Universidade do Minho
DI/INESC

Campus de Gualtar,
4700 Braga
Portugal

Tel: +351-53-604470
Fax: +351-53-612954

Technical Report Series
DI/INESC-94-12-1

J.N. Oliveira

Hash Tables: A Case Study in $\trianglelefteq$-Calculation

Hash Tables: A Case Study in $\trianglelefteq$-Calculation

by

J.N. Oliveira,
DI/INESC
Universidade do Minho,
Gualtar,
4700 Braga
Portugal

December 1994

Abstract

This report describes the process of using the SETS calculus to formally derive the *hash table* implementation of a finite collection of data.

This exercise is a suggestive illustration about handling so-called "ad hoc" data type invariants, that is, invariants not implied by the reification process which are enforced into an implementation to obtain extra advantages (typically, efficiency).

The report 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 report provides a final generalization of the "hashing construction".

Contents

1 Introduction

The purpose of this report is to describe the process of "calculating" the *hash table* [Wir76, HS19] implementation of an abstract database, as an example of handling so-called *"ad hoc" data type invariants* in the SETS calculus [Oli90, Oli92]. For economy of presentation, acquaintance with these references is assumed and only a synopsis of SETS is presented.

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

1.1 Hashing

Hash tables are well known data structures [Wir76, HS19] 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 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*,

$$H : Date \longrightarrow Location$$

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* [HS19]. In this report we will only address the latter.

2 A Synopsis of SETS

This section summarizes the SETS notation for constructive set-theory-based data-type specification, adopted in this report. SETS should not be regarded as a competitor to META-IV [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 — product ($A \times B$), exponentiation (A^B) and co-product ($A + B$) [Oli90].

2.1 The Notation

Product and exponentiation are defined in the usual manner,

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

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 [1]. 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):

$$2^A$$

[1] As in the VDM style notation [Jon80], for instance.

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

SETS	PASCAL	C/C++	**Descrição informal**

$A \times B$	`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;`
		`};`
			'Records' variantes

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

- 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):

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

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

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

  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 $\emptyset$ [2].

- union types [3] (*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:

$*$
$-$
$\times$
$\rightharpoonup$
$+$

For instance, expression

$$X \rightharpoonup A^2 \times B* + C$$

[2] 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, which retains their cardinality only. For instance, instead of enumerating

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

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).

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

will abbreviate

$$(X \rightharpoonup ((A^2) \times (B*))) + C$$

etc.

Properties of these 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 [4].

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 */
        subTrees *R;
        } ;

typedef struct subTrees {
        node *first;
        struct subTrees *next;
        } ;

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

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

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

that is, $DecTree \cong \mathcal{F}(DecTree)$ for

$$\mathcal{F}(X) \cong What \times (Answer \rightharpoonup X)$$

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 [5]

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

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,

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

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

[5] See section 2.2.

Table 2: $Sets$ vs Model Specification.

Sets	VDM [Jon80]	Z [Spi89]	ME-TOO [Hen84]	**Description**
A × B	A B	A × B	tup(A,B)	Tuples
A + B	A | B			Unions
A + 1	[A]			Omissions
B^A				"Arrays"
2^A	A-set	**P** A	set(A)	Finite sets
$A \rightharpoonup B$	$A \stackrel{m}{\rightarrow} B$	$A \nrightarrow B$	ff(A,B)	Mappings
$A*$	A-list	seq A	seq(A)	Finite lists

that is, the following "linear equation":

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

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;
        } ;
```

Table 2 presents a comparison between SETS and other model-oriented specification notations. 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.*

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

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

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

In general, conditional expressions such as

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

are written thus:

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

Partial expressions are special cases of conditional expressions,

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

meaning

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

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 ($A \rightharpoonup B$) 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

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

2.2 The Calculus

SETS [Oli90, Oli92] is an (inequational) calculus of (functional) abstraction invariants. Each law of Set is an inequation of the form

$$A \trianglelefteq_f^{\phi} B \tag{1}$$

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 (1) 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],

$$A \trianglelefteq_f B \tag{2}$$

is accepted as a shorthand of (1) wherever f is a total function, that is, $\phi(b) = TRUE$.

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 (2) will indicate laws which do not induce concrete invariants;
- in many reasoning situations, both f and ϕ in (1) can be left out until explicitly demanded by algorithm synthesis [Oli92]; all laws used along the report are provided with the relevant abstraction map f and invariant ϕ, listed respectively in appendix C.2 and appendix C.3 [6]. Equation numbers are used consistently in the identification of such functions and predicates: $f_{(\text{n})}$ and $\phi_{(\text{n})}$ are respectively the abstraction map and invariant associated with law (n).

[6] Other less obvious fs and ϕs are also given in this appendix, which is a subset of a similar appendix in [Oli91].

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 [7] which formalize real-life conventions, rules, norms or natural constraints. For instance, even a primitive datatype such as $Date$,

  $$Date \cong 31 \times 12 \times I\!N$$

  requires a nontrivial 'ad hoc' invariant:

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

  where

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

  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 a real number — and from several historical attempts to devise a norm able to finitely approximate such a number [8].

- 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 (70), 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 $A*$ which are ordered by some total order on A, which is irrelevant *wrt.* the abstraction function $elems$ (81).

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 restiction of f to B_ϕ:

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

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:

$$A \cong B_\phi \quad (4)$$

Such is the case of

$$2^A \quad \cong_{elems|linear} \quad (A*)_{linear} \quad (5)$$

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

[8] The situation is actually more complicated, for the 1582 discontinuity (10 days omitted in October of that year) of the Gregorian calendar was adopted much later in a few countries — *e.g.* Poland (1586), Hungary (1587), Germany (1700), England (1752) *etc.* [Gof84].

Table 3: Summary of Functorial Calculus.

$\mathsf{F}\,X$	$\mathsf{F}\,f$	$\mathsf{F}\,\phi$
X	f	ϕ
C	1_C	$\lambda c.TRUE$
2^X	$2^f = \lambda s.\{f\,x \mid x \in s\}$	$2^\phi = \lambda s.\forall b \in s : \phi\,b$
$X*$	$f* = \lambda l.[f\,x \mid x \leftarrow l\,]$	$\phi* = \lambda l.\forall 1 \leq i \leq \mathit{length}\,l : \phi(l\,i)$
X^C	$f^C = \lambda\sigma.f \circ \sigma = \lambda\sigma.\left(\begin{array}{c} c \\ f(\sigma\,c) \end{array}\right)_{c\in C}$	$\phi^C = \lambda\sigma.\forall c \in C : \phi(\sigma\,c)$
$C \rightharpoonup X$	$C \rightharpoonup f = \lambda\sigma.f \circ \sigma = \lambda\sigma.\left(\begin{array}{c} c \\ f(\sigma\,c) \end{array}\right)_{c\in dom\sigma}$	$C \rightharpoonup \phi = \lambda\sigma.\forall x \in \mathit{rng}\,\sigma : \phi\,x$
$\mathsf{G}\,X \times \mathsf{H}\,X$	$\mathsf{G}\,f \times \mathsf{H}\,f$	$\lambda(x,y).(\mathsf{G}\,\phi)x \wedge (\mathsf{H}\,\phi)y$
$\mathsf{G}\,X + \mathsf{H}\,X$	$\mathsf{G}\,f + \mathsf{H}\,f$	$\lambda x.\left\{\begin{array}{lcl} x = i_1\,a & \Rightarrow & (\mathsf{G}\,\phi)a \\ x = i_2\,b & \Rightarrow & (\mathsf{H}\,\phi)b \end{array}\right.$
$\mathsf{G}\,X + 1$	$\lambda x.\left\{\begin{array}{lcl} x = i_2\,NIL & \Rightarrow & x \\ x = i_1\,a & \Rightarrow & i_1\langle(\mathsf{G}\,f)a\rangle \end{array}\right.$	$\lambda x.\left\{\begin{array}{lcl} x = i_2\,NIL & \Rightarrow & TRUE \\ x = i_1\,a & \Rightarrow & (\mathsf{G}\,\phi)a \end{array}\right.$

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

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

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*

$$A \trianglelefteq_f^\phi B \tag{6}$$

then

$$\mathcal{F}(A) \trianglelefteq_{\mathcal{F}(f)}^{\mathcal{F}(\phi)} \mathcal{F}(B) \tag{7}$$

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], (7) still holds for this functor, for finite A and B [Jou92]. Table 3 presents a summary of this "functorial calculus" for the most common SETS constructs.

The aim of this report 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 now be stronger than the one 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 (1) 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:

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

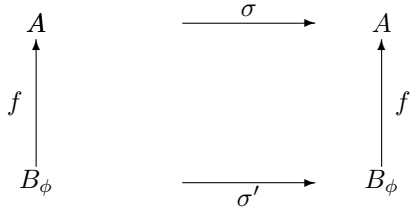
□

On the functionality side, 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 $\sigma : A \to A$ be the specification of a function transforming data of sort A while preserving some ad hoc invariant α on A, that is* [9]*,*

$$\forall a \in A : \alpha(a) \Rightarrow \alpha(\sigma(a)) \tag{8}$$

Then the reification σ' of σ implied by reification step (1),



[9] 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:

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

will still preserve α at B-level.
Proof: σ' is any function satisfying

$$\forall b \in B_\phi : f(\sigma'(b)) = \sigma(f(b)) \tag{9}$$

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 (9). 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 (8). □

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

$$Database \quad \cong \quad 2^A \tag{10}$$

as a convenient (however very primitive!) model for the database itself [10]. That is, our notion of an abstract database "re-uses" the algebra of finite sets:

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

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

$$\begin{array}{rcl} find & : & A \times Database \longrightarrow 2 \\ find(a,\sigma) & \stackrel{\text{def}}{=} & (a \in \sigma) \end{array} \tag{11}$$

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

We start by calculating the hash-table implementation of (10). Then we will proceed to the deduction of the corresponding operators 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

$$2^A \trianglelefteq \ldots$$

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

into something we can identify as an acceptable model of a hash table. Our reasoning starts from the basic result (69), expressing the fact that a data domain can always be refined by adding extra information to it. From an instance of fact (69),

$$A \trianglelefteq_{\pi_2} n \times A \tag{12}$$

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

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

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,

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

we have

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

by theorem 2. By applying the powerset functor $\lambda X.2^X$ to (14) we obtain (*cf.* table 3 above)

$$2^A \trianglelefteq_{f_1} (2^{n \times A})_{\Phi_1} \tag{15}$$

where

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

and

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

We now resort to the "currying" law of exponentiation, (56), established by the $uncurry$ bijection (89). 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 is — *cf.* the righthand side of (15) —

$$2^{n \times A} \cong (2^A)^n \tag{16}$$

By handling datatype invariants according to theorem 3 we obtain

$$(2^{n \times A})_{\Phi_1} \cong_{f_2} ((2^A)^n)_{\Phi_2} \tag{17}$$

where

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

and

$$\begin{array}{rcl} \Phi_2 & = & \Phi_1 \circ uncurry \\ & = & \lambda t.\Phi_1(\{(i,a) \mid i \in n \wedge a \in t(i)\}) \\ & = & \lambda t.\forall i \in n : (\forall a \in t(i) : i = H(a)) \end{array} \tag{18}$$

It is easy to show that $((2^A)^n)_{\Phi_2}$ above — *i.e.* the "array" structure $(n \to 2^A)_{\Phi_2}$ — is in fact a model of hash tables. According to Φ_2 (18), every set in the range of such "arrays" contains *synonyms* [HS19]. All such sets are mutually disjoint, *cf.* the following lemma:

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

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

Proof: By contradition, 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 (18) 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 [11]. We have thus achieved the intended "hashing effect". However, the task of reifying 2^A "recurs" in the co-domain of $n \rightarrow 2^A$, a problem identified in the literature under the headings *overflow handling* or *collision handling* [HS19].

4.1.1 Collision Handling

Let us see how instructive it is to deal with this problem in a formal way. Since refinement steps in SETS are compositional, and bearing in mind that we have been just 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

$$n \rightarrow (m \rightarrow 2^A) \tag{19}$$

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

$$\Phi_3 \quad = \quad \lambda t. \forall i \in n : (\forall j \in m : (\forall a \in t(i)(j) : i = H(a) \wedge j = H_i(a))) \tag{20}$$

But, strictly speaking, *rehashing* is nothing but mutiplying 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 the co-domain of (19).

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 (70) in appendix B, turning hash tables into arrays of sequences of collisions,

$$(n \rightarrow A*)_{\Phi_3}$$

The corresponding invariant Φ_3 is obtained via theorem 1, whereby

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

for

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

cf. table 3. By theorem 3 one has,

$$(2^A)^n \trianglelefteq_{elems^n} (A*)^n \quad \Rightarrow \quad ((2^A)^n)_{\Phi_2} \trianglelefteq_{elems^n} ((A*)^n)\underbrace{\Phi_2 \circ elems^n}_{\Phi_3} \tag{21}$$

that is,

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

$A*$ 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 $A*$ is a well-known fixpoint solution of functor

$$X \quad \cong \quad 1 + A \times X \qquad \textit{/*finite lists on A */} \tag{22}$$

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

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

Then recursion is *removed* by reifying the recursive pattern of (22) 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),

$$A* \trianglelefteq (K \rightharpoonup A \times (K+1)) \times (K+1) \tag{23}$$

where

$$f_{(23)}(\sigma, k) \stackrel{\text{def}}{=} \left\{ \begin{array}{lcll} k = NIL & \Rightarrow & [] & \\ \neg(k = NIL) & \Rightarrow & let & a = \pi_1(\sigma(k)) \\ & & & k' = \pi_2(\sigma(k)) \\ & & in & cons(a, f(\sigma, k')) \end{array} \right.$$

and where

$$\phi_{(23)}(\sigma, k) \stackrel{\text{def}}{=} \phi_{aux}(k, \sigma, \emptyset)$$

for

$$\phi_{aux}(k, \sigma, C) \stackrel{\text{def}}{=} \left\{ \begin{array}{lcl} TRUE & \Leftarrow & k = NIL \\ FALSE & \Leftarrow & (k \neq NIL \wedge k \notin dom(\sigma)) \vee k \in C \\ \phi_{aux}(k', \sigma, C \cup \{k\}) & \Leftarrow & k \in dom(\sigma) \wedge \sigma(k) = \langle a, k' \rangle \end{array} \right. \tag{24}$$

Back to (23) we have, via theorem 1:

$$(A*)^n \trianglelefteq^{\phi_{(23)}{}^n}_{f_{(23)}{}^n} ((K \rightharpoonup A \times (K+1)) \times (K+1))^n \tag{25}$$

$$\cong_{f_{(55)}} \underbrace{(K \rightharpoonup A \times (K+1))^n}_{Z} \times (K+1)^n \tag{26}$$

But there is no need for every linked list having its "own" or "private" heap. In fact, no programming language compiler providing for pointer management would "reuse" the same heap address $k \in K$ for different lists accessible via $(K+1)^n$. That is, for $\sigma \in Z$ of equation (26),

$$\forall i \neq j \in n : dom\sigma(i) \cap dom\sigma(j) = \emptyset$$

holds. This can be exploited in the following "amalgamation" of all such heaps,

$$Z \quad \cong \quad K \times n \rightharpoonup A \times (K+1)$$

permitted via law (62), in which the disjointness property above is re-written thus:

$$\alpha(\sigma) \stackrel{\text{def}}{=} \forall i \neq j \in n : \{k \in K \mid \langle k, i \rangle \in dom\sigma\} \cap \{k \in K \mid \langle k, j \rangle \in dom\sigma\} = \emptyset$$

i.e.

$$\begin{array}{rcl} \alpha(\sigma) & \stackrel{\text{def}}{=} & \forall i \neq j \in n : \langle k, i \rangle \in dom\sigma \Rightarrow \langle k, j \rangle \notin dom\sigma \\ & \Leftrightarrow & fdp(dom\sigma) \end{array} \tag{27}$$

cf. (82). If we can rely on this property α, which can be regarded as another *ad hoc* invariant now guaranteed by the implementation device (*e.g.* C compiler), the "reverse" [12] application of law (75) is enabled:

$$K \times n \rightharpoonup A \times (K+1)_\alpha \quad \cong_{f^{-1}_{(75)}} \quad K \rightharpoonup \underbrace{n}_{(*)} \times (A \times (K+1)) \tag{28}$$

removing the invariant and using the inverse of $f_{(75)}$ (see appendix C). It is useful to calculate the abstraction map encompassing all the steps ever since linked lists were introduced, that is, from (25) to (28):

$$\begin{array}{rcl} f_4 & = & f_{(23)}{}^n \circ f_{(55)} \circ (f_{(62)} \times id) \circ (f^{-1}_{(75)} \times id) \\ & = & f_{(23)}{}^n \circ f_{(55)} \circ ((f_{(62)} \circ f^{-1}_{(75)}) \times id) \\ & = & f_{(23)}{}^n \circ \underbrace{f_{(55)} \circ ((f_{(62)} \circ f^{-1}_{(75)}) \times id)}_{g} \\ & = & \lambda\langle \sigma, t \rangle . \left(\begin{array}{c} i \\ f_{(23)}(\left(\begin{array}{c} k \\ \pi_2(\sigma(k)) \end{array} \right)_{k \in dom\sigma}, t(i)) \end{array} \right)_{i \in n} \end{array}$$

[12] That is, from right to left.

It is immediate to see that this map altogether ignores every $i = \pi_1(\sigma(k))$. That is to say, the n factor marked $(*)$ in (28) became redundant after the introduction of ad hoc invariant α (27) and can be scrapped reducing Z to

$$K \rightharpoonup A \times (K+1)$$

and abstraction map f_4 to:

$$\begin{array}{rcl} f_4(\sigma,t) & \stackrel{\text{def}}{=} & \left(\begin{array}{c} i \\ f_{(23)}(\left(\begin{array}{c} k \\ \sigma(k) \end{array} \right)_{k \in dom\sigma}, t(i)) \end{array} \right)_{i \in n} \\ & = & \left(\begin{array}{c} i \\ f_{(23)}(\sigma, t(i)) \end{array} \right)_{i \in n} \end{array}$$

Putting everything together we reach a final data structure

$$((K \rightharpoonup A \times (K+1)) \times (K+1)^n)_{\Phi_4} \tag{29}$$

— a table of pointers to linked-lists sharing the same heap structure — whose overall invariant can be systematically calculated:

$$\begin{array}{rcl} \Phi_4 & = & {\phi_{(23)}}^n \circ g \wedge \Phi_3 \circ f_4 \\ & = & (\lambda\langle\sigma,t\rangle.\forall i \in n : \phi_{(23)}(\sigma, t(i)) \wedge \Phi_3 \circ f_4 \end{array}$$

that is,

$$\begin{array}{rcl} \Phi_4(\sigma,t) & = & (\forall i \in n : \phi_{aux}(t(i),\sigma,\emptyset)) \wedge (\forall i \in n : (\forall a \in f_{aux}(t(i),\sigma) : i = H(a)) \\ & = & \forall i \in n : \phi_{aux}(t(i),\sigma,\emptyset) \wedge (\forall a \in f_{aux}(t(i),\sigma) : i = H(a)) \end{array}$$

recalling (24) and compressing $elems \circ f_\sigma$ into an auxiliary function:

$$f_{aux}(k,\sigma) \stackrel{\text{def}}{=} \left\{ \begin{array}{rcll} k = NIL & \Rightarrow & \{\} & \\ \neg(k = NIL) & \Rightarrow & let & a = \pi_1(\sigma(k)) \\ & & & k' = \pi_2(\sigma(k)) \\ & & in & \{a\} \cup f_{aux}(k',\sigma) \end{array} \right.$$

In the end, this corresponds in fact to a classical strategy, *cf.* (quoting [HS19]):

> 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.

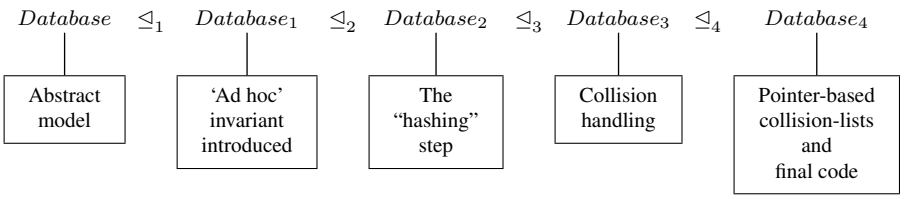
The final step would be to encode (29) (with some extra "syntactic sugar") in a commercial programming language, *e.g.* C

```
typedef  Database4 *Collision[n]; /* hash table */
typedef  struct Collision {
         A Synonym;
         struct Collision *Next;
         } ;
```
(30)

or PASCAL

```
type  Database4  =  array[1..n] of Bucket; (* hash table *)
      Bucket     =  ^ Collision;
      Collision  =  record
                       Synonym:  A;
                       Next:   ^ Collision
                    end;
```
(31)

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



(32)

4.2 Operation Level Calculation

The steps of (32) 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:

$$\begin{aligned} f_1(f_2(t)) &= \{a \in A \mid (i,a) \in \{(i,a) \mid i \in n \wedge a \in t(i)\}\} \\ &= \{a \in A \mid i \in n \wedge a \in t(i)\} \\ &= \{a \in A \mid \exists i \in n : a \in t(i)\} \\ &= \bigcup_{i \in n} \{a \in A \mid a \in t(i)\} \\ &= \bigcup_{i \in n} t(i) \end{aligned} \tag{33}$$

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:

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

leading to the 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:

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

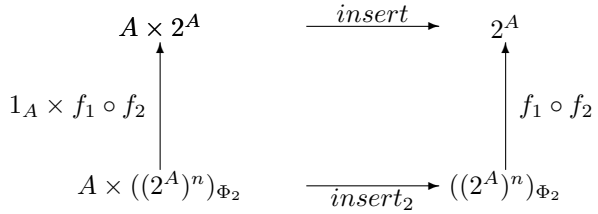
In summary,

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

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 $\bigcup_{i \in n} t(i)$ [13], the efficiency gain is obvious.

4.2.2 The $insert$ Operation

Refinement diagram:



Refinement equation:

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

Reasoning:

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

where t' is the one-entry mapping

$$t' \quad = \quad \begin{pmatrix} H(a) \\ \{a\} \cup t(H(a)) \end{pmatrix}$$

Then,

$$\begin{aligned} t'(H(a)) \cup \bigcup_{i \in n - \{H(a)\}} t(i) &= \bigcup_{i \in n} \begin{cases} i = H(a) & \Rightarrow \quad t'(i) \\ \neg(i = H(a)) & \Rightarrow \quad t(i) \end{cases} \\ &= \bigcup_{i \in n} \begin{cases} i \in domt' & \Rightarrow \quad t'(i) \\ \neg(i \in domt') & \Rightarrow \quad t(i) \end{cases} \\ &= \bigcup_{i \in n} (t \dagger t')(i) \\ &= f_1(f_2(t \dagger t')) \end{aligned}$$

In summary,

$$f_1(f_2(insert_2(a,t))) \quad = \quad f_1(f_2(t \dagger \begin{pmatrix} H(a) \\ \{a\} \cup t(H(a)) \end{pmatrix}))$$

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

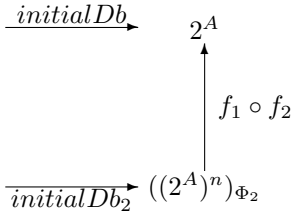
$$insert_2(a,t) \quad = \quad t \dagger \begin{pmatrix} H(a) \\ insert(a, t(H(a))) \end{pmatrix}$$

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 (32).

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

4.2.3 The $initialDb$ Operation

Refinement diagram:



Refinement equation:

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

Reasoning: since

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

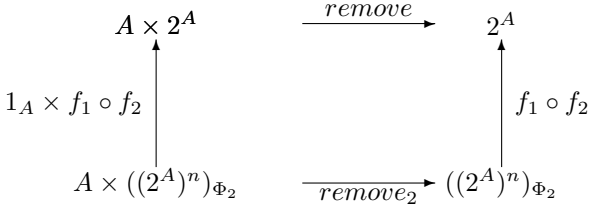
one straightforwardly draws

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

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:



Refinement equation:

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

Reasoning:

$$\begin{array}{rcl}
f_1(f_2(remove_2(a,t))) & = & remove(a, f_1(f_2(t))) \\
& = & f_1(f_2(t)) - \{a\} \\
& = & \bigcup_{i \in n} t(i) - \{a\} \\
& = & (\bigcup_{i \in n} t(i)) \cap \overline{\{a\}} \\
& = & \bigcup_{i \in n} (t(i) \cap \overline{\{a\}}) \\
& = & \bigcup_{i \in n} (t(i) - \{a\}) \\
& = & (t(H(a)) - \{a\}) \cup \bigcup_{i \in n - \{H(a)\}} (t(i) - \{a\}) \\
& = & t'(H(a)) \cup \bigcup_{i \in n - \{H(a)\}} \underbrace{t(i) - \{a\}}_{(*)}
\end{array}$$

where t' is the one-entry mapping

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

At this point one resorts to invariant Φ_2 (18), 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 $(*)$ above to $t(i)$. The reasoning hereupon is similar to that carried out for $insert$, leading to

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

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

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

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 (10), is far too simple a model for database files. A model closer to reality is

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

where A plays the rôle of a domain of keys and B is the data of interest [14].

Note that $2^A \cong_{dom} A \rightharpoonup 1$ (58). So it seems that the new model amounts to generalizing 1 in functor $\lambda X.X \rightharpoonup 1$ to any finite B. And indeed each operation σ over (10) can be generalized to some σ' over (34) provided σ' "preserves σ along the dom morphism", that is, if σ is such that

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

We obtain the classical "dictionary functionality" [15]:

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

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

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

[15] 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 (11) and $a \in dom\sigma = TRUE$ is generalized to any value of B. This follows a technique based on SETS for elaborating or sophisticating specification models which is discussed elsewhere [Oli95].

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

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

With respect to data refinement, this generalization is not immediate since functor $\lambda X.X \rightharpoonup B$ is not so well-behaved as $\lambda X.2^X$ or any other in table 3: only for $f : A \to C$ an isomorphism will

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

be well-defined [16]:

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

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

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

Now revisit (14), 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

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

Now

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

cf. law (62). So we have obtained the "hash effect" again, in step which generalizes (17):

$$((n \times A) \rightharpoonup B)_{\phi \rightharpoonup B} \cong_{{f_{(62)}}^{-1}} (A \rightharpoonup B)^n_{\Phi_2} \tag{37}$$

where

$$
\begin{array}{rcl}
\Phi_2 & = & (\phi \rightharpoonup B) \circ {f_{(62)}}^{-1} \\
& = & \lambda\sigma.\forall x \in dom{f_{(62)}}^{-1}(\sigma) : \phi(x) \\
& = & \lambda\sigma.\forall i \in n : (\forall a \in dom\sigma(i) : i = H(a))
\end{array}
$$

Collision handling may proceed now by resorting to the implementation theory of $A \rightharpoonup B$, *e.g.* [Fie80]. But law (76) provides an easy way of re-using our earlier results by implementing abstract mappings as sets of pairs satisfying a functional dependency (fdp). 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:

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

In the end we will obtain data structure (30) — or (31) — 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.

[16] 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 $A \rightharpoonup 1$ (*i.e.* 2^A) is well-behaved.

6 Conclusions

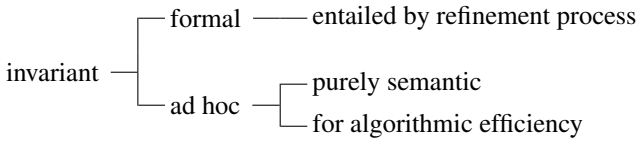
For programmers and program users it is important to know not only which properties hold for the programs they deal with, but also *why*, *when* and *where* they hold.

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 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 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" aplied 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 report 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 [17]
- *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 report, has shown how early in design ad hoc invariants of the last kind above can be introduced in software development, *cf.* step 1 in (32) and (13).

The SETS calculus [Oli90, Oli92] has been extended in this report 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 solutions [Oli99] — has provided evidence, once again, of the benefits of addressing traditional algorithmic "culture" from a formal perspective. In [Oli99] the full spectrum of hashing reification is systematically expounded, namely in respect of collision handling. For instance, one may organize synomyms as binary serarch collision trees rather that linear collision lists, *etc.*.

7 Future Work

Future research will proceed in several directions. First, 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" (16) and (36) in the following way: 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

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

[17] The invariant of $Date$ — recal (3) — is a telling example of this class of invariants.

for $n > 0$ and $H : A \longrightarrow n$ a given hash function. While instances $G(X) = 2^X$ and $G(X) = X \rightharpoonup B$ (for some B) were studied in this report, other interesting Gs such as $G(X) = B \rightharpoonup X$ or $G(X) = X^*$ do not satisfy (38). However, (38) is too strong and may be relaxed to

$$\mathcal{G}(n \times A) \quad \trianglelefteq \quad (\mathcal{G}(A))^n \tag{39}$$

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

$$Book \rightharpoonup Publisher$$

be a database file assigning books to publishers. One may hash this data-structure on $Book$, as we have seen (let $G(X) = X \rightharpoonup Publisher$). 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 $G(X) = Book \rightharpoonup X$. This in fact makes sense with respect to (39), since

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

holds as a generalization of law (74) [18] entailing an invariant enforcing mutually disjoint domains of the n mappings obtained on the left. $G(X) = X^*$ remains as an example of an "un-hashable" data-structure, for

$$(n \times A)^* \quad \trianglelefteq \quad (A^*)^n \tag{40}$$

does not hold in general [19].

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 be at compile-time factored into a collection of n similar (sub-)structures. From another prespective, 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 [OM95] is providing evidence that the "amount of concurrency" gained in a *horizontal refinement* step, expressed in SETS, can 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 synomym 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 [OM95].

Finally, one may regard 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 [Oli95]. This is worthwhile studying in more detail.

Acknowledgements

This report found 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]. It took much longer to write than originally forecasted because it has demanded a series of developments in SETS which were not available at that time. In the meantime, it has become partly available in the unpublished lecture notes of the author's M. Sc. course on reification [Oli91].

This work has been partly supported by the JNICT council under R&D contract nr. PMCT TIT 169.90.

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

[19] This is because it is impossible to reconstruct, from the n sequences on the left-handside of (40), the original interleaving of such sequences. The symmetric of (40) holds under the following surjection:

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

References

[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. (Portuguese translation).

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

[Hen84] P. Henderson. ME TOO: A language for software specification and model-building — preliminary report. Technical report, Univ. Stirling, Dec. 1984.

[HS19] E. Horowitz and S. Sahni. *Fundamentals of Data Structures*. Computer Software Engineering Series. Pitman, 19??. 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*. University of Coimbra, 1992. M. Sc. thesis (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.

[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 Project *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, April 1990.

[Oli91] J. N. Oliveira. *Especificação Formal de Programas*. Univ 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).

[Oli95] J. N. Oliveira. Data sophistication is not data refinement! Technical report, INESC #2361, U. Minho, 1995. (In preparation).

[Oli99] J. N. Oliveira. *A Reification Handbook*. 1999. (In preparation).

[OM95] J. N. Oliveira and F. S. Moura. Can distribution be (statically) calculated? Technical report, DI/INESC, 1995. (in preparation).

[Rod93] C. J. Rodrigues. *Sobre o Desenvolvimento Formal de Bases de Dados*. University of Minho, 1993. M. Sc. thesis (in Portuguese).

[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.

A SETS—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 [20].

A, B, C, ... will denote SETS objects, *i.e.* data domains. n, m, ... will denote initial segments of $\mathbb{N}$, the set of natural numbers. Subscripted f and ϕ denote abstraction functions and concrete invariants, respectively. If the subscript is an equation number, this indicates the law where the given map or predicate arise.

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

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

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

$$A \neq B \Rightarrow X^A \cap X^B = \emptyset \tag{63}$$
$$A \cap B = \emptyset \Rightarrow A \cup B \cong A + B \tag{64}$$

$$A^B \cong A^X \times A^{B-X} \Leftarrow X \subseteq B \tag{65}$$
$$A^n \cong A \times A^{n-1} \tag{66}$$
$$n \neq m \Rightarrow X^n \cap X^m = \emptyset \tag{67}$$

[20] For instance, laws (66,67) are instances of (65,63), respectively; law (76) is obtained from law (75) by making $C = 1$, etc.

B SETS—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.

$$A \preceq A \times B \tag{68}$$
$$B \preceq A \times B \tag{69}$$
$$2^A \preceq A* \tag{70}$$
$$A \rightharpoonup (B \times C) \trianglelefteq (A \rightharpoonup B) \times (A \rightharpoonup C) \tag{71}$$
$$(A \times B) \rightharpoonup C \trianglelefteq A \rightharpoonup (B \rightharpoonup C) \tag{72}$$
$$A \rightharpoonup D \times (B \rightharpoonup C) \trianglelefteq (A \rightharpoonup D) \times ((A \times B) \rightharpoonup C) \tag{73}$$
$$A \rightharpoonup (B + C) \trianglelefteq (A \rightharpoonup B) \times (A \rightharpoonup C) \tag{74}$$
$$A \rightharpoonup B \times C \trianglelefteq (A \times B) \rightharpoonup C \tag{75}$$
$$A \rightharpoonup B \trianglelefteq 2^{A \times B} \tag{76}$$

C Abstraction Invariants

C.1 Basic Functions and Predicates

$$\overset{+}{\bowtie}(\sigma, \tau) \stackrel{\text{def}}{=} i_1 \circ \sigma \cup i_2 \circ \tau \tag{77}$$
$$djd(\sigma, \tau) \stackrel{\text{def}}{=} dom\sigma \cap dom\tau = \emptyset \tag{78}$$
$$dpi(\sigma, \tau) \stackrel{\text{def}}{=} \pi_1[dom(\tau)] \subseteq dom(\sigma) \tag{79}$$
$$eqd(\langle \sigma, \tau \rangle) \stackrel{\text{def}}{=} dom\sigma = dom\tau \tag{80}$$
$$elems(b) \stackrel{\text{def}}{=} \{b(i) \mid i \in length(b)\} \tag{81}$$
$$fdp(r) \stackrel{\text{def}}{=} \forall t, t' \in r : \pi_1(t) = \pi_1(t') \Rightarrow \pi_2(t) = \pi_2(t') \tag{82}$$
$$i_1(a) \stackrel{\text{def}}{=} \langle 1, a \rangle \tag{83}$$
$$i_2(a) \stackrel{\text{def}}{=} \langle 2, a \rangle \tag{84}$$
$$\sigma \bowtie \tau \stackrel{\text{def}}{=} \begin{pmatrix} a \\ \langle \sigma(a), \tau(a) \rangle \end{pmatrix}_{a \in dom\sigma} \tag{85}$$
$$mkf(\rho) \stackrel{\text{def}}{=} \begin{pmatrix} a \\ the(\{b \in B \mid a\rho b\}) \end{pmatrix}_{a \in \pi_1[\rho]} \tag{86}$$
$$\underset{n}{\bowtie}(\sigma, \tau) \stackrel{\text{def}}{=} \begin{pmatrix} a \\ \langle \sigma(a), \begin{pmatrix} b \\ \tau(a, b) \end{pmatrix}_{\langle a, b \rangle \in sel(a, \tau)} \rangle \end{pmatrix}_{a \in dom(\sigma)} \tag{87}$$
$$sel(a, \tau) \stackrel{\text{def}}{=} \{\langle a', b \rangle \in dom(\tau) \mid a' = a\} \tag{88}$$
$$uncurry(t) \stackrel{\text{def}}{=} \lambda(b, c).t(b)(c) \tag{89}$$

C.2 Abstraction Maps

$$f_{(49)} = \lambda\sigma.\langle \sigma(1), \ldots, \sigma(n) \rangle \tag{90}$$
$$f_{(50)} = \lambda x.x \tag{91}$$
$$f_{(43)} = \pi_1 \tag{92}$$

$$f_{(55)} = \lambda\langle x, y\rangle. \begin{pmatrix} a \\ \langle x(a), y(a)\rangle \end{pmatrix}_{a \in A} \tag{93}$$

$$f_{(56)} = uncurry \tag{94}$$

$$f_{(58)} = dom \tag{95}$$

$$f_{(62)} = \lambda\sigma. \begin{pmatrix} \begin{pmatrix} a \\ \sigma(a,c) \end{pmatrix}^{c}_{\langle a,c\rangle \in dom\sigma} \end{pmatrix}_{c \in C} \tag{96}$$

$$f_{(62)}{}^{-1} = \lambda\sigma. \bigcup_{c \in C} \begin{pmatrix} \langle c, a\rangle \\ \sigma(c)(a) \end{pmatrix}_{a \in dom\sigma(c)} \tag{97}$$

$$f_{(75)} = \lambda\sigma. \begin{pmatrix} a \\ \langle b, \sigma(\langle a, b\rangle)\rangle \end{pmatrix}_{\langle a,b\rangle \in dom\sigma} \tag{98}$$

$$f_{(75)}^{-1} = \lambda\sigma. \begin{pmatrix} \langle a, \pi_1(\sigma(a))\rangle \\ \pi_2(\sigma(a)) \end{pmatrix}_{a \in dom\sigma} \tag{99}$$

$$f_{(68)} = \pi_1 \tag{100}$$

$$f_{(69)} = \pi_2 \tag{101}$$

$$f_{(70)} = \mathit{elems} \tag{102}$$

$$f_{(71)} = \bowtie \tag{103}$$

$$f_{(76)} = mkf \tag{104}$$

$$f_{(72)} = \lambda\sigma. \begin{pmatrix} \langle a, b\rangle \\ (\sigma(a))(b) \end{pmatrix}_{a \in dom\sigma \wedge b \in dom\sigma(a)} \tag{105}$$

$$f_{(73)} = \underset{n}{\bowtie} \tag{106}$$

$$f_{(74)} = \overset{+}{\bowtie} \tag{107}$$

C.3 Concrete Invariants

$$\phi_{(71)} = eqd \tag{108}$$

$$\phi_{(75)} = fdp \circ dom \tag{109}$$

$$\phi_{(76)} = fdp \tag{110}$$

$$\phi_{(72)} = \lambda\sigma. (\quad) \notin rng(\sigma) \tag{111}$$

$$\phi_{(73)} = dpi \tag{112}$$

$$\phi_{(74)} = djd \tag{113}$$