7

CONTRACT-ORIENTED PROGRAMMING

The chapters of the first part of this book rely on a type-polymorphic notion of computation, captured by the omnipresent use of the arrow notation

$$B \xleftarrow{f} A$$

where A and B are *types*.

The generalization from functions to relations carried out in the previous two chapters has preserved the same principle — all relational combinators are typed in the same way. There is thus an implicit assumption of *static type checking* in the overall approach — types are checked at "compile time". Expressions which don't type are automatically excluded.

However, examples such as the Alcuin puzzle show that this is insufficient. Why? Because the types involved are most often "too large": the whole purpose of the puzzle is to consider only the inhabitants of type $Bank^{Being}$ — functions that describe all possible configurations in the puzzle — that satisfy the "starvation property", recall (5.76). Moreover, the *carry* _ operation (5.208) *should* preserve this property — something we didn't at all check in the previous chapter!

Let us generalize the situation in this puzzle to that of a function $f : A \to A$ and a predicate $p : A \to \mathbb{B}$ that should be preserved by f. Predicates such as p have become known as *invariants* by software theorists. The preservation requirement is captured by:

$$\langle \forall\, a \;:\; p\, a :\; p\, (f\, a) \rangle$$

Note how the type A is now divided in two parts — a "good one", $\{a \mid a \in A \wedge p\ a\}$ and a "bad one", $\{a \mid a \in A \wedge \neg\ (p\ a)\}$. By identifying p as an invariant, the programmer is *obliged* to ensure a "good" output $f\ a$ wherever a "good" input is passed to f. For "bad" inputs nothing is requested.

The situation above can be generalized to some $f : A \to B$ where B is subject to some invariant $q : B \to \mathbb{B}$. So f is *obliged* to ensure "good" outputs satisfying q. It may well be the case that the only way for f to ensure "good" outputs is to restrict its inputs by some precondition $p : A \to \mathbb{B}$. Thus the proof obligation above generalizes to:

$$\langle \forall\, a \;:\; p\, a :\; q\, (f\, a) \rangle \tag{7.1}$$

One might tentatively try and express this requirement by writing

$$p \xrightarrow{f} q$$

where predicates p and q take the place of the original types A and B, respectively. This is what we shall do, calling assertion $p \xrightarrow{f} q$ a *contract*. Note how are are back to the function-as-a-contract view of section 2.1 but in a wider setting:

> f *commits itself* to producing a "good" B-value (wrt. q) provided it is supplied with a "suitable" A-value (wrt. p).

The main difference compared to section 2.1 is that the well-typing of $p \xrightarrow{f} q$ cannot be mechanically ascertained at "compile time" — it has to be validated by a formal proof — the proof obligation (7.1) mentioned above. This kind of type checking is often referred to as "extended type checking".

In real life software design data type invariants can be arbitrarily complex — think of all legal restrictions imposed on the organized societies of today! The increasing "softwarization" of our times forces us to think that, as in the regular functioning of such organized *societies*, programs should interact with each other via *formal contracts* establishing what they rely upon or guarantee among themselves. This is the only way to ensure *safety* and *security* essential to reliable, mechanized operations.

This chapter will use relation algebra to describe such contracts and develop a simple theory about them, enabling compositionality as before. Relations (including functions) will play a double role — they will not only describe computations but also the data structures involved in such computations, in a unified and elegant way.

7.1 CONTRACTS

It should be routine work for the reader to check that

$$f \cdot \Phi_p \subseteq \Phi_q \cdot f \tag{7.2}$$

means exactly the same as (7.1) above. In software design terminology, this is known as a (functional) *contract*, and we shall write

$$p \xrightarrow{f} q \tag{7.3}$$

to denote it — a notation that generalizes the type $A \to B$ of f, as already observed. Thanks to (5.207), (7.2) can also be written:

$$f \cdot \Phi_p \subseteq \Phi_q \cdot \top \tag{7.4}$$

Predicates p and q in contract $p \xrightarrow{f} q$ shall be referred to as the contract's *precondition* and *postcondition*, respectively. Contracts compose sequentially, see the following exercise.

***Exercise** 7.1. Show that $q \xleftarrow{g \cdot f} p$ holds provided $r \xleftarrow{f} p$ and $q \xleftarrow{g} r$ hold.*
□

WEAKEST PRE-CONDITIONS Note that more than one (*pre*) condition p may ensure (*post*) condition q on the outputs of f. Indeed, contract $false \xrightarrow{f} q$ always holds, but it is useless — pre-condition *false* is "*unacceptably strong*".

Clearly, the weaker p the better. The question is, then: is there a *weakest* such p? We calculate:

$$
\begin{array}{ll}
 & f \cdot \Phi_p \subseteq \Phi_q \cdot f \\
\equiv & \quad \{ \text{ recall (5.207) } \} \\
 & f \cdot \Phi_p \subseteq \Phi_q \cdot \top \\
\equiv & \quad \{ \text{ shunting (5.46); (5.205) } \} \\
 & \Phi_p \subseteq f^\circ \cdot \dfrac{true}{q} \\
\equiv & \quad \{ \text{ (5.52) } \} \\
 & \Phi_p \subseteq \dfrac{true}{q \cdot f} \\
\equiv & \quad \{ \Phi_p \subseteq id \text{ ; (5.58) } \} \\
 & \Phi_p \subseteq id \cap \dfrac{true}{q \cdot f} \\
\equiv & \quad \{ \text{ (5.198) } \} \\
 & \Phi_p \subseteq \Phi_{(q \cdot f)}
\end{array}
$$

We conclude that $q \cdot f$ is such a *weakest* pre-condition. Notation $wp\ (f, q) = q \cdot f$ is often used to denote a *weakest* pre-condition (WP). This is the weakest constraint on inputs for outputs by f to fall within q. The special situation of a weakest precondition is nicely captured by the universal property:

$$f \cdot \Phi_p = \Phi_q \cdot f \quad \equiv \quad p = q \cdot f \tag{7.5}$$

where $p = wp\ (f, q)$ could be written instead of $p = q \cdot f$, as seen above. Property (7.5) enables a "logic-free" calculation of weakest pre-conditions, as we shall soon see: given f and post-condition q, there

always exists a unique (weakest) precondition p such that $\Phi_q \cdot f$ can be replaced by $f \cdot \Phi_p$. Moreover:

$$\frac{f}{\overline{f}} \cdot \Phi_p = \Phi_p \cdot \frac{f}{\overline{f}} \quad \Leftarrow \quad p \leqslant f \tag{7.6}$$

where $\leqslant$ denotes the injectivity preorder (5.230) on functions.[1]

Exercise *7.2. Calculate the weakest pre-condition wp* (f, q) *for the following function / post-condition pairs:*

- $f\ x = x^2 + 1$, $q\ y = y \leqslant 10$ *(in* $\mathbb{R}$*)*
- $f = \mathbb{N}_0 \xrightarrow{\mathsf{succ}} \mathbb{N}_0$, $q =$ *even*
- $f\ x = x^2 + 1$, $q\ y = y \leqslant 0$ *(in* $\mathbb{R}$*)*

□

INVARIANTS In case $p = q$ in a contract (7.3), that is, in case of $q \xrightarrow{f} q$ holding, we say that q is an *invariant* of f, meaning that the "truth value" of q remains unchanged by execution of f. More generally, invariant q is *preserved* by function f provided contract $p \xrightarrow{f} q$ holds and $p \Rightarrow q$, that is, $\Phi_p \subseteq \Phi_q$.

Some pre-conditions are weaker than others wrt. invariant preservation. We shall say that w is the *weakest* pre-condition for f to preserve *invariant* q wherever $wp\ (f, q) = w \wedge q$, where $\Phi_{p \wedge q} = \Phi_p \cdot \Phi_q$.

Recalling the Alcuin puzzle, let us define the *starvation* invariant as a predicate on the state of the puzzle, passing the *where* function as a parameter w:

$$starving\ w \quad = \quad w \cdot CanEat \subseteq w \cdot \underline{Farmer}$$

Then the *contract*

$$starving \xrightarrow{carry\ b} starving$$

would mean that the function *carry b* — that should transfer the beings in b to the other bank of the river — always preserves the invariant:

$$wp\ (carry\ b, starving) = starving.$$

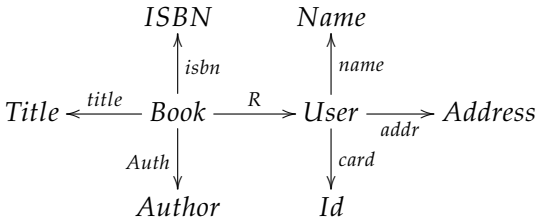
Things are not that easy, however: there is a need for a *pre-condition* ensuring that b includes the farmer together with a good choice of the being to carry!

Let us see some simpler examples first.

1 The interested reader will find the proofs of (7.5) and (7.6) in reference [50].

7.2 LIBRARY LOAN EXAMPLE

Consider the following relational data model of a library involving books and users that can borrow its books:



(7.7)

All arrows denote attributes (functions) but two — *Auth* and *R*. The former is a relation because a book can have more that one author.[2] The latter is the most interesting relation of the model, *u R b* meaning "book *b* currently on loan to library user *u*". Quite a few invariants are required in this model, for instance:

- *the same book cannot be not on loan to more than one user;*
- *no book exists with no authors;*
- *no two different users have the same card Id;*
- *books with the same ISBN should have the same title and the same authors.*

Such properties (invariants) are easy to encode:

- no book on loan to more than one user:

  $Book \xrightarrow{R} User$ is *simple*

- no book without an author:

  $Book \xrightarrow{Auth} Author$ is *entire*

- no two users with the same card Id:

  $User \xrightarrow{card} Id$ is *injective*

- *ISBN* is a *key* attribute:

  $ISBN \xrightarrow{title \cdot isbn^\circ} Title$ and $ISBN \xrightarrow{\Lambda Auth \cdot isbn^\circ} \mathsf{P}\, Author$ are *simple* relations.

Since all other arrows are functions, they are simple and entire.

Let us now spell out such invariants in terms of relational assertions (note the role of the injectivity preorder):

- no book on loan to more than one user:

2 Its power transpose (5.247) — $\Lambda Auth : Book \to \mathsf{P}Author$ — gives the *set* of authors of a book.

$$id \leqslant R^{\circ}$$

equivalent to $\mathsf{img}\ R\ \subseteq\ id$;

- no book without an author:

$$id\ \subseteq\ \mathsf{ker}\ Auth$$

- no two users with the same card Id:

$$id \leqslant card$$

equivalent to $\mathsf{ker}\ card\ \subseteq\ id$.

- *ISBN* is a *key* attribute:

$$title \leqslant isbn\ \wedge\ \Lambda Auth \leqslant isbn$$

equivalent to $\frac{isbn}{isbn}\ \subseteq\ \frac{title}{title}$ and $\frac{isbn}{isbn}\ \subseteq\ \frac{Auth}{Auth}$, respectively.[3]

Below we focus on the first invariant, *no book on loan to more than one user*. To bring life to our model, let us think of two operations on $User \xleftarrow{R} Book$, one that *returns* books to the library and another that *records* new borrowings:

$$(\mathsf{return}\ S)\ R = R - S \tag{7.8}$$

$$(borrow\ S)\ R = S \cup R \tag{7.9}$$

Note that parameter S is of type $User \xleftarrow{R} Book$, indicating which users borrow/return which books. Clearly, these operations only change the *books-on-loan* relation R, which is conditioned by invariant

$$inv\ R\ \ =\ \ \mathsf{img}\ R\ \subseteq\ id \tag{7.10}$$

The question is, then: are the following "types"

$$inv \xleftarrow{\mathsf{return}\ S} inv \tag{7.11}$$

$$inv \xleftarrow{borrow\ S} inv \tag{7.12}$$

valid? Let us check (7.11):

$$\begin{array}{ll} & inv\ (\mathsf{return}\ S\ R) \\ \equiv & \quad \{\ \text{inline definitions}\ \} \\ & \mathsf{img}\ (R - S)\ \subseteq\ id \\ \Leftarrow & \quad \{\ \text{since img is monotonic}\ \} \\ & \mathsf{img}\ R\ \subseteq\ id \\ \equiv & \quad \{\ \text{definition}\ \} \\ & inv\ R \\ \Box & \end{array}$$

3 Note the use of (5.170) in the second case.

So, for all R, $inv\ R \Rightarrow inv\ (\mathsf{return}\ S\ R)$ holds — invariant *inv* is preserved.

At this point note that (7.11) was checked only as a *warming-up* exercise — we don't actually need to worry about it! Why?

> As $R - S$ is smaller than R (exercise 5.41) and *"smaller than injective is injective"* (5.82), it is immediate that *inv* (7.10) is preserved.

To see this better, we unfold and draw definition (7.10) in the form of a diagram:

$$inv\ R \quad = \quad \begin{array}{ccc} Book & \xleftarrow{R^\circ} & User \\ {\scriptstyle R}\downarrow & \subseteq & \downarrow{\scriptstyle id} \\ User & \xleftarrow[id]{} & User \end{array}$$

As R occurs only in the lower-path of the diagram, it can always get smaller.

This "rule of thumb" does not work for *borrow S* because, in general, $R \subseteq borrow\ S\ R$. This time R gets bigger, not smaller, and we do have to check the contract:

$$\begin{array}{ll} & inv\ (borrow\ S\ R) \\ \equiv & \quad \{\ \text{inline definitions}\ \} \\ & \mathsf{img}\ (S \cup R)\ \subseteq\ id \\ \equiv & \quad \{\ \text{exercise 5.15}\ \} \\ & \mathsf{img}\ R\ \subseteq\ id \wedge \mathsf{img}\ S\ \subseteq\ id \wedge S \cdot R^\circ\ \subseteq\ id \\ \equiv & \quad \{\ \text{definition of } inv\ \} \\ & inv\ R \wedge \underbrace{\mathsf{img}\ S\ \subseteq\ id \wedge S \cdot R^\circ\ \subseteq\ id}_{wp\ (borrow\ S, inv)} \end{array}$$

Thus the complete definition of the *borrow* operation becomes, in the notation of section 5.3:

$$\begin{array}{l} Borrow\ (S, R : Book \to User)\ R' : Book \to User \\ \mathsf{pre}\ S \cdot S^\circ\ \subseteq\ id \wedge S \cdot R^\circ\ \subseteq\ id \\ \mathsf{post}\ R' = R \cup S \end{array}$$

Why have we written *Borrow* instead of *borrow* as before? This is because *borrow* has become a *simple* relation

$$Borrow = borrow \cdot \Phi_{\mathsf{pre}}$$

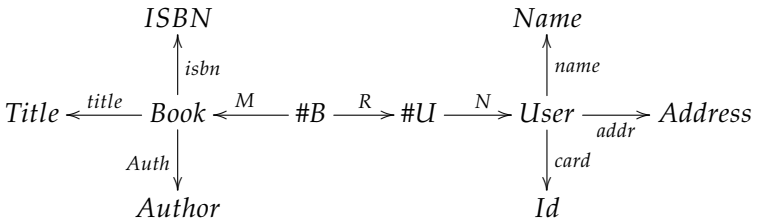
It is no longer a function since its (weakest) precondition is not the predicate *true*. (Recall that lowercase identifiers are reserved to functions only.) This precondition was to be expected, as spelt out by rendering $S \cdot R^\circ\ \subseteq\ id$ in pointwise notation: for all users u, u',

$$\langle \exists\, b \;:\; u\ S\ b : \; u'\ R\ b \rangle \Rightarrow u = u'$$

should hold. So, after the operation takes place, the result state $R' = R \cup S$ won't have the same book on loan twice to different users. (Of course, the same must happen about S itself, which is the same predicate for $R = S$.) Interestingly, the weakest precondition is not ruling out the situation in which $u\ S\ b$ and $u\ R\ b$ hold, for some book b and user u. Not only this does not harm the model but also it corresponds to a kind of renewal of a previous borrowing.

EVOLUTION The library loan model (7.7) given above is not realistic in the following sense — it only "gives life" to the borrowing relation R. In a sense, it assumes that all books have been bought and all users are registered.

How do we improve the model so that new books can be acquired and new users can join the library? Does this evolution entail a complete revision of (7.7)? Not at all. What we have to do is to *add* two new relations, say M and N, the first recording the books currently available in the library and the second the users currently registered for loaning:



Two new datatypes have been added: *#U* (unique identifier of each user) and *#B* (key identifying each book). Relations M and N have to be simple. The operations defined thus far stay the same, provided *#B* replaces *Book* and *#U* replaces *User* — advantages of a polymorphic notation. New operations can be added for

- acquiring new books — will change relation M only;
- registering new users — will change relation N only;
- cancelling users' registrations — will change relation N only.

There is, however, something that has not been considered: think of a starting state where $M = \bot$ and $N = \bot$, that is, the library has no users, no books yet. Then necessarily $R = \bot$. In general, users cannot borrow books that don't exist,

$$\delta\, R \ \subseteq\ \delta\, M$$

and not-registered users cannot borrow books at all:

$$\rho\, R \ \subseteq\ \delta\, N$$

Invariants of this kind capture so-called *referential integrity* constraints. They can be written with less symbols, cf.

$$R \subseteq \top \cdot M$$

and

$$R \subseteq N^\circ \cdot \top$$

respectively. Using the "thumb" rules as above, it is clear that, with respect to *referential integrity*:

- returning books is no problem, because R is only on the lower side of both inclusions;
- *borrow*ing books calls for new contracts — R is on the lower side and it increases!
- registering new users and buying new books are no problem, because M and N are on the upper side only;
- unregistering users calls for a contract because N is on the upper side and decreases — users must return all books before unregistering!

7.3 MOBILE PHONE EXAMPLE

In this example we go back to the *store* operation on a mobile phone list of calls specified by (5.2). Of the three invariants we select (b), the one requiring no duplicate calls in the list. Recall, in Haskell, the function $(!!) :: [a] \to \mathbb{Z} \to a$. This tells how a finite list s is converted into a partial function $(s!!)$ of type $\mathbb{Z} \to a$. In fact, the partiality extends to the negative numbers[4] and so we should regard $(s!!)$ as a *simple* relation[5] even if restricted to the type $a \leftarrow \mathbb{N}_0$, as we shall do below.

The no-duplicates requirement requests $(s!!)$ to be injective: in case $s !! i$ and $s !! j$ are defined, $i \neq j \Rightarrow s !! i \neq s !! j$. Let $L = (s!!)$. Then we can re-specify the operations of *store* in terms of L, as follows:[6]

$$\begin{aligned} &inv\ L = id \leqslant L \\ &\mathsf{filter}\ (c \neq)\ L = L - \underline{c} \\ &c : L = [\underline{c}, L] \cdot \mathsf{in}^\circ \end{aligned}$$

where $\mathsf{in} = [\underline{0}, \mathsf{succ}]$ — the Peano algebra which builds up natural numbers.[7] By (5.121) the definition of $c : L$ can also be written $\underline{c} \cdot \underline{0}^\circ \cup$

4 Try $[2,3,3] !! (-1)$, for instance.
5 Partial functions are *simple* relations, as we know.
6 Knowing that take 10 will always yield its input or a smaller list, and that *smaller than injective is injective* (5.82), we only need to focus on $(c:) \cdot \mathsf{filter}\ (c \neq)$.
7 Recall section 3.1.

$L \cdot \mathsf{succ}^\circ$, explicitly telling that c is placed in position 0 while L is shifted one position up to make room for the new element. We calculate:

$$
\begin{array}{ll}
& \mathit{inv}\ (c : (\mathsf{filter}\ (c \neq)\ L) \\
\equiv & \qquad \{\ \mathit{inv}\ L = \mathit{id} \leqslant L, \text{using the injectivity preorder}\ \} \\
& \mathit{id} \leqslant c : (\mathsf{filter}\ (c \neq)\ L) \\
\equiv & \qquad \{\ \text{in-line definitions}\ \} \\
& \mathit{id} \leqslant [\underline{c}, L - \underline{c}] \cdot \mathsf{in}^\circ \\
\equiv & \qquad \{\ \text{Galois connection (5.233)}\ \} \\
& \mathsf{in} \leqslant [\underline{c}, L - \underline{c}] \\
\equiv & \qquad \{\ \text{(5.235) ; in is as injective as } \mathit{id}\ \} \\
& \mathit{id} \leqslant \underline{c} \wedge \mathit{id} \leqslant L - \underline{c} \wedge \underline{c}^\circ \cdot (L - \underline{c}) \ \subseteq\ \bot \\
\Leftarrow & \qquad \{\ \text{constant functions are injective; } L \ \subseteq\ \top\ \} \\
& \mathit{id} \leqslant L - \underline{c} \wedge \underline{c}^\circ \cdot (\top - \underline{c}) \ \subseteq\ \bot \\
\Leftarrow & \qquad \{\ \text{smaller than injective is injective ; } \underline{c} \cdot (\top - \underline{c}) = \bot\ (5.155)\ \} \\
& \mathit{id} \leqslant L \\
\Box &
\end{array}
$$

Having given two examples of contract checking in two quite different domains, let us prepare for checking that of the Alcuin puzzle. By exercise 5.20 we already know that any of the starting states $w = \underline{Left}$ or $w = \underline{Right}$ satisfy the invariant:

$$\mathit{starving}\ w = w \cdot \mathit{CanEat} \ \subseteq\ w \cdot \underline{\mathit{Farmer}}.$$

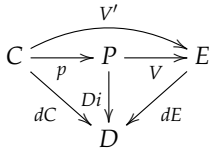
The only operation defined is

$$\mathit{carry}\ \mathit{who}\ \mathit{where} = (\in \mathit{who}) \rightarrow \mathit{cross} \cdot \mathit{where}\ ,\ \mathit{where}$$

Clearly, calculating the weakest precondition for this operation to preserve *starving* is expected to be far more complex than in the previous examples, since *where* is everywhere in the invariant. Can this be made simpler?

The answer is positive provided we understand a technique to be adopted, called *abstract interpretation*. So we postpone the topic of this paragraph to section 7.5, where abstract interpretation will be introduced. In between, we shall study a number of rules that can be used to address contracts in a structured way.

***Exercise** 7.3. Consider the voting system described by the relations of the diagram below,*



where electors can vote in political parties or nominally in members of such parties. In detail: (a) p c denotes the party of candidate c; (b) dC c denotes the district of candidate c; (c) dE e denotes the district of elector e; (d) d Di p records that party p has a list of candidates in district d; (e) e V p indicates that elector e voted in party p; (f) e V′ c indicates that elector e voted nominally in candidate c.

There are several invariants to take into account in this model, namely:

$$inv1\ (V, V') = V : E \leftarrow P \text{ and } V' : E \leftarrow C \text{ are injective} \tag{7.13}$$

$$inv2\ (V, V') = V^\circ \cdot V' = \bot \tag{7.14}$$

since an elector cannot vote in more than one candidate or party;

$$inv3\ (V, V') \quad = \quad dE \cdot [V, V'] \ \subseteq \ [Di, dC] \tag{7.15}$$

since each elector is registered in one district and can only vote in candidates of that district.

When the elections take place, relations p, dC, dE and Di are static, since all lists and candidates are fixed before people can vote. Once it is over, the scrutinity of the votes is carried out function

$$batch\ (V, V', X) = \ldots$$

where $X : E \to (P + C)$ is a batch of votes to be loaded into the system.

Complete the definition of batch and discharge the proof obligations of the contacts that this function must satisfy.

□

7.4 A CALCULUS OF FUNCTIONAL CONTRACTS

The number and complexity of invariants in real life problems invites us to develop *divide & conquer* rules alleviating the proof obligations that have to be discharged wherever contracts are needed. All such rules have definition (7.2) as starting point. Let us see, for instance, what happens wherever the input predicate in (7.3) is a disjunction:

$$\begin{array}{ll}
 & \Phi_q \xleftarrow{f} \Phi_{p_1} \cup \Phi_{p_2} \\
\equiv & \quad \{\ (7.2)\ \} \\
 & f \cdot (\Phi_{p_1} \cup \Phi_{p_2}) \subseteq \Phi_q \cdot f \\
\equiv & \quad \{\ \text{distribution of } (f\cdot) \text{ by } \cup\ (5.60)\ \} \\
 & f \cdot \Phi_{p_1} \ \cup \ f \cdot \Phi_{p_2} \ \subseteq \ \Phi_q \cdot f \\
\equiv & \quad \{\ \cup\text{-universal (5.59)}\ \} \\
 & f \cdot \Phi_{p_1} \subseteq \Phi_q \cdot f \ \wedge \ f \cdot \Phi_{p_2} \subseteq \Phi_q \cdot f \\
\equiv & \quad \{\ (7.2) \text{ twice}\ \} \\
 & \Phi_q \xleftarrow{f} \Phi_{p_1} \ \wedge \ \Phi_q \xleftarrow{f} \Phi_{p_2}
\end{array}$$

Recall that the disjunction $p \vee q$ of two predicates is such that $\Phi_{p \vee q} = \Phi_p \cup \Phi_q$ holds. Se we can write the result above in the simpler notation (7.3) as the contract decomposition rule:

$$q \xleftarrow{f} p \vee r \quad \equiv \quad q \xleftarrow{f} p \ \wedge \ q \xleftarrow{f} r \tag{7.16}$$

The dual rule,

$$\Phi_q \cdot \Phi_r \xleftarrow{f} \Phi_p \quad \equiv \quad \Phi_q \xleftarrow{f} \Phi_p \ \wedge \ \Phi_{q_2} \xleftarrow{f} \Phi_p$$

is calculated in the same way and written

$$q \wedge r \xleftarrow{f} p \quad \equiv \quad q \xleftarrow{f} p \ \wedge \ r \xleftarrow{f} p \tag{7.17}$$

in the same notation, since $\Phi_{p \wedge q} = \Phi_p \cap \Phi_q$. The fact that contracts compose sequentially (exercise 7.1) enables the corresponding decomposition, once a suitable middle predicate r is found:

$$q \xleftarrow{g \cdot h} p \quad \Leftarrow \quad q \xleftarrow{g} r \ \wedge \ r \xleftarrow{h} p \tag{7.18}$$

This follows straight from (7.3, 7.2), as does the obvious rule concerning identity

$$q \xleftarrow{id} p \quad \equiv \quad q \Leftarrow p \tag{7.19}$$

since $p \Rightarrow q \ \Leftrightarrow \ \Phi_p \ \subseteq \ \Phi_q$. The expected

$$p \xleftarrow{id} p$$

immediately follows from (7.19).

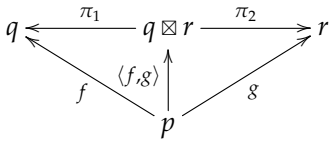
Now suppose that we have contracts $q \xleftarrow{f} p$ and $r \xleftarrow{g} p$. What kind of contract can we infer for $\langle f, g \rangle$? We calculate:

$$\begin{array}{ll}
 & \Phi_q \xleftarrow{f} \Phi_p \quad \wedge \quad \Phi_r \xleftarrow{g} \Phi_p \\
\equiv & \quad \{ \text{ (7.3,7.2) twice } \} \\
 & f \cdot \Phi_p \subseteq \Phi_q \cdot f \ \ \wedge \ \ g \cdot \Phi_p \subseteq \Phi_r \cdot g \\
\equiv & \quad \{ \text{ cancellations (2.22) } \} \\
 & \pi_1 \cdot \langle f, g \rangle \cdot \Phi_p \subseteq \Phi_q \cdot f \ \ \wedge \ \ \pi_2 \cdot \langle f, g \rangle \cdot \Phi_p \subseteq \Phi_r \cdot g \\
\equiv & \quad \{ \text{ universal property (5.103) } \} \\
 & \langle f, g \rangle \cdot \Phi_p \subseteq \langle \Phi_q \cdot f, \Phi_r \cdot g \rangle \\
\equiv & \quad \{ \text{ absorption (5.106) } \} \\
 & \langle f, g \rangle \cdot \Phi_p \subseteq (\Phi_q \times \Phi_r) \cdot \langle f, g \rangle \\
\equiv & \quad \{ \text{ (7.3,7.2) } \} \\
 & \Phi_q \times \Phi_r \xleftarrow{\langle f, g \rangle} \Phi_p
\end{array}$$

Defining $p \boxtimes q$ such that $\Phi_{p \boxtimes q} = \Phi_p \times \Phi_q$ we obtain the contract decomposition rule:

$$q \boxtimes r \xleftarrow{\langle f,g \rangle} p \quad \equiv \quad q \xleftarrow{f} p \ \wedge \ r \xleftarrow{g} p \tag{7.20}$$

which justifies the existence of arrow $\langle f, g \rangle$ in the diagram



(7.21)

where predicates (coreflexives) are promoted to objects (nodes in diagrams).

***Exercise** 7.4. Check the contracts $q \xleftarrow{\pi_1} q \boxtimes r$ and $q \boxtimes r \xrightarrow{\pi_2} r$ of diagram (7.21).*
□

Let us finally see how to handle conditional expressions of the form $if\ (c\ x)\ then\ (f\ x)\ else\ (g\ x)$ which, by (5.213), transform into

$$c \to f\ ,\ g \ = \ f \cdot \Phi_c \cup g \cdot \Phi_{\neg c} \tag{7.22}$$

In this case, (7.4) offers a better standpoint for calculation than (7.2), as the reader may check in calculating the following rule for conditionals:

$$\Phi_q \xleftarrow{c \to f\ ,\ g} \Phi_p \quad \equiv \quad \begin{cases} \Phi_q \xleftarrow{f} \Phi_p \cdot \Phi_c \\ \Phi_q \xleftarrow{g} \Phi_p \cdot \Phi_{\neg\, c} \end{cases} \tag{7.23}$$

This is because it is hard to handle $c \to f\ ,\ g$ on the upper side, $\top$ being more convenient.

Further contract rules can calculated on the same basis, either elaborating on the predicate structure or on the combinator structure. However, all the cases above involve functions only and the semantics of computations are, in general, relations. So our strategy is to generalize definition (7.2) from functions to arbitrary relations.

RELATIONAL CONTRACTS Note that $S = R \cdot \Phi_p$ means

$$b\ S\ a \ \Leftrightarrow \ p\ a \wedge b\ R\ a$$

—- that is, S is R pre-conditioned by p. Dually, $\Phi_q \cdot R$ is the largest part of R which yields outputs satisfying q — R post-conditioned by q. By writing

$$R \cdot \Phi_p \ \subseteq \ \Phi_q \cdot R \tag{7.24}$$

— which is equivalent to

$$R \cdot \Phi_p \ \subseteq \ \Phi_q \cdot \top \tag{7.25}$$

by (5.207) and even equivalent to

$$\Phi_p \ \subseteq \ R \setminus (\Phi_q \cdot \top) \tag{7.26}$$

by (5.159) — we express a very important fact about R regarded as a (possibly non-deterministic, undefined) program R: condition p on the inputs is *sufficient* for condition q to hold on the outputs:

$$\langle \forall\, a \ :\ p\, a :\ \langle \forall\, b \ :\ b\, R\, a :\ q\, b \rangle \rangle$$

Thus we generalize functional contracts (7.2) to arbitrary relations,

$$p \xrightarrow{R} q \quad \equiv \quad R \cdot \Phi_p \ \subseteq \ \Phi_q \cdot R \tag{7.27}$$

a definition equivalent to

$$p \xrightarrow{R} q \quad \equiv \quad R \cdot \Phi_p \ \subseteq \ \Phi_q \cdot \top \tag{7.28}$$

as seen above.

7.5 ABSTRACT INTERPRETATION

The proofs involved in verifying contracts may be hard to perform due to the intricacies of real-life sized software specifications, which may involve hundreds of invariants of arbitrary complexity. Such situations can only be tackled with the support of a theorem prover, and in many situations even this is not enough to accomplish the task. This problem has made software theorists to think of strategies helping designers to simplify their proofs. One such strategy is *abstract interpretation*.

It is often the case that the proof of a given contract does not require the whole model because the contract is only concerned with a particular *view* of the whole thing. As a very simple example, think of a model that is made of two independent parts $A \times B$ and of an invariant that constrains part A only. Then one may safely ignore B in the proofs. This is equivalent to applying projection $\pi_1 : A \times B \to A$ (2.21) to the original model. Note that π_1 is an *abstraction*, since it is a surjective function (recall figure 5.3).

In general, software models are not as "separable" as $A \times B$ is, but abstraction functions exist that yield much simpler models where the proofs can be made easier. Different abstractions help in different proofs — a kind of "on demand" *abstraction* making a model more *abstract* with respect to the *specific* property one wishes to check. In general, techniques of this kind are known as *abstract interpretation* techniques and play a major role in *program analysis*, for instance. To explain abstract interpretation we need to introduce the notion of a *relational type*.

RELATIONS AS TYPES A function h is said to have *relation type* $R \to S$, written $R \xrightarrow{h} S$ if

$$h \cdot R \;\subseteq\; S \cdot h \qquad \begin{array}{ccc} B & \xleftarrow{R} & B \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle h} \\ A & \xleftarrow{S} & A \end{array} \tag{7.29}$$

holds. Note that (7.29) could be written $h\ (S \leftarrow R)\ h$ in the notation of (6.10). In case $h : B \to A$ is surjective, i.e. h is an *abstraction function*, we also say that $A \xleftarrow{S} A$ is an *abstract simulation* of $B \xleftarrow{R} B$ through h.

A special case of relational type defines so-called *invariant functions*. A function of relation type $R \xrightarrow{h} id$ is said to be *R-invariant*, in the sense that

$$\langle \forall\, b, a \;:\; b\ R\ a :\; h\ b = h\ a \rangle \tag{7.30}$$

holds. When h is R-invariant, observations by h are not affected by R-transitions. In pointfree notation, an R-invariant function h is always such that:

$$R \;\subseteq\; \frac{h}{h} \tag{7.31}$$

For instance, a binary operation θ is *commutative* iff θ is swap-invariant, that is

$$\mathsf{swap} \;\subseteq\; \frac{\theta}{\theta} \tag{7.32}$$

holds.

Exercise *7.5. What does (7.29) mean in case R and S are partial orders?*
□

Exercise *7.6. Show that relational types compose, that is* $Q \xleftarrow{k} S$ *and* $S \xleftarrow{h} R$ *entail* $Q \xleftarrow{k \cdot h} R$ *.*
□

Exercise *7.7. Sow that an alternative way of stating (7.27) is*

$$p \xrightarrow{R} q \quad \equiv \quad R \cdot \Phi_p \;\subseteq\; \Phi_q \cdot \top \tag{7.33}$$

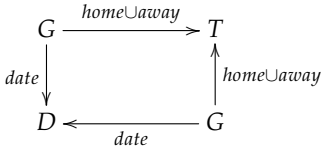
□

***Exercise** 7.8. Recalling exercise 5.12, let the following relation specify that two dates are at least one week apart in time:*

$$d \; Ok \; d' \;\Leftrightarrow\; |\, d - d' \,| > 1 \; week$$

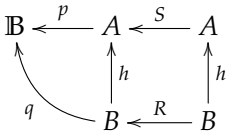
Looking at the type diagram below, say in your own words the meaning of the invariant specified by the relational type (7.29) statement below, on the left:

$$\ker\,(home \cup away) - id \xrightarrow{date} Ok$$



□

ABSTRACT INTERPRETATION Suppose that one wishes to show that $q : B \to \mathbb{B}$ is an invariant of some operation $B \xrightarrow{R} B$, i.e. that $q \xrightarrow{R} q$ holds and you know that $q = p \cdot h$, for some $h : B \to A$, as shown in the diagram. Then one can factor the proof in two steps:



- show that there is an abstract *simulation* S such that $R \xrightarrow{h} S$;
- prove $p \xrightarrow{S} p$, that is, that p is an (abstract) *invariant* of (abstract) S.

This strategy is captured by the following calculation:

$$\begin{array}{ll}
 & R \cdot \Phi_q \;\subseteq\; \Phi_q \cdot \top \\
\equiv & \quad \{\; q = p \cdot h \;\} \\
 & R \cdot \Phi_{(p \cdot h)} \;\subseteq\; \Phi_{(p \cdot h)} \cdot \top \\
\equiv & \quad \{\; (5.205) \text{ etc} \;\} \\
 & R \cdot \Phi_{(p \cdot h)} \;\subseteq\; h^\circ \cdot \Phi_p \cdot \top \\
\equiv & \quad \{\; \text{shunting} \;\} \\
 & h \cdot R \cdot \Phi_{(p \cdot h)} \;\subseteq\; \Phi_p \cdot \top \\
\Leftarrow & \quad \{\; R \xrightarrow{h} S \;\} \\
 & S \cdot h \cdot \Phi_{(p \cdot h)} \;\subseteq\; \Phi_p \cdot \top \\
\Leftarrow & \quad \{\; \Phi_{(p \cdot h)} \;\subseteq\; h^\circ \cdot \Phi_p \cdot h \;(5.210) \;\} \\
 & S \cdot h \cdot h^\circ \cdot \Phi_p \cdot h \;\subseteq\; \Phi_p \cdot \top
\end{array}$$

$$\Leftarrow \qquad \{\ \top = \top \cdot h \ (\mathsf{cancel}\ h);\ \mathsf{img}\ h \ \subseteq\ id\ \}$$
$$S \cdot \Phi_p \ \subseteq\ \Phi_p \cdot \top$$
$$\Box$$

Abstract interpretation techniques usually assume that h is an adjoint of a Galois connection. Our first examples below do not assume this, for an easy start.

7.6 SAFETY AND LIVENESS PROPERTIES

Before showing examples of abstract interpretation, let us be more specific about what was meant by "some operation $B \xrightarrow{R} B$ " above. In section 4.9 a monad was studied called the *state monad*. This monad is inhabited by state-transitions encoding state-based automata known as *Mealy machines*.

With relations one may be more relaxed on how to characterize state automata. In general, functional models generalize to so called *state-based* relational models in which there is

- a set Σ of *states*
- a subset $I \ \subseteq\ \Sigma$ of *initial* states
- a *step* relation $\Sigma \xrightarrow{R} \Sigma$ which expresses transition of states.

We define:

- $R^0 = id$ — no action or transition takes place
- $R^{i+1} = R \cdot R^i$ — all "paths" made of $i+1$ R-transitions
- $R^* = \bigcup_{i \geqslant 0} R^i$ — the set of all possible R-paths.

We represent the set I of initial states by the coreflexive $\Sigma \xrightarrow{\Phi_{(\in I)}} \Sigma$, simplified to $\Sigma \xrightarrow{I} \Sigma$ to avoid symbol cluttering.

Given $\Sigma \xrightarrow{R,I} \Sigma$ (i.e. a nondeterministic automaton, model) there are two kinds of property that one may wish to prove — *safety* and *liveness* properties. *Safety* properties are of the form $R^* \cdot I \ \subseteq\ S$, that is,

$$\langle \forall\ n\ :\ n \geqslant 0 :\ R^n \cdot I \ \subseteq\ S \rangle \tag{7.34}$$

for some safety relation $S : \Sigma \ \to\ \Sigma$, meaning: *All paths in the model originating from its initial states are* bounded *by S*. In the particular case $S = \frac{true}{p}$ [8]

$$\langle \forall\ n\ :\ n \geqslant 0 :\ R^n \cdot I \ \subseteq\ \frac{true}{p} \rangle \tag{7.35}$$

8 Recall that $\frac{true}{p} = \Phi_p \cdot \top$ (5.205).

meaning that formula p holds for every state reachable by R from an initial state. Invariant preservation is an example of a safety property: if starting from a "good" state, the automaton only visits "good" (valid) states.

In contrast to safety properties, the so-called *liveness* properties are of the form

$$\langle \exists\, n \;:\; n \geqslant 0 :\; Q \;\subseteq\; R^n \cdot I \rangle \tag{7.36}$$

for some *target* relation $Q : \Sigma \;\rightarrow\; \Sigma$, meaning: *the target relation Q is eventually* realizable, *after n steps starting from an initial state.* In the particular case $Q = \frac{true}{p}$ we have

$$\langle \exists\, n \;:\; n \geqslant 0 :\; \frac{true}{p} \;\subseteq\; R^n \cdot I \rangle \tag{7.37}$$

meaning that, for a sufficiently large n, formula p will eventually hold.

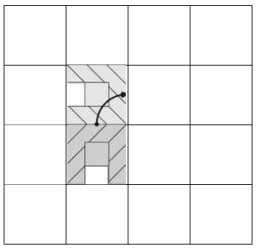
7.7 EXAMPLES

The Alcuin puzzle is an example of a problem that is characterized by a liveness and safety property:

- From initial state $where = \underline{Left}$, state $where = \underline{Right}$ is eventually reachable — a *liveness* property.
- Initial state $where = \underline{Left}$ is valid and no step of the automaton leads to invalid *where* states — a *safety* property.

The first difficulty in ensuring properties such as (7.35) e (7.37) is the quantification on the number of path steps. In the case of (7.37) one can try and find a particular path using a *model checker*. In both cases, the complexity /size of the *state space* may offer some impedance to proving / model checking. Below we show how to circumvent such difficulties by use of *abstract interpretation*.

THE HEAVY ARMCHAIR PROBLEM Let us show a simple, but effective example of abstract interpretation applied to a well-known problem — the *heavy armchair* problem.[9] Consider the following picture:



9 Credits: this version of the problem and the pictures shown are taken from [6].

We wish to move the armchair to an adjacent square, horizontally or vertically. However, because the armchair is too heavy, it can only be rotated over one of its four legs, as shown in the picture.

The standard model for this problem is a pair (p, o) where $p = (y, x)$ captures the square where the armchair is positioned and o is one of the complex numbers $\{i, -i, 1, -1\}$ indicating the orientation of the armchair (that is, it can face N,S,E,W). Let the following the step-relation be proposed,

$$R = P \times Q$$

where P captures the *adjacency* of two squares and Q captures 90° rotations. A *rotation* multiplies an orientation o by $\pm\ i$, depending on choosing a clockwise ($-i$) or anti-clockwise (i) rotation. Altogether:

$$\begin{array}{l} ((y', x'), d')\ R\ ((y, x), d)\ \Leftrightarrow \\ \begin{cases} y' = y\ \pm\ 1 \wedge x' = x \vee y' = y \wedge x' = x\ \pm\ 1 \\ d' = (\pm\ i)\ d \end{cases} \end{array}$$

We want to check the *liveness* property:

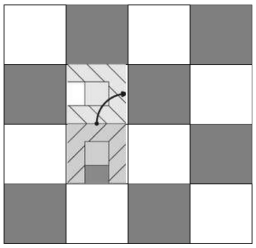
$$\textit{For some } n,\ ((y, x+1), d)\ R^n\ ((y, x), d)\ \textit{holds.} \tag{7.38}$$

That is, we wish to move the armchair to the adjacent square on its right, keeping the armchair's orientation. This is exactly what the pointfree version of (7.38) tells:

$$\langle \exists\ n\ ::\ (id \times (1+)) \times id\ \subseteq\ R^n \rangle$$

In other words: *there is a path with n steps that realizes the* function $move = (id \times (1+)) \times id$.

Note that the state of this problem is arbitrarily large. (The squared area is unbounded.) Moreover, the specification of the problem is non-deterministic. (For each state, there are four possible successor states.) We resort to *abstract interpretation* to obtain a bounded, deterministic (*functional*) model: the floor is coloured as a chess board and the armchair behaviour is abstracted by function $h = col \times dir$ which tells the *colour* of the square where the armchair is and the *direction* of its current orientation:



Since there are two colours (black, white) and two directions (horizontal, vertical), both can be modelled by Booleans. Then the action of

moving to any adjacent square abstracts to *color* negation and any 90° rotation abstracts to *direction* negation:

$$P \xrightarrow{col} (\neg) \tag{7.39}$$

$$Q \xrightarrow{dir} (\neg) \tag{7.40}$$

In detail:

$$\begin{aligned} &col\ (y,x) = even\ (y+x) \\ &dir\ x = x \in \{1,-1\} \end{aligned}$$

For instance, $col\ (0,0) = \mathsf{True}$ (black in the picture), $col\ (1,1) = \mathsf{True}$, $col\ (1,2) = \mathsf{False}$ and so on; $dir\ 1 = \mathsf{True}$ (horizontal orientation), $dir\ (-i) = \mathsf{False}$, and so on. Checking (7.40):

$$\begin{aligned} & dir\ ((\pm\, i)\ x) \\ = & \quad \{\ dir\ x = x \in \{1,-1\}\ \} \\ & (\pm\, i)\ x \in \{1,-1\} \\ = & \quad \{\ multiply\ by\ (\pm\, i)\ \text{within}\ \{1,i,-1,-i\}\ \} \\ & x \in \{-i,i\} \\ = & \quad \{\ \text{the remainder of}\ \{-i,i\}\ \text{is}\ \{1,-1\}\ \} \\ & \neg\ (x \in \{1,-1\}) \\ = & \quad \{\ dir\ x = x \in \{1,-1\}\ \} \\ & \neg\ (dir\ x) \end{aligned}$$

□

Checking (7.39):

$$\begin{aligned} & (\neg) \xleftarrow{col} P \\ \equiv & \quad \{\ (7.29)\ \text{for functions}\ \} \\ & col \cdot P \ \subseteq\ \neg \cdot col \\ \equiv & \quad \{\ \text{shunting ; go pointwise}\ \} \\ & (y',x')\ P\ (y,x) \Rightarrow even\ (y'+x') = \neg\ even\ (y+x) \\ \equiv & \quad \{\ \text{unfold}\ \} \\ & \left\{ \begin{array}{l} y' = y\ \pm\ 1 \wedge x' = x \Rightarrow even\ (y'+x') = \neg\ even\ (y+x) \\ y' = y \wedge x' = x\ \pm\ 1 \Rightarrow even\ (y'+x') = \neg\ even\ (y+x) \end{array} \right. \\ \equiv & \quad \{\ \text{substitutions ; trivia}\ \} \\ & \left\{ \begin{array}{l} even\ (y\ \pm\ 1) = \neg\ even\ y \\ even\ (x\ \pm\ 1) = \neg\ even\ x \end{array} \right. \\ \equiv & \quad \{\ \text{trivia}\ \} \\ & true \end{aligned}$$

□

Altogether:

$$R \xrightarrow{col \times dir} (\neg \times \neg)$$

That is, step *relation* R is simulated by $s = \neg \times \neg$, i.e. the *function*

$$s\ (c, d) = (\neg\ c, \neg\ d)$$

over a state space with 4 possibilities only: wherever the armchair turns over one of its legs, whatever this is, it changes *both* the colour of the square where it is, and its direction.

At this level, we note that *observation* function

$$f\ (c, d) = c\ \oplus\ d \tag{7.41}$$

is *s-invariant* (7.30), that is

$$f \cdot s = f \tag{7.42}$$

since $\neg\ c\ \oplus\ \neg\ d = c\ \oplus\ d$ holds. By induction on n, $f \cdot s^n = f$ holds too.

Expressed under this abstraction, (7.38) is rephrased into: *there is a number of steps n such that* $s^n\ (c, d) = (\neg\ c, d)$ *holds*. Let us check this abstract version of the original property, assuming variable n existentially quantified:

$$\begin{array}{ll}
 & s^n\ (c, d) = (\neg\ c, d) \\
\Rightarrow & \qquad \{\ \text{Leibniz}\ \} \\
 & f\ (s^n\ (c, d)) = f\ (\neg\ c, d) \\
\equiv & \qquad \{\ f \text{ is } s\text{-invariant}\ \} \\
 & f\ (c, d) = f\ (\neg\ c, d) \\
\equiv & \qquad \{\ (7.41)\ \} \\
 & c\ \oplus\ d = \neg\ c\ \oplus\ d \\
\equiv & \qquad \{\ 1\ \oplus\ d = \neg\ d \text{ and } 0\ \oplus\ d = d\ \} \\
 & d = \neg\ d \\
\equiv & \qquad \{\ \text{trivia}\ \} \\
 & \mathit{false}
\end{array}$$

Thus, for all paths of arbitrary length n, $s^n\ (c, d) \neq (\neg\ c, d)$. We conclude that the proposed liveness property does not at all hold!

ALCUIN PUZZLE EXAMPLE Abstract interpretation applies nicely to this problem, thanks to its symmetries. On the one hand, one does not need to work over the 16 functions in $Bank^{Being}$, since starting from

the left margin or from the right margin is irrelevant. Another symmetry can be found in type *Being*, suggesting the following abstraction of beings into three classes:

$$f : Being \to \{\alpha, \beta, \gamma\}$$

$$f = \begin{pmatrix} Goose & \longrightarrow & \alpha \\ Fox & \longrightarrow & \beta \\ Beans & \nearrow & \\ Farmer & \longrightarrow & \gamma \end{pmatrix}$$

The abstraction consists in unifying , the maximum and minimum elements of the "food chain". In fact, the simultanous presence of one α and one β is enough for defining the invariant — which *specific* being eats the other is irrelevant detail. This double abstraction is captured by

$$\begin{array}{ccc} Bank & \xleftarrow{w} & Being \\ {\scriptstyle \underline{Left}}\uparrow & & \downarrow{\scriptstyle f} \\ 1 & \xleftarrow[V]{} & \{\alpha, \beta, \gamma\} \end{array} \qquad V = \underline{Left}^{\circ} \cdot w \cdot f^{\circ}$$

where the choice of $Left$ as reference bank is arbitrary. Thus function w is abstracted by the row *vector* relation V [10] such that:

$$\_\ V\ x = \langle \exists\ b\ :\ x = f\ b :\ w\ b = Left \rangle$$

Vector V tells whether at least one being of class x can be found in the reference bank. Noting that there could be more than one β there, we refine the abstraction a bit so that the number of beings of each class is counted.[11] This leads to the following *state-abstraction* (higher order) function h based on f:

$$h : (Being \to Bank) \to \{\alpha, \beta, \gamma\} \to \{0, 1, 2\}$$
$$h\ w\ x = \langle \sum b\ :\ x = f\ b \wedge w\ b = Left :\ 1 \rangle$$

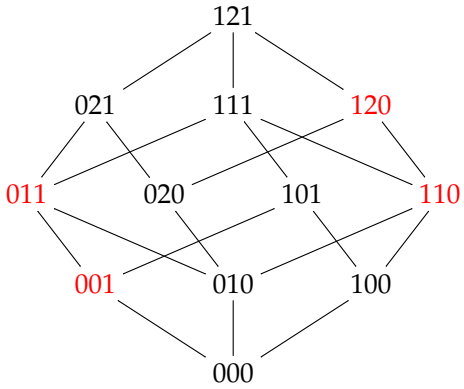
For instance,

$$\begin{array}{rcl} h\ \underline{Left} & = & 121 \\ h\ \underline{Right} & = & 000 \end{array}$$

abbreviating by vector xyz the mapping $\{\alpha \mapsto x, \beta \mapsto y, \gamma \mapsto z\}$.[12] To obtain the other bank just compute: $\overline{x} = 121 - x$. Note that there are

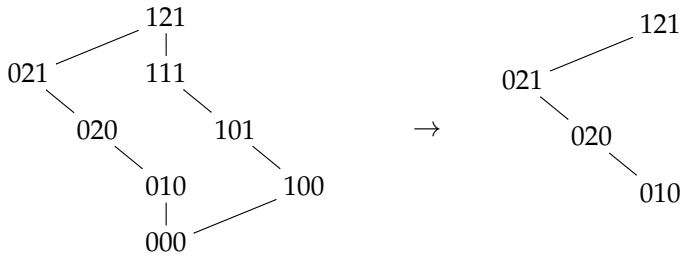
10 A fragment of $! : \{\alpha, \beta, \gamma\} \to 1$, recall section 5.5.
11 This suggests that linear algebra would be a good alternative to relation algebra here!
12 This version of the model is inspired in [6].

$2 \times 3 \times 2 = 12$ possible state vectors, 4 of which are invalid (these are marked in red):

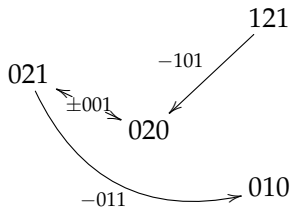


The ordering implicit in the lattice above is pointwise ($\leqslant$). This is complemented by $\bar{x} = 121 - x$, which gives the information of the other bank.

The 8 valid states can be further abstracted to only 4 of them,



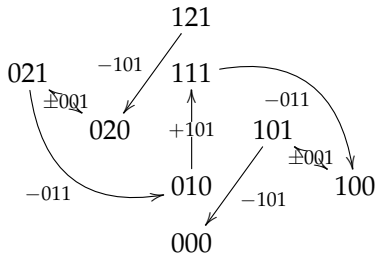
since, due to complementation (cf. the Left-Right margin symmetry), we only need to reach state 010. Then we reverse the path through the complements. In this setting, the automaton is deterministic, captured by the abstract automaton:



Termination is ensured by disabling toggling between states 021 and 020:

$$\begin{array}{r} 121 \\ -101 \\ \hline 020 \\ +001 \\ \hline 021 \\ -011 \\ \hline 010 \end{array}$$

We then take the complemented path $111 \to 100 \to 101 \to 000$. So the abstract solution for the Alcuin puzzle is, finally:



$$\begin{array}{r}
121 \\
-101 \\
\hline
020 \\
+001 \\
\hline
021 \\
-011 \\
\hline
010 \\
+101 \\
\hline
111 \\
-011 \\
\hline
100 \\
+001 \\
\hline
101 \\
-101 \\
\hline
000
\end{array}$$

At this point note that, according to the principles of abstract interpretation stated above, quite a few steps are pending in this exercise: abstract the *starving* invariant to the vector level, find an abstract simulation of *carry*, and so on and so forth. But — why bother doing all that? There no other operation in the problem, so the abstraction found is, in a sense, universal: we should have started from the vector model and not from the *Being* $\to$ *Bank* model, which is not *sufficiently* abstract.

The current scientific basis of programming enables the calculation of programs, following the scientific method. So, programming is lesser and lesser an *art*. Where is creativity gone to? To the *art* of abstract modelling and elegant proving — this is where it can be found nowadays.

Exercise *7.9. Verification of code involves calculations of real numbers and is often done on the basis of an abstract interpretation called sign analysis:*

$$\begin{array}{l}
sign : \mathbb{R} \to \{-, 0, +\} \\
sign\ 0 = 0 \\
sign\ x = \textbf{if}\ x > 0\ \textbf{then}\ +\ \textbf{else}\ -
\end{array}$$

Suppose there is evidence that the operation $\theta : \{-, 0, +\}^2 \to \{-, 0, +\}$ *defined by*

$$\begin{array}{c|ccc}
\theta & - & 0 & + \\
\hline
- & + & 0 & - \\
0 & 0 & 0 & 0 \\
+ & - & 0 & +
\end{array} \tag{7.43}$$

is the abstract simulation induced by sign of a given concrete operation $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$*, that is, that*

$$\theta \cdot (sign \times sign) = sign \cdot f \tag{7.44}$$

holds. It is easy to see, by inspection of (7.43), that θ is a commutative operation, recalling (7.32).

- *Show that sign $\cdot f$ is necessarily commutative as well. (Hint: the free theorem of* swap *can be useful here.)*
- *Does the previous question guarantee that the specific operation f is also commutative? Answer informally.*

□

7.8 "FREE CONTRACTS"

In design by contract, many functional *contracts* arise naturally as corollaries of *free theorems*. This has the advantage of saving us from proving such contracts explicitly.

The following exercises provide ample evidence of this.

***Exercise** 7.10. The type of functional composition $(\cdot)$ is*

$$(b \to c) \to (a \to b) \to a \to c$$

Show that contract composition *(7.18) is a corollary of the free theorem (FT) of this type.*

□

***Exercise** 7.11. Show that contract $q^\star \xleftarrow{\mathsf{map}\, f} p^\star$ holds provided contract $q \xleftarrow{f} p$ holds.*

□

***Exercise** 7.12. Suppose a functional programmer wishes to prove the following property of lists:*

$$\langle \forall\, a, s \;:\; (p\ a) \wedge \langle \forall\, a' \;:\; a' \in \mathsf{elems}\ s :\ p\ a' \rangle :\ \langle \forall\, a'' \;:\; a'' \in \mathsf{elems}\ (a : s) :\ p\ a'' \rangle \rangle$$

Show that this property is a contract arising (for free) from the polymorphic type of the cons *operation $(:)$ on lists.*

□

7.9 REASONING BY APPROXIMATION

Currently in preparation

7.10 BIBLIOGRAPHY NOTES

To be completed