Formal Aspects of Computing (2000) 12: 313–349

Formal Aspects of Computing

Encoding, Decoding and Data Refinement

Ralph-Johan Back and Joakim von Wright

Åbo Akademi University and Turku Centre for Computer Science, Turku, Finland

Abstract. Data refinement is the systematic replacement of a data structure with another one in program development. Data refinement between program statements can on an abstract level be described as a commutativity property where the abstraction relationship between the data structures involved is represented by an abstract statement (a *decoding*). We generalise the traditional notion of data refinement by defining an *encoding operator* that describes the least (most abstract) data refinement with respect to a given abstraction. We investigate the categorical and algebraic properties of encoding and describe a number of special cases, which include traditional notions of data refinement. The dual operator of encoding is *decoding*, which we investigate and give an intuitive interpretation to. Finally we show a number of applications of encoding and decoding.

Keywords: Abstraction; Data refinement; Galois connection; Predicate transformer semantics

1. Introduction

Data refinement is the systematic replacement of a data structure (abstract data) by another one (concrete data) in program development. Traditionally, data refinement has been formalised in terms of a *data refinement relation* which generalises the (algorithmic) refinement relation between programs. In this paper we consider data refinement as an operator rather than a relation. The *encoding* operator $\downarrow$ is defined so that $S \downarrow D$ is the most general (least refined) data refinement of statement S with respect to an *abstraction statement* D (an abstraction statement models the relationship between the concrete and the abstract state space, in a more general way than is possible with relations). Traditional notions of data refinement (forward, functional and backward) are special cases of encoding, where the abstraction statement satisfies some additional conditions. Thus, traditional (syntactic) rules for data refinement of programs can be derived from the general algebraic properties of the encoding operator.

We work within the *refinement calculus* framework described in [BaW98]. This is a formalisation using classical strongly typed higher-order logic of the traditional refinement calculus [Bac80, Bac88, Mor88] which was based on the weakest precondition semantics of programs [Dij76]. The basis of the refinement calculus is the *predicate transformer hierarchy*, where programs are modelled as predicate transformers, which in turn are built from predicates, functions and relations using homomorphic operators (statement constructors). We add the encoding operator to this hierarchy and investigate its algebraic properties and its use in calculational

Correspondence and offprint requests to: Joakim von Wright, Åbo Akademi University, Department of Computer Science, FIN-20520 Turku, Finland. Email: Jockum.Wright@.abo.fi

data refinement. Using Galois connections we find that under certain restrictions there exists a dual *decoding* operator $\uparrow$ which allows us to calculate the least general (most refined) abstraction $S \uparrow D$ of a given (concrete) statement S with respect to abstraction statement D. We also investigate the properties of this operator and illustrate its use.

Data refinement has been the subject of many detailed studies, but mostly in a relational framework (for an overview of different methods and theories of data refinement, see [dRE98]). The relational framework is less expressive than predicate transformers and does not allow a uniform handling of data refinement in the form described here. The idea of describing data refinement in terms of general abstraction statements is not new [Bac80, Bac89, BaW89, GaM91], but our notion of least data refinement (encoding) and its dual have to our knowledge not been investigated in a predicate transformer framework before. Our encoding operator can also be seen as a generalisation of the *calculational* approach to data refinement, where data refinement of a program statement S with respect to abstraction relation R (or, dually, an abstraction of a concrete program) is expressed in explicit form [Bac80, Bac89, Mor89, MoG90].

We use higher-order logic as the underlying logic of our investigation, with lambda notation for functions (e.g., $(\lambda x \cdot x = 1)$) and an infix dot for function application $(f.x)$. Quantifiers are given low precedence and their scope is delimited by parentheses. We use standard notation for logical connectives and F, T for Boolean falsity and truth. Proofs are written in a calculational style, with indented subderivations [BaW98, BGW97]. Most proofs are placed in Appendix B, but a few more important proofs are included in the main text.

The paper is organised as follows. Section 2 gives a short overview of the basic concepts related to the predicate transformer hierarchy and its use in the refinement calculus, and some other background material. Then, in Section 3, we introduce the notion of encoding as part of the data refinement method, and study the algebraic properties of encoding. This study makes only weak assumptions about the abstraction statement D (assuming only monotonicity). In Section 4 we study the properties of encoding when we consider the kinds of abstraction statements that would usually be used in program derivations. In particular, we study encoding in connection with forward and backward data refinement. We show that encodings in such cases can be calculated explicitly. In Section 5 we consider the dual decoding operation, study its properties, and show how it is related to encoding. In Section 6, we look at encoding and decoding of recursive statements in more detail. Finally, in Section 7, we give some examples of how the algebraic results derived in the previous sections can be translated into more concrete properties about program statements expressed in ordinary programming language syntax.

2. Background

We begin with a brief overview of the basic concepts related to the predicate transformer hierarchy.

2.1. States, Predicates and Relations

A *state space* can be any type Σ. In general, we do not assume that this type has any specific internal structure. An element $\sigma : \Sigma$ is called a *state*. A *state function* $f : \Sigma \to \Gamma$ maps states to states (note that the two state spaces can be different). Function composition is written forward (so $(f;g).\sigma = g.(f.\sigma)$) and the unit of composition is the identity function id.

A *state predicate* is a Boolean state function $p : \Sigma \to \mathsf{Bool}$. Since p can be identified with the set of states σ where $p.\sigma \equiv \mathsf{T}$ (i.e., where p holds), we use set notation for meet (intersection) and join (union) of predicates. The unit elements of meet and join are the everywhere true predicate true and the everywhere false predicate false, respectively. The complement (negation) of predicate p is $\neg p$. Predicates are ordered by the subset (stronger-than) ordering $\subseteq$.

A *state transformer* is a function $f : \Sigma \to \Sigma$ that changes the state in a deterministic way. A *state relation* is a binary Boolean state function $R : \Sigma \to \Gamma \to \mathsf{Bool}$. We think of a relation as a *non-deterministic state transformer*, transforming an initial state σ to one of the states γ such that $R.\sigma.\gamma$ holds.

When a relation $R : \Sigma \to \Gamma \to \mathsf{Bool}$ is applied to a single state, the result $R.\sigma$ is a set of states, i.e., a predicate. Extending this partial application of a relation to a set of states, we have a notion of *image* and

inverse image of a predicate under relation:

$$\begin{array}{lcll} \mathsf{im}.R.p & \hat{=} & \{\gamma \mid \exists \sigma \in p \bullet \gamma \in R.\sigma\} & (\textit{image}) \\ \overline{\mathsf{im}}.R.q & \hat{=} & \{\sigma \mid \forall \gamma \in R.\sigma \bullet \gamma \in q\} & (\textit{inverse image}) \end{array}$$

Equivalently, we can write

$$\begin{array}{lcl} \mathsf{im}.R.p.\gamma & \equiv & (\exists \sigma \bullet p.\sigma \wedge R.\sigma.\gamma) \\ \overline{\mathsf{im}}.R.q.\sigma & \equiv & (\forall \gamma \bullet R.\sigma.\gamma \Rightarrow q.\gamma) \end{array}$$

which shows the duality between the two operators: $\mathsf{im}.R.p = \neg\overline{\mathsf{im}}.R^{-1}.(\neg p)$.

Meet (intersection) and join (union) of relations are defined by pointwise extension from predicates, e.g.,

$$(Q \cap R).\sigma \quad \hat{=} \quad Q.\sigma \cap R.\sigma$$

Subset ordering on relations is the extension of the subset ordering on predicates,

$$Q \subseteq R \quad \hat{=} \quad (\forall \sigma \bullet Q.\sigma \subseteq R.\sigma)$$

Composition of relations is also defined as usual:

$$(Q;R).\sigma.\delta \quad \hat{=} \quad (\exists \gamma \bullet Q.\sigma.\gamma \wedge R.\gamma.\delta) \qquad (\textit{composition})$$

The unit elements of composition, meet and join, are the *identity relation* Id, the everywhere true relation True and the everywhere false relation False, respectively.

In addition, we define a *quotient operator* for relations:

$$(Q\backslash R).\sigma.\delta \quad \hat{=} \quad (\forall \gamma \bullet Q.\sigma.\gamma \Rightarrow R.\gamma.\delta) \qquad (\textit{quotient})$$

The quotient operation $(Q\backslash R)$ is similar to the composition $Q;R$. We have that $(Q;R).\sigma.\delta$ holds when there exists some intermediate state γ reachable from σ by Q such that $R.\gamma.\delta$, while $(Q\backslash R).\sigma.\delta$ holds when for every intermediate state γ reachable from σ by Q, we have that $R.\gamma.\delta$ holds. The following *shunting properties* show how these two constructs are related:

$$\begin{array}{lcll} P;Q \subseteq R & \equiv & P^{-1} \subseteq R\backslash Q^{-1} & (\textit{shunt}) \\ P;Q \subseteq R & \equiv & Q \subseteq P^{-1}\backslash R & \end{array}$$

Predicates and functions can easily be coerced into relations, by the following definitions:

$$\begin{array}{lcll} |p|.\sigma.\gamma & \hat{=} & p.\sigma \wedge \sigma = \gamma & (\textit{test relation}) \\ |f|.\sigma.\gamma & \hat{=} & \gamma = f.\sigma & (\textit{mapping relation}) \end{array}$$

Furthermore, the inverse of a function is a relation:

$$f^{-1}.\gamma.\sigma \quad \hat{=} \quad \gamma = f.\sigma \qquad (\textit{inverse of function})$$

2.2. Program Variable Notation

Although most of our investigation is carried out on the algebraic level, we will need a syntactic level with program variables in the examples that illustrate the concepts and results. For this, we need to show how to model expressions and assignments with program variables. For this purpose, we assume that in practice the state space Σ is really a product, $\Sigma = \Sigma_1 \times \ldots \times \Sigma_n$. The projection function $\pi_i : \Sigma \to \Sigma_i$, $1 \leqslant i \leqslant n$, gives the value of the ith component.

A declaration like

$$\mathsf{var}\ y : \mathsf{Bool}, x : \mathsf{Nat}, z : \mathsf{Nat}$$

defines a state space and at the same time introduces more convenient names for the projection functions.

We are only interested in the set of program variables, not in the order in which they are listed. Hence, we assume that any declaration of program variables is equivalent to a declaration where the variables are listed in some standard order (e.g., lexicographical order). In this case, the equivalent declaration with standard ordering would be

$$\mathsf{var}\ x : \mathsf{Nat}, y : \mathsf{Bool}, z : \mathsf{Nat}$$

This declaration denotes the state space $\Sigma = \mathsf{Nat} \times \mathsf{Bool} \times \mathsf{Nat}$. The program variables are identified with the corresponding projection functions, so that $x = \pi_1$, $y = \pi_2$ and $z = \pi_3$.

An *expression* in program variables like $x * (z + 1)$ or a *Boolean expression* like $y \Rightarrow x = z$ is evaluated in a state σ by first applying the program variables to the state. Computing the value of the first expressions in the state $\sigma = (a, b, c)$, we have that

$$(x * (z + 1)).\sigma = x.\sigma * (z.\sigma + 1) = \pi_1.\sigma * (\pi_3.\sigma + 1) = a * (c + 1)$$

A predicate can be described by a Boolean expression over the program variable(s), e.g., $x + z > 0$. A relation can also be described using program variables. For instance, the relation $x < x' + z$ holds between states σ and σ' if $x.\sigma < x.\sigma' + z.\sigma$. Thus, we write x' to denote that the program variable should be applied to the final state rather than the initial state.

We use the *assignment notation* for relations in program statements. The basic form is

$$(x := x' \mid b) \qquad (\textit{relational assignment})$$

where b is a Boolean expression over primed and unprimed program variables. We have that $(x := x' \mid b).\sigma.\sigma'$ holds if $b.\sigma.\sigma'$ holds and state components other than x have the same value in σ and σ' (i.e., if $x = \pi_i$, then $\pi_1.\sigma = \pi_1.\sigma', \ldots, \pi_{i-1}.\sigma = \pi_{i-1}.\sigma', \pi_{i+1}.\sigma = \pi_{i+1}.\sigma', \ldots, \pi_n.\sigma = \pi_n.\sigma'$). For example, $(x := x' \mid x' > x)$ is a relation that increases the value of x by an arbitrary amount, but does not change y or z. This generalises in the obvious way to the situation where x is actually a list of program variables.

Consider two state spaces, Σ determined by $\mathsf{var}\ x, z$ and Σ' determined by $\mathsf{var}\ y, z$, where x, y and z are lists of program variables (with type indications). Here the program variables z are common to both state spaces. We generalise the assignment notation to account for relations over different state spaces, to the form

$$(y/x := y' \mid b) \qquad (\textit{generalised relational assignment})$$

This is a relation from initial state space Σ with program variables x, z to final state space Σ' with program variables y, z. We have that $(y/x := y' \mid b).\sigma.\sigma'$ holds if $b.\sigma.\sigma'$ holds and $z.\sigma = z.\sigma'$ (note that if x and y have different types then the two occurrences of z here denote differently typed projection functions). The form y/x shows that the program variables y are added to the state space and that the program variables x are deleted from the state space. This avoids the need to keep explicit track of the state spaces involved in a program statement, even when we are changing the state space in the middle of a statement.

As an example, the assignment $(y/x := y' \mid x = y' + 1)$ has the following intuition: the program variable x is replaced by program variable y which gets a new value y' related to the old value of x by $x = y' + 1$.

Note also that the simple relational assignment above is a special case of this more general relational assignment, where the same variable is removed and introduced:

$$(x := x' \mid b) = (x/x := x' \mid b)$$

The ordinary (functional, or deterministic) assignment $(x := e)$ is also a special case, which we can define as follows:

$$|x := e| \quad = \quad (x := x' \mid x' = e) \qquad (\textit{functional assignment})$$

Thus $(x := e)$ denotes a state transformer that changes the value of x to the value of e in the initial state. The generalisation to a deterministic assignment $(y/x := e)$ that replaces variables is straightforward:

$$|y/x := e| \quad = \quad (y/x := y' \mid y' = e) \quad (\textit{generalised functional assignment})$$

In Appendix A we give a number of rules for manipulating assignments.

This way of modelling program variables in the refinement calculus is quite simple and straightforward. A disadvantage is that the number of program variables in a state is always fixed. Thus, $\mathsf{var}\ x : \mathsf{Nat}, y : \mathsf{Nat}$ denotes a state space with two components, whereas $\mathsf{var}\ x : \mathsf{Nat}, y : \mathsf{Nat}, z : \mathsf{Bool}$ is a different state space with

three components. The transformation between these two state spaces has to be done explicitly, with coercion functions. In [BaW98] we present a more general and flexible approach to modelling program variables, which avoids the need for coercion functions and is therefore better suited for programming logics. For the purposes of this paper, that added flexibility is not, however, needed, so we will use the simple model here instead.

2.3. Predicate Transformers

A *predicate transformer* is a function that maps predicates to predicates. We use the abbreviation $\Sigma \mapsto \Gamma$ for the predicate transformer type $(\Gamma \to \mathsf{Bool}) \to (\Sigma \to \mathsf{Bool})$. The intuition is that $S : \Sigma \mapsto \Gamma$ maps postconditions over state space Γ to preconditions over state space Σ. Predicate transformers have a *weakest precondition* interpretation. Thus, $S.q$ is a predicate that characterises those initial states from which execution of S is guaranteed to terminate in a final state where postcondition q holds. The *refinement ordering* on $\Sigma \mapsto \Gamma$ is defined by pointwise extension of the ordering on predicates:

$$S \sqsubseteq S' \quad \hat{=} \quad (\forall p \bullet S.p \subseteq S'.p) \qquad (\textit{refinement})$$

Programs are modelled by *monotonic predicate transformers* $\Sigma \mapsto_m \Gamma$. These form a complete lattice with respect to the refinement ordering. In the rest of this paper we will assume monotonicity without usually mentioning this fact explicitly. The algebraic structure of predicate transformers gives us three basic program operators (composition, meet and join) and three corresponding basic constants (the unit element skip, the top element magic and the bottom element abort):

$$\begin{array}{rcll}
(S_1 ; S_2).q & \hat{=} & S_1.(S_2.q) & (\textit{composition}) \\
(\sqcap i \in I \bullet S_i).q & \hat{=} & (\cap i \in I \bullet S_i.q) & (\textit{meet, demonic choice}) \\
(\sqcup i \in I \bullet S_i).q & \hat{=} & (\cup i \in I \bullet S_i.q) & (\textit{join, angelic choice}) \\
\mathsf{skip}.q & \hat{=} & q & (\textit{skip}) \\
\mathsf{magic}.q & \hat{=} & \mathsf{true} & (\textit{magic}) \\
\mathsf{abort}.q & \hat{=} & \mathsf{false} & (\textit{abort})
\end{array}$$

In addition, we will work with fixpoint constructs (to be explained later) and with four homomorphic embedding operators. Two of them embed predicates in predicate transformers and the other two embed relations:

$$\begin{array}{rcll}
\{p\}.q & \hat{=} & p \cap q & (\textit{assertion}) \\
[p].q & \hat{=} & \neg p \cup q & (\textit{guard}) \\
\{R\}.q & \hat{=} & \mathsf{im}.R^{-1}.q & (\textit{angelic update}) \\
[R].q & \hat{=} & \overline{\mathsf{im}}.R.q & (\textit{demonic update})
\end{array}$$

for predicates p and relations R. Using the definitions of images, the two update statements are characterised as follows:

$$\{R\}.q.\sigma \;\equiv\; R.\sigma \cap q \neq \emptyset \;\text{ and }\; [R].q.\sigma \;\equiv\; R.\sigma \subseteq q$$

A special case of demonic update is the *chaotic update* $\mathsf{chaos} = [\mathsf{True}]$. Note also that assertions and guards are special cases of updates:

$$\{p\} \;=\; \{\lambda\sigma\ \sigma' \bullet p.\sigma \wedge \sigma' = \sigma\} \;\text{ and }\; [p] \;=\; [\lambda\sigma\ \sigma' \bullet p.\sigma \wedge \sigma' = \sigma]$$

We also introduce an embedding of state transformers:

$$\langle f \rangle.q \quad \hat{=} \quad f ; q \qquad (\textit{functional update})$$

We will not consider the functional update to be a primitive notion here, since it is a special case of both the other updates: $\{|f|\} = \langle f \rangle = [|f|]$. However, we will occasionally comment on how results for the angelic and demonic updates carry over to functional updates.

There is a strong duality built into the constructs described above. We define the *duality operator* on predicate transformers by

$$S^{\circ}.q \quad \hat{=} \quad \neg S.(\neg q) \qquad (dual)$$

Then $\{R\}$ and $[R]$ are duals, as are $[p]$ and $\{p\}$, while $\langle f \rangle$ is self-dual.

In practice, we use assignments to express state changes. For instance, $\langle x := x + z \rangle ; \langle y, z := \mathsf{T}, x \rangle$ is a sequence of two assignment statements (functional updates), the second one being a multiple assignment statement. A demonic assignment is a demonic update where the relation is an assignment, e.g., $[x := x' \mid x' > x]$.

We can introduce more traditional program constructs using the basic statements. For instance, a conditional statement is defined by

$$\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \ \mathsf{fi} \ \hat{=}\ \{b\}; S_1 \sqcup \{\neg b\}; S_2$$

and a block with local variables is defined by

$$\mathsf{begin\ var}\ x := c; S\ \ \mathsf{end} \ \hat{=}\ \langle x/ := c \rangle ; S ; \langle /x \rangle$$

Here we first introduce a new variable x that is initialised to the constant c (without removing any variables), then we execute S, and finally we remove x (without introducing any new variables).

The well-known *Knaster-Tarski* theorem guarantees that every monotonic function f on a complete lattice has a least fixpoint $\mu.f$ and a greatest fixpoint $\nu.f$. This allows us to define predicate transformers by recursion. For instance, a simple while loop is defined by

$$\mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od} \ \hat{=}\ \mu.(\lambda X \bullet \mathsf{if}\ b\ \mathsf{then}\ S; X\ \mathsf{else\ skip}\ \ \mathsf{fi})$$

The above should be sufficient to indicate that the statements that we can construct from our basic predicate transformer constructs are very general, and allow us to model both specifications as well as executable statements within the same framework. For more details, we refer to [BaW98].

A number of useful algebraic properties can be proved about the basic predicate transformer constructs. As an example, the basic predicate transformer constructs are homomorphic with respect to composition (considering $\cap$ as the composition of predicates). Thus we have

$$\begin{array}{rclcrcl} \{p\};\{q\} & = & \{p \cap q\} & \qquad & [p];[q] & = & [p \cap q] \\ \{Q\};\{R\} & = & \{Q;R\} & & [Q];[R] & = & [Q;R] \\ \langle f \rangle ; \langle g \rangle & = & \langle f;g \rangle & & & & \end{array}$$

for arbitrary predicates p and q, relations Q and R and functions f and g. A more thorough treatment of the homomorphic properties of statements is also given in [BaW98].

2.4. Homomorphic Predicate Transformers and Normal Forms

Predicate transformers can be classified according to basic homomorphism properties (in addition to monotonicity). We say that S is *conjunctive* if it distributes over non-empty meets, i.e., if $S.(\cap i \in I \bullet q_i) = (\cap i \in I \bullet S.q_i)$ for arbitrary non-empty collections $\{q_i \mid i \in I\}$ of predicates. Dually, S is *disjunctive* if it distributes over non-empty joins of predicates. Furthermore, S is *strict* if $S.\mathsf{false} = \mathsf{false}$ and *terminating* if $S.\mathsf{true} = \mathsf{true}$.

If a predicate transformer S is both terminating and conjunctive (i.e., if S distributes over arbitrary meets) then it is said to be *universally conjunctive*. Dually, S is said to be *universally disjunctive* if it is both strict and disjunctive.

It is well known that conjunctivity and disjunctivity each implies monotonicity. Furthermore, universal disjunctivity implies continuity (i.e., distributivity over joins of directed sets; see Section 6) and dually, universal conjunctivity implies cocontinuity.

The basic notation that we introduced above provides normal forms for different classes of predicate transformers. The following lemma summarises the normal forms that will be used in this paper.

Lemma 1. Assume that S is a predicate transformer.

(a) If S is monotonic then it can be written in the form $\{Q\};[R]$ for some relations Q and R.

(b) If S is conjunctive then it can be written in the form $\{p\};[R]$ for some predicate p and some relation R.
(c) If S is universally conjunctive then it can be written in the form $[R]$ for some relation R.
(d) If S is universally disjunctive then it can be written in the form $\{R\}$ for some relation R.
(e) If S is conjunctive and universally disjunctive then it can be written in the form $\{p\};\langle f\rangle$ for some predicate p and some function f.

This *normal form theorem* is proved in [vWr94].

2.5. Galois Connections and Inverse Statements

Assume that $(A,\sqsubseteq)$ and $(B,\sqsubseteq)$ are partially ordered sets and that functions $f : A \to B$ and $g : B \to A$ are given. We say that the pair (f,g) is a *Galois connection* if the following condition holds for all x and y:

$$f.x \sqsubseteq y \quad \equiv \quad x \sqsubseteq g.y \qquad \text{(Galois connection)}$$

For example, the connection between the composition and quotient operations on relations show that $((\lambda X \cdot P;X),(\lambda Y \cdot P^{-1}\backslash Y))$ is a Galois connection. Another example of a Galois connection is the pair $(\mathsf{im}.R, \overline{\mathsf{im}}.R)$, for arbitrary relation R:

$$\begin{array}{ll} & \mathsf{im}.R.p \subseteq q \\ \equiv & \{\text{definitions of image and predicate ordering}\} \\ & (\forall \gamma \cdot (\exists \sigma \cdot R.\sigma.\gamma \wedge p.\sigma) \Rightarrow q.\gamma) \\ \equiv & \{\text{quantfier rules}\} \\ & (\forall \sigma \cdot p.\sigma \Rightarrow (\forall \gamma \cdot R.\sigma.\gamma \Rightarrow q.\gamma)) \\ \equiv & \{\text{definitions of inverse image and predicate ordering}\} \\ & p \subseteq \overline{\mathsf{im}}.R.q \end{array}$$

When the functions f and g are monotonic, the notion of a Galois connection can be expressed algebraically on the level of the functions themselves: (f,g) is a Galois connection if and only if

$$f \circ g \sqsubseteq \mathsf{id} \qquad \text{and} \qquad \mathsf{id} \sqsubseteq g \circ f$$

When the partially ordered sets involved are complete lattices, we can say even more about Galois connections.

Lemma 2. Assume that A and B are complete lattices.

(a) If the pair of monotonic functions $(f : A \to B, g : B \to A)$ is a Galois connection, then f is a (universal) join homomorphism and g is a (universal) meet homomorphism.
(b) If $g : B \to A$ is a meet homomorphism, then there exists a unique monotonic function $f : A \to B$ such that (f,g) is a Galois connection.
(c) If $f : A \to B$ is a join homomorphism, then there exists a unique monotonic function $g : B \to A$ such that (f,g) is a Galois connection.

When (f,g) is a Galois connection, then f is called the *left adjoint* of g and g the *right adjoint* of f.

The following *shunting properties* show how functions in a Galois connection can be manipulated algebraically.

Lemma 3. Assume that the pair of monotonic functions (f,g) is a Galois connection. Then

(a) $f;h \sqsubseteq h' \;\equiv\; h \sqsubseteq g;h'$
(b) $h;g \sqsubseteq h' \;\equiv\; h \sqsubseteq h';f$

for arbitrary monotonic functions h and h'.

Since statements are seen here as monotonic predicate transformers on a complete lattice of predicates, we can apply the theory of Galois connections directly to statements. From the normal form theorem (Lemma 1) we know that a Galois connection (S,T) of statements must be of the form $(\{P\},[Q])$ for some state relations P and Q. The following lemma makes this even clearer.

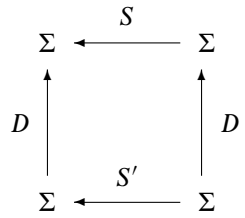

Fig. 1. Data refinement.

Lemma 4. Let state relation R, predicate r and state function f be arbitrary. Then

(a) $\{R\}$ and $[R^{-1}]$ form a Galois connection, and
(b) $\{r\};\langle f\rangle$ and $[f^{-1}];[r]$ form a Galois connection.

When a statement pair (S, T) is a Galois connection we will refer to T as the *inverse statement* of S, denoted by $\overline{S}$. The use of the term 'inverse' can be justified by the properties $S;\overline{S} \sqsubseteq \mathsf{skip}$ and $\mathsf{skip} \sqsubseteq \overline{S};S$ (for details, see [BaW93]). Lemma 4 gives us a general rule for inverse statements. In program variable notation, these rules amount to the following:

$$\overline{\{x := x' \mid b\}} = [x := x' \mid b[x, x' := x', x]]$$
$$\overline{\{p\};\langle x := e\rangle} = [x := x' \mid x = e[x := x']];[p]$$

In addition to this, we will use the fact that

$$\overline{S_1;S_2} = \overline{S_2};\overline{S_1}$$

Inverse statements are investigated in more detail in [BaW93].

A simple example can illustrate the idea of inversion. Consider the statement $S = \langle x := x + 1\rangle$ where x ranges over the natural numbers. The inverse $\overline{S}$ of this statement is

$$\overline{\langle x := x + 1\rangle} = [x > 0];\langle x := x - 1\rangle$$

Executing $S;\overline{S}$ from an initial state where $x = x_0$ leads to an intermediate state where $x = x_0 + 1$ and a final state where $x = x_0$, so $S;\overline{S} = \mathsf{skip}$. On the other hand, $\overline{S};S$ is miraculous when executed in initial state where $x = 0$, but from any other initial state it terminates in the initial state, so $\overline{S};S \sqsupseteq \mathsf{skip}$.

3. Data Refinement and Encoding

The basis for encoding is data refinement, which can on an abstract level be described as a commutativity property: we define the relation $\sqsubseteq_D$ as follows:

$$S \sqsubseteq_D S' \quad \triangleq \quad D;S \sqsubseteq S';D \qquad (\textit{data refinement})$$

and when $S \sqsubseteq_D S'$ holds we say that S *is data refined through* D *by* S' (or that S is D*-refined by* S'). Here $D : \Gamma \mapsto \Sigma$ (the *abstraction statement*), $S : \Sigma \mapsto \Sigma$ (the *abstract statement*) and $S' : \Gamma \mapsto \Gamma$ (the *concrete statement*) are monotonic predicate transformers. Data refinement can be illustrated by a *subcommuting diagram* where a path lower down is always a refinement of a path higher up with the same end points (see Fig. 1).

3.1. Abstraction Category

Data refinement gives us a category, where there is an arrow D (an abstraction) with source S' and target S exactly when $S \sqsubseteq_D S'$ holds. Identity morphisms are of the form skip (note that $\sqsubseteq_{\mathsf{skip}}$ is ordinary algorithmic refinement $\sqsubseteq$, which is reflexive) and the composition of morphisms is sequential composition; if $S \sqsubseteq_D S'$ and

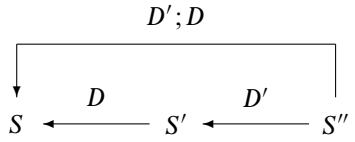

Fig. 2. Abstraction category.

$S' \sqsubseteq_{D'} S''$, then

$$
\begin{array}{ll}
 & D';D;S \\
\sqsubseteq & \{\text{assumption } S \sqsubseteq_D S',\ D' \text{ monotonic}\} \\
 & D';S';D \\
\sqsubseteq & \{\text{assumption } S' \sqsubseteq_{D'} S''\} \\
 & S'';D';D
\end{array}
$$

so $S \sqsubseteq_{D';D} S''$. This is illustrated in the diagram in Fig. 2.

It is well known that data refinement can be handled in category-theoretic terms [JiH90]. However, we prefer to work in the simpler partial order framework where data refinement is expressed in terms of algorithmic refinement. How this is done is shown in the next subsection.

3.2. Encoding

A typical situation in data refinement is that the abstract statement S and the abstraction D are given. The problem is then to find a monotonic predicate transformer S' such that $S \sqsubseteq_D S'$ holds. Furthermore, we are interested in making S' as small as possible (with respect to the refinement ordering), giving us maximal freedom in subsequent refinement steps. Or, to put it in another way, we are interested in the *smallest D-refinement of S*.

Consider therefore $D;S \sqsubseteq X;D$ as an equation in unknown monotonic predicate transformer X. This equation obviously has a solution (set $X = \mathsf{magic}$). Furthermore, if $\{S_i \mid i \in I\}$ is a set of solutions, then $(\sqcap i \in I \cdot S_i)$ is monotonic and

$$
\begin{array}{ll}
 & (\sqcap i \in I \cdot S_i);D \\
= & \{\text{distributivity}\} \\
 & (\sqcap i \in I \cdot S_i;D) \\
\sqsupseteq & \{\text{assumption } D;S \sqsubseteq S_i;D \text{ for all } i \in I\} \\
 & (\sqcap i \in I \cdot D;S) \\
= & \{\text{vacuous meet}\} \\
 & D;S
\end{array}
$$

so there always exists a least solution, which is $\sqcap\{X \in \Sigma \mapsto_m \Sigma \mid D;S \sqsubseteq X;D\}$. This is obviously the smallest D-refinement of S. We introduce the notation $S \downarrow D$ for this solution, and call it the *encoding of S with D*. This means that $S \downarrow D$ is characterised by the following two conditions:

$$
\begin{array}{lll}
S \sqsubseteq_D (S \downarrow D) & & \text{(encode1)} \\
S \sqsubseteq_D X \ \Rightarrow\ S \downarrow D \sqsubseteq X & & \text{(encode2)}
\end{array}
$$

The following alternative characterisation of encoding is generally more useful:

Theorem 5. The encoding $S \downarrow D$ is uniquely defined by the following property:

$$S \downarrow D \sqsubseteq S' \equiv S \sqsubseteq_D S'$$

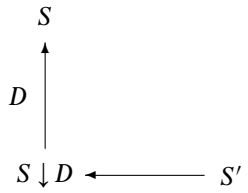

Fig. 3. Factoring out data refinement.

Proof. First assume that conditions (encode1) and (encode2) hold. Then

$$
\begin{array}{ll}
 & (S \downarrow D) \sqsubseteq S' \\
\Rightarrow & \{\text{monotonicity}\} \\
 & (S \downarrow D); D \sqsubseteq S'; D \\
\Rightarrow & \{\text{property (encode1), refinement is transitive}\} \\
 & D; S \sqsubseteq S'; D
\end{array}
$$

and (encode2) with $X := S'$ gives implication in the opposite direction. Now assume that $S \downarrow D \sqsubseteq S' \equiv S \sqsubseteq_D S'$ for arbitrary S'. Then

$$
\begin{array}{ll}
 & D; S \sqsubseteq (S \downarrow D); D \\
\equiv & \{\text{assumption with } S' := S \downarrow D\} \\
 & S \downarrow D \sqsubseteq S \downarrow D \\
\Leftarrow & \{\text{refinement is reflexive}\} \\
 & \mathsf{T}
\end{array}
$$

and

$$
\begin{array}{ll}
 & D; S \sqsubseteq X; D \\
\equiv & \{\text{assumption with } S' := X\} \\
 & S \downarrow D \sqsubseteq X
\end{array}
$$

and the proof is finished. □

Assume that monotonic predicate transformer $S : \Sigma \mapsto \Sigma$ and monotonic abstraction $D : \Gamma \mapsto \Sigma$ are given. Then $S \downarrow D$ has type $\Gamma \mapsto \Gamma$. We think of $S \downarrow D$ as the translation of S to the state space Γ, such that only the data representation is changed. Thus, a data refinement $S \sqsubseteq_D S'$ can be decomposed so that the pure data refinement is 'factored out', and followed by the 'remaining' algorithmic refinement (i.e., the change in non-determinism, miraculousness and termination that is not inherent in the data refinement)

$$S \ \sqsubseteq_D \ S \downarrow D \ \sqsubseteq \ S'$$

This is illustrated in Fig. 3, where the horizontal arrow stands for algorithmic refinement (decoding with skip) and the vertical arrow for data refinement.

3.3. Homomorphism Properties of Encoding

Encoding is a binary operator on predicate transformers. We shall now investigate its homomorphism and other algebraic properties, in both arguments.

Theorem 6. Encoding is monotonic in its first argument:

$$S \sqsubseteq S' \ \Rightarrow \ S \downarrow D \sqsubseteq S' \downarrow D$$

The encoding operation is not, however, monotonic in its second argument. From Theorem 8 below we get

the following three encodings:

$$\begin{aligned} \mathsf{skip} \downarrow \mathsf{abort} &= \mathsf{abort} \\ \mathsf{skip} \downarrow \mathsf{skip} &= \mathsf{skip} \\ \mathsf{skip} \downarrow \mathsf{magic} &= \mathsf{chaos} \end{aligned}$$

Since $\mathsf{abort} \sqsubset \mathsf{chaos} \sqsubset \mathsf{skip}$ (provided that the underlying state space has at least two elements) we have $\mathsf{skip} \sqsubseteq \mathsf{magic}$ but $\mathsf{skip} \downarrow \mathsf{skip} \not\sqsubseteq \mathsf{skip} \downarrow \mathsf{magic}$ (and $\mathsf{abort} \sqsubseteq \mathsf{skip}$ but $\mathsf{skip} \downarrow \mathsf{abort} \not\sqsupseteq \mathsf{skip} \downarrow \mathsf{skip}$). Thus, encoding is neither monotonic nor antimonotonic in its second argument.

Now let us consider how encoding distributes into basic statement constructs. Distribution is important, because it gives rise to *structure-preserving encodings*, i.e., refinements of the form $S \downarrow D \sqsubseteq S'$, where S and S' have the same statement structure.

Theorem 7. Encoding can be refined to preserve the structure of its first argument as follows:

(a) $\mathsf{abort} \downarrow D = \mathsf{abort}$, if D is strict,

(b) $\mathsf{skip} \downarrow D \sqsubseteq \mathsf{skip}$,

(c) $\mathsf{magic} \downarrow D = \mathsf{magic}$, if D is strict and terminating,

(d) $\{p\} \downarrow D \sqsubseteq \{p'\} \;\equiv\; D.p \subseteq p'$,

(e1) $[p] \downarrow D \sqsubseteq [p'] \;\equiv\; p' \subseteq D^{\circ}.p$, if D is universally disjunctive,

(e2) $[p] \downarrow D \sqsubseteq [p'] \;\equiv\; p' \subseteq D.p$, if D is universally conjunctive,

(f) $(S_1 ; S_2) \downarrow D \sqsubseteq (S_1 \downarrow D) ; (S_2 \downarrow D)$,

(g) $(\sqcap i \in I \cdot S_i) \downarrow D \sqsubseteq (\sqcap i \in I \cdot S_i \downarrow D)$, and

(h) $(\sqcup i \in I \cdot S_i) \downarrow D = (\sqcup i \in I \cdot S_i \downarrow D)$, if D is universally disjunctive.

The restriction in Theorem 7 (h) may seem to indicate that we should in general require abstraction D to be universally disjunctive. However, as long as we work within a framework of conjunctive predicate transformers (where the join operator is not included), this restriction need not be made.

Theorem 7 illustrates three different levels of rules for encoding. The equality rules in (a), (c) and (h) have the strongest form. They tell us that abort and magic is each the *least D-refinement* of abort and magic, respectively. The equivalence rules in (d) and (e) are slightly weaker. For example, (d) tells us that $\{D.p\}$ is the *least structure-preserving D-refinement* of $\{p\}$. However, this does not exclude the possibility that $\{p\} \downarrow D$ is strictly less (with respect to the refinement ordering) than $\{D.p\}$. Finally, the rules in (b), (f) and (g) are even weaker, not excluding the possible existence of strictly smaller D-refinements that preserves the structure of the original statement.

3.4. The Abstraction Argument of Encoding

The preceding theorems describe the homomorphism properties of encoding with respect to the first argument. For completeness, it we also investigate the homomorphism properties in the second argument.

Theorem 8. An encoding can be refined to preserve the structure of its second argument, as follows:

(a) $S \downarrow \mathsf{abort} = \mathsf{abort}$,

(b) $S \downarrow \mathsf{skip} = S$,

(c) $S \downarrow \mathsf{magic} = \mathsf{chaos}$,

(d) $S \downarrow (D_1 \sqcup D_2) \sqsubseteq (S \downarrow D_1) \sqcup (S \downarrow D_2)$.

From Theorem 8 we see that encoding is very weakly homomorphic in its second argument. However, for a sequential composition of decodings we have an important result.

Theorem 9. Encodings can be composed as follows:

$$S \downarrow (D_1 ; D_2) \;\sqsubseteq\; (S \downarrow D_2) \downarrow D_1$$

Proof.

$$
\begin{array}{ll}
 & S \downarrow (D_1; D_2) \sqsubseteq (S \downarrow D_2) \downarrow D_1 \\
\Leftarrow & \{\text{Theorem 5}\} \\
 & D_1; D_2; S \sqsubseteq ((S \downarrow D_2) \downarrow D_1); D_1; D_2 \\
\Leftarrow & \{\text{property (encode1), monotonicity}\} \\
 & D_1; D_2; S \sqsubseteq D_1; (S \downarrow D_2); D_2 \\
\Leftarrow & \{\text{property (encode1), monotonicity}\} \\
 & D_1; D_2; S \sqsubseteq D_1; D_2; S \\
\equiv & \{\text{reflexivity}\} \\
 & \mathsf{T}
\end{array}
$$

□

The importance of Theorem 9 is illustrated as follows. Assume that we encode S with D_1 and refine this to S_1:

$$S \downarrow D_1 \ \sqsubseteq \ S_1$$

Now assume that we continue, encoding S_1 with D_2 and refine this to S_2:

$$S_1 \downarrow D_2 \ \sqsubseteq \ S_2$$

Then

$$
\begin{array}{ll}
 & D_2; D_1; S \\
\sqsubseteq & \{\text{definition of encoding}\} \\
 & D_2; (S \downarrow D_1); D_1 \\
\sqsubseteq & \{\text{first assumption, } D_1 \text{ monotonic}\} \\
 & D_2; S_1; D_1 \\
\sqsubseteq & \{\text{definition of encoding}\} \\
 & (S_1 \downarrow D_2); D_2; D_1 \\
\sqsubseteq & \{\text{second assumption}\} \\
 & S_2; D_2; D_1
\end{array}
$$

and we can use Theorem 5 to conclude

$$S \downarrow (D_2; D_1) \sqsubseteq S_2$$

Theorem 9 is sufficient to justify encoding in a stepwise manner. In Section 4.1 we shall see that in an important special case the refinement in Theorem 9 can be strengthened to an equality.

4. Calculating Encodings

So far, we have assumed only that the abstraction D in an encoding is monotonic. From the normal form theorem (Lemma 1 (a)) we know that an abstraction D can be decomposed into a sequential composition

$$D = \{R_a\}; [R_d]$$

(in fact, if D is required to be strict and continuous, then a similar decomposition is possible where R_d is total and image-finite, so $[R_d]$ is strict and continuous [vWr94]). In fact, if $S \sqsubseteq_D S''$, then it is possible to find S' such that

$$S \downarrow [R_d] \sqsubseteq S' \ \text{ and } \ S' \downarrow \{R_a\} \sqsubseteq S''$$

(for details, see [GaM93, vWr94]; S' can be taken to be $S \downarrow [R_d]$). Thus it is reasonable to consider the following two special cases of encoding:

- *forward data refinement* – encoding with a universally disjunctive abstraction, and
- *backward data refinement* – encoding with a universally conjunctive abstraction.

This was noted by Gardiner and Morgan [GaM93], who show that a similar combination of forward and backward data refinement provide a single method that is sound and complete for refinement of abstract data types.

In a relational framework forward and backward data refinement must be defined separately. The more expressive predicate transformer framework permits a single definition of data refinement, with forward and backward data refinement as special cases. In practical program development, forward data refinement is sufficient for most cases. Then the relation R in the abstraction statement can be written as an assignment (we shall see examples of this in Section 7).

4.1. Forward Data Refinement

When the abstraction D is universally disjunctive, encoding can be expressed explicitly.

Theorem 10. Assume that S and T are monotonic predicate transformers and that D is universally disjunctive. Then

$$S \downarrow D = D; S; \overline{D}$$

Theorem 10 expresses the encoding $S \downarrow D$ explicitly, so it can be *calculated*. Since a universally disjunctive predicate transformer can always be written as an angelic update, we can rewrite the result as follows:

$$S \downarrow D = \{R\}; S; [R^{-1}]$$

This shows how our general statement notation allows us to express the least D-refinement of statement S in statement form, when D is universally disjunctive. Thus, encoding reduces to a well-known construction in this special case when the abstraction statement is universally disjunctive (data refinement of S was described as $\{R\}; S; [R^{-1}]$ already in [BaW89]).

We now show how Theorem 9 can be strengthened to an equality:

Theorem 11. Assume that either D_1 or D_2 is universally disjunctive. Then

$$S \downarrow (D_2; D_1) = (S \downarrow D_1) \downarrow D_2$$

Theorem 11 gives us a basis for decomposing data refinements:

Corollary 12. Assume that the data refinement $S \sqsubseteq_{D_2;D_1} S''$ is known, where either D_1 or D_2 is universally disjunctive. Then there exists S' with

$$S \sqsubseteq_{D_1} S' \sqsubseteq_{D_2} S''$$

(for the proof, choose S' to be $S \downarrow D_1$). In fact, it is easily shown that $S \downarrow D_1$ is the smallest solution to the 'equation' $S \sqsubseteq_{D_1} X \sqsubseteq_{D_2} S''$ (in Section 5.4 we shall see that in certain situations this equation also has a greatest solution). Corollary 12 generalises the result mentioned in the introduction to Section 4.

4.2. Calculating Encodings

Theorem 10 gives a closed expression for encodings when the abstraction involved is universally disjunctive. This means that it is possible to *calculate* the encoding $S \downarrow D$ of a program statement S. In a sense, we already have an explicit description $(D; S; \overline{D})$ of $S \downarrow D$, but if the abstraction D involves a change of state space, then we want to express $D; S; \overline{D}$ (or some refinement of it) purely in terms of statements that do not change the state space.

Inverse statements give us a way of calculating the data refinement of any statement S. However, in practice we want to calculate data refinements in a structure-preserving way. In this, we can improve on Theorem 7, using the fact that the decoding D is of the form $\{R\}$. As a result, we get rules that are essentially equivalent to traditional rules for data refinement [Bac80, Bac89, MoG90, Mor89, vWr94]. We number the rules in the same way as in Theorem 7, but we only state the cases where the assumption that D is universally disjunctive gives us a better result than before.

Theorem 13. Assume that D is the universally disjunctive decoding $D = \{R\}$. Then

(c) $\mathsf{magic} \downarrow D = \{\mathsf{dom}.\, R\}; \mathsf{magic}$,
(d) $\{p\} \downarrow D \sqsubseteq \{p'\} \equiv \mathsf{im}.\, R^{-1}.\, p \subseteq p'$, and
(e) $[p] \downarrow D \sqsubseteq [p'] \equiv p' \subseteq \overline{\mathsf{im}}.\, R.\, p$.

Theorem 13 (c) is an example of a rule that is not structure-preserving (of course, the structure-preserving rule $\mathsf{magic} \downarrow D \sqsubseteq \mathsf{magic}$ holds trivially). Similarly, the following rule for the guard statement is valid:

$$[p] \downarrow D \sqsubseteq \{\mathsf{dom}.\, R\}; [\overline{\mathsf{im}}.\, R.\, p]$$

4.3. Rules for Relational Updates

When the form of the abstraction is known, it is also possible to give rules for encoding relational updates.

Theorem 14. Assume that D is the universally disjunctive abstraction $\{R\}$. Then

(a) $[P] \downarrow D \sqsubseteq [P'] \equiv P' \subseteq R \backslash (P; R^{-1})$, and
(b) $\{P\} \downarrow D \sqsubseteq \{P'\} \equiv R; P \subseteq P'; R$.

Proof. For (a), we have

$$\begin{aligned}
& [P] \downarrow \{R\} \sqsubseteq [P'] \\
\equiv\ & \{\text{Theorem 10}\} \\
& \{R\}; [P]; [R^{-1}] \sqsubseteq [P'] \\
\equiv\ & \{\text{Galois connection}\} \\
& [P]; [R^{-1}] \sqsubseteq [R^{-1}]; [P'] \\
\equiv\ & \{\text{homomorphism}\} \\
& [P; R^{-1}] \sqsubseteq [R^{-1}; P'] \\
\equiv\ & \{\text{embedding}\} \\
& P; R^{-1} \supseteq R^{-1}; P' \\
\equiv\ & \{\text{property of relational quotient}\} \\
& (R^{-1})^{-1} \backslash (P; R^{-1}) \supseteq P' \\
\equiv\ & \{\text{property of inverse relation}\} \\
& R \backslash (P; R^{-1}) \supseteq P'
\end{aligned}$$

and for (b),

$$\begin{aligned}
& \{P\} \downarrow \{R\} \sqsubseteq \{P'\} \\
\equiv\ & \{\text{Theorem 5}\} \\
& \{R\}; \{P\} \sqsubseteq \{P'\}; \{R\} \\
\equiv\ & \{\text{homomorphism, embedding}\} \\
& R; P \subseteq P'; R
\end{aligned}$$

□

Theorem 14 (a) shows that the least structure-preserving encoding of $[P]$ under $\{R\}$ is $[R \backslash (P; R^{-1})]$. On the other hand, (b) does not allow explicit calculation of the encoding for an angelic update. Instead, the rule can be used as a *proof rule*, to check whether a suggested encoding $\{P\} \downarrow D \sqsubseteq \{P'\}$ holds. It can also be used for an *implicit calculation* of the encoding. If we can do a stepwise calculation of the form

$$\begin{aligned}
& R; P \\
\subseteq\ & \cdots \\
& P'; R
\end{aligned}$$

then we have calculated a relation P' such that $\{P\} \downarrow D \sqsubseteq \{P'\}$ holds (an example of this is given in Section 7.4).

The functional update is handled by rewriting according to $\langle f \rangle = [|f|]$. Thus the rule is

$$\langle g \rangle \downarrow D \sqsubseteq \langle g' \rangle \equiv |g'| \subseteq R \backslash (|g|; R^{-1})$$

In practical program development, a program is often specified as a general conjunctive specification $\{p\}; [Q]$, in terms of a precondition p and a next-state relation Q. The following generalisation of Theorem 14 (c) gives a rule for encoding such a specification.

Theorem 15. Assume that D is the universally disjunctive abstraction $\{R\}$. Then

$$\{p\}; [Q] \downarrow D \sqsubseteq \{p'\}; [Q'] \equiv (\mathsf{im}.\, R^{-1}.\, p \subseteq p') \wedge (Q' \subseteq (R; |p|) \backslash (Q; R^{-1}))$$

A direct consequence of Theorem 15 is the following calculational rule for encoding a conjunctive specification:

$$\{p\}; [Q] \downarrow D \sqsubseteq \{\mathsf{im}.\, R^{-1}.\, p\}; [(R; |p|) \backslash (Q; R^{-1})]$$

4.4. Functional Data Refinement

An important special case of data refinement is *functional data refinement* where the decoding is of the form $\{r\}; \langle f \rangle$. Here, r is a *concrete invariant* on Γ and $f : \Gamma \to \Sigma$ is an *abstraction function* which computes the unique abstract state σ corresponding to any concrete state γ in r. Functional data refinement was studied previously by Hoare [Hoa72] in Hoare logic and by Back [Bac80] in the refinement calculus.

The structural rules are the same for functional data refinement as for data refinement in general, noting that the inverse of $\{r\}; \langle f \rangle$ is $[f^{-1}]; [r]$ (i.e, a demonic update followed by a guard). Thus we have

$$S \downarrow D = \{r\}; \langle f \rangle; S; [f^{-1}]; [r]$$

where $D = \{r\}; \langle f \rangle$. Here the assertion $\{r\}$ at the beginning and the guard $[r]$ at the end express the requirement that r is an invariant. It then remains to find a way of simplifying $\langle f \rangle; S; [f^{-1}]$. In fact, we get useful special cases of the rules in Theorems 13 and 14:

Theorem 16. Assume that p and r are predicates, f is a function, P is a relation, and $D = \{r\}; \langle f \rangle$. Then

(a) $\{p\} \downarrow D \sqsubseteq \{r\}; \{p'\}; [r]$, if $f; p \subseteq p'$,
(b) $[p] \downarrow D \sqsubseteq \{r\}; [p']; [r]$, if $p' \subseteq f; p$,
(c) $[P] \downarrow D = \{r\}; [|f|; P; f^{-1}]; [r]$, and
(d) $\{P\} \downarrow D \sqsubseteq \{r\}; \{P'\}; [r]$, if $|f|; P \subseteq P'; |f|$.

Again, a functional update is handled by rewriting it into a demonic update first, as $\langle g \rangle = [|g|]$. However, if f is bijective, then we get a simpler rule:

Corollary 17. Assume that $D = \{r\}; \langle f \rangle$ where f is a bijective function, and let f^{-1} (temporarily) stand for the inverse function of f. Then

$$\langle g \rangle \downarrow D = \{r\}; \langle f; g; f^{-1} \rangle; [r]$$

4.5. Backward Data Refinement

In the backward data refinement case (i.e., when the abstraction of an encoding is universally conjunctive), we do not get such strong rules as in the forward case. However, we still get a complete collection of rules that allow us to verify encoding of statements. As before, we only show the cases that improve on the general result in Theorem 7.

Theorem 18. Assume that D is the universally conjunctive abstraction $D = [R]$. Then

(a) $\mathsf{abort} \downarrow D = \mathsf{abort}$, if R is total,
(c) $\mathsf{magic} \downarrow D = \mathsf{magic}$, if R is total,

(d) $\{p\} \downarrow D \sqsubseteq \{p'\} \equiv \overline{\mathsf{im}}.R.p \subseteq p'$, and
(e) $[p] \downarrow D \sqsubseteq [p'] \equiv p' \subseteq \overline{\mathsf{im}}.R.p$.

For the demonic update, the rule is similar to the forward case.

Theorem 19. Assume that D is the universally conjunctive abstraction $[R]$. Then

$$[P] \downarrow D \sqsubseteq [P'] \equiv P'^{-1} \subseteq R \backslash (R; P)^{-1}$$

However, for the angelic update there seems to be no simple rule; there are no algebraic laws that allow us to manipulate the expression $[R]; \{P\} \sqsubseteq \{P'\}; [R]$.

5. Data Refinement and Decoding

Since the encoding $S \downarrow D$ was defined as the least solution to the equation $D; S \sqsubseteq X; D$, it makes sense to consider the dual situation also. We refer to greatest solution to the equation $D; X \sqsubseteq T; D$ as the *decoding of T by D*, written $T \uparrow D$. Essentially, $T \uparrow D$ is the greatest statement that is D-refined to T. However, this equation does not necessarily have a solution at all (set $D = \mathsf{magic}$ and $T = \mathsf{abort}$). Furthermore, even if it has solutions, no greatest solution need exist.

If $T \uparrow D$ does exist, then it satisfies the following properties:

$$T \uparrow D \sqsubseteq_D T \qquad \text{(decode1)}$$
$$X \sqsubseteq_D T \Rightarrow X \sqsubseteq T \uparrow D \qquad \text{(decode2)}$$

5.1. Conditions for Decoding

Before considering in more detail under what conditions the decoding actually exists, let us show its relationship to D-refinement.

Theorem 20. The definition of $T \uparrow D$ can equivalently be expressed by requiring that

$$S \sqsubseteq T \uparrow D \equiv S \sqsubseteq_D T$$

holds for arbitrary S.

It is also interesting to note that from Theorems 5 and 20 we directly find that

$$((\lambda X \cdot X \downarrow D), (\lambda Y \cdot Y \uparrow D))$$

is a Galois connection: Thus, by Lemma 2 we know that this condition can be satisfied only if $(\lambda X \cdot X \downarrow D)$ distributes over arbitrary joins. We have

$$(\lambda X \cdot X \downarrow D).(\sqcup i \in I \cdot S_i) = (\sqcup i \in I \cdot (\lambda X \cdot X \downarrow D).S_i)$$
$$\equiv \{\text{beta reduction}\}$$
$$(\sqcup i \in I \cdot S_i) \downarrow D = (\sqcup i \in I \cdot S_i \downarrow D)$$

and this was in Theorem 7 shown to hold if D is universally disjunctive.

Thus we can summarise: the decoding $T \uparrow D$ exists for arbitrary monotonic predicate transformer T whenever the decoding D is universally disjunctive.

Theorem 20 indicates that decoding is essentially reverse encoding. We can interpret the decoding $S \uparrow D$ as translating the statement S into a more abstract state space, without changing the non-determinism, termination or strictness of the statement. In Section 7 we will see an example of how decoding can be used to reduce a correctness condition to a more abstract one.

Applying the rules $f \circ \overline{f} \sqsubseteq \mathsf{id}$ and $\mathsf{id} \sqsubseteq \overline{f} \circ f$ to the Galois connection $((\lambda X \cdot X \downarrow D), (\lambda Y \cdot Y \uparrow D))$ we get the following result which shows that encoding and decoding each undoes the effect of the other, thus giving further justification for the naming of these operators:

Corollary 21. Assume that abstraction D is universally disjunctive. Then

(a) $S \sqsubseteq (S \downarrow D) \uparrow D$,
(b) $(T \uparrow D) \downarrow D \sqsubseteq T$.

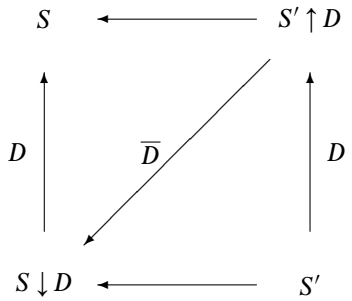

Fig. 4. Relationships derived from $S \sqsubseteq_D T$.

5.2. Decoding and the Abstraction Category

An interesting duality between encoding and decoding (for universally disjunctive abstraction statement D) is the following. Assume that the data refinement $S \sqsubseteq_D T$ is given. Recall from Section 3.2 that we have

$$S \;\sqsubseteq_D\; S \downarrow D \;\sqsubseteq\; T$$

which factors out the pure data refinement first, and then the remaining algorithmic refinement. On the other hand, Theorem 20 shows that we can also reverse the order and factor out the algorithmic refinement first:

$$S \;\sqsubseteq\; T \uparrow D \;\sqsubseteq_D\; T$$

This raises the question whether $S \downarrow D$ and $T \uparrow D$ are related in the abstraction category. We have

$$\begin{aligned}
&\overline{D};(S \downarrow D)\\
=\;&\{\text{Theorem 10}\}\\
&\overline{D};D;S;\overline{D}\\
\sqsubseteq\;&\{\text{assumption } S \sqsubseteq_D T\}\\
&\overline{D};T;D;\overline{D}\\
=\;&\{\text{Theorem 24, below}\}\\
&(T \uparrow D);\overline{D}
\end{aligned}$$

which shows that $S \downarrow D \sqsubseteq_{\overline{D}} T \uparrow D$. Figure 4 illustrates all the relationships that can be deduced once $S \sqsubseteq_D T$ is known, for universally disjunctive D.

5.3. Properties of Decoding

The characterisation of decoding in Theorem 20 allows us to deduce homomorphism properties for decoding similar to those of encoding. First, we consider monotonicity:

Theorem 22. Decoding is monotonic in its first argument:

$$T \sqsubseteq T' \;\Rightarrow\; T \uparrow D \sqsubseteq T' \uparrow D$$

for universally disjunctive D.

Next, we collect other basic homomorphism properties.

Theorem 23. Assume that D is universally disjunctive. Then

(a) $\mathsf{abort} = \mathsf{abort} \uparrow D$ if D is terminating,
(b) $\mathsf{skip} \sqsubseteq \mathsf{skip} \uparrow D$,
(c) $\mathsf{magic} = \mathsf{magic} \uparrow D$,

(d) $\{p\} \sqsubseteq \{p'\} \uparrow D \;\equiv\; p \subseteq \overline{D}.\, p'$,
(e) $[p] \sqsubseteq [p'] \uparrow D \;\equiv\; \overline{D}^{\circ}.\, p' \subseteq p$,
(f) $(S_1 \uparrow D); (S_2 \uparrow D) \sqsubseteq (S_1; S_2) \uparrow D$,
(g) $(\sqcap i \in I \cdot S_i \uparrow D) = (\sqcap i \in I \cdot S_i) \uparrow D$,
(h) $(\sqcup i \in I \cdot S_i \uparrow D) \sqsubseteq (\sqcup i \in I \cdot S_i) \uparrow D$.

The refinements in Theorem 23 may seem to go the wrong way, but the intuition can be explained as follows. Decoding is a kind of reverse refinement, so $S \uparrow D$ is a less refined version of S (with respect to the abstraction D). If S' and D are known, then finding a statement S such $S \sqsubseteq S' \uparrow D$ means finding some statement S that is D-refined by S'.

5.4. Decoding with Inverse Statements

Decoding can also be expressed explicitly when the abstraction D is universally disjunctive.

Theorem 24. Assume that S and T are monotonic predicate transformers and that D is universally disjunctive. Then

$$T \uparrow D = \overline{D}; T; D$$

Theorems 10 and 24 and the shunting properties of inverse statements now allow us to characterise forward data refinement in a number of ways:

Corollary 25. All the following conditions are equivalent, for monotonic S and S' and universally disjunctive D:

(a) $S \downarrow D \sqsubseteq S'$
(b) $D; S; \overline{D} \;\sqsubseteq\; S'$
(c) $D; S \;\sqsubseteq\; S'; D$
(d) $S \sqsubseteq S' \uparrow D$
(e) $S \;\sqsubseteq\; \overline{D}; S'; D$
(f) $S; \overline{D} \;\sqsubseteq\; \overline{D}; S'$

The rule for composing encodings has a counterpart for decodings.

Theorem 26. Decoding can be composed as follows:

$$(S \uparrow D_1) \uparrow D_2 \;=\; S \uparrow (D_1; D_2)$$

In connection with Corollary 12 we stated that if $S \sqsubseteq_{D_2; D_1} T$ where either D_1 or D_2 is universally disjunctive, then $S \downarrow D_1$ is the smallest solution to the equation $S \sqsubseteq_{D_1} X \sqsubseteq_{D_2} T$. In fact, if D_2 is universally disjunctive then $T \uparrow D_2$ is the greatest solution to this same equation (which has a complete lattice of solutions), and we have

$$S \;\sqsubseteq_{D_1}\; S \downarrow D_1 \;\sqsubseteq\; T \uparrow D_2 \;\sqsubseteq_{D_2}\; T$$

5.5. Calculating Decodings

When the abstraction D is explicitly given as an angelic update, we can improve slightly on the results in Theorem 23.

Corollary 27. Assume that D is the universally disjunctive abstraction $D = \{R\}$. Then

(a) $\mathsf{abort} = \mathsf{abort} \uparrow D$ if R is total, and
(d) $\{p\} \sqsubseteq \{p'\} \uparrow D \;\equiv\; p \subseteq \overline{\mathsf{im}}.\, R^{-1}.\, p'$.

For the relational updates, we get duals from the results in Theorem 14.

Theorem 28. Assume that D is the universally disjunctive decoding $\{R\}$. Then

(a) $[P] \sqsubseteq [P'] \uparrow D \;\equiv\; P; R^{-1} \supseteq R^{-1}; P'$, and
(b) $\{P\} \sqsubseteq \{P'\} \uparrow D \;\equiv\; P \subseteq R^{-1} \backslash (P'; R)$.

Again, we do not give a separate rule for functional update; it is handled by rewriting according to $\langle f \rangle = \{|f|\}$.

Exactly as for encoding, we can try to get simpler special cases when the abstraction D used in an decoding is functional, i.e., when D is of the form $\{r\}; \langle f \rangle$.

Theorem 29. Assume that D is the abstraction $D = \{r\}; \langle f \rangle$. Then

(a) $\{p\} \sqsubseteq \{p'\} \uparrow D \;\equiv\; r \cap f; p \subseteq p'$,
(b) $[p] \sqsubseteq [p'] \uparrow D \;\equiv\; r \cap p' \subseteq f; p$, and
(c) $\langle g \rangle \sqsubseteq \langle g' \rangle \uparrow D \;\equiv\; r \subseteq g; r \wedge f; g = g'; f$.

For the relational updates we do not get simpler rules that those in Theorem 28.

6. Recursive Statements

Before looking in more detail on how encoding works with recursive statements, let us state the following very useful *fusion theorem* (attributed to Kleene).

Lemma 30. Assume that $f : \Sigma \to \Sigma$ and $g : \Gamma \to \Gamma$ are monotonic functions on complete lattices and that $h : \Sigma \to \Gamma$ is continuous and $k : \Sigma \to \Gamma$ is co-continuous. Then

(a) $h \circ f \sqsubseteq g \circ h \;\Rightarrow\; h.(\mu.f) \sqsubseteq \mu.g$,
(b) $k \circ f \sqsupseteq g \circ k \;\Rightarrow\; k.(\nu.f) \sqsupseteq \nu.g$.

Here a function f is *continuous* if $f.(\sqcup A) = (\sqcup a \in A \mid f.a)$ for every directed set A (the set A is *directed* if $(\forall x, y \in A \bullet \exists z \in A \bullet x \sqcup y \sqsubseteq z)$, i.e., if every pair of elements in A has an upper bound in A). Dually, a function f is *co-continuous* if $f.(\sqcap A) = (\sqcap a \in A \mid f.a)$ for every co-directed set A (and the set A is *co-directed* if $(\forall x, y \in A \bullet \exists z \in A \bullet z \sqsubseteq x \sqcap y)$).

6.1. Encoding Recursive Statements

Encoding also distributes into the recursive constructs:

Theorem 31. Assume that $f : (\Sigma \mapsto \Sigma) \to (\Sigma \mapsto \Sigma)$ and $g : (\Gamma \mapsto \Gamma) \to (\Gamma \mapsto \Gamma)$ are monotonic functions that map monotonic predicate transformers to monotonic predicate transformers, and that

$$(\forall S \in \Sigma \to_m \Sigma \bullet f.S \downarrow D \sqsubseteq g.(S \downarrow D))$$

Then

(a) $\mu.f \downarrow D \sqsubseteq \mu.g$ if D is strict and continuous, and
(b) $\nu.f \downarrow D \sqsubseteq \nu.g$.

Proof. For (a) we first note that continuity of D implies that the function $(\lambda X \bullet X \downarrow D)$ is continuous (the proof for this follows a straightforward pointwise extension argument). Then,

$$\begin{array}{ll} & (\mu.f) \downarrow D \sqsubseteq \mu.g \\ \Leftarrow & \{\text{fusion (Lemma 30 (a))}\} \\ & (\lambda X \bullet X \downarrow D) \circ f \sqsubseteq g \circ (\lambda X \bullet X \downarrow D) \\ \Leftarrow & \{\text{definitions, beta conversion}\} \\ & (\forall S \bullet f.S \downarrow D \sqsubseteq g.(S \downarrow D)) \end{array}$$

The proof of (b) is more direct:

$$\begin{array}{ll} & (\nu.f) \downarrow D \sqsubseteq \nu.g \\ \Leftarrow & \{\text{fixed-point induction}\} \\ & (\nu.f) \downarrow D \sqsubseteq g.((\nu.f) \downarrow D) \\ \equiv & \{\text{unfold fixed point}\} \\ & f.(\nu.f) \downarrow D \sqsubseteq g.((\nu.f) \downarrow D) \\ \Leftarrow & \{\text{specialisation}\} \\ & (\forall S \bullet f.S \downarrow D \sqsubseteq g.(S \downarrow D)) \end{array}$$

□

Since any useful programming notation will include facilities for μ-recursion (or some form of iteration, which is semantically a special case of recursion), the result in Theorem 31 shows that it makes sense to require that abstraction be strict and continuous.

For the fixpoint constructs, we also get calculational rules:

Theorem 32. Assume that $f : (\Sigma \mapsto \Sigma) \to (\Sigma \mapsto \Sigma)$ is a monotonic function that maps monotonic predicate transformers to monotonic predicate transformers, and that D is universally disjunctive. Then

(a) $\mu.f \downarrow D \sqsubseteq (\mu X \cdot f.(X \uparrow D) \downarrow D)$, and
(b) $\nu.f \downarrow D \sqsubseteq (\nu X \cdot f.(X \uparrow D) \downarrow D)$.

6.2. Iterations and Loops

As an illustration of Theorem 32 we show how a *weak iteration* S^* is encoded. This construct is defined as follows [BaW98]:

$$S^* \triangleq (\nu X \cdot S; X \sqcap \mathsf{skip})$$

and the operational intuition of S^* is that S is repeated some (demonically chosen) finite number of times. When D is universally disjunctive we have the following derivation:

$$\begin{array}{ll} & (\nu X \cdot S; X \sqcap \mathsf{skip}) \downarrow D \\ \sqsubseteq & \{\text{Theorem 32 (b)}\} \\ & (\nu X \cdot (S; (X \uparrow D) \sqcap \mathsf{skip}) \downarrow D) \\ \sqsubseteq & \{\text{Theorem 7}\} \\ & (\nu X \cdot (S \downarrow D); ((X \uparrow D) \downarrow D) \sqcap (\mathsf{skip} \downarrow D)) \\ \sqsubseteq & \{\text{Theorem 7 and Corollary 21}\} \\ & (\nu X \cdot (S \downarrow D); X \sqcap \mathsf{skip}) \end{array}$$

so we get

$$S^* \downarrow D \sqsubseteq (S \downarrow D)^*$$

This illustrates how Theorem 32 is applied: when the encoding is distributed all the way into the recursion expression, the decoding $X \uparrow D$ disappears, by Corollary 21 (the last step of the derivation).

A similar derivation can be used to derive the encoding rule for while-loops. First, we use the basic encoding rules to find that

$$(\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}) \downarrow D \sqsubseteq \mathsf{if}\ b'\ \mathsf{then}\ (S_1 \downarrow D)\ \mathsf{else}\ (S_2 \downarrow D)\ \mathsf{fi}$$

provided that $D.b \subseteq b' \subseteq D^\circ.b$. A derivation like the one for weak iteration then gives

$$(\mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}) \downarrow D \sqsubseteq \mathsf{while}\ \ b'\ \mathsf{do}\ (S \downarrow D)\ \mathsf{od}$$

provided that $D.b \subseteq b' \subseteq D^\circ.b$.

6.3. Decoding Recursive Statements

From the rules for encoding of fixpoints we also get rules for decoding.

Theorem 33. Assume that $f : (\Sigma \mapsto \Sigma) \to (\Sigma \mapsto \Sigma)$ and $g : (\Gamma \mapsto \Gamma) \to (\Gamma \mapsto \Gamma)$ are monotonic functions that map monotonic predicate transformers to monotonic predicate transformers, and that

$$(\forall S \cdot f.(S \uparrow D) \sqsubseteq g.S \uparrow D)$$

Then

(a) $\mu.f \sqsubseteq \mu.g \uparrow D$, if D is strict and continuous, and
(b) $\nu.f \sqsubseteq \nu.g \uparrow D$.

7. Examples and Applications

We shall now illustrate the concepts and rules developed in this paper with a number of examples.

7.1. Encoding with Context Information

The rules above tell us how assertions and guards are transformed in encodings and decodings. However, we are often interested in using assertions and guards as *context information*, to help calculate the encoding or decoding of an associated statement. For details about how assertions and guards can be introduced, propagated and eliminated from a program text, we refer to [BaW98].

In an encoding or decoding transformation of the form $S \downarrow D \sqsubseteq S'$ or $S \sqsubseteq S' \uparrow D$ we call S the *source* and S' the *target* of the transformation. If the source has the form $\{p\}; S$, then the state information represented by predicate p can be used as an assumption in the calculation of $\{p\}; S \downarrow D$. Consider as an example the rule from Theorem 14 (a):

$$[Q] \downarrow D \sqsubseteq [Q'] \equiv Q' \subseteq R \backslash (Q; R^{-1})$$

where $D = \{R\}$. The condition on the right-hand side of this rule can be rewritten as follows:

$$(\forall \sigma\ \gamma\ \gamma' \cdot R.\gamma.\sigma \wedge Q'.\gamma.\gamma' \ \Rightarrow\ (\exists \sigma' \cdot Q.\sigma.\sigma' \wedge R.\gamma'.\sigma'))$$

If the source instead had the form $\{p\}; [Q]$, then the rule would have the form

$$\{p\}; [Q] \downarrow D \sqsubseteq [Q'] \equiv Q' \subseteq R \backslash (|p|; Q; R^{-1})$$

and the condition on the right-hand side could be rewritten as

$$(\forall \sigma\ \gamma\ \gamma' \cdot p.\sigma \wedge R.\gamma.\sigma \wedge Q'.\gamma.\gamma' \ \Rightarrow\ (\exists \sigma' \cdot Q.\sigma.\sigma' \wedge R.\gamma'.\sigma'))$$

Note how the context information p has become an added assumption (a conjunct in the antecedent) that can help in the proof. The same argument holds for all other rules: an assertion in the source of an encoding or decoding transformation can be used as an assumption in the proof of the transformation.

The use of guards is dual to the use of assertions: a guard statement in the target of a transformation can be used as context information. Consider the same example as for assertions above. If the target has the form $[p']; [Q']$, then the rule becomes

$$[Q] \downarrow D \sqsubseteq [p]; [Q'] \equiv Q' \subseteq (|p'|; R) \backslash (Q; R^{-1})$$

where the condition on the right-hand side can be rewritten as

$$(\forall \sigma\ \gamma\ \gamma' \cdot p'.\gamma \wedge R.\gamma.\sigma \wedge Q'.\gamma.\gamma' \ \Rightarrow\ (\exists \sigma' \cdot Q.\sigma.\sigma' \wedge R.\gamma'.\sigma'))$$

Again, the context information appears in the antecedent of the condition, i.e., as an assumption.

7.2. Encoding with Program Variables

The rules for encoding and decoding that we have given so far have reduced reasoning on the predicate transformer level to reasoning on the lower levels in the predicate transformer hierarchy: predicates, functions and relations. These rules are often elegant and algebraic. However, if we want to apply the rules in program derivation, then we need to translate the rules to the level of program variables. It is not our aim to derive a complete collection of such rules, but we shall give an example of how such rules are derived. In practice, what we get are syntactic rules for data refinement, much like those that have been described in detail elsewhere [Bac80, ChU89, MoG90, Mor89].

As an example, consider the rule for encoding a demonic assignment with a functional abstraction (Theorem 16 (c)):

$$[P] \downarrow (\{r\}; \langle f \rangle) = \{r\}; [|f|; P; f^{-1}]; [r]$$

In assignment notation, we have

$$\begin{aligned} & [x := x' \mid b] \downarrow (\{c\}; \langle x/y := e \rangle) \\ = & \{\text{rule above}\} \\ & \{c\}; [(x/y := x \mid x = e); (x := x' \mid b); (y/x := y \mid x = e)]; [c] \end{aligned}$$

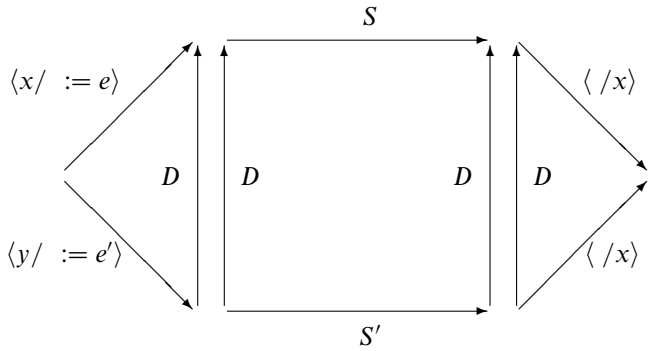

Fig. 5. Simulation between blocks.

$= \{\text{merge assignments and simplify}\}$
$\{c\}; [(x/y := x' \mid b[x := e]); (y/x := y \mid x = e)]; [c]$
$= \{\text{merge assignments and simplify}\}$
$\{c\}; [y := y' \mid b[x, x' := e, e[y := y']]]; [c]$
$= \{\text{rewrite guard as update and merge}\}$
$\{c\}; [y := y' \mid b[x, x' := e, e[y := y']] \wedge c[y := y']]$

Thus we get the following rule for encoding in terms of program variables:

$$[x := x' \mid b] \downarrow (\{c\}; \langle x/y := e\rangle) = \{c\}; [y := y' \mid b[x, x' := e, e'] \wedge c']$$

where e' abbreviates $e[y := y']$ and c' abbreviates $c[y := y']$.

A similar derivation gives the following rule for forward data refinement when the abstraction is of the form $\{x/y := x \mid c\}$ where c is a Boolean expression relating the abstract variable x and the concrete variable y:

$$[x := x' \mid b] \downarrow \{x/y := x \mid c\} \sqsubseteq \{\exists x \cdot c\}; [y := y' \mid (\forall x \cdot c \Rightarrow (\exists x' \cdot c' \wedge b))]$$

where c' abbreviates $c[x, y := x', y']$. In this case the derivation is as follows:

$[x := x' \mid b] \downarrow \{x/y := x \mid c\}$
$= \{\text{encoding rule (Theorem 15)}\}$
$\{\mathsf{im}.(y/x := y \mid c)^{-1}.\mathsf{true}\}; [((x/y := x \mid c); |\mathsf{true}|) \backslash ((x := x' \mid b); (y/x := y \mid c))]$
$= \{\text{merge assignments and simplify}\}$
$\{\exists x \cdot c \wedge \mathsf{T}\}; [(x/y := x \mid c) \backslash (y/x := y \mid (\exists x' \cdot b \wedge c[x := x']])$
$= \{\text{quotient with assignments, simplify}\}$
$\{\exists x \cdot c\}; [y := y' \mid (\forall x \cdot c \Rightarrow (\exists x' \cdot b \wedge c[x, y := x', y']))]$

As before, we have used the rules from Appendix A to manipulate assignments.

7.3. Example: Block Refinement

Consider a block refinement of the form

$$\mathsf{begin\ var}\ x := e; S\ \mathsf{end} \sqsubseteq \mathsf{begin\ var}\ y := e'; S'\ \mathsf{end}$$

A standard method for proving such a refinement is by *simulation*, which in our framework amounts to finding an abstraction statement D such that the following three conditions hold:

$$\begin{array}{rcll} \langle x/ := e\rangle & \sqsubseteq & \langle y/ := e'\rangle; D & (\textit{initialisation simulation}) \\ D; S & \sqsubseteq & S'; D & (\textit{body simulation}) \\ D; \langle /x\rangle & \sqsubseteq & \langle /y\rangle & (\textit{finalisation simulation}) \end{array}$$

The block refinement can then be illustrated by a subcommuting diagram, as in Fig. 5. The results of this paper show that the body simulation condition is equivalent to $S \downarrow D \sqsubseteq S'$. The initialisation and finalisation conditions can also be simplified when the abstraction statement is an (angelic or demonic) assignment, as the following example shows.

We show some details of a simple block refinement, to illustrate data refinement with program variables. Consider the block

$$\textsf{begin var } x := \emptyset; \ldots; \langle x := x \cup \{z\}\rangle; \ldots \textsf{ end}$$

where the ellipses stand for statements that we do not consider further. The set (x) is to be implemented by an array (variables n and y), according to the abstraction statement

$$D = \langle x/n, y := \textsf{arrset}.\, n.\, y\rangle$$

where $\textsf{arrset}.\, n.\, y$ stands for the set $\{a \mid \exists i < n \cdot y[i] = a\}$ (we assume that y is a potentially infinite array, indexed by the natural numbers). This abstraction statement can be justified if elements are never removed from the set.

First we use the rule derived in Section 7.2 to calculate the data refinement of the assignment $\langle x := x \cup \{z\}\rangle$. We have

$$\begin{array}{ll} & \langle x := x \cup \{a\}\rangle \downarrow D \\ = & \{\text{rewrite deterministic assignment as demonic assignment}\} \\ & [x := x' \mid x' = x \cup \{z\}] \downarrow \langle x/n, y := \textsf{arrset}.\, n.\, y\rangle \\ = & \{\text{rule from Section 7.2}\} \\ & [n, y := n', y' \mid \textsf{arrset}.\, n'.\, y' = \textsf{arrset}.\, n.\, y \cup \{z\}] \\ \sqsubseteq & \{\text{standard refinement derivation}\} \\ & \langle n, y[n] := n + 1, z\rangle \end{array}$$

Note that we do not need to manipulate assignments that replace variables; this is taken care of by the rule for encoding. Thus, the resulting derivation follows the same outline as traditional data refinement derivations in the calculational style [MoG90, BaW89]. Other statements inside the block body can be encoded by means of similar calculations.

The block initialisation is calculated by noting that

$$\langle x/\ := e\rangle \sqsubseteq \langle y/\ := e'\rangle; D \equiv \langle x/\ := e\rangle; \overline{D} \sqsubseteq \langle y/\ := e'\rangle$$

when D is universally disjunctive. We have

$$\begin{array}{ll} & \langle x/\ := \emptyset\rangle; \overline{D} \\ = & \{\text{rewrite assignment, rule for inverses}\} \\ & [x/\ := x \mid x = \emptyset]; [n, y/x := n, y \mid x = \textsf{arrset}.\, n.\, y] \\ = & \{\text{homomorphism, merge assignments (see Appendix A)}\} \\ & [n, y/\ := n, y \mid \textsf{arrset}.\, n.\, y = \emptyset] \\ \sqsubseteq & \{\text{make assignment deterministic}\} \\ & \langle n, y/\ := 0, \textsf{arb}\rangle \end{array}$$

where arb stands for some arbitrary element (it does not matter how y is initialised). A similar calculation shows that

$$\langle\ /x\rangle; D \sqsubseteq \langle\ /y\rangle$$

and we have the refinement

$$\begin{array}{ll} & \textsf{begin var } x := \emptyset; \ldots; \langle x := x \cup \{z\}\rangle; \ldots \textsf{ end} \\ \sqsubseteq & \textsf{begin var } n, y := 0, \textsf{arb}; \ldots; \langle n, y[n] := n + 1, z\rangle; \ldots \textsf{ end} \end{array}$$

7.4. Least Data Refinement

The following example illustrates the difference between the least structure-preserving data refinement and the all-over least data refinement. It also shows the calculation of an encoding using the definition. We consider the assignment $\langle a := a + 1 \rangle$ under abstraction $\{a/c := a \mid c = a \operatorname{div} 2\}$. We have

$$
\begin{aligned}
& \langle a := a + 1 \rangle \downarrow \{a/c := a \mid c = a \operatorname{div} 2\} \\
= & \{\text{Theorem 10}\} \\
& \{a/c := a \mid c = a \operatorname{div} 2\}; \langle a := a + 1 \rangle; [c/a := c \mid c = a \operatorname{div} 2] \\
= & \{\text{rewrite demonic update as a deterministic statement}\} \\
& \{a/c := a \mid c = a \operatorname{div} 2\}; \langle a := a + 1 \rangle; \langle c/a := a \operatorname{div} 2 \rangle \\
= & \{\text{merge assignments}\} \\
& \{a/c := a \mid c = a \operatorname{div} 2\}; \langle c/a := (a + 1) \operatorname{div} 2 \rangle \\
= & \{\text{rewrite deterministic statement as angelic update}\} \\
& \{a/c := a \mid c = a \operatorname{div} 2\}; \{c/a := c \mid c = (a + 1) \operatorname{div} 2\} \\
= & \{\text{homomorphism}\} \\
& \{(a/c := a \mid c = a \operatorname{div} 2); (c/a := c \mid c = (a + 1) \operatorname{div} 2)\} \\
= & \{\text{merge assignments}\} \\
& \{c := c' \mid \exists a \cdot c = a \operatorname{div} 2 \wedge c' = (a + 1) \operatorname{div} 2\} \\
= & \{\text{arithmetic, predicate calculus}\} \\
& \{c := c' \mid \exists a \cdot (a = 2c \wedge c' = c) \vee (a = 2c + 1 \wedge c' = c + 1)\} \\
= & \{\text{predicate calculus}\} \\
& \{c := c' \mid c' = c \vee c' = c + 1\} \\
= & \{\text{rewrite angelic update as angelic choice}\} \\
& \mathsf{skip} \sqcup \langle c := c + 1 \rangle
\end{aligned}
$$

Here we have used the rules in Appendix A to manipulate assignments.

Thus the least data refinement of $\langle a := a + 1 \rangle$ is an angelic choice between skip and $\langle c := c + 1 \rangle$. If we apply Theorem 14 (b) (in its syntactic form as given in Section 7.2) then we get

$$
\begin{aligned}
& \langle a := a + 1 \rangle \downarrow \{a/c := a \mid c = a \operatorname{div} 2\} \\
\sqsubseteq & \{\text{syntactic rule for encoding}\} \\
& [c := c' \mid (\forall a \cdot c = a \operatorname{div} 2 \Rightarrow (\exists a' \cdot c' = a' \operatorname{div} 2 \wedge a' = a + 1))] \\
= & \{\text{predicate calculus}\} \\
& [c := c' \mid (\forall a \cdot c = a \operatorname{div} 2 \Rightarrow c' = (a + 1) \operatorname{div} 2)] \\
= & \{\text{arithmetic, predicate calculus}\} \\
& [c := c' \mid (\forall a \cdot (a = 2c \Rightarrow c' = (a + 1) \operatorname{div} 2) \wedge (a = 2c + 1 \Rightarrow c' = (a + 1) \operatorname{div} 2))] \\
= & \{\text{predicate calculus}\} \\
& [c := c' \mid c' = (2c + 1) \operatorname{div} 2 \wedge c' = (2c + 2) \operatorname{div} 2] \\
= & \{\text{arithmetic}\} \\
& [c := c' \mid \mathsf{F}] \\
= & \{\text{definition of magic}\} \\
& \mathsf{magic}
\end{aligned}
$$

which shows that the least structure-preserving data refinement (and in fact, the least conjunctive data refinement) is magic.

7.5. Abstracting Properties

As an application of decoding, we shall now show how a correctness property can be translated to a more abstract level.

Theorem 34. Assume that D is a universally disjunctive abstraction. Then

$$p \,\{\!|\, S \uparrow D \,|\!\}\, \overline{D}.q \Rightarrow D.p \,\{\!|\, S \,|\!\}\, q$$

Theorem 34 gives us the following rule for reducing a correctness condition using decoding:

$$\frac{p' \subseteq D.p \qquad p \,\{\!|\, S \,|\!\}\, q \qquad q \subseteq \overline{D}.q' \qquad S \sqsubseteq S' \uparrow D}{p' \,\{\!|\, S' \,|\!\}\, q'}$$

The idea is that we are free to choose an abstraction D that suits the situation. As a small example, consider the correctness condition

$$\mathsf{true} \,\{\!|\, \mathsf{if}\ x < y\ \mathsf{then}\ x, y := y, x\ \mathsf{else}\ x := y\ \mathsf{fi} \,|\!\}\, (x \geqslant y) \qquad (*)$$

Since the exact values of x and y are not important, it should be possible to reduce this to a correctness condition where the unnecessary details about x and y are abstracted away.

Recall from Section 2.3 that the conditional statement in (*) can be written as

$$S' = \{x < y\}; \langle x, y := y, x\rangle \sqcup \{x \geqslant y\}; \langle x := y\rangle$$

Next, we introduce the abstraction $D = \langle b/x, y := , x \geqslant y)\rangle$ and start calculating the decoding. For the first assertion we have

$$\begin{aligned}
&(x < y)\\
=\ &\{\text{arithmetic}\}\\
&\neg(x \geqslant y)\\
=\ &\{\text{predicate calculus}\}\\
&(\neg b)[b := (x \geqslant y)]\\
=\ &\{\text{assignment property (Appendix A)}\}\\
&\mathsf{true} \cap (b/x, y := (x \geqslant y)); (\neg b)
\end{aligned}$$

so $\{\neg b\} \sqsubseteq \{x < y\} \uparrow D$ by Theorem 29 (a). For the first assignment we then have (with context assumption $x < y$)

$$\begin{aligned}
&(x, y := y, x); (b/x, y := (x \geqslant y))\\
=\ &\{\text{merge assignments}\}\\
&(b/x, y := (y \geqslant x))\\
=\ &\{\text{assumption } x < y\}\\
&(b/x, y := \mathsf{T})\\
=\ &\{\text{split assignment}\}\\
&(b/x, y := (x \geqslant y)); (b := \mathsf{T})
\end{aligned}$$

so $\langle b := \mathsf{T}\rangle \sqsubseteq ([x < y]; \langle x, y := y, x\rangle) \uparrow D$ by Theorem 29 (c) (where the guard statement is context information).

For the second assertion we similarly find $\{b\} \sqsubseteq \{x \geqslant y\} \uparrow D$ and then $\mathsf{skip} \sqsubseteq [x \geqslant y]; \langle x := y\rangle \uparrow D$ (where the guard statement is again context information). Thus we have

$$\{\neg b\}; \langle b := \mathsf{T}\rangle \sqcup \{b\}; \mathsf{skip} \sqsubseteq S' \uparrow D$$

by Theorem 23 (f) and (g). Furthermore, we find $\mathsf{true} \subseteq D.\mathsf{true}$ and also $\overline{D}.(x \geqslant y) = b$. Thus by the rule derived from Theorem 34 we have reduced the correctness condition (*) to the following:

$$\mathsf{true} \,\{\!|\, \mathsf{if}\ \neg b\ \mathsf{then}\ b := \mathsf{T}\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi} \,|\!\}\, b$$

which is then easy to verify. This illustrates how a correctness condition can be reduced to a finite-state level, where automatic verification methods can be used.

8. Conclusion

We have defined an encoding and a decoding operator on predicate transformers and investigated their place in the predicate transformer hierarchy. In particular, we investigated the algebraic properties of encoding and decoding and their application to data refinement. The investigation has led to proliferation of similar and linked results. In order to see the structure, we can consider the results in the light of three independent dichotomies:

1. *Encoding vs. decoding:* Encoding translates an abstract program into a concrete one, so that encoding steps can be included in calculational program derivations. The dual operation is decoding, which translates a concrete program into a more abstract one.
2. *Semantics vs. syntax:* The rules for encoding and decoding are first developed on a semantic level, as properties of predicate transformers that are reduced to properties of the predicates, functions and relations that the predicate transformers are built from. From these rules we then derive syntactic rules that allow us to apply encoding and decoding to programs that are described in terms of program variables and assignments to them.
3. *Forward vs. backward data refinement:* At the abstract level, encoding is defined as a single concept, using the notion of a data refinement with a general abstraction statement. However, in order to derive rules that reduce encoding to the lower levels in the predicate transformer hierarchy there was a need to separate forward data refinement (disjunctive abstraction statements) and backward data refinement (conjunctive abstraction statements).

The idea of calculating data refinements is not new, but most existing work has concentrated on the syntactic level, with rules that explicitly talk about the program variables and the Boolean expressions involved [Bac80, GaM91, Mor89]. Hoare, He and Sanders [HoHS87] make use of the *weakest pre-specification* [HH87] and a dual strongest post-specification to calculate data refinements at an algebraic level. However, they work in a relational framework, which means that they do not model non-termination properly, and they have to handle forward and backward data refinement separately. Data refinement of a statement S with respect to a relation R is expressed by Back and von Wright as $\{R\}; S; [R^{-1}]$ [BaW89]. These formulations correspond to encoding in the special case of forward data refinement, as does the data refinement calculator approach of Morris [Mor89] and Gardiner and Morgan [MoG90]. An analogue of our decoding of a statement S was considered previously by Back [Bac80] in the form $x/y.R; S; y/x.R$ (in a framework without miracles or angelic non-determinism).

Traditionally, theories and methods of data refinement have only considered the standard (forward) notion of data refinement. Gardiner and Morgan were the first to handle forward and backward data refinement in a single rule [GaM93], but they did not consider the calculation of data refinement in this general framework. As noted above, the problem of finding the most general data refinement (also referred to as the *weakest simulation*) has also been handled extensively in a relational framework [dRE98], but the dual problem (which we solve by decoding) does not have a general solution when programs are modelled as relations. Our use of inverse statements in data refinement goes back to older work [Bac89, BaW89, vWr94], but the combination of all the three dichotomies mentioned above has to our knowledge not been considered before.

References

[Bac80] Back, R.: *Correctness Preserving Program Refinements: Proof Theory and Applications*, *Mathematical Centre Tracts*, vol. 131. Mathematical Centre, Amsterdam, 1980.

[Bac88] Back, R.: A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[Bac89] Back, R.: Changing data representation in the refinement calculus. In *21st Hawaii International Conference on System Sciences*, 1989.

[BGW97] Back, R., Grundy, J. and von Wright, J.: Structured calculational proof. *Formal Aspects of Computing*, 9:469–483, 1997.

[BaW89] Back, R. and von Wright, J.: Refinement calculus, Part I: Sequential programs. In *REX Workshop for Refinement of Distributed Systems*, Vol. 430 of *Lecture Notes in Computer Science*, Springer, Berlin, 1989.

[BaW93] Back, R. and von Wright, J.: Statement inversion and strongest postcondition. *Science of Computer Programming*, 20:223–251, 1993.

[BaW98] Back, R. and von Wright, J.: *Refinement Calculus: A Systematic Introduction*. Springer, Berlin, 1998.

[ChU89] Chen, W. and Udding, J.: Towards a calculus of data refinement. In *Mathematics of Program Construction*, Vol. 375 of *Lecture Notes in Computer Science*, Springer, Berlin, 1989.

[Dij76] Dijkstra, E.: *A Discipline of Programming*. Prentice-Hall International, Englewood Cliffs, NJ, 1976.

[dRE98] de Roever, W.-P. and Engelhardt, K.: *Data Refinement: Model-Oriented Proof Methods and their Comparison*, Cambridge Tracts in Theoretical Computer Science 47. Cambridge University Press, Cambridge, UK, 1998.
[GaM91] Gardiner, P. and Morgan, C.: Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.
[GaM93] Gardiner, P. and Morgan, C.: A single complete rule for data refinement. *Formal Aspects of Computing*, 5:367–383, 1993.
[HH87] Hoare, C. and He, J.: The weakest prespecification. *Information Processing Letters*, 24:127–132, 1987.
[HoHS87] Hoare, C., He, J., and Sanders, J.: Prespecification in data refinement. *Information Processing Letters*, 25:71–76, 1987.
[Hoa72] Hoare, C.: Proofs of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.
[JiH90] Jifeng, H. and Hoare, C.: Data refinement in a categorical setting. Techn. Rep. PRG 90, Oxford University Computing Laboratory, 1990.
[MoG90] Morgan, C. and Gardiner, P.: Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.
[Mor88] Morgan, C.: The specification statement. *ACM Transactions on Programming Languages and Systems*, 10:403–419, 1988.
[Mor89] Morris, J.: Laws of data refinement. *Acta Informatica*, 26:287–308, 1989.
[vWr94] von Wright, J.: The lattice of data refinement. *Acta Informatica*, 31:105–135, 1994.

A. Manipulating Assignments

The basic rules for merging and splitting assignments are essentially syntactic versions of the definitions of basic operators on relations and of functions. For the ordinary relational assignment we have the following *merge/split rules*, one for the simple case and one for the general case:

$$(x := x' \mid b);(x := x' \mid c) = (x := x' \mid (\exists x'' \cdot b[x' := x''] \wedge c[x := x'']))$$
$$(y/x := y' \mid b);(z/y := z' \mid c) = (z/x := z' \mid (\exists y' \cdot b \wedge c[y := y']))$$

In addition to relational composition, relational quotient will be used in a number of rules. Thus, we need rules for working with quotients in the assignment notation as well:

$$(x := x' \mid b)\backslash(x := x' \mid c) = (x := x' \mid (\forall x'' \cdot b[x' := x''] \Rightarrow c[x := x'']))$$
$$(y/x := y' \mid b)\backslash(z/y := z' \mid c) = (z/x := z' \mid (\forall y' \cdot b \Rightarrow c[y := y']))$$

For functional assignments, we get the following special cases of the merge/split rules:

$$(x := d);(x := e) = (x := e[x := d])$$
$$(y/x := d);(z/y := e) = (z/x := e[y := d])$$

All these rules require that all assignments involved have the same target (the *target* of an assignment is the collection of variables to which values are assigned). We can use the following *frame rules* to make the targets of two assignments equal.

$$(x := x' \mid b) = (x, z := x', z' \mid b \wedge z' = z)$$
$$(y/x := y' \mid b) = (y, z/x, z := y', z' \mid b \wedge z' = z)$$
$$(x := e) = (x, z := e, z)$$
$$(y/x := e) = (y, z/x, z := e, z)$$

By using the merge/split rules twice in succession, we can reorder assignments. For example, the following commutativity rule can be proved using merge/split and frame rules:

$$(y/x := y' \mid d);(z/w := z' \mid e) = (z/w := z' \mid e);(y/x := y' \mid d)$$

provided that w does not occur free in d and x does not occur free in e. Finally, we note the following rule for evaluation of an expression after an assignment to a variable:

$$(x := d);e = e[x := d]$$

Note that an expression is really a function that maps states to values; thus it can be composed with an assignment (which is a function that maps states to states).

B. Additional Proofs

Proof of Lemma 4

Since the pair $(\mathsf{im}.R, \overline{\mathsf{im}}.R)$ is a Galois connection the definitions immediately give us that $[R^{-1}]$ is the inverse of $\{R\}$. Now (b) follows from this, by setting $R = |r|;|f|$.

Proof of Theorem 6

Assume that $S \sqsubseteq S'$ holds. Then

$$
\begin{array}{ll}
 & S \downarrow D \sqsubseteq S' \downarrow D \\
\Leftarrow & \{\text{property (encode2)}\} \\
 & D;S \sqsubseteq (S' \downarrow D);D \\
\Leftarrow & \{\text{assumption } S \sqsubseteq S'\} \\
 & D;S' \sqsubseteq (S' \downarrow D);D \\
\equiv & \{\text{property (encode1)}\} \\
 & \mathsf{T}
\end{array}
$$

Proof of Theorem 7

The proofs of the different parts of this theorem all follow the same general pattern, making use of the specific algebraic properties of the statement in question. For (a) we assume that D is strict. Then

$$
\begin{array}{ll}
 & \mathsf{abort} \downarrow D \sqsubseteq \mathsf{abort} \\
\equiv & \{\text{Theorem 5}\} \\
 & D;\mathsf{abort} \sqsubseteq \mathsf{abort};D \\
\equiv & \{D \text{ strict implies } D;\mathsf{abort} = \mathsf{abort}, \text{ general rule } \mathsf{abort};D = \mathsf{abort}\} \\
 & \mathsf{abort} \sqsubseteq \mathsf{abort} \\
\equiv & \{\text{refinement is reflexive}\} \\
 & \mathsf{T}
\end{array}
$$

which shows that $\mathsf{abort} \downarrow D = \mathsf{abort}$. For (b) we have

$$
\begin{array}{ll}
 & \mathsf{skip} \downarrow D \sqsubseteq \mathsf{skip} \\
\equiv & \{\text{Theorem 5}\} \\
 & D;\mathsf{skip} \sqsubseteq \mathsf{skip};D \\
\equiv & \{\mathsf{skip} \text{ is unit}\} \\
 & D \sqsubseteq D \\
\equiv & \{\text{refinement is reflexive}\} \\
 & \mathsf{T}
\end{array}
$$

For (c) we assume that D is strict and terminating. We then have for arbitrary S:

$$
\begin{array}{ll}
 & \mathsf{magic} \downarrow D \sqsubseteq S \\
\equiv & \{\text{Theorem 5}\} \\
 & D;\mathsf{magic} \sqsubseteq S;D \\
\equiv & \{D \text{ terminating implies } D;\mathsf{magic} = \mathsf{magic}\} \\
 & \mathsf{magic} \sqsubseteq S;D \\
\equiv & \{D \text{ is strict}\} \\
 & S = \mathsf{magic}
\end{array}
$$

which shows that $\mathsf{magic} \downarrow D = \mathsf{magic}$. For (d) we have

$\{p\} \downarrow D \sqsubseteq \{p'\}$
$\equiv$ {Theorem 5}
$D; \{p\} \sqsubseteq \{p'\}; D$
$\equiv$ {definitions}
$(\forall q \cdot D.(p \cap q) \subseteq p' \cap D.q)$
$\equiv$ {mutual implication}
• $(\forall q \cdot D.(p \cap q) \subseteq p' \cap D.q)$
$\Rightarrow$ {specialise $q := p$}
$D.p \subseteq p' \cap D.p$
$\equiv$ {lattice property}
$D.p \subseteq p'$
• $(\forall q \cdot D.(p \cap q) \subseteq p' \cap D.q)$
$\Leftarrow$ {general rule $S.(p \cap q) \subseteq S.p \cap S.q$}
$(\forall q \cdot D.p \cap D.q \subseteq p' \cap D.q)$
$\Leftarrow$ {monotonicity}
$D.p \subseteq p'$
· $D.p \subseteq p'$

Next, for (e) when D is universally disjunctive:

$[p] \downarrow D \sqsubseteq [p']$
$\equiv$ {Theorem 5}
$D; [p] \sqsubseteq [p']; D$
$\equiv$ {definitions}
$(\forall q \cdot D.(\neg p \cup q) \subseteq \neg p' \cup D.q)$
$\equiv$ {D assumed disjunctive}
$(\forall q \cdot D.(\neg p) \cup D.q \subseteq \neg p' \cup D.q)$
$\equiv$ {mutual implication}
• $(\forall q \cdot D.\neg p \cup D.q \subseteq \neg p' \cup D.q)$
$\Rightarrow$ {specialise $q :=$ false}
$D.(\neg p) \cup D.\mathsf{false} \subseteq \neg p' \cup D.\mathsf{false}$
$\equiv$ {D assumed strict}
$D.(\neg p) \subseteq \neg p'$
$\equiv$ {antimonotonicity of negation, definition of dual}
$D^\circ.p \subseteq p'$
• $(\forall q \cdot D.(\neg p) \cup D.q \subseteq \neg p' \cup D.q)$
$\Leftarrow$ {monotonicity}
$D.(\neg p) \subseteq \neg p'$
$\equiv$ {antimonotonicity of negation, definition of dual}
$D^\circ.p \subseteq p'$
· $D.p \subseteq p'$

Now consider (e). Before the main proof we note the following for arbitrary conjunctive S:

$S.(\neg p \cup q) \subseteq \neg S.p \cup S.q$
$\equiv$ {shunting rule}
$S.p \cap S.(\neg p \cup q) \subseteq S.q$
$\equiv$ {S conjunctive}
$S.(p \cap (\neg p \cup q)) \subseteq S.q$

$\equiv$ {lattice properties}
$S.(p \cap q) \subseteq S.q$
$\equiv$ {S monotonic}
T

Now assume that D is universally conjunctive. Then we have

$[p] \downarrow D \sqsubseteq [p']$
$\equiv$ {Theorem 5}
$D; [p] \sqsubseteq [p']; D$
$\equiv$ {definitions}
$(\forall q \cdot D.(\neg p \cup q) \subseteq \neg p' \cup D.q)$
$\equiv$ {mutual implication}
• $(\forall q \cdot D.(\neg p \cup q) \subseteq \neg p' \cup D.q)$
$\Rightarrow$ {specialise $q := p$}
$D.\mathsf{true} \subseteq \neg p' \cup D.p)$
$\equiv$ {D terminating implies $D.\mathsf{true} = \mathsf{true}$, shunting}
$p' \subseteq D.p$
• $(\forall q \cdot D.(\neg p \cup q) \subseteq \neg p' \cup D.q)$
$\Leftarrow$ {derivation above, D assumed conjunctive}
$(\forall q \cdot \neg D.p \cup D.q \subseteq \neg p' \cup D.q)$
$\Leftarrow$ {monotonicity}
$\neg D.p \subseteq \neg p'$
$\equiv$ {property of complement}
$p' \subseteq D.p$
• $p' \subseteq D.p$

For (f) we have

$(S_1; S_2) \downarrow D \sqsubseteq (S_1 \downarrow D); (S_2 \downarrow D)$
$\equiv$ {Theorem 5}
$D; S_1; S_2 \sqsubseteq (S_1 \downarrow D); (S_2 \downarrow D); D$
$\Leftarrow$ {property (encode1)}
$D; S_1; S_2 \sqsubseteq (S_1 \downarrow D); D; S_2$
$\Leftarrow$ {monotonicity}
$D; S_1 \sqsubseteq (S_1 \downarrow D); D$
$\equiv$ {property (encode1)}
T

Now, for (g) we have

$(\sqcap i \in I \cdot S_i) \downarrow D \sqsubseteq (\sqcap i \in I \cdot S_i \downarrow D)$
$\equiv$ {Theorem 5}
$D; (\sqcap i \in I \cdot S_i) \sqsubseteq (\sqcap i \in I \cdot S_i \downarrow D); D$
$\equiv$ {distributivity}
$D; (\sqcap i \in I \cdot S_i) \sqsubseteq (\sqcap i \in I \cdot (S_i \downarrow D); D)$
$\Leftarrow$ {D monotonic implies $D; (\sqcap i \in I \cdot S_i) \sqsubseteq (\sqcap i \in I \cdot D; S_i)$}
$(\sqcap i \in I \cdot D; S_i) \sqsubseteq (\sqcup i \in I \cdot (S_i \downarrow D); D)$
$\equiv$ {property (encode1)}
T

Finally, for (h) we have

$$
\begin{array}{ll}
& (\sqcup i \in I \cdot S_i) \downarrow D \sqsubseteq (\sqcup i \in I \cdot S_i \downarrow D) \\
\Leftarrow & \{\text{property (encode2)}\} \\
& D ; (\sqcup i \in I \cdot S_i) \sqsubseteq (\sqcup i \in I \cdot S_i \downarrow D) ; D \\
\equiv & \{\text{distributivity; } D \text{ universally disjunctive}\} \\
& (\sqcup i \in I \cdot D ; S_i) \sqsubseteq (\sqcup i \in I \cdot (S_i \downarrow D) ; D) \\
\equiv & \{\text{property (encode1)}\} \\
& \mathsf{T}
\end{array}
$$

Proof of Theorem 8

For (a) we have

$$
\begin{array}{ll}
& S \downarrow \mathsf{abort} \sqsubseteq \mathsf{abort} \\
\equiv & \{\text{Theorem 5}\} \\
& \mathsf{abort} ; S \sqsubseteq \mathsf{abort} ; \mathsf{abort} \\
\equiv & \{\text{properties of } \mathsf{abort}\} \\
& \mathsf{T}
\end{array}
$$

Next, for (b) we have

$$
\begin{array}{ll}
& S \downarrow \mathsf{skip} \sqsubseteq S \\
\equiv & \{\text{Theorem 5}\} \\
& \mathsf{skip} ; S \sqsubseteq S ; \mathsf{skip} \\
\equiv & \{\mathsf{skip} \text{ is unit}\} \\
& \mathsf{T}
\end{array}
$$

and

$$
\begin{array}{ll}
& S \sqsubseteq S \downarrow \mathsf{skip} \\
\equiv & \{\mathsf{skip} \text{ is unit}\} \\
& \mathsf{skip} ; S \sqsubseteq (S \downarrow \mathsf{skip}) ; \mathsf{skip} \\
\equiv & \{\text{property (encode1)}\} \\
& \mathsf{T}
\end{array}
$$

For (c) we have

$$
\begin{array}{ll}
& S \downarrow \mathsf{magic} \sqsubseteq S' \\
\equiv & \{\text{Theorem 5}\} \\
& \mathsf{magic} ; S \sqsubseteq S' ; \mathsf{magic} \\
\equiv & \{\text{definitions}\} \\
& (\forall q \cdot \mathsf{true} \subseteq S' . \mathsf{true}) \\
\equiv & \{\mathsf{chaos} \text{ is the least of all terminating predicate transformers}\} \\
& \mathsf{chaos} \sqsubseteq S'
\end{array}
$$

which shows that $S \downarrow \mathsf{magic} = \mathsf{chaos}$. Finally, for (d) we have

$$
\begin{array}{ll}
& S \downarrow (D_1 \sqcup D_2) \sqsubseteq (S \downarrow D_1) \sqcup (S \downarrow D_2) \\
\equiv & \{\text{Theorem 5}\} \\
& (D_1 \sqcup D_2) ; S \sqsubseteq ((S \downarrow D_1) \sqcup (S \downarrow D_2)) ; (D_1 \sqcup D_2) \\
\equiv & \{\text{distributivity}\} \\
& D_1 ; S \sqcup D_2 ; S \sqsubseteq (S \downarrow D_1) ; D_1 \sqcup (S \downarrow D_1) ; D_2 \sqcup (S \downarrow D_2) ; D_1 \sqcup (S \downarrow D_2) ; D_2
\end{array}
$$

$\Leftarrow$ {monotonicity of join}
$D_1; S \sqcup D_2; S \sqsubseteq (S \downarrow D_1); D_1 \sqcup (S \downarrow D_2); D_2$
$\equiv$ {property (encode1)}
T

Proof of Theorem 10

$S \downarrow D \sqsubseteq D; S; \overline{D}$
$\equiv$ {Theorem 5}
$D; S \sqsubseteq D; S; \overline{D}; D$
$\equiv$ {property of inverses: $\mathsf{skip} \sqsubseteq \overline{D}; D$}
T

and

$D; S; \overline{D} \sqsubseteq S \downarrow D$
$\equiv$ {shunting (Lemma 3)}
$D; S \sqsubseteq (S \downarrow D); D$
$\equiv$ {property (encode1)}
T

Proof of Theorem 11

If D_1 is universally disjunctive, then

$(S \downarrow D_1) \downarrow D_2 \sqsubseteq S \downarrow (D_2; D_1)$
$\equiv$ {Theorems 10 and 5}
$D_2; D_1; S; \overline{D_1} \sqsubseteq (S \downarrow (D_2; D_1)); D_2$
$\equiv$ {shunting}
$D_2; D_1; S \sqsubseteq (S \downarrow (D_2; D_1)); D_2; D_1$
$\equiv$ {Theorem 5}
T

Similarly, if D_2 is universally disjunctive, then

$(S \downarrow D_1) \downarrow D_2 \sqsubseteq S \downarrow (D_2; D_1)$
$\equiv$ {Theorem 5, shunting}
$S \downarrow D_1 \sqsubseteq \overline{D_2}; (S \downarrow (D_2; D_1)); D_2$
$\equiv$ {Theorem 5, shunting back}
$D_2; D_1; S \sqsubseteq S \downarrow (D_2; D_1); D_2; D_1$
$\equiv$ {Theorem 5}
T

Refinement in the other direction was proved in Theorem 9.

Proof of Theorem 13

For (c) we have

$(\mathsf{magic} \downarrow D).q.\gamma$
$\equiv$ {D is the decoding $\{R\}$, Theorem 10}
$(\{R\}; \mathsf{magic}; [R^{-1}]).q.\gamma$

$\equiv$ {definition of magic}
$\{R\}.\,\mathsf{true}.\,\gamma$
$\equiv$ {definition of angelic update}
$(\exists \sigma \cdot R.\,\gamma.\,\sigma)$
$\equiv$ {definition of domain}
$\mathsf{dom}.\,R.\,\gamma$
$\equiv$ {definitions}
$(\{\mathsf{dom}.\,R\};\mathsf{magic}).\,q.\,\gamma$

The other two cases follow directly from the corresponding results in Theorem 7.

Proof of Theorem 15

The proof follows the same idea as the proof of Theorem 14 but it uses a number of algebraic properties of conjunctive specifications, so we omit it for brevity.

Proof of Theorem 16

We have

$\{p\} \downarrow D \sqsubseteq \{r\};\{p'\};[r]$
$\equiv$ {definition of encoding, assumption $D = \{r\};\langle f\rangle$}
$\{r\};\langle f\rangle;\{p\};[f^{-1}];[r] \sqsubseteq \{r\};\{p'\};[r]$
$\Leftarrow$ {monotonicity of ;}
$\langle f\rangle;\{p\};[f^{-1}] \sqsubseteq \{p'\}$
$\Leftarrow$ {definition of encoding}
$\{p\} \downarrow \langle f\rangle \sqsubseteq \{p'\}$
$\equiv$ {Theorem 13 (d)}
$\mathsf{im}.\,f^{-1}.\,p \subseteq p'$
$\equiv$ {definitions of subset, of image, and of f^{-1}}
$(\forall \gamma \cdot (\exists \sigma \cdot p.\,\sigma \wedge \sigma = f.\,\gamma) \Rightarrow p'.\,\gamma)$
$\equiv$ {one-point rule}
$(\forall \gamma \cdot p.\,(f.\,\gamma) \Rightarrow p'.\,\gamma)$
$\equiv$ {definition of composition and subset}
$f;p \subseteq p'$

which proves (a). The proofs of (b) and (d) are similar. For (c) we have

$[P] \downarrow D$
$=$ {definition of encoding, assumption $D = \{r\};\langle f\rangle$}
$\{r\};\langle f\rangle;[P];[f^{-1}];[r]$
$=$ {rewrite $\langle f\rangle = [|f|]$, homomorphism}
$\{r\};[|f|;P;f^{-1}];[r]$

Proof of Theorem 18

For (c) we assume that R is total. Then

$\mathsf{magic} \downarrow D \sqsubseteq S'$
$\equiv$ {Theorem 5}
$[R];\mathsf{magic} \sqsubseteq S';[R]$

$\equiv$ {definitions}
$(\forall q\ \gamma \cdot (\forall \sigma \cdot R.\gamma.\sigma \Rightarrow \mathsf{T}) \Rightarrow S'.(\lambda\gamma' \cdot \forall\sigma' \cdot R.\gamma'.\sigma' \Rightarrow q.\sigma').\gamma)$
$\equiv$ {predicate calculus}
$(\forall q\ \gamma \cdot S'.(\lambda\gamma' \cdot \forall\sigma' \cdot R.\gamma'.\sigma' \Rightarrow q.\sigma').\gamma)$
$\Rightarrow$ {specialise $q := \mathsf{false}$, simplify}
$(\forall \gamma \cdot S'.(\lambda\gamma' \cdot \forall\sigma' \cdot \neg R.\gamma'.\sigma').\gamma)$
$\equiv$ {R assumed total}
$(\forall \gamma \cdot S'.\mathsf{false}.\gamma)$
$\equiv$ {definition of magic}
$S' = \mathsf{magic}$

which shows that $\mathsf{magic} \downarrow D = \mathsf{magic}$. All the other cases follow directly from the corresponding results in Theorem 7 and from the definitions of duals, images and inverse images.

Proof of Theorem 19

$[P] \downarrow [R] \sqsubseteq [P']$
$\equiv$ {Lemma 25}
$[R]; [P] \sqsubseteq [P']; [R]$
$\equiv$ {homomorphism}
$[R; P] \sqsubseteq [P'; R]$
$\equiv$ {embedding}
$R; P \supseteq P'; R$
$\equiv$ {property of relation quotient}
$P'^{-1} \subseteq R \backslash (R; P)^{-1}$

Proof of Theorem 20

First, assume that conditions (decode1) and (decode2) hold. Then

$S \sqsubseteq T \uparrow D$
$\Rightarrow$ {monotonicity}
$D; S \sqsubseteq D; T \uparrow D$
$\Rightarrow$ {property (decode1), transitivity}
$D; S \sqsubseteq T; D$

and (decode2) with $X := S$ gives implication in the opposite direction. Now assume that $S \sqsubseteq T \uparrow D \equiv S \sqsubseteq_D T$ for arbitrary S. Then

$D; (T \uparrow D) \sqsubseteq T; D$
$\equiv$ {assumption with $S := T \uparrow D$}
$T \uparrow D \sqsubseteq T \uparrow D$
$\equiv$ {refinement is reflexive}
T

and

$D; X \sqsubseteq T; D$
$\equiv$ {assumption with $S := X$}
$X \sqsubseteq T \uparrow D$

Proof of Theorem 22

$$
\begin{aligned}
& T \uparrow D \sqsubseteq T' \uparrow D \\
\equiv\ & \{\text{Galois connection}\} \\
& (T \uparrow D) \downarrow D \sqsubseteq T' \\
\Leftarrow\ & \{\text{Corollary 21 (b)}\} \\
& T \sqsubseteq T'
\end{aligned}
$$

Proof of Theorem 23

We prove (d) as an example of how to use the corresponding properties for encoding. The proofs for the other parts are not hard (and they are even simpler if the result in Theorem 24 is used).

$$
\begin{aligned}
& \{p\} \sqsubseteq \{p'\} \uparrow D \\
\equiv\ & \{\text{Galois connection}\} \\
& \{p\} \downarrow D \sqsubseteq \{p'\} \\
\equiv\ & \{\text{Theorem 7 (d)}\} \\
& D.p \subseteq p' \\
\equiv\ & \{\text{characterisation of Galois connection}\} \\
& p \subseteq \overline{D}.p'
\end{aligned}
$$

Proof of Theorem 24

We have, for arbitrary S,

$$
\begin{aligned}
& S \sqsubseteq T \uparrow D \\
\equiv\ & \{\text{Galois connection}\} \\
& S \downarrow D \sqsubseteq T \\
\equiv\ & \{\text{Theorem 10}\} \\
& D;S;\overline{D} \sqsubseteq T \\
\equiv\ & \{\text{shunting (Lemma 3)}\} \\
& S \sqsubseteq \overline{D};T;D
\end{aligned}
$$

which shows $T \uparrow D = \overline{D};T;D$.

Proof of Theorem 26

$$
\begin{aligned}
& (S \uparrow D_1) \uparrow D_2 \\
=\ & \{\text{Theorem 24}\} \\
& \overline{D_2};\overline{D_1};S;D_1;D_2 \\
=\ & \{\text{property of inverses}\} \\
& \overline{D_1;D_2};S;D_1;D_2 \\
=\ & \{\text{Theorem 24}\} \\
& S \uparrow (D_1;D_2)
\end{aligned}
$$

Proof of Theorem 28

For (a), we have

$$
\begin{aligned}
& [P] \sqsubseteq [P'] \uparrow D \\
\equiv\ & \{\text{Galois connection}\} \\
& [P] \downarrow D \sqsubseteq [P'] \\
\equiv\ & \{\text{as in proof of Theorem 14}\} \\
& P;R^{-1} \supseteq R^{-1};P'
\end{aligned}
$$

and for (b),

$$
\begin{array}{ll}
& \{P\} \sqsubseteq \{P'\} \uparrow D \\
\equiv & \{\text{Galois connection}\} \\
& \{P\} \downarrow D \sqsubseteq \{P'\} \\
\equiv & \{\text{Theorem 14}\} \\
& R; P \subseteq P'; R \\
\equiv & \{\text{property of relation quotient}\} \\
& P \subseteq R^{-1} \backslash (P'; R)
\end{array}
$$

Proof of Theorem 29

For (a) we have

$$
\begin{array}{ll}
& \{p\} \sqsubseteq \{p'\} \uparrow D \\
\equiv & \{\text{property of encoding, assumption } D = \{r\}; \langle f \rangle\} \\
& \{r\}; \langle f \rangle; \{p\} \sqsubseteq \{p'\}; \{r\}; \langle f \rangle \\
\equiv & \{\text{rewrite as angelic updates, homomorphism}\} \\
& \{|r|; |f|; |p|\} \sqsubseteq \{|p'|; |r|; |f|\} \\
\equiv & \{\text{homomorphism}\} \\
& |r|; |f|; |p| \subseteq |p'|; |r|; |f| \\
\equiv & \{\text{definitions}\} \\
& (\forall \gamma\ \sigma \cdot r.\gamma \wedge \sigma = f.\gamma \wedge p.\sigma \Rightarrow p'.\gamma \wedge r.\gamma \wedge \sigma = f.\gamma) \\
\equiv & \{\text{one-point rule}\} \\
& (\forall \gamma \cdot r.\gamma \wedge p.(f.\gamma) \Rightarrow p'.\gamma) \\
\equiv & \{\text{definitions}\} \\
& r \cap f; p \subseteq p'
\end{array}
$$

and the proof of (b) is similar. Finally, for (c) we have

$$
\begin{array}{ll}
& \langle g \rangle \sqsubseteq \langle g' \rangle \uparrow D \\
\equiv & \{\text{property of encoding, assumption } D = \{r\}; \langle f \rangle\} \\
& \{r\}; \langle f \rangle; \langle g \rangle \sqsubseteq \langle g \rangle; \{r\}; \langle f \rangle \\
\equiv & \{\text{rewrite as angelic updates, homomorphisms, definitions}\} \\
& (\forall \gamma\ \sigma' \cdot (\exists \sigma \cdot r.\gamma \wedge \sigma = f.\gamma \wedge \sigma' = g.\sigma) \Rightarrow (\exists \gamma' \cdot \gamma' = g'.\gamma \wedge r.\gamma' \wedge \sigma' = f.\gamma')) \\
\equiv & \{\text{one-point rule}\} \\
& (\forall \gamma \cdot r.\gamma \Rightarrow r.(g.\gamma) \wedge g.(f.\gamma) = f.(g'.\gamma)) \\
\equiv & \{\text{quantifier rules}\} \\
& (\forall \gamma \cdot r.\gamma \Rightarrow r.(g.\gamma)) \wedge (\forall \gamma \cdot r.\gamma \Rightarrow g.(f.\gamma) = f.(g'.\gamma)) \\
\equiv & \{\text{definitions}\} \\
& r \subseteq g; r \wedge f; g = g'; f
\end{array}
$$

Proof of Theorem 32

For (a) we have

$$
\begin{array}{ll}
& \mu.f \downarrow D \sqsubseteq (\mu X \cdot f.(X \uparrow D) \downarrow D) \\
\Leftarrow & \{\text{Theorem 31}\} \\
& (\forall S \cdot f.S \downarrow D \sqsubseteq f.((S \downarrow D) \uparrow D) \downarrow D)
\end{array}
$$

$\Leftarrow$ {monotonicity of encoding (Theorem 6) and of f}
$(\forall S \cdot S \sqsubseteq (S \downarrow D) \uparrow D)$
$\equiv$ {Corollary 21}
T

and the derivation for the greatest fixpoint is similar.

Proof of Theorem 33

For (a) we have

$\mu.f \sqsubseteq \mu.g \uparrow D$
$\equiv$ {Galois connection}
$(\mu.f) \downarrow D \sqsubseteq \mu.g$
$\Leftarrow$ {Theorem 31}
$(\forall S \cdot f.S \downarrow D \sqsubseteq g.(S \downarrow D))$
$\Leftarrow$ {generalisation ($\forall$ introduction)}

$f.S \downarrow D \sqsubseteq g.(S \downarrow D)$
$\equiv$ {Galois connection}
$f.S \sqsubseteq g.(S \downarrow D) \uparrow D$
$\Leftarrow$ {Corollary 21 (a), monotonicity}
$f.((S \downarrow D) \uparrow D) \sqsubseteq g.(S \downarrow D) \uparrow D$
$\Leftarrow$ {specialise $S := S \downarrow D$}
$(\forall S \cdot f.(S \uparrow D) \sqsubseteq g.S \uparrow D$

$\cdot$ $(\forall S \cdot f.(S \uparrow D) \sqsubseteq g.S \uparrow D)$

The proof for (b) is similar.

Proof of Theorem 34

$D.p \ \{\!|\ S\ |\!\}\ q$
$\equiv$ {definition of correctness}
$D.p \ \subseteq \ S.q$
$\Leftarrow$ {property of inverses, monotonicity}
$D.p \ \subseteq \ (D;\overline{D};S;D;\overline{D}).q$
$\equiv$ {decoding in terms of inverses}
$D.p \ \subseteq \ D.((S \uparrow D).(\overline{D}.q))$
$\Leftarrow$ {monotonicity}
$p \ \subseteq \ (S \uparrow D).(\overline{D}.q)$
$\equiv$ {definition of correctness}
$p \ \{\!|\ S \uparrow D\ |\!\}\ \overline{D}.q$

Received May 1999
Accepted in revised form November 2000 by B. C. Pierce