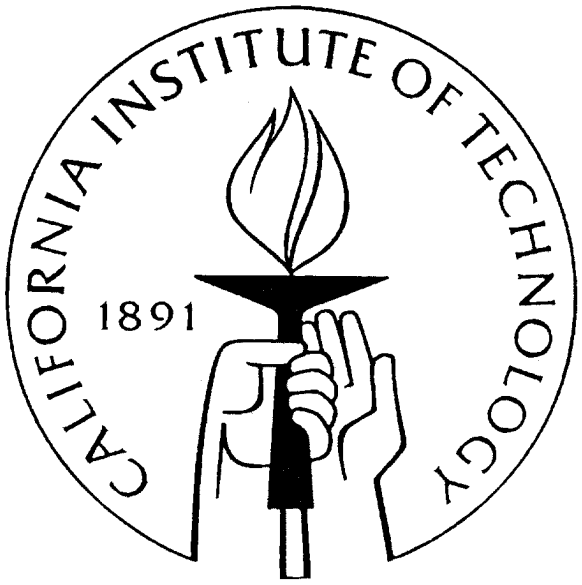

Refinement Calculus, Lattices and Higher Order Logic

R. J. R. Back

Computer Science Department
California Institute of Technology

Caltech-CS-TR-92-22

Refinement Calculus, Lattices and Higher Order Logic

R.J.R. Back

Abstract

We show how to describe refinement calculus in a lattice theoretic framework. The predicate transformer lattice is built by pointwise extensions from the truth value lattice. A simple but very powerful statement language characterizes the monotonic predicate transformer lattice. We show how to formalize the calculus in higher order logic. We also show how to express data refinement in this framework. A theorem of data refinement with stuttering is proved, to illustrate the algebraic nature of reasoning in this framework.

1 Introduction

Statements of a programming language can be interpreted as predicate transformers, functions mapping postconditions to preconditions. Such a *weakest precondition* semantics is useful for reasoning about total correctness of programs. It was introduced by [12], and has been further developed in, e.g., [16, 13]. The predicate transformer semantics provides the basis for the *refinement calculus*, introduced in [2] and further developed in, e.g., [4, 23, 25, 7, 24] and others.

In this paper we show how reasoning about predicate transformers can be formalized in higher order logic (simple type theory), using a lattice theoretic basis. Statements are interpreted as monotonic predicate transformers, and the set of all statements forms the complete lattice of all monotonic predicate transformers. This is a sublattice of the complete boolean lattice of all predicate transformers. The latter gives us the general framework in which to carry out proofs of program refinements and program equivalences. It has been constructed by pointwise extension from the truth value lattice and the lattice of predicates (sets), and inherits the basic properties of these. In fact, one can reason about predicate transformers (program statements) in a manner very similar to the way one reasons about properties in propositional calculus and predicate calculus. The formalization permits an algebraic style of reasoning similar to the one put forward in [20, 18].

The formalization of predicate transformers and refinement calculus described here has been implemented in the HOL system [15], an interactive proof checker that is based on higher order logic and ML. An early implementation is described in [?], whereas [9] describes a new implementation of the refinement calculus. The earlier implementation used a different formalization of states and state predicates than the one described here, one which turned out to be cumbersome to use in practice. An implementation of the predicate transformer logic using this formalization of states and state predicates but based on the book by Dijkstra and Scholten [13] is described in [1]. Refinement calculus has been implemented as a definitional theory in HOL. This means that no new axioms are introduced, only new definitions are added to the basic theory. Hence, the consistency of the theory follows from the consistency of higher order logic itself. Errors in proofs are avoided by the mechanical proof checking performed by the HOL system.

The paper is organized as follows. We first give an overview of the lattice theoretic approach to predicate transformers and refinement. In Section 2, we define a simple language which is complete, in the sense of describing all monotonic predicate transformers. In Section 3, we look

$\texttt{bool}$ (boolean values T and F)
σ (set of states)
$\texttt{Pred}_\sigma = \sigma \to \texttt{bool}$ (state predicates)
$\texttt{Oper}_{\sigma,\tau} = \sigma \to \tau$ (state transformers)
$\texttt{Ptran}_{\sigma,\tau} = \texttt{Pred}_\sigma \to \texttt{Pred}_\tau$ (predicate transformers)
$\texttt{Mtran}_{\sigma,\tau} = \texttt{Pred}_\sigma \to_m \texttt{Pred}_\tau$ (monotonic predicate transformers)

Figure 1: Basic domains

a little bit closer on the way states and state spaces are defined, in order to get a simple formalization of predicate transformers in higher order logic. Section 4 defines the usual program constructs in terms of the simple language. Section 5 gives a very brief overview of data refinement. Finally, Section 6 is a case study of using the refinement logic, where we prove a theorem about data refinement of loops that may introduce stuttering actions. Most of the results here are based on joint work with J. von Wright, as reported in [8, 7].

2 Predicate transformer lattice

Truth value lattice The set of truth values

$$\texttt{bool} = \{T, F\}$$

forms a complete boolean lattice $(\texttt{bool}, \leq)$ under the implication ordering:

$$a \leq b \quad \textit{iff} \quad a \Rightarrow b.$$

That $\texttt{bool}$ is a complete lattice means that arbitrary meets and joins exists in it. The meet $a \wedge b$ and the join $a \vee b$ have their usual truth function interpretations (conjunction and disjunction). The bottom element is F and the top element is T. The lattice has a complement (negation) because it is boolean.

Predicate lattice The set of *state predicates on* σ, $\texttt{Pred}_\sigma$, is defined as

$$\texttt{Pred}_\sigma = \sigma \to \texttt{bool}.$$

The ordering on $\texttt{bool}$ is extended to state predicates by pointwise extension: for p and q in $\texttt{Pred}_\sigma$, we have that

$$p \leq q \quad \textit{iff} \quad (\forall u \in \sigma.\, p\, u \Rightarrow q\, u).$$

As $(\texttt{Pred}_\sigma, \leq)$ is a pointwise extension of the truth value lattice, it will also be a complete boolean lattice.

The meet $p \wedge q$ is the conjunction of the two predicates and the join $p \vee q$ is the disjunction of the predicates. The bottom element is *false*, the identically false predicate, and the top element is *true*, the identically true predicate. These operations are defined, for each $u \in \sigma$, by

$$(p \wedge q)u = p\, u \wedge q\, u, \quad (p \vee q)u = p\, u \vee q\, u \quad \textit{and} \quad (\neg p)\, u = \neg(p\, u)$$
$$\textit{false}\, u = F \quad \textit{and} \quad \textit{true}\, u = T.$$

Predicate transformer lattice The set of *predicate transformers from σ to τ*, $\mathtt{Ptran}_{\sigma,\tau}$, is defined as

$$\mathtt{Ptran}_{\sigma,\tau} = \mathtt{Pred}_{\sigma} \rightarrow \mathtt{Pred}_{\tau}.$$

The ordering on $\mathtt{Pred}_{\tau}$ is extended to an ordering on predicate transformers, again by pointwise extension: for $s, t \in \mathtt{Ptran}_{\sigma,\tau}$, we have

$$s \leq t \quad \textit{iff} \quad (\forall p \in \mathtt{Pred}_{\sigma}.\, s\, p \leq t\, p).$$

Hence $(\mathtt{Ptran}_{\sigma,\tau}, \leq)$ is also a complete boolean lattice. This ordering is the *refinement ordering* of [2, 3].

Arbitrary meets and joins thus exist for predicate transformers also. We interpret the meet $s \wedge t$ operationally as the *demonic choice* between the predicate transformers s and t, while the join $s \vee t$ is interpreted as the *angelic choice* between these two alternatives. These operations are defined pointwise, by

$$(s \wedge t)\, p = s\, p \wedge t\, p \quad \textit{and} \quad (s \vee t)\, p = s\, p \vee t\, p.$$

The bottom element is *abort*, the predicate transformer that does not establish any postcondition. The top element is *magic*, the predicate transformer that establishes every postcondition. These are defined for $q \in \mathtt{Pred}_{\sigma}$, by

$$\textit{abort}\, q = \textit{false} \quad \textit{and} \quad \textit{magic}\, q = \textit{true}.$$

We may also define the negated predicate transformer, $\neg s$,

$$(\neg s)\, q = \neg(s\, q).$$

Functional operations The predicate transformers are functions from predicates to predicates, so we can compose them as functions. For $s \in \mathtt{Ptran}_{\sigma,\tau}$ and $t \in \mathtt{Ptran}_{\rho,\sigma}$, we define functional composition $s;\, t \in \mathtt{Ptran}_{\rho,\tau}$ as

$$(s;\, t)\, q = s(t\, q).$$

This models *sequential composition* of program statements. The identity element of functional composition is the identity function $\mathit{skip} \in \mathtt{Ptran}_{\sigma,\sigma}$:

$$\mathit{skip}\, q = q.$$

Monotonic predicate transformer lattice Let $\mathtt{Mtran}_{\sigma,\tau}$ be the set of *monotone* predicate transformers in $\mathtt{Ptran}_{\sigma,\tau}$, i.e., predicate transformers s which satisfy the condition

$$(\forall p, q \in \mathtt{Pred}.\, p \leq q \Rightarrow s\, p \leq s\, q).$$

Then $(\mathtt{Mtran}_{\sigma,\tau}, \leq)$ is a complete lattice (although it is not boolean), and is a sublattice of $(\mathtt{Ptran}_{\sigma,\tau}, \leq)$. The monotonic predicate transformers will correspond to program statements.

The domains introduced above are listed in Figure 1.

Total correctness and refinement Total correctness for predicate transformers is defined as follows. For predicates $p \in \mathtt{Pred}_{\tau}$ and $q \in \mathtt{Pred}_{\sigma}$, the predicate transformer $s \in \mathtt{Ptran}_{\sigma,\tau}$ is *totally correct* with respect to precondition p and postcondition q, denoted $p[s]q$, if $p \Rightarrow s\, q$.

Refinement preserves total correctness, in the following sense:

$$s \leq t \quad \textit{iff} \quad (\forall p, q.\, p[s]q \Rightarrow p[t]q).$$

In other words, t satisfies every total correctness specification that s satisfies if $s \leq t$, and this condition characterizes refinement exactly.

3 Statements

To define program statements, we need some basic predicate transformers to start with. First, for any predicate $g \in \mathtt{Pred}_\sigma$, we define two predicate transformers, the *guard statement* $[g]$ and the *assert statement* $\{g\}$, for $p \in \mathtt{Pred}_\sigma$, :

$$[g]\,p = (g \Rightarrow p) \quad \textit{and} \quad \{g\}\,p = (g \wedge p).$$

The guard statement behaves as a skip statement if g holds, and otherwise as magic. The assert statement also behaves as a skip statement if g holds but otherwise as an abort statement.

We also need a way of changing states. For any state transformer $f : \sigma \to \tau$, we define a corresponding predicate transformer $\langle f \rangle \in \mathtt{Ptran}_{\tau,\sigma}$, the *update statement*: for any $q \in \mathtt{Pred}_\tau$ and any $u \in \sigma$, we have

$$\langle f \rangle\, q\, u = q(f\, u).$$

In other words, $\langle f \rangle$ is guaranteed to establish postcondition q in initial state u if and only if q holds for the (final) state $f\, u$.

Statements We will identify program statements with predicate transformers. Combining lattice operations, functional operations and basic predicate transformers gives us a simple base language for predicate transformers.

$$A ::= [g] \mid \{g\} \mid \langle f \rangle \mid A_1;\, A_2 \mid \bigwedge_{i \in I} A_i \mid \bigvee_{i \in I} A_i$$

Here A, A_i are predicate transformers, I is any set, g is a state predicate and f is a state transformer. The types of the predicate transformers in sequential composition, meet and join have to match for the operations to be defined, as described earlier. We will refer to predicate transformers described in this language as *statements*.

Note that the index set I in meets and joins may be infinite. Hence, the statement language itself is infinitary. We may consider statements as an idealized programming language that captures the idea of program execution, even though not all program statements can be executed on a real (finite) computer.

The bottom and top of the predicate transformer lattice, as well as the identity transformation, are definable in terms of the basic predicate transformers:

$$\begin{aligned} \mathit{abort} &= \bigvee \emptyset \\ \mathit{magic} &= \bigwedge \emptyset \\ \mathit{skip} &= \{\mathit{true}\} = [\mathit{true}] = \langle \mathit{Id} \rangle \end{aligned}$$

where $\mathit{Id} : \sigma \to \sigma$ stands for the identity state transformation, $\mathit{Id}\, u = u$. For predicate transformers in $\mathtt{Ptran}_{\sigma,\sigma}$ (the state space is not changed) we have $\mathit{abort} = \{\mathit{false}\}$ and $\mathit{magic} = [\mathit{false}]$.

Completeness of statements All statements A are monotonic, i.e., $A \in \mathtt{Mtran}_{\sigma,\tau}$ for some σ and τ. In fact, the reverse also holds: every monotonic predicate transformer can be described as a statement, so statements coincide with the set of monotonic predicate transformers [8]. The power of the statement language derives from the arbitrary meets and joins permitted.

Notation Arbitrary meets of predicate transformers have similar properties as universal quantification, motivating us to write $\bigwedge_{i \in I} A_i$ as $(\forall i \in I.\, A_i)$. Similarly, we write $\bigvee_{i \in I} A_i$ as $(\exists i \in I.\, A_i)$.

Construct	$\leq$	$\bot$	$\top$	$\wedge$	$\vee$	$\neg$	$\wedge - c$	$\vee - c$	$\bigwedge$	$\bigvee$
$\langle f \rangle$	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
$\{g\}$	yes	yes	**no**	yes	yes	**no**	yes	yes	yes	yes
$[g]$	yes	**no**	yes	yes	yes	**no**	yes	yes	yes	yes
;	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
$\wedge$	yes	yes	yes	yes	**no**	**no**	yes	yes	yes	**no**
$\vee$	yes	yes	yes	**no**	yes	**no**	yes	yes	**no**	yes
continuous $\wedge$	yes	yes	yes	yes	yes	**no**	yes	yes	yes	**no**
continuous $\vee$	yes	yes	yes	yes	yes	**no**	yes	yes	**no**	yes
infinite $\wedge$	yes	yes	yes	yes	**no**	**no**	yes	**no**	yes	**no**
infinite $\vee$	yes	yes	yes	**no**	yes	**no**	**no**	yes	**no**	yes

Table 1: Homomorphic properties of statements

Homomorphic properties We define the following homomorphism properties for predicate transformers:

bottom homomorphism ($\bot$)	$A\,false = false$	
top homomorphism ($\top$)	$A\,true = true$	
negation hom. ($\neg$)	$A\,(\neg p) = \neg(A\,p)$	
finite meet hom. ($\wedge$)	$A\,(p \wedge q) = A\,p \wedge A\,q$	
positive meet hom.	$A\,(\forall i \in I.\, q_i) = (\forall i \in I.\, A\,q_i)$,	any $I \neq \emptyset$
universal meet hom. ($\bigwedge$)	$A\,(\forall i \in I.\, q_i) = (\forall i \in I.\, A\,q_i)$,	any I
meet continuous ($\wedge - c$)	$A\,(\forall i \in I.\, q_i) = (\forall i \in I.\, A\,q_i)$,	down chain $q_i, i \in I$
finite join hom. ($\vee$)	$A\,(p \vee q) = A\,p \vee A\,q$	
positive join hom.	$A\,(\exists i \in I.\, q_i) = (\exists i \in I.\, A\,q_i)$,	any $I \neq \emptyset$
universal join hom. ($\bigvee$)	$A\,(\exists i \in I.\, q_i) = (\exists i \in I.\, A\,q_i)$,	any I
join continuous ($\vee - c$)	$A\,(\exists i \in I.\, q_i) = (\exists i \in I.\, A\,q_i)$,	up chain $q_i, i \in I$

Note that universal meet homomorphism implies bottom homomorphism, and similarly universal join homomorphism implies top homomorphism, by choosing the index set to be empty. Table 1 summarizes the properties that statements have. For the predicate transformers, we indicate for each homomorphic property whether it has this property. For the statement constructors (sequential composition, meet and join), we indicate whether they preserve the homomorphic property in question.

The term *non-miraculous* is also used for bottom homomorphism, the term *terminating* for top homomorphism, the terms *conjunctive* and *demonic* for meet homomorphism, and the terms *disjunctive* and *angelic* for join homomorphism.

Monotonicity of replacement Besides preserving monotonicity, the statement constructors are also monotonic themselves, as functions on the predicate transformer lattice: If $A_i \leq A'_i, i \in I$, then

$$\begin{aligned} A_1;\, A_2 &\leq A'_1;\, A'_2 \\ (\forall i \in I.\, A_i) &\leq (\forall i \in I.\, A'_i) \\ (\exists i \in I.\, A_i) &\leq (\exists i \in I.\, A'_i). \end{aligned}$$

In general, for any statement A containing a substatement B, $A = A(B)$, we have that

$$B \leq B' \Rightarrow A(B) \leq A(B').$$

This means that we may always replace a statement by its refinement, in any context. This is the essential property needed in top-down program derivation.

Changing behavior Let A be a statement. We define the *non-miraculous domain* or *guard* of A, denoted gA, to be $gA = \neg A\, false$. We define the *termination domain* of A, denoted tA, to be $tA = A\, true$. For any statement A, we have that

$$A = [gA];\ A = \{tA\};\ A.$$

We can use these predicates to change the behavior of statements. For a statement A, $sA = \{gA\};\ A$ is a statement that behaves as A, except that when A would terminate miraculously, sA aborts. Similarly, $mA = [tA];\ A$ is a statement that behaves as A, except that where A may abort, mA terminates miraculously.

Recursive composition The language of statements does not include recursion. Recursion is not needed, because the language is complete. However, it is convenient, and we can define recursion without problems in this framework.

Let $C(X)$ be a statement containing the variable X ranging over monotonic predicate transformers. Then, by the monotonicity of statement constructors, we have that

$$C \quad = \quad \lambda X.C(X) : \mathtt{Mtran} \rightarrow \mathtt{Mtran}$$

is monotonic. Hence, by the Knaster-Tarski fixpoint theorem, this function has a least fixed point $\mu X.C(X) : \mathtt{Mtran}$, which we take to be the meaning of a recursively defined statement.

The fixed point can be expressed explicitly in the predicate transformer lattice:

$$\mu X.C(X) = \bigwedge \{s \mid C\, s \leq s\}$$

The recursive statement $\mu X.C(X)$ is also monotonic with respect to substatement replacement.

4 States and state spaces

We will define a *state space* to be a product type

$$\sigma = \sigma_1 \times \ldots \times \sigma_m, \tag{1}$$

where $m \geq 0$. A *state* in this state space is an element $(x_1, \ldots, x_m)$ of type σ. For $m = 0$, we have $\sigma = \mathtt{unit}$, the trivial type with only one element.

As a special case, the state space may contain a single component. This means that any type can be instantiated for the state space, i.e., the approach we have taken does not restrict the interpretation of states.

Each variable is associated with a type in higher order logic. Thus, a tuple is really of the form $v = (x_1 : \sigma_1, \ldots, x_m : \sigma_m)$. The types may either be explicitly stated or may be inferred from the context. We say that the tuple v *declares* the state space $\sigma_1 \times \ldots \times \sigma_m$.

State predicates Let $P : \texttt{bool}$ be a boolean term. Assume that we want to interpret P as a predicate on the program variables $v = (x_1 : \sigma_1, \ldots, x_m : \sigma_m)$. Then

$$\lambda v.\, P$$

is the state predicate corresponding to P. *Program variables* are thus bound variables in a predicate, while the free variables are *logical variables*, whose values are determined by the environment. A state predicate is thus determined by a boolean term $P : \texttt{bool}$ and a declaration v of program variables. The boolean term alone is not sufficient to determine a state predicate.

For instance, if P is $x + y < z + 1$ and the program variables are $(x : \texttt{int}, y : \texttt{int})$, this determines the state predicate

$$p \quad = \quad \lambda(x : \texttt{int}, y : \texttt{int}).\, x + y < z + 1.$$

Here z is free, and is thus a logical variable. The predicate p states that the sum of the first and the second state component is less than $z + 1$. The variables x and y are only used as local names for the state components, whereas z is some logical variable whose value is assumed to be determined by the environment. Assuming $z = 3$, e.g., the predicate p is true in state $(0, 1)$ but false in state $(2, 2)$.

Hoare logic uses predicates with implicit program variables. A correctness formula in this logic has the form $P\{S\}Q$, where P and Q are boolean terms. Which free variables in P and Q are program variables and which are logical variables is not determined by the formula. This forces one to use syntactic conventions to make this distinction when using the logic (such as, e.g., that x_0 is a logical varible, while x is a program variable). By being explicit about the program variables, we avoid this problem, and can reason about statements containing both program varibles and logical variables without having to introduce any additional syntactic conventions.

Explicit vs implicit program variables A predicate $P : \texttt{bool}$ with implicit program variables v determines the state predicate $\lambda v.\, P$. In the other direction, any state predicate $p : \texttt{Pred}_\sigma$ can be described as a predicate with given implicit program variables v: we have by η-reduction that

$$p = \lambda v.\, p\, v$$

whenever v is a declaration of the state space σ. Here $p\, v : \texttt{bool}$ is the boolean term with implicit program variables v.

A number of syntactic operations need to be done on predicates with implicit program variables, to express proof rules in programming logics. We show below what these operations correspond to for state predicates.

Renaming Assume that $P : \texttt{bool}$ is a predicate with implicit program variables v. Typically, we may want to rename some of the program variables in P, essentially changing the program variables v to w to get the predicate $P[w/v]$.

Making the program variables explicit gives us the state predicate $\lambda v.P$. The meaning of the predicate does not depend on the way program variables are named. By α-conversion, we are therefore free to rename the program variables in predicates:

$$\lambda v.\, P = \lambda w.\, P[w/v].$$

Substitution Another important operation on predicates is *substitution*. Given a predicate P on implicit program variables v, we want to substitute the terms t for the program variables v. The result is denoted $P[t/v]$ in e.g. Hoare logic. When the program variables are explicit, the effect of substitution is achieved with β-reduction. We have that

$$\lambda v.\, p\, t \quad = \quad \lambda v.(\lambda v.\, P)t \quad = \quad \lambda v.\, P[t/v]$$

(Application is assumed to bind stronger than lambda-abstraction.)

For example, the result of substituting $x + y$ for x in the predicate $\lambda(x, y).\, x > y$ is denoted by the term:

$$\lambda(x, y).\, (\lambda(x, y).\, x > y)(x + y, y) \quad (= \lambda(x, y).\, x + y > y).$$

We use β-reduction to compute the result. Note how the application to $(x+y, y)$ shows explicitly that no substitution is made for y. Simultaneous substitution is also easily expressed in this way.

Changing state space When program variables are implicit, we can easily mix predicate expressions over different state spaces. For example, if $x < y$ has state space (x, y) and $x = z+1$ has state space (x, z), we can simply write

$$x < y \wedge x = z + 1$$

for the conjunction of these two predicates in, e.g., Hoare logic. Implicitly, we have projected both predicates onto the state space (x, y, z).

Such a projection must be made explicit when working with state predicates. Assume that v and w are state tuples. If p is a predicate over the state declared by v, then the corresponding predicate over w is

$$\lambda w.\, p\, v$$

This projection changes the program variables in $v - w$ to logical variables, and changes the logical variables $v - w$ to program variables. Hence, care has to be taken in using this operation, so that the effect achieved is the one intended.

Consider as an example the state predicate $\lambda(x, y).\, x < y$. Projecting this on the larger state space (x, y, z) gives us the state predicate

$$\lambda(x, y, z).\, (\lambda(x, y).x < y)(x, y) = \lambda(x, y, z).\, x < y.$$

5 Derived constructs

A large number of useful programming notations have been developed for sequential programs. These constructs can be treated as abbreviations for statements as they are defined in the previous section. We describe some such common constructs below.

Assignment statements An ordinary assignment statement is easily expressed as an update statement. If the program variables are x, y and z, say, then the state is a tuple (x, y, z). The assignment statement $x := x + 1$ will then correspond to a mapping $f : (x, y, z) \rightarrow (x + 1, y, z)$. The corresponding update statement is

$$\langle f \rangle \quad = \quad \langle \lambda(x, y, z).\, (x + 1, y, z) \rangle$$

We have that

$$\langle f \rangle\, q\, (x, y, z) = q((\lambda(x, y, z).\, (x + 1, y, z))(x, y, z)) = q(x + 1, y, z)$$

As $q(x+1, y, z) = q(x, y, z)[x+1/x]$, we have the standard weakest precondition characterization of assignment statements.

Nondeterministic assignments We may also introduce statements that change the state space in a nondeterministic way, according to some specified state relation. These statements permit the nondeterminism to be unbounded, and are useful for specifying computations [2, 4, 23, 7].

Let $r : \sigma \rightarrow \tau \rightarrow \mathbf{bool}$ be a state relation. Then r can be viewed as a function from initial states in σ to sets of possible final states in τ. We want to capture the notion of a statement which chooses nondeterministically between these possible final states. Since we permit both demonic and angelic nondeterminism, we add two constructs to our notation:

$$\begin{aligned} [r]\, q\, v &\stackrel{\text{df}}{=} \forall v'.\, r\, v\, v' \Rightarrow q\, v' \\ \{r\}\, q\, v &\stackrel{\text{df}}{=} \exists v'.\, r\, v\, v' \wedge q\, v' \end{aligned}$$

As an example, consider the state relation $r = \lambda(x, y)(x', y').\, x' > x$, which specifies that the final value of x should be greater than the initial value. The following calculation applies $[r]$ to the predicate $q = \lambda(x, y).\, x > 1$:

$$\begin{aligned} & [r]\, q\, (x, y) \\ = \; & \{\text{Definitions}\} \\ & \forall(x', y').\, (\lambda(x, y)(x', y').x' > x)(x, y)(x', y') \Rightarrow (\lambda(x, y).\, x > 1)(x', y') \\ = \; & \{\text{Beta reduction}\} \\ & \forall(x', y').\, x' > x \Rightarrow x' > 1 \\ = \; & \{\text{Arithmetic}\} \\ & x \geq 1. \end{aligned}$$

Thus, to guarantee that $x > 1$ holds in the final state, $x \geq 1$ must hold prior to the nondeterministic assignment.

A convenient abbreviation for demonic and angelic assignment statements is as follows. Let $v = u, w$. Then define

$$\begin{aligned} [u := u'.Q] &\stackrel{\text{df}}{=} [\lambda v\, v'.Q \wedge w' = w] \\ \{u := u'.Q\} &\stackrel{\text{df}}{=} \{\lambda v\, v'.Q \wedge w' = w\} \end{aligned}$$

Both statements assign some values u' to u such that condition Q becomes satisfied, leaving w unchanged. In the first case, the choice when there is more than one possible assignment is demonic, in the second case the choice is angelic. In the first case, the result is miraculous termination if there is no possible assignment, in the second case, the result is abortion.

Guarded commands and conditionals A *guarded command* $g \rightarrow S$ is defined as

$$g \rightarrow S = [g]; S.$$

The *conditional statement* $\mathtt{if}\ A_1 \parallel \ldots \parallel A_n\ \mathtt{fi}$ is defined by

$$\mathtt{if}\ A_1 \parallel \ldots \parallel A_n\ \mathtt{fi} \quad = \quad \{gA_1 \vee \ldots \vee gA_n\}; (A_1 \wedge \ldots \wedge A_n)$$

Note that $A_i \leq A_i'$ for $i = 1, \ldots, n$ is not sufficient to guarantee that

$$\mathtt{if}\ A_1 \parallel \ldots \parallel A_n\ \mathtt{fi} \leq \mathtt{if}\ A_1' \parallel \ldots \parallel A_n'\ \mathtt{fi}$$

holds. This is because of the assert statement in the definition of the conditional; the refinement $A_i \leq A_i'$ may strengthen the guard, so that $gA_i \neq gA_i'$. If we also show that $gA_1 \vee \ldots \vee gA_n = gA_1' \vee \ldots \vee gA_n'$, then the refinement does hold.

Iteration We define iteration as a special case of recursion in the usual way:

$$\mathbf{do}\ \ b \to s\ \mathbf{od} \stackrel{\text{df}}{=} (\mu X.(b \to s; X) \wedge (\neg b \to \mathbf{skip}))$$

Straighforward calculations show that

$$\mathbf{do}\ \ b \to s\ \mathbf{od}\, q = (\mu x.(b \wedge s\, x) \vee (\neg b \wedge q))$$

which corresponds to the usual fixpoint definition of iteration.

Blocks and local variables A block with local variables is described with the help of an *enter* statement and an *exit* statement:

$$\mathit{enter}\, y_0 = \langle \lambda(x).(x, y_0) \rangle \qquad \mathit{exit} = \langle \lambda(x, y).(x) \rangle.$$

For $\mathit{enter} : \tau \to \mathtt{Mtran}_{\sigma\times\tau,\sigma}$, $s : \mathtt{Mtran}_{\sigma\times\tau,\sigma\times\tau}$ and $\mathit{exit} : \mathtt{Mtran}_{\sigma,\sigma\times\tau}$, we have

$$\mathbf{block}\, y_0\, s = \mathit{enter}\, y_0;\ s;\ \mathit{exit}.$$

This definition initializes the new local variables to given values y_0. We may abbreviate this construct as $[\![\ \mathbf{var}\ y := y_0;\ S\ ]\!]$ (the variable y does not really serve any purpose here, it is added only for conformance with standard notation).

If one wants to initialize the local variables to some arbitrary (demonically chosen) value establishing relation $r : \sigma \to \sigma \times \tau \to \mathtt{bool}$, then one can define the enter statement as

$$\mathit{enter}\, r = [r].$$

For instance, choosing $r\,(x)\,(x', y) = (x = x')$ gives an enter statement that chooses arbitrary any value of the correct type for the new local variables.

6 Data refinement

Data refinement is general technique by which one can change the state space in a refinement. The general data refinement method [19, 21, 17], has been formalized in refinement calculus in [2, 22, 5, 26, 11, 7, 14]. We show here that the framework introduced above is sufficient to defined data refinement in a very general setting, as a special case of the ordinary refinement notion.

Definition of data refinement Let $S \in \mathtt{Mtran}_{\sigma,\sigma}$ and $S' \in \mathtt{Mtran}_{\sigma',\sigma'}$ be statements on the respective state spaces σ and σ'. Let $r \in \sigma' \to \sigma \to \mathtt{bool}$ be a relation between these state spaces. We say that S *is data refined by* S' *via* r, denoted $S \leq_r S'$, if

$$\{r\};\ S \leq S';\ \{r\}.$$

Let us write r^{-1} for the inverse of relation r, defined by $r^{-1}\, v\, v' = r\, v'\, v$. Alternative and equivalent characterizations of data refinement are then

$$S;\ [r^{-1}] \leq [r^{-1}];\ S', \quad S \leq [r^{-1}];\ S';\ \{r\}, \quad \{r\};\ S;\ [r^{-1}] \leq S'.$$

These characterizations follow from the fact that $\{r\}$ and $[r^{-1}]$ are each others inverses, in the sense that

$$\{r\};\ [r^{-1}] \leq \mathbf{skip} \quad \textit{and} \quad \mathbf{skip} \leq [r^{-1}];\ \{r\}.$$

Computing the conditions for data refinement shows that $S \leq_r S'$ holds iff

$$(\forall q.\forall v, v'.\, S\, q\, v \wedge r\, v'\, v \Rightarrow S'\, (\lambda v'.\, \exists v.\, r\, v'\, v \wedge q\, v)).$$

Data refinement as described here is *forward data refinement*, also called *downward simulation. Backward data refinement* or *upward simulation*, introduced in [10], can be described in an analogous fashion [28].

Piecewise data refinement We can do refinement via encoding in a piecewise fashion. We have that

$$
\begin{array}{ll}
S_1; S_2 \leq_r S_1'; S_2', & \textit{if} \quad S_1 \leq_r S_1' \textit{ and } S_2 \leq_r S_2' \\
\bigwedge S_i \leq_r \bigwedge S_i', & \textit{if} \quad S_i \leq_r S_i' \textit{ for each } i \\
\bigvee S_i \leq_r \bigvee S_i', & \textit{if} \quad S_i \leq_r S_i' \textit{ for each } i
\end{array}
$$

Using these relations, we can compute the conditions under which data refinement holds for the derived constructs. For instance, for an iteration statement, we have that

$$\texttt{do}\ b \rightarrow S\ \texttt{od} \leq_r \texttt{do}\ b' \rightarrow S'\ \texttt{od} \quad \textit{if} \quad b \rightarrow S \leq_r b' \rightarrow S' \quad \textit{and} \quad [\neg b] \leq_r [\neg b'].$$

Data refinement of blocks We can use the above results to show how a block statement is refined. We have that

$$
\begin{array}{ll}
 & \texttt{block}_{y_0}\ s \\
= & \{\text{definition}\} \\
 & \textit{enter}\ y_0;\ s;\ \textit{exit} \\
\leq & \{\text{Assuming (a) } \textit{enter}\ y_0 \leq \textit{enter}\ y_0';\ \{r\}\} \\
 & \textit{enter}\ y_0';\ \{r\};\ s;\ \textit{exit} \\
\leq & \{\text{ Assuming (b) } \{r\};\ s \leq s';\ \{r\}\ \} \\
 & \textit{enter}\ y_0';\ s';\ \{r\};\ \textit{exit} \\
\leq & \{\text{ Assuming (c) } \{r\};\ \textit{exit} \leq \textit{exit}'\ \} \\
 & \textit{enter}\ y_0';\ s';\ \textit{exit}' \\
= & \{\text{definition}\} \\
 & \texttt{block}_{y_0'}\ s'
\end{array}
$$

It turns out that condition (a) is equivalent to $(\forall x.\, r\,(x, y_0')\,(x, y_0))$, condition (b) expresses the data refinement requirement $s \leq_r s'$, while condition (c) is always satisfied. Hence, refinement of a block amounts to finding an abstraction relation r that is initialized on block entry and which gives data refinement of the block bodies.

Initialized loops Combining this with the rule for data refinement of iteration gives us the following rule:

$$[\![\ \texttt{var}\ \ y := y_0;\ \texttt{do}\ \ A\ \ \texttt{od}\ ]\!] \leq [\![\ \texttt{var}\ \ y' := y_0';\ \texttt{do}\ \ A'\ \ \texttt{od}\ ]\!]$$

if there exists a relation $R(x, y, y')$, such that

(i) $R(x, y_0, y_0')$,

(ii) $A \leq_r A'$ and

(iii) $[\neg g A] \leq_r [g \neg A']$.

where $r = \lambda(x', y').\, \lambda(x, y).\, R(x, y, y') \wedge x' = x$.

7 Data refinement theorem

We apply this framework to the proof of a data refinement theorem, where refinement of loops may introduce stuttering actions, i.e., concrete actions that correspond to skip statements on

the abstract level. We assume in the sequel that all statements are positively conjunctive and continuous.

A alternative proof of the data refinement theorem with stuttering, based on characterizing the iteration statement in terms of continuous meets and joins, rather than in terms of action sequences as we will do here, is given in [27]. Corresponding results for backward data refinement are also proved in that paper.

Iteration We give an alternative characterization of iteration, in terms of the possible sequences of actions. Define for any action A:

1. $A^0 = \mathit{skip}$ and
2. $A^{n+1} = A^n;\ A$, for $n \geq 0$.

Then we can define *iteration* do A od as follows:

$$\textbf{do}\ A\ \textbf{od} = (\forall n \geq 0.\ A^n;\ [\neg gA]) \wedge (\exists n \geq 0.\ A^n;\ \mathit{abort}). \tag{2}$$

The first part of the conjunct corresponds to the partial correctness requirement and the second part to the termination requirement for the loop.

This characterization of iteration needs the assumption that the action A is continuous. It can be shown to be equivalent to the definition of iteration in terms of fixed points, under the assumption of positive conjunctivity and continuity.

Action sequences We generalize the above to consider arbitrary sequences of actions over a given finite set S of actions. Let $S = \{A_1, \ldots, A_n\}$. Let S^n be the set of all sequences of actions of length n, let S^* be set of all finite sequences of actions($S^* = \cup S^n$) and S^∞ the set of all infinite sequences of actions. Let $gS = \vee\{gA : A \in S\}$.

Let $\mathit{seq}\ a$ denote the sequential composition of actions in finite sequence a, i.e., $\mathit{seq}\langle\rangle = \mathit{skip}$ and $\mathit{seq}(a;\ A) = (\mathit{seq}\ a);\ A$. For an infinite sequence $a \in S^\infty$, we define

$$\mathit{seq}\ a = (\exists n.\ \mathit{seq}\ a \downarrow n;\ \mathit{abort}),$$

where $a \downarrow n$ stands for the first n elements of a.

The definition of iteration (2) gives us an equivalent definition of iteration in terms of the possible action sequences, by choosing $A = A_1 \wedge \ldots \wedge A_n$:

$$\textbf{do}\ A\ \textbf{od} = (\forall a \in S^*.\ \mathit{seq}\ a;\ [\neg gS]) \wedge (\forall a \in S^\infty.\ \mathit{seq}\ a).$$

LEMMA 1 (BOUNDEDNESS) *Assume that S is a finite set of continuous actions. Then*

$$(\forall a \in S^\infty.\ \mathit{seq}\ a) = (\exists n \geq 0.\forall a \in S^\infty.\ \mathit{seq}\ a \downarrow n;\ \mathit{abort}).$$

Note: Defining the termination condition as $(\forall a \in S^\infty.\ \mathit{seq}\ a)$ is correct even in the case that S has infinitely many actions, whereas the right hand side would not necessarily then give the right answer. When S is finite, these two characterizations coincide.

THEOREM 1 *Assume that $\alpha = [r]$ for some relation r. Assume also*

(i) *Commutativity: (a) $A;\ \alpha \leq \alpha;\ A1$, (b) skip; $\alpha \leq \alpha;\ A2$ and (c) $B;\ \alpha \leq \alpha;\ B1$,*

(ii) *Enabledness: (a) $[\neg gA];\ \alpha \leq \alpha;\ [\neg gA1 \wedge \neg gA2]$ and $[\neg gB];\ \alpha \leq \alpha;\ [\neg gB1]$, and*

(iii) *Termination: $\alpha;\ A2^\infty = \mathit{magic}$.*

Then

$$\textbf{do}\ A \parallel B\ \textbf{od};\ \alpha \quad \leq \quad \alpha;\ \textbf{do}\ A1 \parallel A2 \parallel B1\ \textbf{od}.$$

Proof Let us denote $S = \{A, B\}$ and $T = \{A1, A2, B1\}$. By the definition of iteration, we have that

$$\alpha;\ \mathtt{do}\ A1 \mid A2 \parallel B1\ \mathtt{od} \quad = \quad \alpha;\ ((\forall c \in T^*.\ seq\ c;\ [\neg gT]) \wedge (\forall c \in T^{\infty}.\ seq\ c))$$

For any sequence c of actions $A1, A2, B1$ (finite or infinite) , let $\overline{c}$ denote the sequence that we get from c when each $A1$ is replaced by A, each $B1$ by B, and each $A2$ is omitted.

We first prove Lemmas 2 – 6. The theorem follows then directly from Lemmas 4 and 6:

$\alpha;\ \mathtt{do}\ A1 \mid A2 \parallel B1\ \mathtt{od}$

$=$ {definition of iteration }

$\alpha;\ ((\forall c \in T^*.\ seq\ c;\ [\neg gT]) \wedge (\forall c \in T^{\infty}.\ seq\ c))$

$=$ {conjunctivity }

$\alpha;\ (\forall c \in T^*.\ seq\ c;\ [\neg gT]) \wedge \alpha;\ (\forall c \in T^{\infty}.\ seq\ c)$

$\geq$ {Lemma 4 and Lemma 6 }

$(\forall d \in S^* :\ seq\ d;\ [\neg gS]);\ \alpha \wedge (\forall d \in S^*.\ seq\ d;\ [\neg gS]);\ \alpha \wedge (\forall d \in S^{\infty}.seq\ d);\ \alpha$

$=$ {simplify }

$(\forall d \in S^* :\ seq\ d;\ [\neg gS]);\ \alpha \wedge (\forall d \in S^{\infty}.seq\ d);\ \alpha$

$=$ {conjunctivity }

$((\forall d \in S^* :\ seq\ d;\ [\neg gS]) \wedge (\forall d \in S^{\infty}.seq\ d));\ \alpha$

$=$ {definition of iteration }

$\mathtt{do}\ A \parallel B\ \mathtt{od};\ \alpha$

Q.E.D

LEMMA 2 *Let* $c \in T^*$. *Then* $\alpha;\ seq\ c \geq seq\ \overline{c};\ \alpha$.

Proof The proof is by induction on the length of c. If $c = \langle\rangle$, then $\overline{c} = \langle\rangle$ also, so the result holds trivially. Assume that $\alpha;\ seq\ c \geq seq\ \overline{c};\ \alpha, c \in T^*$. Then assumption (i) gives us that

1. $\alpha;\ seq\ c;\ A1 \geq seq\ \overline{c};\ \alpha;\ A1 \geq seq\ \overline{c};\ A;\ \alpha = seq\ \overline{c;\ A1};\ \alpha$,
2. $\alpha;\ seq\ c;\ A2 \geq seq\ \overline{c};\ \alpha;\ A2 \geq seq\ \overline{c};\ \alpha = seq\ \overline{c;\ A2};\ \alpha$,
3. $\alpha;\ seq\ c;\ B1 \geq seq\ \overline{c};\ \alpha;\ B1 \geq seq\ \overline{c};\ B;\ \alpha = seq\ \overline{c;\ B1};\ \alpha$.

Q.E.D

LEMMA 3 $\alpha;\ [\neg gT] \geq [\neg gS];\ \alpha$.

Proof

$\alpha;\ [\neg gT]$

$=$ {definition of gT }

$\alpha;\ [\neg gA1 \wedge \neg gA2 \wedge \neg gB1]$

$=$ { property of guard }

$\alpha;\ ([\neg gA1 \wedge \neg gA2] \vee [\neg gB1])$

$\geq$ { monotonicity, property of join }

$\alpha;\ [\neg gA1 \wedge \neg gA2] \vee \alpha;\ [\neg gB1]$

$$
\begin{array}{ll}
\geq & \{\text{assumption (ii)}\} \\
& [\neg gA];\ \alpha \vee [\neg gB];\ \alpha \\
= & \{\text{ property of guard }\} \\
& [\neg gA \wedge \neg gB];\ \alpha \\
= & \{\text{ definition }\} \\
& [\neg gS];\ \alpha
\end{array}
$$

Q.E.D

LEMMA 4 $\alpha;\ (\forall c \in T^*.\ seq\ c;\ [\neg gT]) \geq (\forall d \in S^* :\ seq\ d;\ [\neg gS]);\ \alpha$.

Proof

$$
\begin{array}{ll}
& \alpha;\ (\forall c \in T^*.\ seq\ c;\ [\neg gT]) \\
= & \{\text{conjunctivity }\} \\
& (\forall c \in T^*.\ \alpha;\ seq\ c;\ [\neg gT]) \\
\geq & \{\text{Lemma 2 }\} \\
& (\forall c \in T^*.\ seq\ \overline{c};\ \alpha;\ [\neg gT]) \\
\geq & \{\text{Lemma 3 }\} \\
& (\forall c \in T^*.\ seq\ \overline{c};\ [\neg gS];\ \alpha) \\
= & \{\text{property of conjuncts }\} \\
& (\forall c \in T^*.\ seq\ \overline{c};\ [\neg gS]);\ \alpha \\
\geq & \{\ \{\overline{c} | c \in T^*\} \subseteq S^*\ \} \\
& (\forall d \in S^*.\ seq\ d;\ [\neg gS]);\ \alpha
\end{array}
$$

Q.E.D

LEMMA 5 *Let* $c \in T^\infty$. *Then* $\alpha;\ seq\ c \geq seq\ \overline{c};\ [\neg gS];\ \alpha$, *if* $c = c_1;\ A2^\infty$ *for some* c_1, *and* $\alpha;\ seq\ c \geq seq\ \overline{c};\ \alpha$ *otherwise.*

Proof Consider first the case when $c = c_1;\ A2^\infty$ for some c_1. Then

$$
\begin{array}{ll}
& \alpha;\ seq\ c \\
= & \{\text{ assumption }\} \\
& \alpha;\ seq\ c_1;\ seq\ A2^\infty \\
\geq & \{\text{ Lemma 2 }\} \\
& seq\ \overline{c_1};\ \alpha;\ seq\ A2^\infty \\
= & \{\text{ assumption (iii) }\} \\
& seq\ \overline{c_1};\ magic \\
\geq & \{\ magic \text{ is greatest element }\} \\
& seq\ \overline{c_1};\ [\neg gS];\ \alpha \\
\geq & \{\ \overline{c_1;\ A2^\infty} = \overline{c}\ \} \\
& seq\ \overline{c};\ [\neg gS];\ \alpha
\end{array}
$$

Consider then the case when c does not end in a infinite trailing sequence of $A2$ actions. In this case, we have that $\overline{c}$ is also infinite, and $\overline{c}$ is the limit of the $\overline{c \downarrow n}$ sequence. We have

$$\alpha;\ seq\ c$$

$=$ { definition of infinite composition }

$\alpha; (\exists n \geq 0.\, seq\ c \downarrow n;\ abort)$

$\geq$ { monotonicity, property of disjunction }

$(\exists n \geq 0.\, \alpha;\ seq\ c \downarrow n;\ abort)$

$\geq$ { Lemma 2 }

$(\exists n \geq 0.\, seq\ \overline{c \downarrow n};\ \alpha;\ abort)$

$=$ { property of *abort* }

$(\exists n \geq 0.\, seq\ \overline{c \downarrow n};\ abort;\ \alpha)$

$=$ { property of disjunction }

$(\exists n \geq 0.\, seq\ \overline{c \downarrow n};\ abort);\ \alpha$

$=$ { by assumption, $\overline{c \downarrow n} = \overline{c} \downarrow m$ for some m }

$(\exists m \geq 0.\, seq\ \overline{c} \downarrow m;\ abort);\ \alpha$

$=$ { definition }

$seq\ \overline{c};\ \alpha$

Q.E.D

LEMMA 6 $\alpha; (\forall c \in T^{\infty}.\, seq\ c) \geq (\forall d \in S^{*}.\, seq\ d; [\neg gS]);\ \alpha \wedge (\forall d \in S^{\infty}.seq\ d);\ \alpha$

Proof

$\alpha; (\forall c \in T^{\infty}.\, seq\ c)$

$=$ {conjunctivity }

$(\forall c \in T^{\infty}.\, \alpha;\ seq\ c)$

$=$ {partition T^{∞} so that $U = \{c_1;\ A2^{\infty} | c_1 \in T^{*}\}$ and $V = T^{\infty} - U$ }

$(\forall c \in U.\, \alpha;\ seq\ c) \wedge (\forall c \in V.\, \alpha;\ seq\ c)$

$\geq$ {Lemma 5 }

$(\forall c \in U.\, seq\ \overline{c};\ [\neg gS];\ \alpha) \wedge (\forall c \in V.\, seq\ \overline{c};\ \alpha)$

$\geq$ {$\{\overline{c} | c \in U\} \subseteq S^{*}$ and $\{\overline{c} | c \in V\} \subseteq S^{\infty}$ }

$(\forall d \in S^{*}.\, seq\ d;\ [\neg gS];\ \alpha) \wedge (\forall d \in S^{\infty}.\, seq\ d;\ \alpha)$

$=$ {property of meet }

$(\forall d \in S^{*}.\, seq\ d;\ [\neg gS]);\ \alpha \wedge (\forall d \in S^{\infty}.\, seq\ d);\ \alpha$

Q.E.D

8 Conclusion

Our purpose here has been to give an overview of the lattice theoretic framework for refinement calculus, and its formalization in higher order logic. We have shown how the predicate transformers form a complete boolean lattice, which arises by pointwise extending first the truth value lattice and then the resulting state predicate lattice.

Program statements and specifications correspond to the complete sublattice of monotonic predicate transformers, and can all be described in the simple language of statements that we presented. Traditional (sequential) program constructs can all be described as abbreviations for statements in the simple statement language. The formalization of states and state predicates that we propose makes it simple to formalize the refinement calculus in higher order logic. A

rather extensive formalization of the refinement calculus already exist, implemented in the HOL system.

We also showed that data refinement can be described in the simple framework introduced, so that there is no need to introduce a separate relation of data refinement. This is a consequence of the fact that our statements themselves can change the state space. As a case study of using this logic, we showed how to prove the correctness of a data refinement where the refinement may introduce stuttering actions. This proof rule turns out to be very useful in stepwise refinement of parallel programs [6]. The proof is intended to illustrate the more or less algebraic style that results from carrying out the proofs on the level of predicate transformers alone. This should be contrasted with the traditional style of reasoning, where predicate transformers are applied to postconditions, and the reasoning is carried out in terms of predicates or directly on boolean terms. The latter approach easily leads to very complicated and long formulas, which tend to hide the essential issues in the proofs.

Acknowledgements

I would like to thank Robert Barta, Ulla Binau, Marcel van de Groot, Peter Hofstee, Rustan Leino, Carroll Morgan, Jan van de Snepscheut and Joakim von Wright for very helpful discussions on the topics treated here.

References

[1] S. Aagerholm. Mechanizing program verification in hol. Technical Report Daimi IR-111, Aarhus University, 1992.

[2] R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.

[3] R. J. R. Back. On correct refinement of programs. *J. Computer and Systems Sciences*, 23(1):49 – 68, August 1981.

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

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

[6] R. J. R. Back. Refinement of parallel and reactive programs. Technical report, Marktoberdorf Summer School on Programming Logics, 1992.

[7] R. J. R. Back and J. von Wright. Refinement calculus I: Sequential nondeterministic programs. Reports on computer science and mathematics 92, Åbo Akademi, 1989.

[8] R. J. R. Back and J. von Wright. Duality in specification languages: A lattice theoretic approach. *Acta Informatica*, 27(7):583–625, 1990.

[9] R. J. R. Back and J. von Wright. Predicate transformers and higher order logic. In *REX Workshop on Semantics: Foundations and Applications*, Beekbergen, the Netherlands, 1992.

[10] J. H. C.A.R. Hoare and J. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25:71–76, 1987.

[11] W. Chen and J. T. Udding. Towards a calculus of data refinement. In *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, Groningen, The Netherlands, June 1989. Springer–Verlag.

[12] E. W. Dijkstra. *A Discipline of Programming.* Prentice–Hall International, 1976.

[13] E. W. Dijkstra and C. Scholten. *Predicate Calculus and Program Semantics.* Springer–Verlag, 1990.

[14] P. Gardiner and C. Morgan. Data refinement of predicate transformers. *Theoretical Comput. Sci.*, 87(1):143–162, 1991.

[15] M. J. Gordon. Hol: A proof generating system for higher order logic. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.

[16] D. Gries. *The Science of Programming.* Springer–Verlag, New York, 1981.

[17] D. Gries and J. Prins. A new notion of encapsulation. In *Proc. SIGPLAN Symp. Language Issues in Programming Environments*, June 1985.

[18] W. Hesselink. Command algebras, recursion and program transformation. *Formal Aspects of Computing*, 2:60–104, 1990.

[19] C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.

[20] C. A. R. Hoare, I. J. Hayes, J. He, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. Spivey, and A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987.

[21] C. Jones. *Software Development: A Rigorous Approach.* Prentice–Hall International, 1980.

[22] C. C. Morgan. Data refinement by miracles. *Information Processing Letters*, 26:243–246, January 1988.

[23] C. C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, July 1988.

[24] C. C. Morgan. *Programming from Specifications.* Prentice–Hall, 1990.

[25] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.

[26] J. M. Morris. Laws of data refinement. *Acta Informatica*, 26:287–308, 1989.

[27] J. von Wright. Data refinement with stuttering. Reports on computer science and mathematics 137, Åbo Akademi, 1992.

[28] J. von Wright. The lattice of data refinement. Reports on computer science and mathematics 130, Åbo Akademi, 1992.