision segments", each one addressed by the relevant hash index computed by H [13]. We have thus achieved the intended "hashing effect". However, the task of reifying 2^A "recurs" in the co-domain of $n \to 2^A$ — a kind of "fractal" effect on this type of data structuring. This problem is identified in the literature under the headings *overflow handling* or *collision handling* [HS78].

4.1.1. Collision Handling

Let us show how SETS helps in dealing with this problem in a formal way. Since refinement steps in SETS are compositional, and bearing in mind that we have just been trying to refine sets into hash tables, what about doing the same for each collision set?

This solution is known as *rehashing* and leads to something like

$$n \to (m \to 2^A) \tag{21}$$

[13] In fact, such collision segments are but the equivalence classes of the kernel relation K_H induced by the restriction of H to db:

$$a \; K_H \; a' \Leftrightarrow H(a) = H(a')$$

cf. $A_1, A_2, A_3, \ldots, A_n$, figure 1.

under a more elaborate invariant involving n possibly different sub-hashing functions H_i $(i = 1, \ldots, n)$:

$$\Phi_3 \quad = \quad \lambda t.\forall i \in n : (\forall j \in m : (\forall a \in t(i)(j) : i = H(a) \wedge j = H_i(a))) \qquad (22)$$

But, strictly speaking, *rehashing* is nothing but multiplying the address space of a given hashing table by some factor (m), since

$$((2^A)^m)^n \cong (2^A)^{n \times m}$$

— *cf.* law (56) — where the hash function is

$$H'(a) = \langle H(a), H_{H(a)}(a) \rangle$$

Typically $m < n$, and collision sets get smaller, but 2^A still recurs in (21).

For a more effective solution to the problem we need to analyse the reification theory of 2^A, and many options are available. The most immediate is perhaps given by law (65) in appendix B, turning hash tables into arrays of sequences of collisions:

$$(n \to A^\star)_{\Phi_3}$$

The corresponding invariant Φ_3 is obtained via theorem 1, whereby

$$2^A \trianglelefteq_{elems} A^\star \quad \Rightarrow \quad (2^A)^n \trianglelefteq_{elems^n} (A^\star)^n$$

for

$$elems^n(t) = \lambda i.elems(t(i))$$

cf. table 2. By theorem 3 one has,

$$(2^A)^n \trianglelefteq_{elems^n} (A^\star)^n \quad \Rightarrow \quad ((2^A)^n)_{\Phi_2} \trianglelefteq_{elems^n} ((A^\star)^n)_{\underbrace{\Phi_2 \circ elems^n}_{\Phi_3}} \qquad (23)$$

that is,

$$\begin{array}{rcl} \Phi_3 & = & \Phi_2 \circ elems^n \\ & = & \lambda t.\forall i \in n : (\forall a \in elems(t(i)) : i = H(a)) \end{array}$$

$A^\star$ is not readily available in programming languages like C or PASCAL. Its reification towards such programming environments has been studied in detail elsewhere (*e.g.* [Oli90, Jou92, Oli92]) as a particular case of refining recursive data types. We sketch here only the outline of the reasoning. First, recursion is *introduced* by realizing that $A^\star$ is a well-known fixpoint solution of the functorial equation

$$X \quad \cong \quad 1 + A \times X \quad \text{/*finite lists on } A \text{ */} \qquad (24)$$

Then recursion is *removed* by reifying the recursive pattern of (24) in terms of references or pointers $(K + 1)$ which are either NIL or point to a location in a *heap* or *object store*, addressed by K, a natural-number isomorphic domain of "keys":

$$A^\star \quad \trianglelefteq \quad \underbrace{(K \hookrightarrow A \times (K + 1))}_{\text{heap}} \times \underbrace{(K + 1)}_{\text{pointer}} \qquad (25)$$

The corresponding abstraction function is (*cf.* [Oli92]):

$$f_{(25)}(\sigma, x) \quad \stackrel{\text{def}}{=} \quad \left\{ \begin{array}{rcll} x = i_2(NIL) & \Rightarrow & <> & \\ x = i_1(k) & \Rightarrow & let & a = \pi_1(\sigma(k)) \\ & & & x' = \pi_2(\sigma(k)) \\ & & in & <a> \frown f_{(25)}(\sigma, x')) \end{array} \right.$$

which is coupled to the following low-level invariant, ensuring acyclic pointer definedness,

$$\phi_{(25)}(\sigma, x) \quad \stackrel{\text{def}}{=} \quad \gamma(x, \sigma, \{\})$$

where:

$$\gamma(x, \sigma, C) \stackrel{\text{def}}{=} \left\{ \begin{array}{rcl} x = i_2(NIL) & \Rightarrow & TRUE \\ x = i_1(k) \wedge k \notin dom(\sigma)) \vee k \in C & \Rightarrow & FALSE \\ x = i_1(k) \wedge k \in dom(\sigma)) & \Rightarrow & \gamma(\pi_2(\sigma(k)), \sigma, C \cup \{k\}) \end{array} \right. \tag{26}$$

Concerning $(A^\star)^n$ we will have:

$$(A^\star)^n \quad \trianglelefteq \quad (K \hookrightarrow A \times (K+1)) \times (K+1)^n \tag{27}$$

(leading to an array of pointers to linked-lists sharing the same heap structure) where

$$f_{(27)}(\sigma, t) \quad \stackrel{\text{def}}{=} \quad \left(\begin{array}{c} i \\ f_{(25)}(\sigma, t(i)) \end{array} \right)_{i \in n}$$

and

$$\phi_{(27)}(\sigma, t) \quad \stackrel{\text{def}}{=} \quad \forall i \in n : \phi_{(25)}(\sigma, t(i))$$

The overall invariant at this point,

$$\Phi_4 \quad = \quad \phi_{(27)} \wedge \Phi_3 \circ f_{(27)}$$

can be systematically calculated. First, we have:

$$\begin{array}{rl} & \Phi_4(\sigma, t) = \\ & \quad \forall i \in n : \phi_{(25)}(\sigma, t(i)) \wedge \Phi_3 \circ f_{(27)} \\ = & \quad (\forall i \in n : \gamma(t(i), \sigma, \{\})) \wedge \forall i \in n : (\forall a \in elems(f_{(25)}(\sigma, t(i))) : i = H(a)) \\ = & \quad \forall i \in n : \gamma(t(i), \sigma, \{\}) \wedge (\forall a \in elems(f_{(25)}(\sigma, t(i))) : i = H(a)) \\ = & \quad \forall i \in n : \gamma(t(i), \sigma, \{\}) \wedge \varphi(t(i), \sigma, i) \end{array}$$

where φ is easily shown to be the following predicate:

$$\varphi(k, \sigma, i) \quad \stackrel{\text{def}}{=} \quad \left\{ \begin{array}{rcll} k = NIL & \Rightarrow & TRUE & \\ \neg(k = NIL) & \Rightarrow & let & a' = \pi_1(\sigma(k)) \\ & & & k' = \pi_2(\sigma(k)) \\ & & in & i = H(a) \wedge \varphi(k', \sigma, i) \end{array} \right.$$

Further "fold-unfold" reasoning will lead to

$$\Phi_4(\sigma, t) \quad = \quad \forall i \in n : \Phi_{4aux}(t(i), \sigma, i, \{\})$$

where Φ_{4aux} is an auxiliary predicate inter-combining γ and φ which realizes the expected linked-list downwards traversal checking for simultaneous hash-index consistency and pointer definedness and acyclicness:

$$\Phi_{4aux}(k, \sigma, i, C) \stackrel{\text{def}}{=}$$

$$\left\{\begin{array}{rcl} k = NIL & \Rightarrow & TRUE \\ k \neq NIL & & \\ \wedge & & \\ k \notin dom(\sigma) & \Rightarrow & FALSE \\ \vee & & \\ k \in C & & \\ k \in dom(\sigma) & \Rightarrow & let \quad a' = \pi_1(\sigma(k)) \\ & & \quad\;\; k' = \pi_2(\sigma(k)) \\ & & in \quad i = H(a) \wedge \Phi_{4aux}(k', \sigma, i, C \cup \{k\}) \end{array}\right.$$

Putting everything together, we have calculated the following data structure:

$$((K \hookrightarrow A \times (K+1)) \times (K+1)^n)_{\Phi_4} \tag{28}$$

In the end, this corresponds in fact to a classical strategy, *cf.* the following quotation from [HS78]:

Many of the comparisons being made could be saved if we maintained lists of identifiers, one list per bucket, each list containing all the synonyms for that bucket (...) Since the size of these lists is not known in advance, the best way to maintain them is as linked chains.

The final step would be to encode (28) (with some extra "syntactic sugar") in a commercial programming language, *e.g.* C

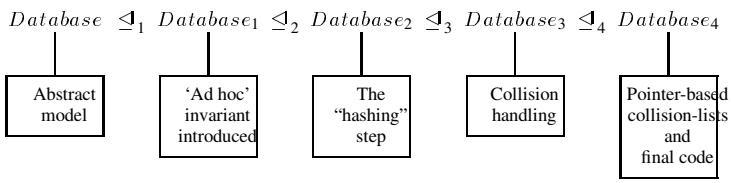
```
typedef  struct Collision {
         A Synonym;
         struct Collision *Next;
         };
typedef  struct Collision *Bucket;
typedef  Bucket Database4[n]; /* hash table */
```
(29)

or PASCAL

```
type  Collision  =  record
                       Synonym: A;
                       Next: ^ Collision
                    end;
      Bucket     =  ^ Collision;
      Database4  =  (* hash table *)
                    array[1..n] of Bucket;
```
(30)

in the documentation of which Φ_4 should be recorded, at conceptual level.

The following diagram provides a summary of the reasoning carried out in this section:



(31)

4.2. Operation Level Calculation

The steps of (31) which interest us are the "hashing ones", that is from $Database$ to $Database_2$. The corresponding abstraction map retrieves the abstract database by

re-assembling all disjoint collision sets:

$$\begin{aligned} f_1(f_2(t)) &= \{a \in A \mid (i,a) \in \{(i,a) \mid i \in n \wedge a \in t(i)\}\} \\ &= \{a \in A \mid i \in n \wedge a \in t(i)\} \\ &= \{a \in A \mid \exists i \in n : a \in t(i)\} \\ &= \bigcup_{i \in n} \{a \in A \mid a \in t(i)\} \\ &= \bigcup_{i \in n} t(i) \end{aligned} \tag{32}$$

This map will now be relevant for composing refinement diagrams which will enable calculation of the operations — $initialDb$, $insert$, $find$, $remove$ — as they should be implemented at $Database_2$-level.

4.2.1. The $find$ Operation

The refinement diagram for $find$ is:

$$\begin{array}{ccc} A \times 2^A & \xrightarrow{\quad find \quad} & 2 \\ \uparrow{\scriptstyle 1_A \times (f_1 \circ f_2)} & & \uparrow{\scriptstyle 1_2} \\ A \times ((2^A)^n)_{\Phi_2} & \xrightarrow[\quad find_2 \quad]{} & 2 \end{array}$$

leading to refinement equation

$$find_2(a,t) = find(a, f_1(f_2(t)))$$

The reasoning is as follows:

$$\begin{aligned} find_2(a,t) &= find(a, f_1(f_2(t))) \\ &= a \in f_1(f_2(t)) \\ &= a \in \bigcup_{i \in n} t(i) \\ &= \bigvee_{i \in n} a \in t(i) \\ &= \exists i \in n : a \in t(i) \end{aligned}$$

Now, taking ad hoc invariant Φ_2 into account, one has:

$$\begin{aligned} \bigvee_{i \in n} a \in t(i) &= a \in t(H(a)) \vee \bigvee_{i \in n \wedge i \neq H(a)} a \in t(i) \\ &= (a \in t(H(a)) \vee \bigvee_{i \in n \wedge i \neq H(a)} FALSE \\ &= a \in t(H(a)) \\ &= find(a, t(H(a))) \end{aligned}$$

In summary,

$$find_2(a,t) = find(a, t(H(a)))$$

It can be easily observed that $find_2$ is nothing but $find$ tuned to the relevant collision bucket. Since the cardinality of $t(H(a))$ is smaller than that of $\bigcup_{i \in n} t(i)$ [14], the efficiency gain is obvious.

4.2.2. The $insert$ Operation

Refinement diagram:

$$\begin{array}{ccc} A \times 2^A & \xrightarrow{\quad insert \quad} & 2^A \\ {\scriptstyle 1_A \times (f_1 \circ f_2)} \uparrow & & \uparrow {\scriptstyle f_1 \circ f_2} \\ A \times ((2^A)^n)_{\Phi_2} & \xrightarrow[\quad insert_2 \quad]{} & ((2^A)^n)_{\Phi_2} \end{array}$$

Refinement equation:

$$f_1(f_2(insert_2(a,t))) = insert(a, f_1(f_2(t)))$$

Reasoning:

$$\begin{array}{rcl} f_1(f_2(insert_2(a,t))) & = & insert(a, f_1(f_2(t))) \\ & = & \{a\} \cup f_1(f_2(t)) \\ & = & \{a\} \cup \bigcup_{i \in n} t(i) \\ & = & \{a\} \cup t(H(a)) \cup \bigcup_{i \in n - \{H(a)\}} t(i) \\ & = & t'(H(a)) \cup \bigcup_{i \in n - \{H(a)\}} t(i) \end{array}$$

where t' is the one-entry mapping

$$t' = \left(\begin{array}{c} H(a) \\ \{a\} \cup t(H(a)) \end{array} \right)$$

Then,

$$\begin{array}{rcl} t'(H(a)) \cup \bigcup_{i \in n - \{H(a)\}} t(i) & = & \bigcup_{i \in n} \left\{ \begin{array}{lcl} i = H(a) & \Rightarrow & t'(i) \\ \neg(i = H(a)) & \Rightarrow & t(i) \end{array} \right. \\ & = & \bigcup_{i \in n} \left\{ \begin{array}{lcl} i \in dom(t') & \Rightarrow & t'(i) \\ \neg(i \in dom(t')) & \Rightarrow & t(i) \end{array} \right. \\ & = & \bigcup_{i \in n} (t \dagger t')(i) \\ & = & f_1(f_2(t \dagger t')) \end{array}$$

In summary,

$$f_1(f_2(insert_2(a,t))) = f_1(f_2(t \dagger \left(\begin{array}{c} H(a) \\ \{a\} \cup t(H(a)) \end{array} \right)))$$

[14] Provided that H is not a constant function, an obvious "bad" hash function.

or, by removing $f_1 \circ f_2$ from both sides of the equality,

$$insert_2(a,t) \quad = \quad t \dagger \left(\begin{array}{c} H(a) \\ insert(a, t(H(a)) \end{array} \right)$$

It can be observed that $insert_2$ is nothing but $insert$ confined to the relevant collision bucket, taking advantage of its smaller cardinality in subsequent refinements such as depicted in (31).

4.2.3. The $initialDb$ Operation

Refinement diagram:

$$\begin{array}{ccc} & \xrightarrow{initialDb} & 2^A \\ & & \uparrow f_1 \circ f_2 \\ & \xrightarrow[initialDb_2]{} & ((2^A)^n)_{\Phi_2} \end{array}$$

Refinement equation:

$$f_1(f_2(initialDb_2)) = initialDb$$

Reasoning: since

$$\bigcup_{i \in n} initialDb_2(i) \quad = \quad \{\}$$

one straightforwardly draws

$$initialDb_2 \quad = \quad \left(\begin{array}{c} i \\ \{\} \end{array} \right)_{i \in n}$$

We obtain the conventional "for loop" ($i = 1, n$) initialization of the hash table setting every bucket to the empty set.

4.2.4. The $remove$ Operation

Refinement diagram:

$$\begin{array}{ccc} A \times 2^A & \xrightarrow{remove} & 2^A \\ \uparrow 1_A \times f_1 \circ f_2 & & \uparrow f_1 \circ f_2 \\ A \times ((2^A)^n)_{\Phi_2} & \xrightarrow[remove_2]{} & ((2^A)^n)_{\Phi_2} \end{array}$$

Refinement equation:

$$f_1(f_2(remove_2(a,t))) = remove(a, f_1(f_2(t)))$$

Reasoning:

$$\begin{array}{rcl} f_1(f_2(remove_2(a,t))) & = & remove(a, f_1(f_2(t))) \\ & = & f_1(f_2(t)) - \{a\} \end{array}$$

$$
\begin{array}{lll}
 & = & \bigcup_{i \in n} t(i) - \{a\} \\
 & = & (\bigcup_{i \in n} t(i)) \cap \overline{\{a\}} \\
 & = & \bigcup_{i \in n} (t(i) \cap \overline{\{a\}}) \\
 & = & \bigcup_{i \in n} (t(i) - \{a\}) \\
 & = & (t(H(a)) - \{a\}) \cup \bigcup_{i \in n - \{H(a)\}} (t(i) - \{a\}) \\
 & = & t'(H(a)) \cup \bigcup_{i \in n - \{H(a)\}} \underbrace{t(i) - \{a\}}_{(*)}
\end{array}
\tag{33}
$$

where t' is the one-entry mapping

$$t' = \left(\begin{array}{c} H(a) \\ t(H(a)) - \{a\} \end{array} \right)$$

At this point one resorts to invariant Φ_2 (20), from which we infer

$$
\begin{array}{c}
\forall i \in n : (\forall a \in t(i) : i = H(a)) \\
\updownarrow \\
\forall i \in n, a \in A : a \in t(i) \Rightarrow i = H(a) \\
\updownarrow \\
\forall i \in n, a \in A : i \neq H(a) \Rightarrow a \notin t(i) \\
\updownarrow \\
\forall i \in n, a \in A : i \neq H(a) \Rightarrow t(i) - \{a\} = t(i)
\end{array}
$$

which is enough to reduce $(*)$ in (33) above to $t(i)$. The reasoning here-upon is similar to that carried out for $insert$, leading to

$$f_1(f_2(remove_2(a,t))) = f_1(f_2(t \dagger \left(\begin{array}{c} H(a) \\ t(H(a)) - \{a\} \cup t(H(a)) \end{array} \right)))$$

i.e. (by removing $f_1 \circ f_2$ from both sides of the equality)

$$remove_2(a,t) = t \dagger \left(\begin{array}{c} H(a) \\ remove(a, t(H(a)) \end{array} \right)$$

Similarly to $insert_2$, it can be observed that $remove_2$ is nothing but $remove$ confined to the relevant collision bucket. But, unlike $insert_2$, $remove_2$ resorted to the low-level ad hoc invariant in its refinement process.

5. A Standard Elaboration

The algebra of finite subsets of A, 2^A (12), is far too simple a model for database files. A model closer to reality is

$$Database' = A \hookrightarrow B \tag{34}$$

where A plays the rôle of a domain of keys and B is the data of interest [15].

Note that $2^A \cong_{dom} A \hookrightarrow 1$ (58). So it seems that the new model amounts to generalizing 1 in functor $\lambda X.X \hookrightarrow 1$ to any finite B. And indeed each operation σ over (12) can be generalized to some σ' over (34) provided that σ' "preserves σ along the dom morphism", that is, if σ is such that

$$dom(\sigma'(x,\ldots)) = \sigma(dom(x),\ldots)$$

We obtain the classical "dictionary functionality" [16]:

$$\begin{array}{rcl} initialDb' & : & \longrightarrow Database' \\ initialDb' & \stackrel{\text{def}}{=} & (\ \ ) \end{array}$$

$$\begin{array}{rcl} insert' & : & A \times B \times Database' \longrightarrow Database' \\ insert'(a,b,\sigma) & \stackrel{\text{def}}{=} & \left\{ \begin{array}{rcl} a \in dom(\sigma) & \Rightarrow & \sigma \\ \neg(a \in dom(\sigma)) & \Rightarrow & \sigma \cup \left(\begin{array}{c} a \\ b \end{array} \right) \end{array} \right. \end{array}$$

$$\begin{array}{rcl} find' & : & A \times Database' \longrightarrow B \cup \{NIL\} \\ find'(a,\sigma) & \stackrel{\text{def}}{=} & \left\{ \begin{array}{rcl} a \in dom(\sigma) & \Rightarrow & \sigma(a) \\ \neg(a \in dom(\sigma)) & \Rightarrow & NIL \end{array} \right. \end{array}$$

$$\begin{array}{rcl} remove' & : & A \times Database' \longrightarrow Database' \\ remove'(a,\sigma) & \stackrel{\text{def}}{=} & \sigma \setminus \{a\} \end{array}$$

With respect to data refinement, this generalization is not immediate since functor $\lambda X.X \hookrightarrow B$ is not so well-behaved as $\lambda X.2^X$ or any other in table 2 (see [OB97]). But for $f : A \to C$ an isomorphism,

$$f \hookrightarrow B : (A \hookrightarrow B) \longrightarrow (C \hookrightarrow B)$$

is well-defined [17]:

$$(f \hookrightarrow B)(\sigma) = \left(\begin{array}{c} f(a) \\ \sigma(a) \end{array} \right)_{a \in dom(\sigma)} \tag{35}$$

Clearly, theorem 4 holds also for $\lambda X.X \hookrightarrow B$:

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

[15] The "mapping" model of dictionary tables or database files specified by (34) is well-known from the literature, see *e.g.* [Fie80].

[16] We assume $NIL \notin B$ in $find'$ for the sake of simplicity, otherwise $B + 1$ should replace $B \cup \{NIL\}$ in the signature. NIL is the counterpart of $a \in dom(\sigma) = FALSE$ in (13) and $a \in dom(\sigma) = TRUE$ is generalized to any value of B. This follows a technique based on SETS and discussed elsewhere [Oli97] for elaborating or sophisticating specification models.

[17] Ill-definedness will arise wherever $\exists a, a' \in dom(\sigma) : \sigma(a) \neq \sigma(a') \wedge f(a) = f(a')$ in (35). This situation is discarded wherever f is injective (ruling out $f(a) = f(a')$) or B is a singleton set (ruling out $\sigma(a) \neq \sigma(a')$). The latter condition explains why $A \hookrightarrow 1$ (*i.e.* 2^A) is well-behaved.

$$= \quad (A \hookrightarrow B)_{\phi \hookrightarrow B}$$

Now revisit (16), our starting point in section 4.1:

$$A \quad \trianglelefteq^{\phi}_{\pi_2|\phi} \quad n \times A$$

In fact, $\pi_2|\phi$ is an isomorphism, for a fixed H. So

$$A \hookrightarrow B \quad \trianglelefteq^{\phi \hookrightarrow B}_{(\pi_2|\phi) \hookrightarrow B} \quad (n \times A) \hookrightarrow B$$

Now

$$(n \times A) \hookrightarrow B \quad \cong \quad (A \hookrightarrow B)^n \tag{36}$$

cf. law (62). So we have obtained the "hash effect" again, in a step which generalizes (19),

$$((n \times A) \hookrightarrow B)_{\phi \hookrightarrow B} \quad \cong_{f^{-1}_{(62)}} \quad (A \hookrightarrow B)^n_{\Phi_2}$$

where

$$\begin{aligned} \Phi_2 &= (\phi \hookrightarrow B) \circ f^{-1}_{(62)} \\ &= \lambda\sigma.\forall x \in dom(f^{-1}_{(62)}(\sigma)) : \phi(x) \\ &= \lambda\sigma.\forall i \in n : (\forall a \in dom(\sigma(i)) : i = H(a)) \end{aligned}$$

Collision handling may proceed now by resorting to the implementation theory of $A \hookrightarrow B$, see *e.g.* [Fie80]. But law (71) provides an easy way of re-using our earlier results by implementing abstract mappings as sets of pairs satisfying a functional dependency, *cf.* fdp (77). So we are back to n-dimensional arrays of sets,

$$(2^{A \times B})^n_{\Phi_3}$$

satisfying not only the ad hoc hash-constraint (Φ_2) but also the extra functional dependency requirement:

$$\begin{aligned} \Phi_3 &= fdp^n \wedge \Phi_2 \circ mkf^n \\ &= \lambda\sigma.\forall i \in n : fdp(\sigma(i)) \wedge (\forall a \in \pi_1[\sigma(i)] : i = H(a)) \end{aligned}$$

In the end we will obtain data structure (29) — or (30) — again, where the `struct` — or `record` — constructs are enriched with an extra B-field. The operations will be reified much in the same way as in section 4.2.

6. Conclusions

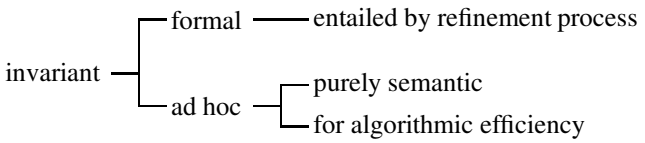
For programmers and program users it is important to know not only which properties hold for the programs they deal with, but also *why* and *when* they arise in the development process.

Particularly relevant properties of programs are those which hold forever along programs lifetime — these are called *invariant properties*. Other such "eternal" properties may hold as consequences of the underlying invariants.

Data invariants are a burden for the software developer because formal arguments have to be produced ensuring that they are preserved by all the operations which create the corresponding data. So, in terms of design effort, it is important to have as weak overall data type invariants as possible.

Traditional engineering pragmatics dealing with continuous domains (*e.g.* space, time, temperature, voltage) normally impose so-called "safety overheads" (*e.g.* "10% stronger than calculated") in order to guarantee design implementation, in face of instrumental inaccuracy, for instance. The same "culture" applied to informal software design tends to overstrengthen presumed properties (invariants) of software artifacts. But a contrast can be found here: discreteness of data (supported by inductive arguments) should make software (theoretically) 100% accurate!

The aim of this paper has been to provide evidence that invariants can not only be kept accurately under control but also that a classification can be given for every implementation-level invariant component, according to its origin in design:



as commented next:

- *formal invariants entailed by the refinement process* — the SETS calculus ensures these as weak as necessary in order to guarantee faithful data representation;
- *purely semantic ad hoc invariants* — arbitrarily stated at the highest specification level, these invariants mirror real-life conventions, rules, norms or natural constraints [18];
- *ad hoc invariants for algorithmic efficiency* — arbitrarily stated at intermediate or final reification levels, these invariants exploit the target machine architecture for efficiency.

The process of formally calculating the hash table implementation of a database file, undertaken in this paper, has shown how early in design ad hoc invariants of the last kind above can be introduced in software development — recall invariant (15) introduced in step 1 of (31).

The SETS calculus [Oli90, Oli92] has been extended in this paper in order to handle such invariants in a systematic way, via a result (theorem 3) which "pushes" such invariants down the refinement process, and another result (theorem 5) which ensures that ad hoc invariant preservation arguments need to be produced only once.

The chosen case study (hash tables) — which is part of an on-going research plan aiming at producing a *reification cook-book* of software problem (formal) solutions — has provided evidence, once again, of the benefits of addressing traditional algorithmic "culture" from a formal perspective. In [Oli96] a wider spectrum of hashing reification is systematically expounded, namely in respect of collision handling. For instance, one may organize synonyms as binary search collision trees rather that linear collision lists, *etc.*

7. Future Work

Future research may proceed in several directions.

[18] The invariant of $Date$ — recall (5) — is a telling example of this class of invariants.

7.1. The "Hash Effect" in General

In the first place, it seems that the hash-table paradigm is more general then it appears as first sight. The "hash effect" is not a privilege of "tabular" data structures, and a general criterion for deciding which data-structures can be "optimized" by "hashing them" arises from generalizing "hashing steps" (18) and (36) as follows.

Let $G(X)$ be a data-structure parameterized by X, that is, a SETS endofunctor. A condition necessary for applying the "hashing step" to G certainly is

$$\mathcal{G}(n \times A) \quad \cong \quad (\mathcal{G}(A))^n \tag{37}$$

for $n > 0$ and a given hash function $H : A \longrightarrow n$. While instances $G(X) = 2^X$ and $G(X) = X \hookrightarrow B$ (for some B) were studied in this paper, other interesting Gs such as $G(X) = B \hookrightarrow X$ or $G(X) = X^*$ do not satisfy (37). However, (37) is too strong and may be relaxed to

$$\mathcal{G}(n \times A) \quad \trianglelefteq \quad (\mathcal{G}(A))^n \tag{38}$$

as can be easily checked. This opens the hashing optimization to many other data-structures. For instance, let

$$Book \hookrightarrow Publisher$$

be a database file assigning books to publishers. One may hash this data-structure on $Book$, as we have seen (let $G(X) = X \hookrightarrow Publisher$). But, if we are searching for books by publishers, a "hash-on-publisher" structue would be perhaps more effective, that is, one would like to hash for $G(X) = Book \hookrightarrow X$. This in fact makes sense with respect of (38), since

$$B \hookrightarrow n \times A \quad \trianglelefteq \quad (B \hookrightarrow A)^n$$

holds as a generalization of law (69) [19] entailing an invariant enforcing mutually disjoint domains of the n mappings obtained on the left.

For $G(X) = X^*$ we apparently have

$$(n \times A)^* \quad \cong \quad (A^*)^n \tag{39}$$

since both sides of (39) have the same cardinality ($\aleph_0$). Yet intuition may tell us that X^* is an "un-hashable" data-structure, for it seems impossible to reconstruct, from the n sequences on the right-hand side of (39), the original interleaving of such sequences. This calls for further research [20].

---

[19] Recall (49) and (50) for the necessary simplifications.

[20] A surjection can be easily found from the left-hand side of (39) to the right-hand side,

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

It is the opposite surjection which is difficult to define. In this respect, see [Bar95] for an extension of the $\trianglelefteq$-ordering based on ordinal rather than cardinal numbers. [Bar95] shows that ordinal numbers can be of help to filter implementation bias [Jon80] but fail to solve this problem.

7.2. Distribution and Concurrency

Another direction for generalizing the hash-effect has to do with distribution and concurrency. From (49) we draw

$$(\mathcal{G}(A))^n \quad \cong \quad \underbrace{\mathcal{G}(A) \times \ldots \mathcal{G}(A)}_{n}$$

which means that a hashed structure, regarded as a monolithic structure, can at compile-time be factored into a collection of n similar (sub-)structures — a kind of "fractal" effect on data structuring. From another perspective, this can be viewed as a step in the *horizontal refinement* [Gog86] of a state-based software component [Oli91, OC93] into n "smaller" components, which can be regarded as concurrent, communicating processes. On-going research [OBM97] seems to provide evidence that the "amount of concurrency" gained in a *horizontal refinement* step can, if expressed in SETS, be calculated and reasoned about. This is particularly relevant concerning hashed structures, whose "gain in concurrency" is the fact that 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 $i \neq j$).

In summary, regarding the hash-effect as mere data optimization is a limited view of hashing. Instead of "data hashing" one should talk about "process hashing", a topic also addressed in [OBM97].

7.3. Hashing as a Classification Device

Last but not least, the hash effect can be reduced to a *classification mechanism*. In the literal sense, classification means "organize into classes". The recent outspread of object-oriented modelling and programming has made the advantage of (hierarchical) classification evident to the programming community. Hashing, which is much older a programming device, can be regarded as "flat", or unordered classification. In fact, let us interpret $A \hookrightarrow B$ as a *B-classification of A*. That is, for a given $\sigma \in (A \hookrightarrow B)$ and $a \in A$, just regard $\sigma(a) \in B$ as "the name of the class to which a belongs".

Any database file $db \in 2^A$ can always be represented by some classification of its data since

$$2^A \quad \trianglelefteq_{dom} \quad A \hookrightarrow B \tag{40}$$

for $B \neq 0$. The class-building effect is captured by (72), a transformation of $A \hookrightarrow B$ which renders the "kernel quotient" explicit: from each "class name" $b \in B$ we retrieve the corresponding members (2^A_+). But it is well-known that

$$B \hookrightarrow 2^A_+ \quad \cong \quad \left(2^A\right)^B$$

the right-hand side of which can be recognized as the right-hand side of (18) for the $B = n$ instantiation. The rôle of the hashing-function H is just that of enforcing a fixed (run-time independent) classification, *i.e.* enforcing that datatype invariant

$$\lambda\sigma.\forall a \in dom(\sigma) : \sigma(a) = H(a)$$

holds over $A \hookrightarrow n$ (40).

7.4. Final Remarks

The exercise reported in this paper has contributed to the evolution of the SETS calculus in a way which deserves some final comments.

First, one may regard the hash effect on operations (recall section 4.2) as a special kind of *promotion* as understood in the Z literature [Woo92] or of *sophistication* as understood in [Oli97].

Second, it seems from section 5 that the functoriality of SETS can be extended to functors other than $\lambda X.2^X$ or those of table 2. This topic is addressed in [OB97].

Third, obvious extensions to two SETS theorems are on demand: one is the extension of theorem 4 to recursive $\mathcal{F}$, leading to

$$\mu\mathcal{F}(A_\phi) = (\mu\mathcal{F}(a))_{\mu\mathcal{F}(\phi)}$$

which can be accomplished in the context of categorial fixpoint semantics [MA86]; the other is perhaps more laborious and is concerned with extending a recursion-removal theorem central to SETS [21]:

Theorem 6. Let $X = \mathcal{F}(X)$ be a recursive definition in SETS and let $\langle X_0, \delta \rangle$ be a fixpoint of $\mathcal{F}$. Let $\mathcal{G}$ be another functor in SETS. Then

$$\mathcal{G}(X_0) \trianglelefteq (K \hookrightarrow \mathcal{F}(K)) \times \mathcal{G}(K)$$

□

Acknowledgements

This paper finds its earliest motivation in the invariant handling rules proposed by Isabel Jourdan (Dept. of Mathematics, University of Coimbra) in her M. Sc. project supervised by the author [Jou92].

The current version of the paper took much longer to write than originally planned because it demanded a series of developments in SETS which were not available at the time. In the meantime, it has become partly available in the lecture notes of the author's M. Sc. course on reification [Oli94a] and as a technical report [Oli94b]. Special thanks go to Luís Barbosa and César Rodrigues for many helpful discussions on the foundations of SETS.

This research has been partly supported by the JNICT council under R&D Contract ref. PMCT/TIT/169.90.

References

[Bac78] 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.

[Bar95] L. S. Barbosa. Ordinal reification. Technical Report DI-LSB-95:7:1, Departamento de Informática, Universidade do Minho, July 1995.

[Fie80] E. Fielding. The specification of abstract mappings and their implementation as B^+-trees. Technical Report PRG-18, Oxford University, September 1980.

[Gof84] J. Le Goff. *Calendário*, volume I, chapter 8, pages 260–292. I.N.-C.M., 1984. EINAUDI Encyclopedia (Portuguese translation).

[21] See [Oli92, Jou92]. This extension, which consists in generalizing X_0 to $\mathcal{G}(X_0)$, is indeed required by step (27) in section 4.1.1. But this lies outside the scope of this paper and is thus left for future work. This should include comparing these SETS results with work developed by Möller [Moe93] in the same area.

[Gog86] J. A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16–28, 1986.

[HS78] E. Horowitz and S. Sahni. *Fundamentals of Data Structures*. Computer Software Engineering Series. Pitman, 1978. E. Horowitz (Ed.).

[Jon80] C. B. Jones. *Software Development — A Rigorous Approach*. Series in Computer Science. Prentice-Hall International, 1980. C. A. R. Hoare.

[Jon86] C. B. Jones. *Systematic Software Development Using* VDM. Series in Computer Science. Prentice-Hall International, 1986. C. A. R. Hoare.

[Jou92] I. S. Jourdan. Reificação de tipos abstractos de dados: Uma abordagem matemática. Master's thesis, University of Coimbra, 1992. (In Portuguese).

[MA86] E. G. Manes and M. A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1986. D. Gries.

[MG90] C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.

[Moe93] B. Moeller. Derivation of graph and pointer algorithms. In S.A. Schuman, B. Möller, and H.A. Partsch, editors, *Formal Program Development*, number 755 in Lecture Notes in Computer Science, pages 123–160. Springer, 1993.

[OB97] J. N. Oliveira and L. S. Barbosa. SETS (subtyped) polymorphism as natural transformation. Technical report, DI/INESC, Jan. 1997.

[OBM97] J. N. Oliveira, L. S. Barbosa, and F. S. Moura. Can distribution be (statically) calculated? Technical report, DI/INESC, 1997. (In preparation).

[OC93] J. N. Oliveira and A. M. Cruz. Formal calculi applied to software component knowledge elicitation. Technical Report C19-WP2D, DI/INESC, December 1993. IMI Contract *C.1.9. Sviluppo di Metodologie, Sistemi e Servizi Innovativi in Rete*.

[Oli90] J. N. Oliveira. A reification calculus for model-oriented software specification. *Formal Aspects of Computing*, 2:1–23, April 1990.

[Oli91] J. N. Oliveira. *Especificação Formal de Programas*. University of Minho, 1^{st} edition, 1991. Lecture Notes for M.Sc. Course in Computing (in Portuguese; incl. extended abstract in English; last edition 1994).

[Oli92] 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).

[Oli94a] J. N. Oliveira. *Desenvolvimento Formal de Programas*. University of Minho, 3^{rd} edition, 1994. Lecture Notes for M.Sc. Course in Computing.

[Oli94b] J. N. Oliveira. Hash tables — a case study in $\trianglelefteq$-calculation. Technical Report DI/INESC 94-12-1, DI/INESC, U. Minho, December 1994.

[Oli96] J. N. Oliveira. *Métodos Formais de Programação*. Univ. of Minho, 3^{rd} edition, 1996. Textbook in preparation. Postscript of current version (427pages in Portuguese) available from `http://www.di.uminho.pt/~jno/ps/mfp.ps.tar.gz`.

[Oli97] J. N. Oliveira. Data sophistication is not data refinement. Technical report, DI/INESC, U. Minho, 1997. (In preparation).

[Rod93] C. J. Rodrigues. Sobre o desenvolvimento formal de bases de dados. Master's thesis, University of Minho, 1993. (In Portuguese).

[Rod97] C. J. Rodrigues. Categorial pre-order calculi. Technical report, University of Minho, Jan. 1997. (Submitted to FME'97, Grätz, Austria, Sept. 1997).

[Spi89] J. M. Spivey. *The Z Notation — A Reference Manual*. Series in Computer Science. Prentice-Hall International, 1989. C. A. R. Hoare.

[Wir76] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

[Woo92] J. C. P. Woodcock. Implementing promoted operations in Z. In *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 367–378. Springer-Verlag, 8–10 January 1992. (Invited paper).

A. SETS—The Equational Calculus

It is not the aim of this and the following appendices to present the collection of all SETS results developed so far in works such as [Oli90, Oli92, Jou92, Rod93, OC93]. The idea is to present a minimally representative superset of the results actually used in the main

body of this paper. Some laws may be regarded as instances or consequences of others, as can be easily checked [22].

Capital letters $A, B, C, \ldots$ will denote SETS objects, *i.e.* data domains. Small letters $n, m, \ldots$ will denote initial segments of $\mathbb{N}$, the set of natural numbers. Subscripted f and ϕ denote abstraction functions and concrete invariants, respectively. Should subscript be equation numbers, these will indicate the laws where the given maps or predicates arise.

A.1. Underlying Abelian Semi-ring

$$A \times B \cong B \times A \tag{41}$$
$$A \times (B \times C) \cong (A \times B) \times C \tag{42}$$
$$A \cong A \times 1 \tag{43}$$
$$A + B \cong B + A \tag{44}$$
$$A + (B + C) \cong (A + B) + C \tag{45}$$
$$A + 0 \cong A \tag{46}$$
$$A \times 0 \cong 0 \tag{47}$$
$$A \times (B + C) \cong (A \times B) + (A \times C) \tag{48}$$

$$\underbrace{A \times \ldots \times A}_{n} \cong A^n \tag{49}$$
$$\underbrace{A + \ldots + A}_{n} \cong n \times A \tag{50}$$

A.2. Exponentiation and Derivatives

$$A^0 \cong 1 \tag{51}$$
$$A^1 \cong A \tag{52}$$
$$1^A \cong 1 \tag{53}$$
$$A^{(B+C)} \cong A^B \times A^C \tag{54}$$
$$(B \times C)^A \cong B^A \times C^A \tag{55}$$
$$A^{B \times C} \cong (A^C)^B \tag{56}$$
$$A \hookrightarrow B \cong (B + 1)^A \tag{57}$$
$$2^A \cong A \hookrightarrow 1 \tag{58}$$
$$(B + C) \hookrightarrow A \cong (B \hookrightarrow A) \times (C \hookrightarrow A) \tag{59}$$
$$0 \hookrightarrow A \cong 1 \tag{60}$$
$$1 \hookrightarrow A \cong A + 1 \tag{61}$$
$$(A \hookrightarrow B)^C \cong (C \times A) \hookrightarrow B \tag{62}$$

[22] For instance, law (71) is obtained from law (70) by making $C = 1$, etc.

B. SETS—The Inequational Calculus

In this appendix (as in the main text), $A \preceq B$ is a shorthand of $A \trianglelefteq B$ wherever the abstraction map is a total function, that is, the concrete invariant is the everywhere $TRUE$ predicate.

$$A \preceq A \times B \tag{63}$$
$$B \preceq A \times B \tag{64}$$
$$2^A \preceq A^\star \tag{65}$$
$$A \hookrightarrow (B \times C) \trianglelefteq (A \hookrightarrow B) \times (A \hookrightarrow C) \tag{66}$$
$$(A \times B) \hookrightarrow C \trianglelefteq A \hookrightarrow (B \hookrightarrow C) \tag{67}$$
$$A \hookrightarrow D \times (B \hookrightarrow C) \trianglelefteq (A \hookrightarrow D) \times ((A \times B) \hookrightarrow C) \tag{68}$$
$$A \hookrightarrow (B + C) \trianglelefteq (A \hookrightarrow B) \times (A \hookrightarrow C) \tag{69}$$
$$A \hookrightarrow B \times C \trianglelefteq (A \times B) \hookrightarrow C \tag{70}$$
$$A \hookrightarrow B \trianglelefteq 2^{A \times B} \tag{71}$$
$$A \hookrightarrow B \trianglelefteq B \hookrightarrow 2^A_+ \tag{72}$$

C. Abstraction Invariants

For basic functions and predicates occurring below see section C.3.

C.1. Abstraction Maps

$$f_{(41)} = [\pi_2, \pi_1]$$
$$f_{(42)} = [\pi_1 \circ \pi_1, [\pi_2 \circ \pi_1, \pi_2]]$$
$$f_{(43)} = \pi_1$$
$$f_{(44)} = i_2 \mid i_1$$
$$f_{(45)} = (i_1 \mid i_2 \circ i_1) \mid (i_2 \circ i_2)$$
$$f_{(46)} = i_1$$
$$f_{(47)} = 1_0$$
$$f_{(48)} = [\pi_1, i_1 \circ \pi_2] \mid [\pi_1, i_2 \circ \pi_2]$$
$$f_{(49)} = \lambda\sigma.\langle\sigma(1), \ldots, \sigma(n)\rangle$$
$$f_{(50)} = \lambda x.x$$
$$f_{(51)} = \lambda x.(\quad)$$
$$f_{(52)} = \lambda a.\begin{pmatrix} NIL \\ a \end{pmatrix}$$
$$f_{(53)} = \lambda x.\begin{pmatrix} a \\ NIL \end{pmatrix}_{a \in A}$$
$$f_{(54)} = \lambda\langle\sigma_1, \sigma_2\rangle.A^{i_1}(\sigma_1) \cup A^{i_2}(\sigma_2)$$

$$
\begin{aligned}
f_{(55)} &= \lambda\langle x, y\rangle.\left(\begin{array}{c} a \\ \langle x(a), y(a)\rangle \end{array}\right)_{a\in A} \\
f_{(56)} &= uncurry \\
f_{(57)} &= \lambda\sigma.\left(\begin{array}{c} a \\ b \end{array}\right)_{a\in A\wedge\sigma(a)=i_1(b)} \\
f_{(58)} &= dom \\
f_{(59)} &= \lambda\langle\sigma_1,\sigma_2\rangle.(i_1 \hookrightarrow A)(\sigma_1)\cup(i_2 \hookrightarrow A)(\sigma_2) \\
f_{(60)} &= \lambda x.(\quad) \\
f_{(61)} &= \lambda x.\left\{\begin{array}{lcl} x = i_1(a) & \Rightarrow & \left(\begin{array}{c} NIL \\ a \end{array}\right) \\ \neg(x = i_1(a)) & \Rightarrow & (\quad) \end{array}\right. \\
f_{(62)} &= \lambda\sigma.\left(\begin{array}{c} c \\ \left(\begin{array}{c} \pi_1(p) \\ \sigma(p) \end{array}\right)_{p\in dom(\sigma)\wedge\pi_2(p)=c} \end{array}\right)_{c\in C} \\
f^{-1}_{(62)} &= \lambda\sigma.\bigcup_{c\in C}\left(\begin{array}{c} \langle c, a\rangle \\ \sigma(c)(a) \end{array}\right)_{a\in dom(\sigma)(c)} \\
f_{(63)} &= \pi_1 \\
f_{(64)} &= \pi_2 \\
f_{(65)} &= elems \\
f_{(66)} &= \bowtie \\
f_{(67)} &= \lambda\sigma.\left(\begin{array}{c} \langle a, b\rangle \\ (\sigma(a))(b) \end{array}\right)_{a\in dom(\sigma)\wedge b\in dom(\sigma(a))} \\
f_{(68)} &= \bowtie_n \\
f_{(69)} &= \stackrel{+}{\bowtie} \\
f_{(70)} &= \lambda\sigma.\left(\begin{array}{c} a \\ \langle b, \sigma(\langle a, b\rangle)\rangle \end{array}\right)_{\langle a,b\rangle\in dom(\sigma)} \\
f^{-1}_{(70)} &= \lambda\sigma.\left(\begin{array}{c} \langle a, \pi_1(\sigma(a))\rangle \\ \pi_2(\sigma(a)) \end{array}\right)_{a\in dom(\sigma)} \\
f_{(71)} &= mkf \\
f_{(72)} &= finv
\end{aligned}
$$

C.2. Concrete Invariants

$$
\begin{aligned}
\phi_{(66)} &= eqd \\
\phi_{(67)} &= \lambda\sigma.(\quad) \notin rng(\sigma) \\
\phi_{(68)} &= dpi \\
\phi_{(69)} &= djd \\
\phi_{(70)} &= fdp \circ dom
\end{aligned}
$$

$$\phi_{(71)} = fdp$$
$$\phi_{(72)} = mdjfam$$

C.3. Basic Functions and Predicates

$$djd(\sigma,\tau) \stackrel{\text{def}}{=} dom(\sigma) \cap dom(\tau) = \{\} \tag{73}$$
$$dpi(\sigma,\tau) \stackrel{\text{def}}{=} \pi_1[dom(\tau)] \subseteq dom(\sigma) \tag{74}$$
$$eqd(\langle\sigma,\tau\rangle) \stackrel{\text{def}}{=} dom(\sigma) = dom(\tau) \tag{75}$$
$$elems(b) \stackrel{\text{def}}{=} \{b(i) \mid i \in length(b)\} \tag{76}$$
$$fdp(r) \stackrel{\text{def}}{=} \forall t,t' \in r : \pi_1(t) = \pi_1(t') \Rightarrow \pi_2(t) = \pi_2(t') \tag{77}$$
$$mdjfam(\sigma) \stackrel{\text{def}}{=} \forall b,b' \in dom(\sigma) : b \neq b' \Rightarrow \sigma(b) \cap \sigma(b') = \{\} \tag{78}$$
$$i_1(a) \stackrel{\text{def}}{=} \langle 1,a\rangle \tag{79}$$
$$i_2(a) \stackrel{\text{def}}{=} \langle 2,a\rangle \tag{80}$$
$$\sigma \bowtie \tau \stackrel{\text{def}}{=} \left(\begin{array}{c} a \\ \langle \sigma(a), \tau(a)\rangle \end{array} \right)_{a \in dom(\sigma)} \tag{81}$$
$$\overset{+}{\bowtie}(\sigma,\tau) \stackrel{\text{def}}{=} i_1 \circ \sigma \cup i_2 \circ \tau \tag{82}$$
$$\underset{\pi}{\bowtie}(\sigma,\tau) \stackrel{\text{def}}{=} \left(\begin{array}{c} a \\ \langle \sigma(a), \left(\begin{array}{c} b \\ \tau(a,b) \end{array} \right)_{\langle a,b\rangle \in sel(a,\tau)} \rangle \end{array} \right)_{a \in dom(\sigma)} \tag{83}$$
$$mkf(\rho) \stackrel{\text{def}}{=} \left(\begin{array}{c} a \\ the(\{b \in B \mid a\rho b\}) \end{array} \right)_{a \in \pi_1[\rho]} \tag{84}$$
$$sel(a,\tau) \stackrel{\text{def}}{=} \{\langle a',b\rangle \in dom(\tau) \mid a' = a\} \tag{85}$$
$$uncurry(t) \stackrel{\text{def}}{=} \left(\begin{array}{c} \langle b,c\rangle \\ t(b)(c) \end{array} \right)_{b \in B \wedge c \in C} \tag{86}$$
$$finv(\sigma) \stackrel{\text{def}}{=} \bigcup_{b \in dom(\sigma)} \left(\begin{array}{c} a \\ b \end{array} \right)_{a \in \sigma(b)} \tag{87}$$