Bringing CAMILA and SETS Together — the `bams.cam` and `ppd.cam` CAMILA Toolset Demos

J.J. Almeida, L.B. Barbosa, F.L. Neves & J.N. Oliveira [1]

Dep. Informática, Universidade do Minho
Campus de Gualtar, 4700 Braga, Portugal
Tel: 351+53-604470
Fax: 351+53-612954.

December 13, 1997

[1]Email: {jj,lsb,fln,jno}@di.uminho.pt; Http: http://www.di.uminho.pt/∼{jj,lsb,fln,jno}.

Introduction

The presentation of the CAMILA [ABNO97a] Toolset at AMAST'97 [ABNO97b] includes two demos — `bams.cam` and `ppd.cam` — which show how incremental data refinement is supported by the prototype / implementation hybridization primitives available in the Toolset.

The process of animating each of these data refinement case studies is hard to understand if unaccompanied by the underlying mathematical calculations in the SETS calculus [Oli90, Oli92, Oli97] [1].

Bridging the gap between theory and practice, this document not only outlines such mathematical calculations which lead to the provided relational model implementations, but also shows how such a process can be partly automated and animated in the Toolset.

A synopsis of SETS is presented in Appendix A. This should be read in advance by those unfamiliar with this program calculus. Reference [AB91] is suggested for those unfamiliar with the CAMILA syntax and environment.

By inspecting the present LaTeX source, readers will have an opportunity to understand how additional tools in the Toolset — namely CAMTeX (a CAMILA into LaTeX embedder) and PRETTY (a LaTeX output CAMILA pretty printer) — can be used in order to produce documentation in an integrated fashion.

1. The `bams.cam` Prototype

CAMILA prototype `bams.cam` is listed in Appendix D.1.

> ***Demo step 1.*** *At* UNIX *shell level type*
>
> `camila bams.cam`
>
> *in order to start the animation of the prototype.*

1.1 The Specification

This is the first section of the prototype. It presents the animation of the formal specification of a toy "bank account management system" ($BAMS$). This is really a very simple specification example based on the following abstract syntax:

$$\begin{array}{lcl} BAMS & \cong & AccId \hookrightarrow Account \\ Account & \cong & \begin{array}{lcl} H & : & 2^{AccHolder} \\ B & : & Amount \end{array} \times \\ AccId & \cong & STR \\ AccHolder & \cong & STR \\ Amount & \cong & INT \end{array} \quad (1.1)$$

[1] SETS's URL is http://www.di.uminho.pt/~jnohtml/setshp.html.

Accounts (*Account*) are uniquely identified by account identifiers (*AccId*). Every account records both a set of account holders ($2^{AccHolder}$) and the account's current balance (*Amount*).

Prototype `bams.cam` has an internal state $bams \in BAMS$ which is manipulated by the obvious operations for opening accounts, adding new account holders, removing accounts, crediting and withdrawal operations, *etc.*:

$$OpenAccount : AccId \times 2^{AccHolder} \times Amount \longrightarrow Void$$
$$OpenAccount(n, h, m) \stackrel{\text{def}}{=}$$
$$\quad bams' = \begin{pmatrix} n \\ Account(h, m) \end{pmatrix} \dagger\, bams$$

$$RemoAccount : AccId \longrightarrow Void$$
$$RemoAccount(n) \stackrel{\text{def}}{=}$$
$$\quad bams' = bams \setminus \{n\}$$

$$AddAccHolders : AccId \times 2^{AccHolder} \longrightarrow Void$$
$$AddAccHolders(n, h) \stackrel{\text{def}}{=}$$
$$\quad bams' = bams \dagger \begin{array}{ll} let & addh = \lambda(x).Account(H(x) \cup h, B(x)) \\ in & \star \hookrightarrow addh(bams \mid \{n\}) \end{array}$$

$$Credit : AccId \times Amount \longrightarrow Void$$
$$Credit(n, m) \stackrel{\text{def}}{=}$$
$$\quad bams' = bams \dagger \begin{array}{ll} let & addm = \lambda(x).Account(H(x), B(x) + m) \\ in & \star \hookrightarrow addm(bams \mid \{n\}) \end{array}$$

$$Withdraw : AccId \times Amount \longrightarrow Void$$
$$Withdraw(n, m) \stackrel{\text{def}}{=}$$
$$\quad bams' = bams \dagger \begin{array}{ll} let & subm = \lambda(x).\ \begin{array}{ll} let & b = B(x) \\ & c = \left\{ \begin{array}{lll} b \geq m & \Rightarrow & b - m \\ \neg(b \geq m) & \Rightarrow & b \end{array} \right. \\ in & Account(H(x), c) \end{array} \\ in & \star \hookrightarrow subm(bams \mid \{n\}) \end{array}$$

$$GoodCostumers : Amount \longrightarrow 2^{AccHolder}$$
$$GoodCostumers(m) \stackrel{\text{def}}{=}$$
$$\quad \bigcup(\{H(bams(k)) \mid k \in dom(bams) \wedge B(bams(k)) \geq m\})$$

The prototype is furthermore equipped with an initialization operation (*Init*) which sets up the following initial internal state:

$bams =$

AccId	AccHolders	Balance
0636024550431	J N Oliveira + L.S. Barbosa	3000
0636024000999	Neves, Fernando L. + J N Oliveira	9000
0636099999000	J.J.A.	9500
0636055566677		10

Demo step 2. *At prototype shell level type*

```
Init();
```

in order to initialize the prototype. Then type [a]

```
view0(bams);
```

in order to see the above initialization data.
You can exercise the prototype as you wish. For instance, open your own account by typing

```
OpenAccount("999",{"*Your name*"},10000);
```

Now, upon typing

```
view0(bams);
```

once again, you should obtain on the screen the following system data:

AccId	AccHolders	Balance
999	**Your name**	*10000*
0636024550431	*J N Oliveira + L.S. Barbosa*	*3000*
0636024000999	*Neves, Fernando L. + J N Oliveira*	*9000*
0636099999000	*J.J.A.*	*9500*
0636055566677		*10*

Next, credit e.g. *200 currency units into your account by typing*

```
Credit("999",200);
```

and, if you want to add a friend as another owner of you account, type

```
AddAccHolders("999",{"*Your friend's name*"});
```

etc. etc.

[a] `view0` is a *BAMS* display function.

1.2 The Reification Process

See section 2 of prototype `bams.cam` is listed in Appendix D.1.

1.2.1 Data Reification

We start by re-writing (1.1) as a *Set*-expression:

$$BAMS \quad = \quad AccId \hookrightarrow (2^{AccHolder} \times Amount) \qquad (1.2)$$

Bearing the relational model in mind as the target implementation media, data reification calculation proceeds as follows [1]:

$$
\begin{array}{ll}
 & BAMS \\
= & AccId \hookrightarrow (2^{AccHolder} \times Amount) \\
\cong_1 & AccId \hookrightarrow (Amount \times 2^{AccHolder}) \\
\cong_2 & AccId \hookrightarrow (Amount \times (AccHolder \hookrightarrow 1)) \\
\trianglelefteq_3 & (AccId \hookrightarrow Amount) \times ((AccId \times AccHolder) \hookrightarrow 1)) \\
\cong_4 & (AccId \hookrightarrow Amount) \times (2^{AccId \times AccHolder}) \\
\trianglelefteq_5 & 2^{AccId \times Amount} \times 2^{AccId \times AccHolder} \\
= & BAMS_5
\end{array}
$$

We are lead to the expected relational tables, one keeping record of the relationship between account identifiers and account balances ($2^{AccId \times Amount}$) and the other of the relationship between account identifiers and account holders ($2^{AccId \times AccHolder}$). For instance, for the initialization data provided above we will have, at implementation level, the following two tables:

AccId	**Balance**
0636024550431	3000
0636024000999	9000
0636099999000	9500
0636055566677	10

AccId	**AccHolder**
0636024550431	J N Oliveira
0636024550431	L.S. Barbosa
0636024000999	Neves, Fernando L.
0636024000999	J N Oliveira
0636099999000	J.J.A.

(1.3)

1.3 Abstraction Invariant Synthesis

For each step $i = 1, 5$ above we record abstraction function f_i (that is, function `abs`i in `bams.cam` [2]) and concrete invariant ϕ_i (implicitly assumed in `bams.cam`):

$$
\begin{array}{ll}
\cong_1 & \left\{ \begin{array}{l} f_1 = AccId \hookrightarrow [\pi_2, \pi_1] \\ \phi_1 = \lambda x \cdot TRUE \end{array} \right. \\
\cong_2 & \left\{ \begin{array}{l} f_2 = AccId \hookrightarrow (id \times dom) \\ \phi_2 = \lambda x \cdot TRUE \end{array} \right. \\
\trianglelefteq_3 & \left\{ \begin{array}{l} f_3 = \bowtie_n \\ \phi_3 = dpi \end{array} \right. \\
\cong_4 & \left\{ \begin{array}{l} f_4 = id \times ffAset \\ \phi_4 = \lambda x \cdot TRUE \end{array} \right.
\end{array}
$$

[1] See *e.g.* [Oli97].

[2] Equation (1.2) above corresponds to step 0 in `bams.cam`.

$$\trianglelefteq_5 \quad \left\{ \begin{array}{l} f_5 = mkf \times id \\ \phi_5 = fdp \times (\lambda x \cdot TRUE) \end{array} \right.$$

Prototype `bams.cam` includes, for the purpose of animation, an inverse *representation* function for each f_i [3]. These are the `rep`i functions in section 2 of source file `bams.cam`.

Two subsequent reification steps (6 and 7) are included in the prototype which have to do with the specific relational model animators available as CAMILA libraries.

Beyond step 6 all data is in the "finite mapping set" (ffs) format which is handled by library `ffs.cam`. This data type is very expressive, as it can be regarded as modelling object populations, relational tables (with NULL values), feature bundles, *etc.* Additional libraries enriching `ffs.cam` (such as *e.g.* the emerging `sql.cam` library) specify a simple relational database engine at this level of abstraction.

Step 7 has to do with data persistency. In order to show how heterogeneous implementation media can interface with each other in the CAMILA Toolset, the $2^{AccId \times Amount}$ table is recorded in a file as a CAMILA ffs-object, while the other table is recorded as a "`awk`-like" textual file. This means that, at low-level, the implementation puts two distinct relational database engines together: one specified on top of `ffs.cam` and another directly implementing the latter on UNIX by resorting to shell commands such as such as `join`, `sort`, `awk` *etc.*

1.3.1 Overall Abstraction Function Synthesis

This follows rule (A.9):

$$\begin{array}{rcl} f & = & f_1 \circ f_2 \circ f_3 \circ f_4 \circ f_5 \\ & = & (AccId \hookrightarrow [\pi_2, \pi_1]) \circ \\ & & (AccId \hookrightarrow (id \times dom)) \circ \bowtie \circ (id \times ffAset) \circ \\ & & (mkf \times id) \end{array}$$

1.3.2 Overall Concrete Invariant Synthesis

Also following rule (A.9):

$$\begin{array}{rcl} \phi(\langle \rho, \gamma \rangle) & = & \phi_5(\langle \rho, \gamma \rangle) \wedge \\ & & \phi_4(f_5(\langle \rho, \gamma \rangle)) \wedge \\ & & \phi_3(f_4(f_5(\langle \rho, \gamma \rangle))) \wedge \\ & & \phi_2(f_3(f_4(f_5(\langle \rho, \gamma \rangle)))) \wedge \\ & & \phi_1(f_2(f_3(f_4(f_5(\langle \rho, \gamma \rangle))))) \\ & = & \phi_5(\langle \rho, \gamma \rangle) \wedge \phi_3(f_4(f_5(\langle \rho, \gamma \rangle))) \\ & = & fdp(\rho) \wedge \phi_3(f_4(\langle mkf(\rho), \gamma \rangle)) \\ & = & fdp(\rho) \wedge dpi(\langle mkf(\rho), ffAset(\gamma) \rangle) \\ & = & fdp(\rho) \wedge 2^{\pi_1}(\gamma) \subseteq dom(mkf(\rho)) \\ & = & fdp(\rho) \wedge 2^{\pi_1}(\gamma) \subseteq 2^{\pi_1}(\rho) \end{array}$$

In summary, invariant ϕ

- imposes a functional dependence on the "amounts table" ρ (=the balance of any account is not ambiguous);
- allows for more than one holder per account (*cf.* "account-holder table" γ);
- guarantees that every account held by somebody is a valid account.

[3] Note that there exist more than one representation function for a given non-injective abstraction function.

1.3.3 Operation Reification

Thanks to the morphism calculus, operation reification in SETS is quite naturally performed by "solving reification equations" implicit in so-called "reification diagrams". The following picture is an example of such diagrams,

$$\begin{tikzcd}
AccNr \times Amount \times BAMS \arrow[r, "credit"] & BAMS \\
AccNr \times Amount \times BAMS_5 \arrow[u, "id \times id \times f"] \arrow[r, "credit_5"'] & BAMS_5 \arrow[u, "f"']
\end{tikzcd}$$

where f is the abstraction function compiled in section 1.3.1 and *credit* is the state transforming function of operation $Credit$, that is,

$$Credit(k,q) \stackrel{\mathrm{def}}{=} bams' = credit(k,q,bams)$$

This kind of algorithm refinement process is explained in appendix B. In this section we instantiate it for the above operation, that is, for refinement equation

$$credit(k,q,f(x)) \quad = \quad f(credit_5(k,q,x)) \tag{1.4}$$

which we want to solve for $credit_5$.

Steps 2 to 4 of the $BAMS$ data calculation are fairly common and well-studied in SETS. The abstraction function corresponding to this sub-calculation,

$$(AccId \hookrightarrow (id \times dom)) \circ \bowtie_n \circ (id \times ffAset) \tag{1.5}$$

can be shown to be equivalent to

$$\bowtie \circ [\pi_1, \dagger \circ ((AccId \hookrightarrow \overline{\{\}}) \times collect)] \tag{1.6}$$

— see [BO97].

In order to perform our calculations at morphism-level we re-write *credit* in curried form,

$$\begin{aligned} credit_{k,q} &: BAMS \longrightarrow BAMS \\ credit_{k,q} &\stackrel{\mathrm{def}}{=} \dagger \circ [id, (AccId \hookrightarrow id \times add_q) \circ |_{\{k\}}] \end{aligned}$$

where add_q abbreviates $+ \circ [id, \overline{q}]$ and $|_{\{k\}}$ abbreviates $| \circ [id, \overline{\{k\}}]$.

In this way, it is easy to check that $credit_{k,q}$ is an instance of the very common "selective update" construct [BO97],

$$\mathcal{U}_S(f) \stackrel{\mathrm{def}}{=} \dagger \circ [id, (A \hookrightarrow f) \circ |_S] \tag{1.7}$$

for $A = AccId$, $f = id \times add_q$ and $S = \{k\}$.

For this construct, the following generic result holds,

$$\mathcal{U}_S(f \times g) \circ (1.6) \quad = \quad \bowtie \circ [\mathcal{U}_S(f) \times \mathcal{U}_S(g)] \circ [\pi_1, \dagger \circ ((A \hookrightarrow \overline{\{\}}) \times collect)] \tag{1.8}$$

and, for $g = id$,

$$\mathcal{U}_S(f \times id) \circ (1.6) \quad = \quad (1.6) \circ (\mathcal{U}_S(f) \times id) \tag{1.9}$$

that is, generic refinement diagram

$$\begin{tikzcd}
A \hookrightarrow C \times 2^B \arrow[r, "\mathcal{U}_S(f \times id)"] & A \hookrightarrow C \times 2^B \\
(A \hookrightarrow B) \times 2^{A \times B} \arrow[u, "(1.6)"] \arrow[r, "\mathcal{U}_S(f) \times id"'] & (A \hookrightarrow B) \times 2^{A \times B} \arrow[u, "(1.6)"']
\end{tikzcd}$$

commutes. The product of $\mathcal{U}_S(f)$ and id ("parallel application") at the implementation-level means that only the first data-structure $A \hookrightarrow C$ of $(A \hookrightarrow B) \times 2^{A \times B}$ will be subject to update. In our case, this is the "amounts table", as expected.

Because the lefthand side of our refinement equation is a bit more elaborate than that of (1.8), it must be subject to some extra calculation effort [4]:

$$\begin{array}{ll}
& credit_{k,q} \circ f \\
= & \mathcal{U}_{\{k\}}(id \times add_q) \circ (AccId \hookrightarrow [\pi_2, \pi_1]) \circ (1.6) \circ (mkf \times id) \\
= & (AccId \hookrightarrow [\pi_2, \pi_1]) \circ \mathcal{U}_{\{k\}}(add_q \times id) \circ (1.6) \circ (mkf \times id) \\
= & (AccId \hookrightarrow [\pi_2, \pi_1]) \circ (1.6) \circ (\mathcal{U}_{\{k\}}(add_q) \times id) \circ (mkf \times id) \\
= & (AccId \hookrightarrow [\pi_2, \pi_1]) \circ (1.6) \circ (\mathcal{U}_{\{k\}}(add_q) \circ mkf \times id) \\
= & (AccId \hookrightarrow [\pi_2, \pi_1]) \circ (1.6) \circ (\dagger \circ [mkf, mkf \circ 2^{AccId \times add_q} \circ sel_{1,\{k\}}] \times id) \\
= & (AccId \hookrightarrow [\pi_2, \pi_1]) \circ (1.6) \circ (mkf \times id) \circ (rplus \circ [id, 2^{AccId \times add_q} \circ sel_{1,\{k\}}] \times id) \\
= & f \circ (\underbrace{rplus \circ [id, 2^{AccId \times add_q} \circ sel_{1,\{k\}}]}_{\lambda x.sqlUpdate(x, 2, add_q, \lambda n.n = k)} \times id)
\end{array}$$

In summary, we have

$$credit_{k,q} \circ f \quad = \quad f \circ (\lambda x.sqlUpdate(x, 2, add_q, \lambda n.n = k) \times id)$$

meaning that $credit_{k,q}$ is implemented in terms of the semantics of *sqlUpdate*, a function available in CAMILA library `sql.cam`. This leads straight to `Credit7` as written in the prototype listed in Appendix D.1.

1.4 The Implementation

Section 3 of the prototype presents the implementation, at refinement level 7, of the operations offered by the specification, each of these calculated such as described above for *Credit*.

As was explained above, formal calculations go as far as level 5. Level 6 has to do with the adopted relational model animator (CAMILA library `sql.cam` on top of `ffs.cam`), and level 7 amounts to making information persistent (saving data in files) in heterogeneous media — the *amounts* relationship is saved as a CAMILA `ffs`-object while the other relationship becomes a textual file. By exercising the prototype as follows, one sees how incremental data refinement is supported by the prototype / implementation hybridization primitives available in the CAMILA Toolset.

[4] See section A.6.3 for the basic functions introduced in this calculation and [BO97] for the justification of some of its steps on the basis of natural transformation and other properties of the morphism-level SETS notation.

Demo step 3. *For each operation* `Op` *of the specification, the prototype shell offers its implementation operation* `Op7`*. For instance, type*

```
Init7();
```

This is the initialization operation at implementation level.
By typing [a]

```
view6(abs7());
```

you should be able to see on the screen the two relational tables depicted in (1.3) [b]*. Your and your friend's account is not there because it has been opened at abstract level. Therefore, type*

```
OpenAccount7("999",{"*Your name*"},10000);

AddAccHolders7("999",{"*Your friend's name*"});
```

and then repeat

```
view6(abs7());
```

A similar effect would have been achieved should you have typed [c]

```
ROLLBACK2ABS();
```

which forces the implementation to synchronize with the specification, as far as state data are concerned. The inverse rollback operation,

```
ROLLBACK2REP();
```

may be invoked wherever you want the state of the specification to mirror that of the implementation, supposing that the specification and implementation processes were subject to independent stimuli in the meantime.

[a] `view6` is a $BAMS_6$ display function.

[b] Type

```
sh("more _bamsAmsTable _bamsAccHolTable");
```

in case you want to browse the actual implementation level state saving files.

[c] `ROLLBACK2ABS` (="rollback to abstraction") is a function which overwrites the implementation state with a representation of the abstraction state. `ROLLBACK2REP` (="rollback to representation") is the function inverse to `ROLLBACK2ABS`.

In a sense, one is able to run two concurrent processes — one (at high-level) is the animation of the prototype of the specification and the other (at low-level) is the implementation itself [5]. These processes intercommunicate with each other via the abstraction / representation functions, and may synchronize via

[5] Or a sub-implementation, as in `bams.cam`. In this case, typical of stepwise refinement, part of the prototype has reached the final implementation platform while the other is still at a higher level of abstraction. This is easy to achieve wherever data refinement calculations lead to product structures ($A_1 \times \ldots \times A_n$) at implementation-level.

dedicated "rollback" operations, such as illustrated above.

2. The `ppd.cam` Prototype

2.1 The Specification

The first section of this prototype presents the animation of the formal specification of a "toy factory" production database, built around the following data model:

$$
\begin{array}{lcl}
PPD & \cong & S : Stock \times \\
 & & P : Pricelist \times \\
 & & E : EquipDb \\
Stock & \cong & Unid \hookrightarrow NAT \\
Pricelist & \cong & Comp \hookrightarrow Currency \\
EquipDb & \cong & Equip \hookrightarrow Unid \hookrightarrow NAT \\
Unid & \cong & Equip + Comp \\
Comp & \cong & K : NAT \\
Equip & \cong & K : NAT \\
Currency & \cong & NAT \\
NAT & \cong & INT
\end{array}
\tag{2.1}
$$

In words, this means that:

- this factory builds *equipments* whose production involves individual *components* obtained from a second source;
- each individual component has a *cost* and is stocked in some *quantity*;
- components participate, in fixed quantities, in the production of *equipments* (*e.g.* PC board ref. `X` has n integrated circuits of ref. `Y`);
- equipments may contain, in fixed quantities, other equipments as *sub-blocks* (*e.g.* personal computer ref. `Z` has m PC boards of ref. `X`).
- (sub)equipments are stocked along with components in order to be readily available for production.

In summary, a production tree must exist for each equipment, which may involve individual components and/or other production sub-trees.

Demo step 4. *At* UNIX *shell level type*

`camila ppd.cam`

in order to start the animation of this prototype. Once in the prototype shell, type

`Init();`

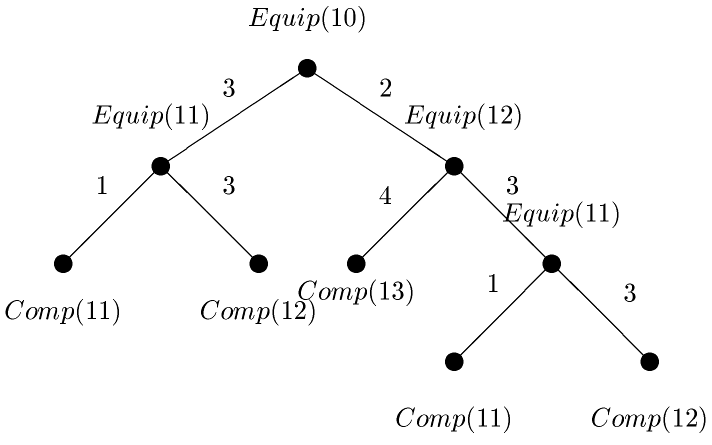
in order to set up some initial state configuration. We will be particularly concerned with the EquipDb structure. So, type

`E(ppd);`

in order to see the start-up production database contents:

```
[
  Equip(
     11 )-> [
       Comp(
          11 )-> 1
       Comp(
          12 )-> 3 ]
  Equip(
     12 )-> [
       Comp(
          13 )-> 4
       Equip(
          11 )-> 3 ]
  Equip(
     10 )-> [
       Equip(
          11 )-> 3
       Equip(
          12 )-> 2 ]]
```

Production tree for say Equip(10) *is, therefore, as follows:*



One of the most interesting operations which can be defined over the PPD data model specification above is that which (recursively) computes the *part explosion* (bill of materials) of some given equipment, according to its production tree:

$$
\begin{array}{rcl}
explode & : & Equip \times EquipDb \longrightarrow (Comp \hookrightarrow \mathbb{N}) \\
explode(k,\sigma) & \stackrel{\text{def}}{=} & let \quad ms = \left\{ \begin{array}{lcl} k \in dom(\sigma) & \Rightarrow & \sigma(k) \\ k \notin dom(\sigma) & \Rightarrow & (\ ) \end{array} \right. \\
 & & in \quad \bigoplus_{x \in dom(ms)} \left\{ \begin{array}{lcl} \text{is-}Comp(x) & \Rightarrow & ms \mid \{x\} \\ \text{is-}Equip(x) & \Rightarrow & ms(x) \otimes explode(x, db \setminus \{k\}) \end{array} \right.
\end{array}
$$

where

- $\oplus$ is multiset ("bag") union [1]:

$$
\begin{array}{rcl}
x \oplus y & \stackrel{\text{def}}{=} & \quad x \setminus dom(y) \\
 & & \cup \;\; y \setminus dom(x) \\
 & & \cup \;\; \left(\begin{array}{c} a \\ x(a) + y(a) \end{array} \right)_{a \in dom(x) \cap dom(y)}
\end{array} \tag{2.2}
$$

- $\otimes$ is called multiset external multiplication (by analogy with vector spaces):

$$
\begin{array}{rcl}
\otimes & : & \mathbb{N} \times (A \hookrightarrow \mathbb{N}) \longrightarrow (A \hookrightarrow \mathbb{N}) \\
n \otimes \sigma & \stackrel{\text{def}}{=} & A \hookrightarrow (\lambda a . n \times a)(\sigma)
\end{array} \tag{2.3}
$$

Demo step 5. *Check the behaviour of the Explode operation provided in the prototype by typing, for instance,*

```
Explode(Equip(10));
```

You should obtain the following output:

```
[
  Comp(
     11 )-> 9
  Comp(
     12 )-> 27
  Comp(
     13 )-> 8 ]
```

which corresponds to the production tree depicted above.

2.2 The Reification Process

2.2.1 Data Reification

The SETS reification of the PPD data model is presented in detail in [Oli97]. As a case-study in automatic data reification, it is also available in URL http://www.di.uminho.pt/~jno/html/setsan.html.

[1] These and other multiset algebra operators are available in CAMILA's standard `mse.cam` library.

Demo step 6. *Run*

Init7();

in the `ppd.cam` *prototype, obtaining*

```
-rw-rw-r--   1 jno      jno            95 Dec 13 00:10 _ppdSP
-rw-rw-r--   1 jno      jno            24 Dec 13 00:10 _ppdEETable
-rw-rw-r--   1 jno      jno            24 Dec 13 00:10 _ppdECTable
```

and then browse the generated `_ppd*` *files in order to get a picture of how this refinement works. Check that both the "stock" and "pricelist" mappings are kept at* CAMILA *level, while the equipment database is split in two tables:*

- *the "equipments into components decomposition" table (file* `_ppdECTable`*);*
- *the "equipments into sub-equipments decomposition" table (file* `_ppdEETable`*).*

Both of these files are textual, "awk-like" data files.

2.2.2 Operation Reification

By far, the most expressive operation reification exercise concerning the *PPD* data model is that of the *explode* function (called by the *Explode* operation), because it is non-trivially recursive.

This exercise, whereby *explode* is refined into a `while`-loop thanks to the "vectorial" properties of multiset operators $\otimes$ and $\oplus$, can be found in *e.g.* [Oli97]. In the `ppd.cam` prototype, this iterative refinement of *explode* is materialized into a shell (see file `ppdExplode.pl`) which the prototype operation *Explode*7 interacts with.

Demo step 7. *Still at* `ppd.cam` *prototype shell level, type*

```
Explode7(Equip(10));
```

in order to activate the part-explosion operation at implementation level. The shell runs in a verbose mode whereby you should obtain the following trace and output:

```
[ ppdExplode.pl: looping... ]
-rw-rw-r--   1 jno      jno            10 Dec 13 00:10 _ff
[ ppdExplode.pl: looping... ]
-rw-rw-r--   1 jno      jno             5 Dec 13 00:10 _ff
[ ppdExplode.pl: looping... ]
-rw-rw-r--   1 jno      jno             5 Dec 13 00:10 _ff
[ ppdExplode.pl: exiting ]
-rw-rw-r--   1 jno      jno            16 Dec 13 00:10 _r
+------+-----+
| Comp | Qty |
+------+-----+
|  11  |  9  |
|  12  | 27  |
|  13  |  8  |
+------+-----+
```

Demo step 8. *Finally, consider* `CompCost7`*, which is an illustration of an intermediate step in refinement: because* `mseDiff` *has not yet been replaced by a proper implementation, the (provisional) implementation ends up switching to abstract mode before producing its output.*
Type

```
CompCost7(Equip(10),10);
```

and you should obatin the following:

```
[ ppdExplode.pl: looping... ]
-rw-rw-r--   1 jno      jno            10 Dec 13 00:10 _ff
[ ppdExplode.pl: looping... ]
-rw-rw-r--   1 jno      jno             5 Dec 13 00:10 _ff
[ ppdExplode.pl: looping... ]
-rw-rw-r--   1 jno      jno             5 Dec 13 00:10 _ff
[ ppdExplode.pl: exiting ]
-rw-rw-r--   1 jno      jno            16 Dec 13 00:10 _r

[ mseExMul successfully performed at UNIX level ]

[ Refinement memo: mseDiff still at abstract level ]

Result is:
[
  Comp(
     11 )-> 80
  Comp(
     12 )-> 265
  Comp(
     13 )-> 79 ]
()
```

3. The Reification Laboratory

ART [1] is a rather elaborate tool of the CAMILA Toolset. Its main purpose is to automate data reification calculations such as the one presented in section 1.2.1. It accepts a `*.cam` file and, according to some parameters (*e.g.*, an indication of the target implementation media), it outputs a valid implementation for a selection of its data types, equipped with the sequence of abstraction / representation functions [2].

ART animates a significant subset of SETS and is based on a (to the best of our knowledge) novel genetic-algorithm approach to term re-writing. Details are provided in appendix C. The following trace was produced by ART along its calculation of $BAMS_5$ from $BAMS$ (recall section 1.2.1):

```
[fln@camila-mobile msc]$ camila art.cam

UM-XMETOO version 1.62 .DLL.COMM.USERTYPE (aap/pem/cjr/jj  1997/11/04 17:39:43)
camila    version 1.77 (1997/11/07 18:36:58)
```

[1] ART ="Automatic Reification Tool" [Nev97].
[2] These can be "copy/pasted" into the prototype source file itself, as in section 2 of the `bams.cam` and `ppd.cam` prototypes.

```
/home/fln/.metoorc")
art.cam
?- CAM2SETS("bams.cam");
_tmp
?- rfPREPARE("BAMS");
x
?- rfPSTARTBAMS(x,1,3);
CS (Chromsize)              = 15
PS (Population Size)        = 30
SS (Segment Size)           = 0 (for binary representation only)
PC (Probability-Crossover) = 25
PM (Probability-Mutation)  = 0
VC (Variable Size Chrom.)  = false
GR (Gene Representation)   = 1 (0=binary, 1=decimal)
MV (Maximum Gene Value)    = 39 (for decimal representation only)
> Best(0): 25%
> thinking... best (1): 33%
> thinking... best (2): 100%
> thinking... best (3): 100%
> thinking...< (  (  2  ^  (  AccId  *  Amount  )  )  *  (  2  ^  (  AccId  *  AccHolder  )  )  ) >
?- rfSHOWREIF(x);
> applying 30... (( AccId -> ( ( AccHolder -> 1 ) * Amount ) ))
> abs = ((*-> prod( dom ,id) ))
> rep = ((*-> prod( ffAset ,id) ))
> applying 1... (( AccId -> ( Amount * ( AccHolder -> 1 ) ) ))
> abs = ((*-> const(p2,p1) ))
> rep = ((*-> const(p2,p1) ))
> applying 36... (( ( AccId -> Amount ) * ( ( AccId * AccHolder ) -> 1 ) ))
> abs = (nJoin)
> rep = (nJoinInv)
> applying 38... (( ( 2 ^ ( AccId * Amount ) ) * ( ( AccId * AccHolder ) -> 1 ) ))
> abs = (prod( mkf ,id))
> rep = (prod( mkr ,id))
> applying 29... (( ( 2 ^ ( AccId * Amount ) ) * ( 2 ^ ( AccId * AccHolder ) ) ))
> abs = (prod(id, ffAset ))
> rep = (prod(id, dom ))
```

4. Future Work

The CAMILA Toolset can be improved in many ways.

While the rapid prototyping environment has reach some stability, the ART tool, for instance, is still in prototype phase (`art.cam` is a non-trivial CAMILA prototype). The genetic calculation engine is expected to be extended to operation calculation in the future.

Moreover, the reification laboratory is expected to be very soon supported by a library a graphical objects (which is currently being formally specified [Fer97]) which should replace the current `view` functions, in an automatic way. Reifications of this library into the most common visual programming environments (*e.g.*

JAVA, TCL/TK, Borland DELPHI, *etc.*) will also be provided. On-going work in this area also shows that such a (formal) presentation layer generator can accommodate some OLAP-typical functionality in a natural way.

The on-going LOGCOMP [1] project includes an integration of SETS with the COQ [2] theorem prover.

Bibliography

[AB91] J. J. Almeida and L. S. Barbosa. CAMILA by example. Technical Report DI/INESC 91-11-1, INESC, November 1991.

[ABNO97a] J. J. Almeida, L. S. Barbosa, F. L. Neves, and J. N. Oliveira. CAMILA: Formal software engineering supported by functional programming. In A. De Giusti, J. Diaz, and P. Pesado, editors, *Proc. II Conf. Latino Americana de Programacisn Funcional (CLaPF97)*, pages 1343–1358, La Plata, Argentina, October 1997. Centenario UNLP.

[ABNO97b] J. J. Almeida, L. S. Barbosa, F. L. Neves, and J. N. Oliveira. CAMILA: Prototyping and refinement of constructive specifications, 1997. To be presented at AMAST'97: Sixth International Conference on Algebraic Methodology and Software Technology, 13–17 December 1997, Macquarie University, Sydney, Australia.

[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.

[BO97] L. S. Barbosa and J. N. Oliveira. "Natural" data refinement in SETS. Technical report, DI/UM, Dec. 1997. (Submitted for publication).

[Fer97] L.G. Ferreira. Master's thesis in preparation, Dec. 1997. (in Portuguese).

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

[Mor90] C. Morgan. *Programming from Specification.* Series in Computer Science. Prentice-Hall International, 1990. C. A. R. Hoare, series editor.

[Nev97] F.L. Neves. Reificação "Genética" de estruturas de dados. Master's thesis, University of Minho, Dec. 1997. (in Portuguese).

[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):1–23, April 1990.

[1] URL: http://www.di.uminho.pt/%7Ejmv/htmls/logcomp.html.

[2] URL: http://zenon.inria.fr:8003/Equipes/COQ-eng.html.

[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).

[Oli97] J. N. Oliveira. Sets - a data structuring calculus and its application to program development. Technical report, UNU/IIST, May 1997. Lecture Notes of Course at UNU/IIST, Macau, 5-16 May 1997, 125p.

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

A. A Synopsis of SETS

SETS stands for "*Set*-based specification" (and calculus). This notation should not be regarded as a competitor to standard specification languages such as VDM-SL or Z, 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 *Set* of sets.

Despite its categorial foundation (see *e.g.* [Oli97] and [BO97] for details), the use of category theory at SETS practitioner level is rather mild and fairly obvious.

A.1 The Object-Level Notation

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

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

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].

The following table will help in introducing *Set*'s primitive constructs to "program hackers":
It 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. Integer tag introduction can be abbreviated via co-product injections, *cf. e.g.* i_1 and i_2 in section A.6.

Table A.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

- 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 {} [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 parentheses, the following operator priorities are assumed: $^\star > \overline{\_} > \times > \hookrightarrow > +$. For instance, expression

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

will abbreviate

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

[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, of which only cardinality is retained. 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 A.1).

[3] Or "pointers", see the analogy of Table A.1.

etc.

Properties of both primitive and derived SETS constructs are listed in sections A.4 and A.5. 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 */
struct subTrees *R;
};

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

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

---

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

$$DecTree \cong What \times (Answer \hookrightarrow DecTree) \tag{A.1}$$

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

A more direct illustration of recursive specification 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) */
};
```

---

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

which — on the basis of the analogy of Table A.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 A.1 can also be used for coding purposes. Consider, for instance, the following SETS specification of a linked list of objects A,

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

that is, the following "linear equation":

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

¿From Table A.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 A.1 again:

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

A.2 The Morphism-Level Notation

The *Set* operator (morphism) repertoire is rich and well structured:

- *Set* Operator
  - Set-theoretic — *e.g.* $\mathbb{N} \xrightarrow{\lambda n . n^2} \mathbb{N}$
  - Category-theoretic
    - Implicit — *e.g.* $A \times B \xrightarrow{\pi_2} B$
    - Universal — *e.g.* $A \xrightarrow{[f,g]} B \times C$
    - Functorial — *e.g.* $X^\star \xrightarrow{f^\star} X^\star$
    - Natural — *e.g.* $X^\star \xrightarrow{elems} 2^X$

This includes all set-theoretical operators typical of constructive specification languages (such as *e.g.* VDM-SL or Z) dealing with lists (A^*), sets (2^A), finite mappings ($A \hookrightarrow B$) and so on. The main advantage of this approach to a specification language is that operators are inherently accompanied of their (categorial) properties, which experience shows to be vital in abstract calculations. Indeed, most VDM-like typical operators, such as *e.g.* *elems*, †, \ *etc.*, are "natural" operators (*i.e.* natural transformations) [BO97].

SETS's *functorial* constructs provide for much textual and reasoning economy, there being a functional construct per each SETS data construct. The most common are, for A, B, C, D given SETS data domains and $f : A \longrightarrow B$ and $g : C \longrightarrow D$ two SETS morphisms: 2^f, f^*, $f \times g$, $f + g$ and $C \hookrightarrow f$, defined as follows [5]:

$$\begin{array}{rcl} 2^f & : & 2^A \longrightarrow 2^B \\ 2^f(S) & \stackrel{\mathrm{def}}{=} & \{f(a) \mid\ a \in S\} \end{array}$$

$$\begin{array}{rcl} f^* & : & A^* \longrightarrow B^* \\ f^*(s) & \stackrel{\mathrm{def}}{=} & <f(a) \mid a \leftarrow s> \end{array}$$

$$\begin{array}{rcl} f \times g & : & A \times B \longrightarrow C \times D \\ f \times g(\langle a, b\rangle) & \stackrel{\mathrm{def}}{=} & \langle f(a), g(b)\rangle \end{array}$$

$$\begin{array}{rcl} f + g & : & A + B \longrightarrow C + D \\ f + g(x) & \stackrel{\mathrm{def}}{=} & \left\{ \begin{array}{rcl} x = i_1(a) & \Rightarrow & i_1(f(a)) \\ x = i_2(b) & \Rightarrow & i_2(g(b)) \end{array} \right. \end{array}$$

$$\begin{array}{rcl} C \hookrightarrow f & : & (C \hookrightarrow A) \longrightarrow (C \hookrightarrow B) \\ (C \hookrightarrow f)(\sigma) & \stackrel{\mathrm{def}}{=} & \left(\begin{array}{c} c \\ f(\sigma(c)) \end{array} \right)_{c \in dom(\sigma)} \end{array}$$

The following universal operations involving products and co-products are also provided: *construction* [Bac78],

$$\begin{array}{rcl} [f, g] & : & A \longrightarrow B \times C \\ [f, g](a) & \stackrel{\mathrm{def}}{=} & \langle f(a), g(a)\rangle \end{array} \tag{A.2}$$

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

$$\begin{array}{rcl} g \mid h & : & (A + B) \longrightarrow C \\ h \mid g(x) & \stackrel{\mathrm{def}}{=} & \left\{ \begin{array}{rcl} x = i_1(a) & \Rightarrow & h(a) \\ x = i_2(b) & \Rightarrow & g(b) \end{array} \right. \end{array} \tag{A.3}$$

Finally, for every $c_0 \in C$, the polymorphic "everywhere c_0" constant function is denoted as follows,

$$\begin{array}{rcl} \overline{c_0} & : & A \longrightarrow C \\ \overline{c_0}(a) & \stackrel{\mathrm{def}}{=} & c_0 \end{array} \tag{A.4}$$

and, of course, we have:

$$\overline{c_0} \circ f = \overline{c_0} \tag{A.5}$$

$$f \circ \overline{c_0} = \overline{f(c_0)} \tag{A.6}$$

[5] See [Oli97] for a discussion on the necessary distinction between ("covariant") $\mathcal{P}(A)$ and ("contravariant") 2^A at morphism level, which is not harmful if we restrict 2^f to injective f.

Lambda-expressions are used wherever convenient. Conditional expressions are written as is classical in mathematics, *e.g.*

$$x! = \begin{cases} x = 0 & \Rightarrow & 1 \\ \neg(x = 0) & \Rightarrow & x \times (x-1)! \end{cases}$$

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, expressions such as

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

are written thus:

$$\begin{cases} p(x) & \Rightarrow & g(x) \\ \neg p(x) & \Rightarrow & h(x) \end{cases}$$

Partial expressions are special cases of conditional expressions,

$$\left\{ \; p(x) \quad \Rightarrow \quad g(x) \right.$$

meaning

$$\begin{cases} p(x) & \Rightarrow & g(x) \\ \neg p(x) & \Rightarrow & \perp \end{cases}$$

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

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.*:

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

A.3 The Calculus

Instead of collecting all SETS results exploited so far in the SETS literature [6], this section restricts itself to a minimally representative superset of the results actually used in the main body of this document.

A.3.1 Basics

SETS is an (inequational) calculus of (functional) *abstraction invariants* as understood by *e.g.* [Mor90]. Each law of the calculus is an inequation of the form

$$A \trianglelefteq_f^\phi B \tag{A.7}$$

meaning:

[6]Se *e.g.* [Oli90, Oli92, Jou92, Rod93, OC93, Oli97].

- 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 [Mor90], equation (A.7) states that the following abstraction invariant holds between the abstract variables $a \in A$ and the concrete variables $b \in B$:

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

As in [Oli90, Oli92], the notation

$$A \preceq_f B \tag{A.8}$$

is accepted as a shorthand of (A.7) 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:

$$\trianglelefteq \left\{ \begin{array}{l} f \stackrel{\text{def}}{=} \bigcirc_{i=1}^{n} f_i \\ \phi \stackrel{\text{def}}{=} \lambda x.\ \bigwedge_{i=n}^{1} \phi_i((\bigcirc_{j=i+1}^{n} f_j)(x)) \end{array} \right. \tag{A.9}$$

SETS is presented in the following sections, which are organized as follows:

- the equational ($\cong$) sub-calculus, chiefly involving primitive constructs, is presented first, in section A.4;
- the inequational sub-calculus, chiefly involving derived constructs, follows in section A.5; shorthand (A.8) will indicate laws which do not induce concrete invariants;
- in many reasoning situations, both f and ϕ in (A.7) can be left out until explicitly demanded by algorithm synthesis; all laws are provided with the relevant abstraction map f and invariant ϕ, listed respectively in section A.6.1 and section A.6.2.

  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).

A.4 The Equational Calculus

In the following, capital letters A, B, C, ... will denote SETS objects, *i.e.* data domains. Small letters 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. Should subscript be equation numbers, these will indicate the laws where the given maps or predicates arise.

A.4.1 Underlying Abelian Semi-ring

$$\begin{array}{rcll}
A \times B & \cong & B \times A & \text{(A.10)} \\
A \times (B \times C) & \cong & (A \times B) \times C & \text{(A.11)} \\
A & \cong & A \times 1 & \text{(A.12)} \\
A + B & \cong & B + A & \text{(A.13)} \\
A + (B + C) & \cong & (A + B) + C & \text{(A.14)} \\
A + 0 & \cong & A & \text{(A.15)} \\
A \times 0 & \cong & 0 & \text{(A.16)} \\
A \times (B + C) & \cong & (A \times B) + (A \times C) & \text{(A.17)}
\end{array}$$

$$\underbrace{A \times \ldots \times A}_{n} \cong A^n \quad \text{(A.18)}$$

$$\underbrace{A + \ldots + A}_{n} \cong n \times A \quad \text{(A.19)}$$

A.4.2 Exponentiation and Derivatives

$$\begin{array}{rcll}
A^0 & \cong & 1 & \text{(A.20)} \\
A^1 & \cong & A & \text{(A.21)} \\
1^A & \cong & 1 & \text{(A.22)} \\
A^{(B+C)} & \cong & A^B \times A^C & \text{(A.23)} \\
(B \times C)^A & \cong & B^A \times C^A & \text{(A.24)} \\
A^{B \times C} & \cong & \left(A^C\right)^B & \text{(A.25)} \\
A \hookrightarrow B & \cong & (B + 1)^A & \text{(A.26)} \\
2^A & \cong & A \hookrightarrow 1 & \text{(A.27)} \\
(B + C) \hookrightarrow A & \cong & (B \hookrightarrow A) \times (C \hookrightarrow A) & \text{(A.28)} \\
0 \hookrightarrow A & \cong & 1 & \text{(A.29)} \\
1 \hookrightarrow A & \cong & A + 1 & \text{(A.30)} \\
(A \hookrightarrow B)^C & \cong & (C \times A) \hookrightarrow B & \text{(A.31)}
\end{array}$$

A.5 The Inequational Calculus

In this section $A \preceq B$ 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.

$$\begin{array}{rcll}
A & \preceq & A \times B & \text{(A.32)} \\
B & \preceq & A \times B & \text{(A.33)} \\
2^A & \preceq & A^\star & \text{(A.34)} \\
A \hookrightarrow (B \times C) & \trianglelefteq & (A \hookrightarrow B) \times (A \hookrightarrow C) & \text{(A.35)} \\
(A \times B) \hookrightarrow C & \trianglelefteq & A \hookrightarrow (B \hookrightarrow C) & \text{(A.36)}
\end{array}$$

$$
\begin{aligned}
A \hookrightarrow D \times (B \hookrightarrow C) &\trianglelefteq (A \hookrightarrow D) \times ((A \times B) \hookrightarrow C) && \text{(A.37)}\\
A \hookrightarrow (B + C) &\trianglelefteq (A \hookrightarrow B) \times (A \hookrightarrow C) && \text{(A.38)}\\
A \hookrightarrow B \times C &\trianglelefteq (A \times B) \hookrightarrow C && \text{(A.39)}\\
A \hookrightarrow B &\trianglelefteq 2^{A \times B} && \text{(A.40)}\\
A \hookrightarrow B &\trianglelefteq B \hookrightarrow 2^A_+ && \text{(A.41)}
\end{aligned}
$$

A.6 Abstraction Invariants

For basic functions and predicates occurring below see section A.6.3.

A.6.1 Abstraction Maps

$$
\begin{aligned}
f_{(A.10)} &= [\pi_2, \pi_1]\\
f_{(A.11)} &= [\pi_1 \circ \pi_1, [\pi_2 \circ \pi_1, \pi_2]]\\
f_{(A.12)} &= \pi_1\\
f_{(A.13)} &= i_2 \mid i_1\\
f_{(A.14)} &= (i_1 \mid i_2 \circ i_1) \mid (i_2 \circ i_2)\\
f_{(A.15)} &= i_1\\
f_{(A.16)} &= 1_0\\
f_{(A.17)} &= [\pi_1, i_1 \circ \pi_2] \mid [\pi_1, i_2 \circ \pi_2]\\
f_{(A.18)} &= \lambda\sigma.\langle \sigma(1), \ldots, \sigma(n)\rangle\\
f_{(A.19)} &= \lambda x.x\\
f_{(A.20)} &= \lambda x.\left(\quad\right)\\
f_{(A.21)} &= \lambda a.\begin{pmatrix} NIL \\ a \end{pmatrix}\\
f_{(A.22)} &= \lambda x.\begin{pmatrix} a \\ NIL \end{pmatrix}_{a \in A}\\
f_{(A.23)} &= \lambda\langle \sigma_1, \sigma_2\rangle.A^{i_1}(\sigma_1) \cup A^{i_2}(\sigma_2)\\
f_{(A.24)} &= \lambda\langle x, y\rangle.\begin{pmatrix} a \\ \langle x(a), y(a)\rangle \end{pmatrix}_{a \in A}\\
f_{(A.25)} &= uncurry\\
f_{(A.26)} &= \lambda\sigma.\begin{pmatrix} a \\ b \end{pmatrix}_{a \in A \wedge \sigma(a) = i_1(b)}\\
f_{(A.27)} &= dom\\
f_{(A.28)} &= \lambda\langle \sigma_1, \sigma_2\rangle.(i_1 \hookrightarrow A)(\sigma_1) \cup (i_2 \hookrightarrow A)(\sigma_2)\\
f_{(A.29)} &= \lambda x.\left(\quad\right)\\
f_{(A.30)} &= \lambda x.\begin{cases} x = i_1(a) & \Rightarrow \begin{pmatrix} NIL \\ a \end{pmatrix} \\ \neg(x = i_1(a)) & \Rightarrow \left(\quad\right) \end{cases}
\end{aligned}
$$

$$
\begin{array}{rcl}
f_{(A.31)} & = & \lambda\sigma.\left(\begin{array}{c} c \\ \left(\begin{array}{c} \pi_1(p) \\ \sigma(p) \end{array} \right)_{p \in dom(\sigma) \wedge \pi_2(p) = c} \end{array} \right)_{c \in C} \\
f_{(A.32)} & = & \pi_1 \\
f_{(A.33)} & = & \pi_2 \\
f_{(A.34)} & = & elems \\
f_{(A.35)} & = & \bowtie \\
f_{(A.36)} & = & \lambda\sigma.\left(\begin{array}{c} \langle a, b \rangle \\ (\sigma(a))(b) \end{array} \right)_{a \in dom(\sigma) \wedge b \in dom(\sigma(a))} \\
f_{(A.37)} & = & \bowtie_n \\
f_{(A.38)} & = & \overset{+}{\bowtie} \\
f_{(A.39)} & = & \lambda\sigma.\left(\begin{array}{c} a \\ \langle b, \sigma(\langle a, b \rangle) \rangle \end{array} \right)_{\langle a, b \rangle \in dom(\sigma)} \\
f_{(A.40)} & = & mkf \\
f_{(A.41)} & = & finv
\end{array}
$$

A.6.2 Concrete Invariants

$$
\begin{array}{rcl}
\phi_{(A.35)} & = & eqd \\
\phi_{(A.36)} & = & \lambda\sigma.(\quad) \notin rng(\sigma) \\
\phi_{(A.37)} & = & dpi \\
\phi_{(A.38)} & = & djd \\
\phi_{(A.39)} & = & fdp \circ dom \\
\phi_{(A.40)} & = & fdp \\
\phi_{(A.41)} & = & mdjfam
\end{array}
$$

A.6.3 Basic Functions and Predicates

$$i_1(a) \stackrel{\text{def}}{=} \langle 1, a \rangle \tag{A.42}$$

$$i_2(a) \stackrel{\text{def}}{=} \langle 2, a \rangle \tag{A.43}$$

$$djd(\sigma, \tau) \stackrel{\text{def}}{=} dom(\sigma) \cap dom(\tau) = \{\} \tag{A.44}$$

$$dpi(\sigma, \tau) \stackrel{\text{def}}{=} \pi_1[dom(\tau)] \subseteq dom(\sigma) \tag{A.45}$$

$$eqd(\langle \sigma, \tau \rangle) \stackrel{\text{def}}{=} dom(\sigma) = dom(\tau) \tag{A.46}$$

$$elems(b) \stackrel{\text{def}}{=} \{b(i) \mid i \in length(b)\} \tag{A.47}$$

$$fdp(r) \stackrel{\text{def}}{=} \forall t, t' \in r : \pi_1(t) = \pi_1(t') \Rightarrow \pi_2(t) = \pi_2(t') \tag{A.48}$$

$$mdjfam(\sigma) \stackrel{\text{def}}{=} \forall b, b' \in dom(\sigma) : b \neq b' \Rightarrow \sigma(b) \cap \sigma(b') = \{\} \tag{A.49}$$

$$\sigma \bowtie \tau \stackrel{\text{def}}{=} \left(\begin{array}{c} a \\ \langle \sigma(a), \tau(a) \rangle \end{array} \right)_{a \in dom(\sigma)} \tag{A.50}$$

$$\overset{+}{\bowtie}(\sigma, \tau) \stackrel{\text{def}}{=} i_1 \circ \sigma \cup i_2 \circ \tau \tag{A.51}$$

$$\bowtie_n \stackrel{\text{def}}{=} \bowtie \circ [\pi_1, \dagger \circ ((A \hookrightarrow \overline{(\quad)}) \times pcurry)] \tag{A.52}$$

$$ffAset(r) \stackrel{\mathrm{def}}{=} \left(\begin{array}{c} p \\ NIL \end{array}\right)_{p \in r} \tag{A.53}$$

$$mkf(\rho) \stackrel{\mathrm{def}}{=} \left(\begin{array}{c} \pi_1(t) \\ \pi_2(t) \end{array}\right)_{t \in \rho} \tag{A.54}$$

$$mkr(\sigma) \stackrel{\mathrm{def}}{=} \{\langle a, \sigma(a)\rangle \mid a \in dom(\sigma)\} \tag{A.55}$$

$$collect(\rho) \stackrel{\mathrm{def}}{=} \left(\begin{array}{c} a \\ \{x \in B \mid a\rho x\} \end{array}\right)_{a \in \pi_1[\rho]} \tag{A.56}$$

$$uncurry(t) \stackrel{\mathrm{def}}{=} \left(\begin{array}{c} \langle b, c\rangle \\ t(b)(c) \end{array}\right)_{b \in B \wedge c \in C} \tag{A.57}$$

$$pcurry(f) \stackrel{\mathrm{def}}{=} \left(\begin{array}{c} a \\ \left(\begin{array}{c} \pi_2(p) \\ f(p) \end{array}\right)_{p \in dom(f) \wedge \pi_1(p) = a} \end{array}\right)_{a \in 2^{\pi_1}(dom(f))} \tag{A.58}$$

$$puncurry(f) \stackrel{\mathrm{def}}{=} \bigcup_{a \in dom(f)} \begin{array}{ll} let & g = f(a) \\ in & \left(\begin{array}{c} \langle a, b\rangle \\ g(b) \end{array}\right)_{b \in dom(g)} \end{array} \tag{A.59}$$

$$finv(\sigma) \stackrel{\mathrm{def}}{=} \bigcup_{b \in dom(\sigma)} \left(\begin{array}{c} a \\ b \end{array}\right)_{a \in \sigma(b)} \tag{A.60}$$

$$sel_{i,s}(\tau) \stackrel{\mathrm{def}}{=} \{t \in \tau \mid \pi_i \in s\} \tag{A.61}$$

$$rplus(s, r) \stackrel{\mathrm{def}}{=} r \cup \{t \in s \mid \pi_1(t) \notin 2^{\pi_1}(r)\} \tag{A.62}$$

B. Program Synthesis by Solving Reification Diagrams

In the Sets method, transformational algorithmic refinement (concrete operation calculation) involves the following steps:

1. Calculation of a concrete-level simulation of the high-level operation (or "get rid of the abstraction function").

2. Calculation of an efficient version of 1. (*e.g.* "get rid of recursion" *etc.*).

3. Encoding of 2. (or "get rid of mathematics altogether!").

In general, let

$$\sigma : A \longrightarrow B$$

be a *Set*-morphism specifying some operation involving data-types A and B. (See later a generalization of σ to pre/post-condition operational specification.)

B.1 The refinement equation

Suppose we have already used Sets to refine A and B:

$$\begin{tikzcd} A \arrow[r, "\sigma"] & B \\ A_1 \arrow[u, "f"] & B_1 \arrow[u, "g"'] \end{tikzcd}$$

The idea is to "close" the above diagram,

$$\begin{tikzcd} A \arrow[r, "\sigma"] & B \\ A_1 \arrow[u, "f"] \arrow[r, "\sigma_1"'] & B_1 \arrow[u, "g"'] \end{tikzcd}$$

so that σ_1 "simulates" the behaviour of σ at concrete-level,

$$\forall a_1 \in A_1 : g(\sigma_1(a_1)) = \sigma(f(a_1)) \tag{B.1}$$

that is, the diagram commutes.

Equation (B.1) is called the refinement equation of σ.

B.1.1 Step 1

We will regard σ_1 as the "unknown" of the refinement equation (B.1). This step consists of solving this equation for σ_1.

Note that there may be many solutions for σ_1 in equation (B.1) — the typical "one-abstract to many-concrete" relationship of (functional) refinement.

"Simulation" amounts to exhibiting the same observational behaviour:

$$\sigma(a) \stackrel{\text{def}}{=} \begin{array}{ll} \mathit{let} & a_1 \in \{a' \in A_1 \mid f(a') = a\} \\ \mathit{in} & g(\sigma_1(a_1)) \end{array}$$

An injective range abstraction map g will lead to unique solutions because g^{-1} exists and diagram also commutes in this way:

$$\sigma_1(a_1) = g^{-1}(\sigma(f(a_1)))$$

Then the idea is to calculate...

$$\begin{array}{rcl} \sigma_1(a_1) & = & g^{-1}(\sigma(f(a_1))) \\ & = & \ldots \\ & \vdots & \\ & = & \ldots \sigma_1 \ldots a_1 \ldots \quad /* \textit{expression free of } f \textit{ and } g\ */ \end{array}$$

For non-injective g there exist more than one solution and idea is to transform refinement equation (B.1) so that f eventually gives place to g and g is eventually "pulled" up to the outermost place on the righthand side…

$$
\begin{array}{rcl}
g(\sigma_1(a_1)) & = & \sigma(f(a_1)) \\
 & = & \ldots \\
 & \vdots & \\
 & = & \ldots g \ldots \sigma_1 \ldots a_1 \ldots \\
 & \vdots & \\
 & = & g(E(\ldots \sigma_1 \ldots a_1 \ldots))
\end{array}
$$

so that abstraction map g can be factored out from both sides of the obtained equation:

$$
\sigma_1(a_1) \stackrel{\mathrm{def}}{=} \ldots E(\ldots \sigma_1 \ldots a_1 \ldots)
$$

(Of course, there may exist another E' such that $g(E'(\ldots \sigma_1 \ldots a_1 \ldots))$ can also be obtained by alternative calculation.)

B.1.2 Step 2

- "Efficiency" can be obtained in many ways which are dependent on the target machine architecture.
- Concrete version of the overall local invariant can be brought in (if not *already* considered in step 1) if it helps in reducing time/space complexity.
- In the case of a target sequential state-based machine (programming language), recursion-removal transformations are very often used to synthesize `for/while`-loops from recursive equations.

B.1.3 Step 3

Gode generation consists of carrying "program" transformation even further,

$$
\begin{array}{rcl}
 & \vdots & \\
\sigma_1(a_1) & = & \ldots E \ldots a_1 \ldots \\
 & = & \ldots \\
 & \vdots & \\
 & = & [\![\mathtt{P}]\!]
\end{array}
$$

until the formal semantics $[\![\mathtt{P}]\!]$ of some executable piece of code `P` (in the target programming language) is found.

B.2 Concrete invariants in refinement equations

In general, f and g have associated concrete invariants, say ϕ and ψ, respectively.

Refinement equation (B.1) becomes

$$
\forall a_1 \in A_1 : \phi(a_1) \Rightarrow \psi(\sigma_1(a_1)) \wedge g(\sigma_1(a_1)) = \sigma(f(a_1))
$$

B.3 What about σ?

The equations above regard σ as a (total) Set-morphism, that is, a total function. Although a simplistic assumption, all other situations (but one, see below) resort to it in a way or another.

B.3.1 Partial operations

Wherever σ is partial, pre-σ (its pre-condition) has to be added to the refinement equation:

$$\forall a_1 \in A_1 : \phi(a_1) \wedge \text{pre-}\sigma(f(a_1)) \Rightarrow \psi(\sigma_1(a_1)) \wedge g(\sigma_1(a_1)) = \sigma(f(a_1))$$

B.3.2 Deterministic state transformers

The typical constructive specification model involves operations which may deliver a result while changing the model's internal state. This can only be captured by a post-condition predicate

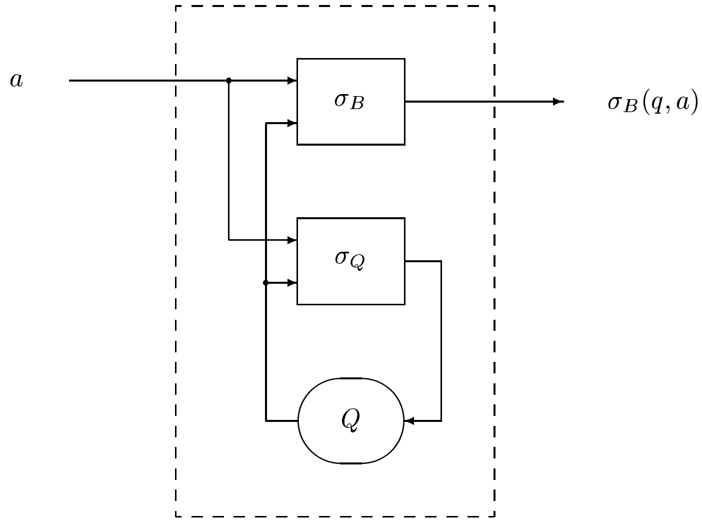
$$\text{post-}\sigma \quad : \quad Q \times A \times Q \times B \longrightarrow 2$$

(Q = internal state data-type).

In many situations the behaviour of σ is deterministic in the sense that it can be "factored" into two functions: a <u>result</u> function σ_B and a <u>state</u> transforming function σ_Q:

$$\text{post-}\sigma(q, a, q', b') \stackrel{\text{def}}{=} q' = \sigma_Q(q, a) \wedge b' = \sigma_B(q, a)$$

In a picture:



So we just have to refine functions σ_Q and σ_B, that is, solve for x the equations implicit in diagrams

$$
\begin{tikzcd}
A \times Q \arrow[r, "\sigma_B"] & B \\
A_1 \times Q_1 \arrow[u, "f \times f_Q"] \arrow[r, "x"'] & B_1 \arrow[u, "g"']
\end{tikzcd}
$$

and

$$
\begin{tikzcd}
A \times Q \arrow[r, "\sigma_Q"] & Q \\
A_1 \times Q_1 \arrow[u, "f \times f_Q"] \arrow[r, "x"'] & Q_1 \arrow[u, "f_Q"']
\end{tikzcd}
$$

but please note:

- In many situations A and B stay the same ($f = 1_A$ and $g = 1_B$) and only Q is refined.
- Both σ_Q and σ_B are partial, in general.

B.3.3 Non-deterministic state transformers

In this case post-$\sigma(q, a, q', b')$ is an arbitrary predicate. We may still regard it as a Boolean-function and adapt refinement equation (B.1) to this situation, for f_Q the state abstraction function.

Leaving invariants and pre-conditions aside, for the moment, we will have:

$$
\begin{array}{l}
\forall q_1, q_1' \in Q_1, a_1 \in A_1, b_1' \in B_1 : \\
\qquad \text{post-}\sigma_1(q_1, a_1, q_1', b_1') = \\
\qquad \text{post-}\sigma(f_Q(q_1'), f(a_1), f_Q(q'), g(b_1'))
\end{array}
\tag{B.2}
$$

But logical equality ($=$) in (B.2) is too strong: it should be relaxed to implication ($\Rightarrow$),

$$
\begin{array}{l}
\forall q_1, q_1' \in Q_1, a_1 \in A_1, b_1' \in B_1 : \\
\qquad \text{post-}\sigma_1(q_1, a_1, q_1', b_1') \Rightarrow \\
\qquad \text{post-}\sigma(f_Q(q_1'), f(a_1), f_Q(q'), g(b_1'))
\end{array}
\tag{B.3}
$$

since increasing determinism is a valid refinement decision (recall the *strengthen postcondition* law of [Mor90]).

B.3.4 Last but not least...

If an operation is known to preserve an invariant α, then any of its proper reifications, as entailed by the SETS calculus, will preserve it as well (see [Oli97]).

C. Automatic Data Reification in the Camila Toolset

This appendix describes how the CAMILA ART tool [1] works. This tool is an automatic "reifier" which performs term-rewriting on a genetic algorithm basis. ART is intended to process the specification of a collection of type definitions, written in the CAMILA notation and, by converting them to the SETS abstract notation, to refine them via the laws of SETS downto elementary, reusable data models available in a data refinement repository.
The reification process is governed by genetic algorithms. These have the potential to support inequational term-rewriting, such as is required in animating the SETS calculus. Together with the CAMILA prototype system the "reifier" is able to rewrite CAMILA data models into concrete implementations such as relational DBMSs, JAVA, C++, and so on.

C.1 Automatic Reification of SETS Models

C.1.1 Some Basic Notions

The animation of a calculus via genetic algorithms implies the availability of some kind of evaluation function over formal expressions. In the SETS case, the fitness of a refinement process will be the evaluation of the result of applying a finite sequence of rules from the SETS calculus to a given SETS expression.
All rules taken from the SETS calculus have, in ART, a unique numeric identifier, *e.g.*

$$\left(\begin{array}{ccc} 1 & 2 & 3 \\ A \times B \longrightarrow B \times A & A \times (B \times C) \longrightarrow (A \times B) \times C & \cdots \end{array} \right) \tag{C.1}$$

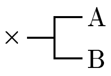
The "genetic engine" represents each potential solution as a chromosome of integer genes. Chromosome size fixes the number of refinement steps which can be applied to a given SETS expression. For example, if 10 rules are available (numbered from 1 to 10, inclusive) and we want to evaluate the result of composing 5 successive refinement steps, then each chromosome will be represented by a sequence such as

$$\left(\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\ r(10) & r(10) & r(10) & r(10) & r(10) \end{array} \right)$$

where $r(x), x \in \mathbb{N}$ returns a random number between 1 and x. In order to illustrate the basic behaviour of the reifier we will consider an evaluation function which guides the refinement process on a *Component Oriented Specification* basis. This means that, given a SETS expression representing the model we want to refine, the reifier must be able to refine such a model until a collection of known, reusable data models (also represented by SETS expressions) are identified. Some relevant functionality concerning expression-handling follows.

[1] ART ="Automatic Reification Tool" [Nev97].

Rules are applied on a *First Match* basis, that is, every rule applied to a given SETS expression will rewrite the outermost leftmost match occurrence. For example, rule $A \times B \longrightarrow B \times A$ applied to $X \hookrightarrow (Y+Z) \times W$ returns $X \hookrightarrow W \times (Y+Z)$.
SETS expression complexity is calculated as the number of nodes of the corresponding expression tree. For example, the complexity of $A \times B$ is 3 because



has three nodes.
Another important operation is $MetaMatch$. For e a SETS expression and p a pattern (which is also a SETS expression), $MetaMatch$ replaces the *First Match* of p in e by the meta-character $*$. For example, for $e = X \hookrightarrow W \times (Y+Z)$ and $p = A + B$, then $MetaMatch(e,p) = X \hookrightarrow W \times *$.
Finally, evaluation function $Eval$ computes the following formula, given a meta-expression e and a pattern p in SETS:

$$Eval(e,p) = \begin{cases} 1 & \Leftarrow match(e,p) \\ \frac{nMeta(x)}{complex(e) - nOper(e)} & \Leftarrow \neg match(e,p) \end{cases}$$

where $nOper(x)$ yields the number of symbols of operations of x and $nMeta(x)$ yields the number of meta-symbols of x.

C.1.2 SETS Expression Representations

The SETS abstract syntax fragment which the reifier takes into account is written in CAMILA syntax as follows:

```
 ; SETS sublanguage

    Sets = Cst | Var | Pset | List | Ffun | Prod | Plus;
    Cst  :: A:STR;
    Var  :: A:STR;
    List :: A:Sets;
    Prod :: A:Sets-list;
    Plus :: A:Sets-list;
    Pset :: A:Sets B:Sets;
    Ffun :: A:Sets B:Sets;

; end of SETS sublanguage
```

However, the reifier is not directly based on this syntax (this approach would lead to a very specific tool restricted to SETS's term re-writing only). Insted, a generic-expression data sort `Exp` is introduced,

```
  Exp    = VarOrTerm;
  VarOrTerm = Var | Term;
  Term   :: O: STR St: VarOrTerm-seq;
  Var    = STR;
  STRs   = STR-seq;
```

where variables and operators are modelled by strings, and constants are modelled by nullary operators. This accommodates any kind of notation and calculus (*e.g.* it is expected to represent SETS's morphism calculus in the future).
For instance, the representation of SETS expression

$$A \times (B + C)$$

will be

```
   Term( "x", < "A", Term( "+", < "B","C">)>)
```

and that of rule 1 in (C.1) will be ordered-pair

```
< Term( "x", < "A","B">), Term( "x", < "B","A">)>;
```

C.1.3 A Small Example

Consider the *DecTree* model (A.1) written in CAMILA syntax:

```
TYPE
  What = INT;
  Answer = INT;
  DecTree = A:What B:Answer -> DecTree;
ENDTYPE
```

Suppose the reifier's goal is to automate *DecTree*'s refinement process towards the relational model (as is given in [Oli92]). In this case $p = 2^K$. First, it will translate this piece of CAMILA syntax into the above `Sets` abstract syntax,

```
     What  ⟼  Cst(INT)
   Answer  ⟼  Cst(INT)
  DecTree  ⟼  Prod(<Var(What),Ffun(Var(Answer),Var(DecTree))>)
```

Then, recursion removal law

$$X_0 \trianglelefteq^{\phi}_{f} K \times K \hookrightarrow \mathcal{F}(\mathcal{K})$$

— *cf.* [Oli92] — is immediately selected and transforms *DecTree* into a non-recursive expression $DecTree'$:

```
     What  ⟼  Cst(INT)
   Answer  ⟼  Cst(INT)
 DecTree'  ⟼  Prod(<Var(K),Ffun(Var(K),Var(What),Ffun(Var(Answer),Var(K)))>)
```

$DecTree'$ is described by the following `Exp`-object

```
Term(
   "x",
   < "K",
     Term(
        "->",
        < "K",
```

```
Term(
   "x",
   < "What",
     Term(
        "x",
        < "Answer","K">)>)>)>)>);
```

which the reifier processes, starting with population size = 20 and chromosome size = 5. Three generations later, the outcome is as follows:

```
> Best(0): 20%
> thinking... best (1): 66%
> thinking... best (2): 66%
> thinking... best (3): 66%
> Results:
> applying 35... (K * (K -> (What * (Answer -> K))))
> applying 36... (K * ((K -> What) * ((K * Answer) -> K)))
> applying 38... (K * ((2 ^ (K * What)) * ((K * Answer) -> K)))
> applying 38... (K * ((2 ^ (K * What)) * (2 ^ ((K * Answer) * K))))
> applying 31... (K * ((2 ^ (K * What)) * (2 ^ ((K * Answer) * K))))
```

On each generation, all the individuals will apply their own chromosome to the expression (corresponding to the `thinking...` message above). After this, one (or more than one) will be classified as the best one(s), *i.e.*, the one(s) responsible for the best evaluation(s). Then, the best score is finally shown (this corresponds to the `best(i): v%` message). Then, the genetic process (*selection*, *crossover* and *mutation*) will continue for the next $n-1$ generations (3 in the example above). In the end, the best reification process (corresponding to the best descendent, *i.e.*, a "child" of the best "fathers" and "mothers", "grandfathers" and "grandmothers", and so on, will be shown on the screen.
The whole process can be analysed as follows, where A and W abbreviate *Answer* and *What*, respectively:

1. $K \times (K \hookrightarrow (W \times (A \hookrightarrow K)) \xrightarrow{35} \underbrace{K \times (K \hookrightarrow (W \times (A \hookrightarrow K))}_{x}$
   (a) $y = MetaMatch(x, 2^K) = x$
   (b) $nOper(y) = 4$
   (c) $nMeta(y) = 0$
   (d) $nComplex(y) = 9$
   (e) $eval(y) = \frac{0}{9-4} = 0$
2. $K \times (K \hookrightarrow (W \times (A \hookrightarrow K)) \xrightarrow{36} \underbrace{K \times (K \hookrightarrow W) \times (K \times A \hookrightarrow K)}_{x}$
   (a) $y = MetaMatch(x, 2^K) = x$
   (b) $nOper(y) = 5$
   (c) $nMeta(y) = 0$
   (d) $nComplex(y) = 11$
   (e) $eval(x) = \frac{0}{11-5} = 0$
3. $K \times (K \hookrightarrow W) \times (K \times W \hookrightarrow K) \xrightarrow{38} \underbrace{K \times (2^{K \times W}) \times (K \times A \hookrightarrow K)}_{x}$

(a) $y = MetaMatch(x, 2^K) = K \times (* \times (K \times A \hookrightarrow K))$

(b) $nOper(y) = 4$

(c) $nMeta(y) = 1$

(d) $nComplex(y) = 9$

(e) $eval(y) = \frac{1}{9-4} = 0.33$

4. $K \times (K \hookrightarrow W) \times (K \times W \hookrightarrow K) \xrightarrow{38} \underbrace{K \times (2^{K \times W}) \times (2^{K \times W \times K})}$

(a) $y = MetaMatch(x, 2^K) = K \times (* \times *)$

(b) $nOper(y) = 2$

(c) $nMeta(y) = 2$

(d) $nComplex(y) = 5$

(e) $eval(y) = \frac{2}{5-2} = 0.66$

C.1.4 Quick Reference Guide

- `FUNC CAM2SETS(f:STR):MSets` — This operation loads CAMILA specification file `f`, and stores the result into variable `M`. The result of such a "transformation" will be stored into variable `M`. The following step will prepare the resultant representation in order to start the reification process.

- `FUNC rfPREPARE(s:STR)` — Given starting symbol `s`, this operation prepares model `M` in order to be automatically reified. The preparation process includes, for instance, the checking for recursive entities and the application of recursion removal laws, the splitting of model `M` into variable independent sub-models, the conversion of model `M` to the generic-expression data sort *Exp*, and so on.

- `FUNC rfSTART(e,t,n)` — Given an *Exp*-expression e, a reification type t, and a number of generations n, it reifies e until a collection of known, reusable data models are identified. In the current prototype, type t identifies:

  - the ER model when $t = 1$, i.e., pattern $p = 2^K$
  - the FF model when $t = 3$, i.e., pattern $p = A \hookrightarrow B$
  - the ER/FF model when $t = 5$, i.e., $p = 2^K$ or $p = A \hookrightarrow B$

- `FUNC rfPSTART(e,t,n)` — Similar to the above but it will only use a subset of more specific, ER-model oriented SETS rules.

- `FUNC rfPSTARTPPD(e,t,n)` — Similar to the `rfPSTART` function above but it will only use a subset of more specific, ER-model oriented SETS rules. Additionally, it will modify some of the genetic parameters for efficiency purposes.

- `FUNC rfPSTARTBAMS(e,t,n)` — Similar to the `rfPSTART` function above but it will only use a subset of more specific, ER-model oriented SETS rules. Additionally, it will modify some of the genetic parameters for efficiency purposes.

- `FUNC rfSHOWREIF(e)` — Given *Exp*-expression e, this shows the complete reification process which e was subject to.

- `FUNC rfSAVEREIF(e,f)` — Given *Exp*-expression e, this saves the complete reification process which was applied to e into file f. The generated output is a LaTeX file which can then be processed independently.

- FUNC RRSHOW(i) — This shows all the information available for the SETS law number i (in the ART repertoire).
- FUNC RRSAVE() — This saves all the information available for the SETS law number i (in the ART repertoire) into LaTeX format.

C.1.5 A Sample Session

```
UM-XMETOO version 1.10.DLL.USERTYPE (aap/pem/cjr/jj  97/02/26 )
/home/fln/.metoorc")
arf.cam
?- CAM2SETS("bams.cam");
_tmp
?- M;
[ BAMS ->
     Ffun(
        Var(
           AccId ),
        Var(
           Account ))
  Account ->
     Prod(
        <
          Pset(
             Cst(
                2 ),
             Var(
                AccHolder ))
          Var(
             Amount )>)
  AccId ->
     Cst(
        STR )
  AccHolder ->
     Cst(
        STR )
  Amount ->
     Cst(
        INT )]
?- rfPREPARE("BAMS");
x
?- rfPSTARTBAMS(x,1,3);
CS (Chromsize)              = 15
PS (Population Size)        = 30
SS (Segment Size)           = 0 (for binary representation only)
PC (Probability-Crossover) = 25
PM (Probability-Mutation)  = 0
VC (Variable Size Chrom.)  = false
GR (Gene Representation)   = 1 (0=binary, 1=decimal)
MV (Maximum Gene Value)    = 39 (for decimal representation only)
> Best(0): 25%
> thinking... best (1): 33%
```

```
> thinking... best (2): 100%
> thinking... best (3): 100%
> thinking...< (  (  2  ^  (  AccId  *  Amount  )  )  *  (  2  ^  (  AccId  *  AccHolder  )  )  ) >
?- rfSHOWREIF(x);
> applying 30... (( AccId -> ( ( AccHolder -> 1 ) * Amount ) ))
> abs = ((*-> prod( dom ,id) ))
> rep = ((*-> prod( ffAset ,id) ))
> applying 1... (( AccId -> ( Amount * ( AccHolder -> 1 ) ) ))
> abs = ((*-> const(p2,p1) ))
> rep = ((*-> const(p2,p1) ))
> applying 36... (( ( AccId -> Amount ) * ( ( AccId * AccHolder ) -> 1 ) ))
> abs = (nJoin)
> rep = (nJoinInv)
> applying 38... (( ( 2 ^ ( AccId * Amount ) ) * ( ( AccId * AccHolder ) -> 1 ) ))
> abs = (prod( mkf ,id))
> rep = (prod( mkr ,id))
> applying 29... (( ( 2 ^ ( AccId * Amount ) ) * ( 2 ^ ( AccId * AccHolder ) ) ))
> abs = (prod(id, ffAset ))
> rep = (prod(id, dom ))
```

D. Prototype Listings

D.1 bams.cam

```
#include io.cam
#include sets.cam
#include ffs.cam
#include txt.cam
#include sql.cam

/****************************************************************************
 *                                                                          *
 *      1. The Specification                                                *
 *                                                                          *
 ****************************************************************************/

TYPE
        BAMS = AccId -> Account;
        Account :: H: AccHolder-set
                   B: Amount;
        AccId   = STR;
        AccHolder = STR;
        Amount  = INT;
;       Void    = SYM;

ENDTYPE
```

```
STATE bams: BAMS;

;-------------------------------------------------------------------------------
FUNC Init():Void
;-------------------------------------------------------------------------------
STATE
bams <-
[
  "0636024550431"-> Account( { "J N Oliveira","L.S. Barbosa"}, 3000),
  "0636024000999"-> Account( { "Neves,  Fernando L.","J N Oliveira"}, 9000 ),
  "0636099999000"-> Account( { "J.J.A."   },   9500 ),
  "0636055566677"-> Account( {            },   10 )
];

;-------------------------------------------------------------------------------
FUNC OpenAccount(n:AccId,h:AccHolder-set,m:Amount):Void
;-------------------------------------------------------------------------------
STATE bams <- [ n -> Account(h,m) ] + bams;

;-------------------------------------------------------------------------------
FUNC RemoAccount(n:AccId):Void
;-------------------------------------------------------------------------------
STATE bams <- bams \ { n };

;-------------------------------------------------------------------------------
FUNC AddAccHolders(n:AccId,h:AccHolder-set):Void
;-------------------------------------------------------------------------------
STATE bams <- bams + let (addh=lambda(x).Account(H(x) U h,B(x)))
                     in (*->addh)(bams/{n});


;-------------------------------------------------------------------------------
FUNC Credit(n:AccId,m:Amount):Void
;-------------------------------------------------------------------------------
STATE
bams <- bams + let (addm=lambda(x).Account(H(x), B(x) .+ m))
               in (*->addm)(bams/{n});


;-------------------------------------------------------------------------------
FUNC Withdraw(n:AccId,m:Amount):Void
;-------------------------------------------------------------------------------
STATE
bams <- bams + let (subm=lambda(x).let (b=B(x),
                                        c= if b >= m then b .- m else b)
                                   in Account(H(x), c))
               in (*->subm)(bams/{n});

;-------------------------------------------------------------------------------
FUNC GoodCostumers(m:Amount):AccHolder-set
;-------------------------------------------------------------------------------
RETURN UNION( { H(bams[k]) | k <- dom(bams): B(bams[k]) >= m });


;-------------------------------------------------------------------------------
```

```
; Initialization
;---------------------------------------------------------------------------
;Init();
;---------------------------------------------------------------------------

/***************************************************************************
 *                                                                         *
 *      2. The Reification Process                                         *
 *                                                                         *
 ***************************************************************************/

abs0 <- (*->lambda(x).Account(p1(x),p2(x)));
rep0 <- (*->const(lambda(x).H(x),lambda(x).B(x)));

abs1 <- (*->const(p2,p1));
rep1 <- (*->const(p2,p1));

abs2 <- (*->prod(id,dom));
rep2 <- (*->prod(id,ffAset));

abs3 <- nJoin;
rep3 <- nJoinInv;

abs4 <- prod(id,ffAset);
rep4 <- prod(id,dom);

abs5 <- prod(mkf,id);
rep5 <- prod(mkr,id);

abs6 <- prod(ff2pair-set,ff2pair-set);
rep6 <- prod(seq2ff-set,seq2ff-set);

abs7()= let (ss = abs7_2(),
             _ = nload("_bamsAmsTable"))
        in < _bamsAmsTable, ss >;
abs7_2() = ioAwk2afs("_bamsAccHolTable","%");
rep7(r) = do(_bamsAmsTable<-p1(r),
             ioObjSave('_bamsAmsTable,p1(r)),
             ioAfs2awk("_bamsAccHolTable",p2(r),"%")
            );

abs6to0(r)=abs0(abs1(abs2(abs3(abs4(abs5(abs6(r)))))));
rep0to6(a)=rep6(rep5(rep4(rep3(rep2(rep1(rep0(a)))))));

a()=abs6to0(abs7());
r()=rep7(rep0to6(bams));

/***************************************************************************
 *                                                                         *
 *      3. The Implementation (Level 7)                                    *
 *                                                                         *
 ***************************************************************************/


;---------------------------------------------------------------------------
```

```
FUNC Init7():Void
;-----------------------------------------------------------------------------
STATE rep7(rep0to6(
[
  "0636024550431"-> Account( { "J N Oliveira","L.S. Barbosa"}, 3000),
  "0636024000999"-> Account( { "Neves,  Fernando L.","J N Oliveira"}, 9000 ),
  "0636099999000"-> Account( { "J.J.A."   },   9500 ),
  "0636055566677"-> Account( {            },   10 )
]));

;-----------------------------------------------------------------------------
FUNC OpenAccount7(n:AccId,h:AccHolder-set,m:Amount):Void
;-----------------------------------------------------------------------------
STATE do(ioObjLoad('_bamsAmsTable),
         if card(sqlSelect(_bamsAmsTable, {1,2}, 1, lambda(x).x==n))==0
         then let (a=_bamsAmsTable U { [ 1 -> n, 2 -> m ] },
                   b= { [ 1 -> n, 2 -> i ] | i <- h })
              in do(
                    _bamsAmsTable <- a,
                    ioObjSave('_bamsAmsTable,a),
                    ioAfs2awk("_tmp0",b,"%"),
                    sh("cat _tmp0 _bamsAccHolTable > _tmp1"),
                    sh("cp  _tmp1 _bamsAccHolTable "),
                    sh("ls -l     _bamsAccHolTable ")
                   )
         else "\nSorry, account exists");

;-----------------------------------------------------------------------------
FUNC RemoAccount7(n:AccId):Void
;-----------------------------------------------------------------------------
STATE do(ioObjLoad('_bamsAmsTable),
         _bamsAmsTable <- sqlDelete(_bamsAmsTable, 1, lambda(x).n==x),
         ioObjSave('_bamsAmsTable,_bamsAmsTable),
         sh("awk '{ FS=\"%\"; if ($1!="
            ++ n ++
            ") print }' _bamsAccHolTable > _tmp"),
         sh("cp _tmp _bamsAccHolTable"),
         "\n[ successful awk call ]"
        );

;-----------------------------------------------------------------------------
FUNC Withdraw7(n:AccId,m:Amount):Void
;-----------------------------------------------------------------------------
STATE do(ioObjLoad('_bamsAmsTable),
         if card(
                sqlSelect(sqlSelect(_bamsAmsTable, {1,2}, 1, lambda(x).x==n),
                          {1,2}, 2, lambda(x).x>=m),
               )==0
         then "\nNon-existing account or insufficient balance"
         else do(
              _bamsAmsTable <-
              sqlUpdate(_bamsAmsTable, 2, lambda(x).x .- m, 1, lambda(x).n==x),
              ioObjSave('_bamsAmsTable,_bamsAmsTable)
                )
        );
```

```
;----------------------------------------------------------------------------
FUNC AddAccHolders7(n:AccId,h:AccHolder-set):Void
;----------------------------------------------------------------------------
STATE let (a = sqlSelect(_bamsAmsTable,{1},1,lambda(x).x==n),
           b = { [ 1 -> n, 2 -> i ] | i <- h },
           c = ffsJoin(a,b))
      in do(
              ioAfs2awk("_tmp0",c,"%"),
              sh("cat _tmp0 _bamsAccHolTable | sort -u > _tmp1"),
              sh("cp  _tmp1 _bamsAccHolTable "),
              sh("ls -l     _bamsAccHolTable "),
              "\n[ ffsJoin performed at CAMILA level ]"
           );

;----------------------------------------------------------------------------
FUNC Credit7(n:AccId,m:Amount):Void
;----------------------------------------------------------------------------
STATE do(ioObjLoad('_bamsAmsTable),
         _bamsAmsTable <-
               sqlUpdate(_bamsAmsTable, 2, lambda(x).m .+ x, 1, lambda(x).n==x),
         ioObjSave('_bamsAmsTable,_bamsAmsTable)
        );

;----------------------------------------------------------------------------
FUNC GoodCostumers7(m:Amount):AccHolder-set
;----------------------------------------------------------------------------
RETURN let (a = sqlSelect(_bamsAmsTable, {1}, 2, lambda(x).x>=m),
            _ = do(
                   ioAfs2awk("_tmp",a,"%"),
                   sh("sort _tmp > _tmp0"),
                   sh("sort _bamsAccHolTable > _tmp1"),
                   sh("ls -l _tmp0 _tmp1 "),
                   sh("join -t % _tmp0 _tmp1 > _tmp"),
                   princ("[ join performed at UNIX level ]\n\n")
                  ),
            b = ioAwk2afs("_tmp","%"))
       in ffsPro(2,b);

/****************************************************************************
 *                                                                          *
 *      4. Extra Reification Laboratory Functionality                       *
 *                                                                          *
 ****************************************************************************/

ROLLBACK2REP()= bams <- abs6to0(abs7());

ROLLBACK2ABS()= rep7(rep0to6(bams));

/****************************************************************************
 *                                                                          *
 *      5. Screen and LaTeX Interface Functions                             *
 *                                                                          *
 ****************************************************************************/
```

```
view0(x)=ioAfsPrint(show0(rep0(x)));
tex0(x)=ioTxtPrint(ffs2TeXtxt(show0(rep0(x))));
show0(x)= let (f=lambda(x).strCAT(seqAddSep(<a | a <- x>, " + ")),
               y=(*->prod(f,itoa))(x),
               z= { [ "AccId"->a,
                      "AccHolders"->p1(y[a]),
                      "Balance"->p2(y[a])
                    ]
                  | a <- dom(y) })
          in z;

tex6(x)= let (a=show6(x),
              f=lambda(x).ioTxtPrint(ffs2TeXtxt(x)))
         in < f(p1(a)), princ(" ~~~~~\n "), f(p2(a)) >;
view6(x)= let (_=prod(ioAfsPrint,ioAfsPrint)(show6(x))) in "";
show6 <- prod((lambda(x).ffsRename([1->"AccId",2->"Balance"],x)),
              (lambda(x).ffsRename([1->"AccId",2->"AccHolder"],x)));
```

D.2 ppd.cam

```
#include io.cam
#include mse.cam
#include sets.cam
#include ffs.cam
#include txt.cam
#include sql.cam

/******************************************************************************
 *                                                                            *
 *      1. The Specification                                                  *
 *                                                                            *
 ******************************************************************************/

TYPE
        PPD  ::  S: Stock
                 P: Pricelist
                 E: EquipDb;
        Stock  = Unid -> NAT;
        Pricelist = Comp -> Currency;
        EquipDb = Equip -> (Unid -> NAT);
        Unid  = Equip | Comp;
        Comp  :: K: NAT;
        Equip :: K: NAT;
        Currency = NAT;
        NAT = INT;

ENDTYPE

STATE ppd: PPD;

;-----------------------------------------------------------------------------
FUNC Explode(k:Equip):Comp -> NAT
;-----------------------------------------------------------------------------
RETURN explode(k,E(ppd));
```

```
;---------------------------------------------------------------------------
FUNC CompCost(k:Equip,n:NAT):Comp -> NAT
;---------------------------------------------------------------------------
RETURN let (ms=mseExMul(explode(k,E(ppd)),n))
       in mseDiff(ms,S(ppd));

;---------------------------------------------------------------------------
FUNC explode(k:Equip,db:EquipDb):Comp -> NAT
;---------------------------------------------------------------------------
RETURN let (ms=if k in dom(db) then db[k] else [])
       in mseCUP(< if is-Comp(x) then ms / { x }
                                 else mseExMul(explode(x,db \ {k}),ms[x])
                 | x <- dom(ms) >);

;---------------------------------------------------------------------------
FUNC Init():SYM
;---------------------------------------------------------------------------
STATE
ppd <- PPD(
[
Comp(11) -> 10,
Comp(12) ->  5,
Comp(13) ->  1,
Comp(14) -> 20
],
[],
[
  Equip(
     11 )->[
       Comp(
          11 )->1 ,
       Comp(
          12 )->3  ],
  Equip(
     12 )->[
       Comp(
          13 )->4 ,
       Equip(
          11 )->3  ],
  Equip(
     10 )->[
       Equip(
          11 )->3 ,
       Equip(
          12 )->2  ] ]);

;---------------------------------------------------------------------------
; Initialization
;---------------------------------------------------------------------------
;Init();
;---------------------------------------------------------------------------

/***************************************************************************
 *                                                                         *
```

```
 *      2. The Reification Process                                           *
 *                                                                           *
 *****************************************************************************/

abs_Unid(x) = let (i=p1(x),y=p2(x)) in if i==2 then Comp(y) else Equip(y);
rep_Unid(x) = < if is-Comp(x) then 2 else 1,K(x)>;

rep0 <- lambda(x).< <(rep_Unid->*)(S(x)), (K->*)(P(x))>,
                  (*->(rep_Unid->*))((K->*)(E(x)))>;

rep1 <- prod(id,uncurry);

rep2 <- prod(id,j606o->*);

rep3 <- prod(id,pJoinInv);

rep4 <- prod(id,prod(mkr,mkr));

rep5 <- prod(id,prod(flat-set,flat-set));

rep6 <- prod(id,prod(seq2ff-set,seq2ff-set));

rep7(t) = do(_ppdSP<-p1(t),
             ioObjSave('_ppdSP,p1(t)),
             ioAfs2awk("_ppdEETable",p1(p2(t)),"%"),
             ioAfs2awk("_ppdECTable",p2(p2(t)),"%")
            );

abs6to0(r)=abs0(abs1(abs2(abs3(abs4(abs5(abs6(r)))))));
rep0to6(a)=rep6(rep5(rep4(rep3(rep2(rep1(rep0(a)))))));

/*****************************************************************************
 *                                                                           *
 *      3. The Implementation (Level 7)                                      *
 *                                                                           *
 *****************************************************************************/

;-----------------------------------------------------------------------------
FUNC Explode7(k:Equip):Comp -> NAT
;-----------------------------------------------------------------------------
RETURN do(
           sh("perl ppdExplode.pl " ++ itoa(K(k))),
           ioAfsPrint(ffsRename([1->"Comp",2->"Qty"],ioAwk2afs("_r","%")))
         );

;-----------------------------------------------------------------------------
FUNC Init7():SYM
;-----------------------------------------------------------------------------
STATE rep7(rep0to6(
PPD(
[
        Comp(11) -> 10,
        Comp(12) ->  5,
        Comp(13) ->  1,
        Comp(14) -> 20
```

```
],
[],
[
  Equip(
     11 )->[
       Comp(
          11 )->1 ,
       Comp(
          12 )->3  ],
  Equip(
     12 )->[
       Comp(
          13 )->4 ,
       Equip(
          11 )->3  ],
  Equip(
     10 )->[
       Equip(
          11 )->3 ,
       Equip(
          12 )->2  ] ])
));

;------------------------------------------------------------------------------
FUNC CompCost7(k:Equip,n:NAT):Comp -> NAT
;------------------------------------------------------------------------------
RETURN do(
           sh("perl ppdExplode.pl " ++ itoa(K(k))),
           sh("awk -f otimes.awk m=" ++ itoa(n) ++ " _r > _tmp0 "),
           princ("\n[ mseExMul successfully performed at UNIX level ]\n"),
           ioObjLoad('_ppdSP),
           let (a=(abs_Unid->*)(p1(_ppdSP)),
                b=ioAwk2afs("_tmp0","%"),
                c=[ Comp(atoi(t[1])) -> atoi(t[2]) | t <- b ])
           in princ("\n[ Refinement memo: mseDiff still at abstract level ]\n",
                    "\nResult is:\n",pp(mseDiff(c,a)),),
           ""
         );
```