On Practical Application of Relational Calculus

Olaf Owe
Department of Informatics
University of Oslo
Norway

May 15, 1992

Contents

1 Introduction

We use relations as a means to describe program behaviour. A program is seen as the relation consisting of all possible pairs (a, b) such that there is a normally terminating execution from initial state a resulting in state b. Thus, one may define the relational semantics of programs, and one may specify programs by means of relations. By a relational calculus one may reason about programs and specifications, and refine specifications into programs [2, 3].

Relational calculus may seem superior to traditional Hoare Logic as a formalism for reasoning about programs, especially from a mathematical point of view, for instance "adaptation" is much simpler — but it is unclear how to best apply this formalism on actual programs. We will focus on practicality, by suggesting a relational language and a calculus which are useful for specification, refinement and reasoning about actual programs. Our priming convention allows relations to be expressed directly by means of program variables, without the use of dummy variable or state names. We give a clear correspondence to Hoare Logic.

Programming typically focus on state changes rather than "non-changes"; and the same is often desirable in specifications. However, in the relational calculus any non-changes must be explicitly specified. We therefore introduce a "framing operator" which allows specification of changes in some program variables with the understanding that other variables are unchanged. The relational semantics of program constructs is conveniently defined by means of framed relations.

We first consider a relational calculus in which the variable space is assumed to be fixed and known. However, in actual program development it is often the case that the set of program variables is not fully determined. More program variables could be added when a specification/program is refined. And different program states may have different variable spaces. It is therefore desirable to be able to define a calculus which does not presume knowledge of the whole variable space. Framed specifications are valuable in this setting. We develop a relational calculus for framed relations based on an unknown and flexible variable space.

Finally we consider the case that the expression language contains partial functions. Program reasoning is complicated by the fact that the evaluation of an expression terminates normally only if the expression has a (welldefined) value. Our relational calculus is extended to this case.

1.1 Relationship to other work

In contrast to Hoare and He in [2, 3], we do not insist that programs define total relations; in particular, we let the empty relation correspond to abnormal termination or non-termination. The refinement-operator $\subseteq$ then captures partial correctness, since the empty relation is included in any relation. And so does the weakest prespecification operator ($\backslash$) as well as the strongest postspecification operator ($/$). In our framework one cannot distinguish the program 'output 1 or abort' from the program which only outputs 1. In total correctness reasoning it is essential to distinguish the two, but in partial correctness reasoning it is not, since they have the same set of (partially correct) refinements.

Another essential difference from [2] concerns refinement of so-called partial relations. A relation is said to be *total* if it contains an output state for every input state, and otherwise *partial*. Specifications defining partial relations frequently occur in practice, for example, the relation $\{(x, x') \mid x * x' = 1\}$ is partial when the input x and the output x' range over rational numbers. In the framework of [2], a partial relation may be refined, but never into a program — whereas in our framework any partial relation may be refined into a program. Consider the above example:

In our framework an implementation must abort for input 0; whereas in the framework of [2] neither abort nor any other implementation satisfies the specification. Since it can be difficult to tell whether a given specification is total or not, this is of practical importance. For this reason [2] offers a + operator which makes any specification total, however, the mathematical complexity added by a + application is in general non-trivial.

Even though the refinement-operator captures partial correctness, the equality relation allows total correctness reasoning, expressing equivalence between relations, and also programs, provided the union-operator is interpreted as "angelic (fair) choice" rather than non-deterministic ("demonic") choice. For such programs one may even strengthen the refinement operator so that it captures total correctness.

2 Statically known variable space

Assume for now that the program variable space is fixed, and let v denote the total list of program variables (in some order). A relation may then be seen as a set of pairs of input states and output states, and may be expressed as

$$\{(v, v') \mid p\}$$

where p is a Boolean expression in v and v', and where v' denotes the list of primed program variables. A state is here seen as a list of values, one value for each program variable. (A more expressive notion of state can be used, but this easily leads to a more complex syntax for expressing relations.) We may abbreviate the above relation to the Boolean expression

$$p$$

provided the primed and unprimed variables are understood as above, letting an unprimed program variable denote its input value and a primed program variable denote its output value. Thus, T denotes the relation consisting of all possible pairs, and F denotes the empty relation. The identity relation may be written as $v' = v$, denoted I. This *priming convention* follows the tradition of the Z language. For our purposes the following advantages are essential:

- We may express relations without help of other variables than the program variables themselves and their primed versions. In particular no "dummy" variables are needed.
- A Boolean expression without primes, say b, has the same meaning as in [2]: it specifies the set of input and output states such that the input state satisfies b and there is no restriction on the output state.

2.1 Notation and definitions

We use the following conventions:

x, y, v, w denote (lists of distinct) unprimed variables

e denotes (lists of) expressions over unprimed and primed variables. An expression is said to be *unprimed* if it does not contain primed variables.

e' denotes the expression e with all (unprimed) program variables occurrences primed

p, q denote Boolean expressions over unprimed and primed variables

$V(e)$ denotes the set of unprimed variables which occur unprimed in e (excluding proof or program constants)

$W(e)$ denotes the set of unprimed variables which occur primed in e

$x{\leftarrow}e$ denotes the substitution of e for the variable x; e and x may be lists of the same length, and $x_1, x_2, .., x_n{\leftarrow}e_1, e_2, .., e_n$ (where the x-es are distinct variables) denotes the simultaneous substitution of e_i for x_i $(i = 1, .., n)$,

s denotes substitutions

$W(s)$ denotes the set of variables occurring to the left of $\leftarrow$ in the substitution s. From now on, we consider substitutions such that $W(s)$ is unprimed.

$[s1]e[s2]$ denotes the expression e with unprimed variables substituted as specified by the substitution $s1$, and primed variables substituted as specified (on their unprimed versions) by the substitution $s2$. For instance, $[x{\leftarrow}2](x' = x + 1)[x{\leftarrow}3]$ denotes $3 = 2 + 1$

w used as a substitution, denotes the substitution $w{\leftarrow}w$

w' used as a substitution, denotes the substitution $w{\leftarrow}w'$.

Thus $[v]p[v']$ is the same as p, and we abbreviate $[v]p[s]$ to $p[s]$, and $[s]p[v']$ to $[s]p$. And e' is the same as $[v']e$ (provided there are no other variables than v).

Following [2], we introduce the following operators on relations; however, due to our priming convention, we may define these operators directly by means of predicate calculus:

$\breve{p}$	$==$	$[v']p[v]$	*(converse (inverse relation))*
$\overline{p}$	$==$	$\neg p$	*(complement)*
$p?$	$==$	$\exists v' : p$	*(domain)*
$p \ ; \ q$	$==$	$\exists s : p[s] \wedge [s]q$	*(composition)*
$p \setminus q$	$==$	$\forall s : [v']p[s] \Rightarrow q[s]$	*(weakest prespecification)*
p/q	$==$	$\forall s : [s]q[v] \Rightarrow [s]p$	*(strongest postspecification)*
$p \subseteq q$	$==$	$\forall v, v' : p \Rightarrow q$	*(inclusion)*
$p \sqsubseteq q$	$==$	$p \subseteq q \wedge \forall v : p? \Leftrightarrow q?$	*(stong inclusion)*
$p \cup q$	$==$	$p \vee q$	*(union)*
$p \cap q$	$==$	$p \wedge q$	*(intersection)*

where $W(s)$ is v. A meta-formula quantified over a substitution with a given W-set, say $\exists s : p$ where $W(s)$ is w, denotes $\exists z : pp$ where z is a list of fresh variables of the same length as w and pp is p with all occurrences of s replaced by the substitution $[w{\leftarrow}z]$. For a programs S, $S?$ is the condition that S may terminate normally, and for angelic programs the $\sqsubseteq$-operator expresses totally correct refinement.

2.2 Some immediate results

The / and $\setminus$ operators may be used for program refinement, since $p; (q/p) \subseteq q$ and $(p \setminus q); p \subseteq q$ follow from the above definitions. We may derive the following equivalences concerning refinement:

$$
\begin{array}{lcl}
(p1; p2) \setminus q & \Leftrightarrow & p1 \setminus (p2 \setminus q) \\
(p1 \cup p2) \setminus q & \Leftrightarrow & (p1 \setminus q) \cap (p2 \setminus q) \\
(p1 \cap p2) \setminus q & \Leftrightarrow & (p1 \setminus q) \cup (p2 \setminus q) \\
\textsc{f} \setminus q & \Leftrightarrow & \textsc{t} \\
\textsc{t} \setminus q & \Leftrightarrow & \forall v' : q \\
q \setminus \textsc{f} & \Leftrightarrow & \neg(\exists v' : q)' \\
q \setminus \textsc{t} & \Leftrightarrow & \textsc{t}
\end{array}
$$

$$
\begin{array}{lcl}
q/(p1; p2) & \Leftrightarrow & (q/p1)/p2 \\
q/(p1 \cup p2) & \Leftrightarrow & (q/p1) \cup (q/p2) \\
q/(p1 \cap p2) & \Leftrightarrow & (q/p1) \cap (q/p2) \\
q/\textsc{f} & \Leftrightarrow & \textsc{t} \\
q/\textsc{t} & \Leftrightarrow & \forall v : q \\
\textsc{t}/q & \Leftrightarrow & \textsc{t} \\
\textsc{f}/q & \Leftrightarrow & \neg(\exists v : q)[v]
\end{array}
$$

Furthermore, we derive

$$
\begin{array}{lcll}
p; \textsc{t} & \Leftrightarrow & p? & \\
b? & \Leftrightarrow & b & b \textit{ unprimed}
\end{array}
$$

The above definitions assume a fixed set of program variables (v). When the program variable space is not yet decided, the meta variable v does not represent a known entity. Rules and definitions using v are then impractical. However, notice that program variables not mentioned in p or q may be added or removed from v without affecting the above equational definitions (in Section **??**). One may therefore reformulate the equations without using the assumption that the variable space is known, and instead use the variable space locally present in p and q:

$$
\begin{array}{lcll}
\check{p} & \Leftrightarrow & [v']p[v] & v = V(p) \cup W(p) \\
\overline{p} & \Leftrightarrow & \neg p & \\
p? & \Leftrightarrow & \exists v' : p & v = W(p) \\
p \;;\; q & \Leftrightarrow & \exists s : p[s] \wedge [s]q & W(s) = W(p) \cup V(q) \\
p \setminus q & \Leftrightarrow & \forall s : [v']p[s] \Rightarrow q[s] & v = V(p), W(s) = W(p, q) \\
p/q & \Leftrightarrow & \forall s : [s]q[v] \Rightarrow [s]p & v = W(q), W(s) = V(p, q) \\
p \subseteq q & \Leftrightarrow & \forall v, w' : p \Rightarrow q & v = V(p, q), w = W(p, q) \\
p \sqsubseteq q & \Leftrightarrow & p \subseteq q \wedge \forall v : p? \Leftrightarrow q? & v = V(p, q) \\
p \cup q & \Leftrightarrow & p \vee q & \\
p \cap q & \Leftrightarrow & p \wedge q &
\end{array}
$$

Local variables. Let the construct **var** x **in** p **end** introduce x as a local program variable with scope p, and define $W($**var** x **in** p **end**$)$ as $W(p) - x$.

$$(\textbf{var}\ \ x\ \ \textbf{in}\ \ p\ \ \textbf{end}) \Leftrightarrow (\forall x : \exists x' : p)$$

However, if there is an outer program variable x, it is hidden inside p and we get

$$(\textbf{var}\ \ x\ \ \textbf{in}\ \ p\ \textbf{end}) \Leftrightarrow x' = x \wedge (\forall x : \exists x' : p)$$

2.3 Programs as relations

A program is seen as the relation consisting of all possible pairs (a, b) such that there is an execution from initial state a which terminates normally, resulting in state b. Thus a non-terminating

program corresponds to the empty relation (F); and also an abnormally terminating program, such as **abort**, corresponds to the empty relation. And **skip** corresponds to the identity relation. Non-deterministic choice corresponds to the $\cup$-operator. The $\cap$-operator corresponds to parallel computation provided $W(p)$ and $W(q)$ are disjoint: $p \cap q$ may then be implemented by doing p and q in parallel provided updating on variables in $V(p,q) \cap W(p,q)$ is done on local copies (which are (initially) copied from and (finally) to the real ones). It would even be possible to allow $W(p)$ and $W(q)$ to be non-disjoint, letting $p \cap q$ abort when p and q give conflicting final values of variables in $W(p) \cap W(q)$.

Consider programs such that the set of initial states from which there are normally terminating executions, is disjoint from those from which there are abnormal or non-terminating executions. Clearly, deterministic programs belong to this category, and so does non-deterministic programs provided the $\cup$-operator represent "angelic choice", in the sense that $p \cup q$ terminates if either p or q terminates. Angelic choice may be implemented by concurrency, letting the first normally terminating alternative kill the other alternatives. Thus $p \cup$ **abort** is equivalent to p, even with respect to termination. And the above implementation of the $p \cap q$ also belongs to this category when $W(p)$ and $W(q)$ are disjoint.

For programs restricted as above, the relational semantics captures termination aspects, since $p?$ expresses that p terminates normally; and total correctness reasoning is possible. In particular, $p = q$ means that p and q are equal in all respects, including termination aspects; and $p \sqsubseteq q$ expresses that p is a totally correct refinement of q.

2.3.1 Framing

In programming, one typically focus on changes being made, and unmentioned program variables are usually unchanged. In contrast a relation gives no restriction on changes on primed variables not mentioned. For instance, the assignment statement $x := x + 1$ does not correspond to the relation $x' = x+1$ because the relation does not restrict the final values of other program variables than x. In particular one need to know the total variable space in order to express the identity relation.

We introduce the notation $< p >$ to denote the relation $p \wedge w' = w$ where w is the list of program variables not occurring primed in p, i.e., w is the total variable space except $W(p)$. For instance, $x := x + 1$ clearly corresponds to the relation $< x' = x + 1 >$; and $< \mathrm{T} >$ is the identity relation. A convenient corollary is that $< p \wedge y' = y >$ is equivalent to $< p >$ when y is not in $W(p)$.

Notice that $W(< e >)$ and $V(< e >)$ depend on the total variable space. Thus the equations above for unknown variable space (Section 2.2) are not well suited for reasoning about framed specifications. The framing operator is not really useful unless one develops a calculus for framed relations which does not depend on V and W-sets of framed specifications.

The framing operator is natural and practical for specification purposes, as well as for explaining the effect of statements. We give below the relational semantics of some basic program constructs:

Assignments. The simultaneous assignment $w := e$ where e is an unprimed expression list of the same length as w, corresponds to the relation $< w' = e >$.

Tests and guards. Let b be unprimed. It follows that the relation $< b >$ corresponds to testing b as in

$$\textbf{if } b \rightarrow \textbf{skip } [] \ \neg b \rightarrow \textbf{abort fi}$$

And the if-statement **if** ... [] $b_i \rightarrow S_i$ [] ... **fi** corresponds to the relation

$$... \cup (< b_i >; R_i) \cup ...$$

where R_i denotes the relation corresponding to the statement S_i. Notice that $< b_i >; R_i$ is equivalent to $b_i \cap R_i$.

Recursion. Relations are naturally ordered by $\subseteq$, with F as the minimum. The meaning of a recursively defined program, or relation, is the least fixpoint, defined as follows:

$$\mu X.F(X) == lim_i \ F^i(\mathbf{abort})$$

where i ranges over the natural numbers, provided F is monotonic in the sense that $p \subseteq q \Rightarrow F(p) \subseteq F(q)$, and continuous (each increasing chain has a limit). By monotonicity, we have $F^i(\mathrm{F}) \subseteq F^{i+1}(\mathrm{F})$, and by continuity, this really is a fixpoint. It suffices that F is em quasi-monotonic, defined as $\mathrm{F} \subseteq q \Rightarrow F(\mathrm{F}) \subseteq F(q)$.

The relation operators and constructs introduced above are all quasi-monotonic and continuous. In particular, the framing operator is continuous and quasi-monotonic (but not monotonic); therefore $< \mu X.F(X) >$ reduces to $\mu X.< F(X) >$, i.e., $\cup_i < F^i(\mathbf{abort}) >$. In contrast, the framing operator is not continuous in the setting of [2]. (And neither is the ; operator, when not restricted to programs, as pointed out to us by Bjørn Kirkerud).

A practical proof rule for reasoning about recursion is that from $F(p) \subseteq p$ one may conclude $\mu X.F(X) \subseteq p$. This rule specializes to invariant-like reasoning in the case of tail-recursion.

All in all we have given the relational semantics of the programming language consisting of assignments, skips, aborts, non-deterministic choice, tests, guarded commands, recursion, sequential as well as restricted parallel composition.

3 Relationship to traditional Hoare logic

The partial correctness specification $\{p\}S\{q\}$ of traditional Hoare logic, may be rewritten within the relational notation as

$$S \subseteq \{p, q\}$$

letting $\{p, q\}$ denote

$$\forall w : p \Rightarrow q'$$

where w is the list of proof-constants, i.e., non-program variables occurring in p or q (but not in S), assuming neither p nor q contains primes. For example, we have $x := x + 1 \subseteq \{x = 1, x = 2\}$ and also $x := x+1 \subseteq \{x = x0, x = x0+1\}$ where $x0$ is a proof constant. Proof constants denoting initial values are quite common in Hoare reasoning.

It is desirable to avoid proof constants when possible (especially when the total variable space is not yet decided). By allowing primed variables in p and q, one may avoid typical proof constants. It is then convenient to redefine $\{p, q\}$ as denoting

$$\forall w : p[v] \Rightarrow \check{q}$$

(As before, w is the proof constants, if any). For instance, $x := x + 1 \subseteq \{\mathrm{T}, x = x' + 1\}$. We still have that the Hoare sentence $\{p\}S\{q\}$ is the same as $S \subseteq \{p, q\}$ for unprimed p and q. Furthermore, the Hoare sentence

$$\{v = v' \wedge p\}S\{q\}$$

is also equivalent to $S \subseteq \{p, q\}$ (p, q may now contain primes). The redefinition of $\{.., ..\}$ is superior to the first one, in the sense that an arbitrary relation p can now be expressed without proof constants as $\{\mathrm{T}, \check{p}\}$ (compared to $\{v = v0, [v{\leftarrow}v0]p\}$ with the original definition). And thus, $S \subseteq p$ is equivalent to the the Hoare sentence $\{v = v'\}S\{\check{p}\}$. Thus, relational program refinement corresponds directly to Hoare logic.

If proof constants are used to denote final values, say $\{p\}S\{v = v' \wedge q\}$, this may be expressed as

$$S \subseteq \{p', q[v]\}$$

(or as $S \subseteq \{p, q\}$ with the first definition of $\{.., ..\}$).

With the above notation, all rules and axioms of traditional Hoare logic may easily be rewritten into the relational framework. One may even simulate Hoare-like reasoning within the relational framework, without added complications; the fact that all relational operators are defined explicitly in predicate calculus may simplify reasoning significantly. In addition, one may prove soundness and completeness of Hoare logic based on the relational semantics of the programming constructs.

For instance, the Hoare axiom

$$\{[x{\leftarrow}e]q\}\ x := e\ \{q\}$$

where q and e are unprimed, can be rewritten as

$$x := e \subseteq \{[x{\leftarrow}e]q, q\}$$

which can be rewritten as

$$< x' = e > \subseteq ([x{\leftarrow}e]q \Rightarrow [v']q)$$

With the below calculus for framed relations this reduces directly to T.

4 A calculus for framed relations

We here try to develop a calculus for framed relations in the case where the variable space is unknown. It turns out that this is possible for all relational operators except complement; $\overline{< p >}$ obviously depends on the total variable space. The calculus is based on the partial knowledge about the program variable space which is locally present in the relations: The primed variables used to express a relation must be among the program variables (at that point), i.e., $W(p)$ must be part of $W(< p >)$. There may be different variable spaces at different points (states) in a program/specification.

Observe that if I denotes the identity relation, then $\check{I} = (I; I) = (I \setminus I) = (I/I) = (I \cup I) = (I \cap I) = I$ and $I \subseteq I$ and $I?$ are true. Thus, program variables not occurring in p or q do not cause difficulties when p or q are framed: for the first group of operators we simply frame the result. The non-trivial program variables are those which occur primed in one of p and q, but not in both. Let the variable list x be $W(p) - W(q)$ and y be $W(q) - W(p)$. This leads to the following equations where the relations are "framed":

$$
\begin{array}{lcll}
< \check{p} > & \Leftrightarrow & < [v']p[v] > & v = V(p) \cup W(p) \\
< p >? & \Leftrightarrow & \exists v' : p & v = W(p) \\
< p > \ ; \ < q > & \Leftrightarrow & < \exists s : p[s] \wedge [s\&x']q > & W(s) = W(p) \cap W(q) \\
< p > \ \backslash \ < q > & \Leftrightarrow & < \forall s : [v']p[s] \Rightarrow (q \wedge x' = x)[s] > & v = V(p), W(s) = W(p) \\
< p > / < q > & \Leftrightarrow & < \forall s : [s]q[v] \Rightarrow [s](p \wedge y' = y) > & v = W(q), W(s) = W(q) \\
< p > \subseteq < q > & \Leftrightarrow & \forall v, w' : p \Rightarrow q[y] \wedge x' = x & v = V(p,q), w = W(p) \\
< p > \sqsubseteq < q > & \Leftrightarrow & < p > \subseteq < q > \wedge \forall v : p? \Leftrightarrow q? & v = V(p,q) \\
< p > \cup < q > & \Leftrightarrow & < p \wedge y' = y \vee q \wedge x' = x > & \\
< p > \cap < q > & \Leftrightarrow & < p[x] \wedge q[y] > &
\end{array}
$$

where $s1\&s2$ denotes the simultaneous composition of the substitutions $s1$ and $s2$, requiring that $W(s1)$ and $W(s2)$ are disjoint. Notice that $< \check{p} >$ is equivalent to $< p >$, $< p >?$ is equivalent to $p?$, and $< p > \cap < q >$ is equivalent to $< p \cap q >$ (using the results of Section 2.2).

Proof: We first rewrite $< p >$ and $< q >$ as $< p \wedge y' = y >$ and $< q \wedge x' = x >$, respectively. Since they now have the same variable sets, we may use the original definitions above (with $W(p,q)$ as the variable space, v), if we ignore other program variables. And since the operators above give identity ($\subseteq$ and ? give true) when p and q are identity, we may frame the result when other program variables are taken into consideration (for $\subseteq$ and ? other variables cause no problems). Thus $< p > / < q >$ is equivalent to $< ((p \wedge y' = y) \ / \ (q \wedge x' = x)) >$ and $< p > \subseteq < q >$ is equivalent to $(p \wedge y' = y) \subseteq (q \wedge x' = x)$. We show the derivations for the /-operator in detail, the others are similar or simpler.

$$
\begin{array}{lll}
< p > / < q > & & \\
\Leftrightarrow & < (p \wedge y' = y) \ / \ (q \wedge x' = x) > & \\
\Leftrightarrow & < \forall s : [s](q \wedge x' = x)[v] \Rightarrow [s](p \wedge y' = y) > & v = W(s) = W(p,q) \\
\Leftrightarrow & < \forall s : [s]q[v] \wedge x = [s]x \Rightarrow [s](p \wedge y' = y) > & v = W(q), W(s) = W(p,q) \\
\Leftrightarrow & < \forall s : [s]q[v] \Rightarrow [s](p \wedge y' = y) > & v = W(q), W(s) = W(q)
\end{array}
$$

4.1 Framed refinements

In program refinement it is often the case that a loose, unframed specification q is (partly) refined into a framed specification/program p. By the operators above we may do refinement in the following ways: $p \subseteq q$, $p; q/p \subseteq q$ and $p \setminus q; p \subseteq q$
It is interesting then to look at the situation where p is framed and q is not. For such situations we derive the following rules:

$$
\begin{array}{lcll}
< p > \ \backslash \ q & \Leftrightarrow & \forall s : [v']p[s] \Rightarrow q[s] & v = V(p), W(s) = W(p) \\
p/ < q > & \Leftrightarrow & \forall s : [s]q[v] \Rightarrow [s]p & v = W(q), W(s) = W(q) \\
< p > \subseteq q & \Leftrightarrow & \forall v, w' : p \Rightarrow q[y] & v = V(p,q), w = W(p), y = W(q) - W(p) \\
< p > \sqsubseteq q & \Leftrightarrow & < p > \subseteq q \wedge \forall v : p? \Leftrightarrow q? & v = V(p,q)
\end{array}
$$

Proof: Not included.

Let us demonstrate the usefulness of the last equations by deriving the weakest prespecification of assignments and tests.

Assignments. By the above relational semantics for assignments, we have that

$$x := e \setminus q$$

by definition is $< x' = e > \setminus q$ which using the equations of Sec. 4.1 reduces to

$$\forall w : (w = e') \Rightarrow q[x{\leftarrow}w])$$

which simplifies to

$$q[x{\leftarrow}e']$$

Taking our priming convention into account, this reflects the weakest precondition known from traditional Hoare logic.

Tests. Let b be unprimed. The prespecification

$$< b > \setminus q$$

reduces directly to

$$b' \Rightarrow q$$

since $W(b)$ is empty. As guarded commands are defined in terms of tests, semicolon and union, their prespecifications follow from the rules above.

Recursion. The prespecification

$$(\mu X.F(X)) \setminus q$$

reduces to

$$\cup_i \ (F^i(\mathbf{abort}) \setminus q)$$

which is equal to the limit of $F^i(\mathbf{abort}) \setminus q$ when i grows.

Local variables. Let the construct **var** x **in** p **end** introduce x as a local program variable with scope p, and define $W(\mathbf{var}\ \ x\ \ \mathbf{in}\ \ p\ \ \mathbf{end})$ as $W(p) - x$.

$$(\mathbf{var}\ \ x\ \ \mathbf{in}\ \ p\ \ \mathbf{end}) \Leftrightarrow (\forall x : \exists x' : p)$$

However, if there is an outer program variable x, it is hidden inside p and we get

$$(\mathbf{var}\ \ x\ \ \mathbf{in}\ \ p\ \mathbf{end}) \Leftrightarrow x' = x \wedge (\forall x : \exists x' : p)$$

In either case we have

$$< \mathbf{var}\ \ x\ \ \mathbf{in}\ \ p\ \ \mathbf{end} > \ \Leftrightarrow \ < \forall x : \exists x' : p >$$

which again shows the strength of framed specifications.

5 Undefined expressions

We now consider the case that the expression language contains partial functions, and thus that an expression need not have a (well-defined) value in all states. A formula p representing a relation is now reinterpreted as

$$\{(v, v') \mid p == \mathrm{T}\}$$

i.e., the set of all pairs for which p has a (well-defined) value, and is true. We let $==$ denote strong equality (equal in all respects), which is non-monotonic and therefore not part of our expression language, whereas $=$ denotes strict and executable equality. For instance, with our reinterpretation

the relation $x' = 1/x$ does not contain any pair with 0 as input value for x; and thus defines the same relation as $x * x' = 1$.

The main question to investigate now is: Do the equations above for the relational operator generalize to the case of partial functions? In order to make these equations meaningful we adopt the validity concept of WS logic [5, 4]; which means that a formula p is valid if it is well-defined and true, i.e., $p == \mathrm{T}$.[1] The Boolean operators $(\Rightarrow, \wedge, \vee, \neg)$ are generalized as suggested by Kleene, and are non-strict but monotonic, and may be considered executable (using angelic choice).

As in WS we let Δe denote the welldefinedness of the expression e, for instance $\Delta(x' = 1/x)$ reduces to $x \neq 0$, whereas $\Delta(x * x' = 1)$ reduces to T, given the usual semantics for rational numbers. The Δ-operator is non-monotonic and not part of our expression language. It may be defined constructively, as in WS, defining $\Delta(p \cup q)$ as $\Delta p \vee \Delta q$ (angelic choice). Alternatively, $\Delta p \wedge \Delta q$ reflects the welldefinedness requirement for non-deterministic demonic choice.

The relational operators defined above (in Section 4) are now redefined as follows: The three occurrences of $\Rightarrow$ in the right hand sides are replaced by the operator $\supset$, defined by

$$p \supset q == (p == \mathrm{T}) \Rightarrow (q == \mathrm{T})$$

expressing that validity of p implies validity of q. This operator is non-monotonic, and therefore not part of our expression language.

The above relational definitions of the programming constructs (in Section 4) generalize to the case of possibly ill-defined expressions without modifications. For instance, the assignment $x := e$ is defined as $< x' = e >$ which in WS is the same as $(\Delta e) \wedge < x' = e >$, which shows that the assignment does not terminate normally when the right hand side is ill-defined.

However, the above examples of derivations of prespecifications must be redone since the underlying logic is now changed. For instance, reconsider the derivations of the prespecification of $x := e$. As before we have that

$$x := e \setminus q$$

is by definition the same as

$$< x' = e > \setminus q$$

which now is equivalent to

$$\forall w : (w = e') == \mathrm{T}) \Rightarrow (q[x \leftarrow w] == \mathrm{T})$$

Using WS logic this simplifies to $\Delta e' \Rightarrow (q[x \leftarrow e'] == \mathrm{T})$ which is valid if and and only if

$$\Delta e' \Rightarrow q[x \leftarrow e']$$

is valid. This result shows that the assignment behaves like abort when e' is ill-defined.

Similarly, we obtain that validity of $< b > \setminus q$ now is equivalent to the validity of $(\Delta b' \wedge b') \Rightarrow q$ which is the same as $b' \supset q$.

[1] Since we are not here concerned about validity of formulas with assumption parts, we might as well have used LPF [1].

6 Conclusion

We suggest extensions of relational calculus intended to enhance its practical applications with respect to program reasoning. The main contribution of this paper is the use of framing mechanism allowing flexibility in the state space or set of program variables, including the siutuation that the set of program variables is changing when a specification or program is refined. This is for instance relevant for object-oriented class inheritance. We develop a relational calculus for framed relations allowing an unknown and flexible variable space.

In addition we consider the case that the expression language contains partial functions. In this case program reasoning is complicated by the fact that the evaluation of an expression terminates normally only if the expression has a (welldefined) value. And the presence of partial functions affects refinement with respect to to partial and total correctness. Our relational calculus is extended to this case.

References

[1] H. Barringer, J.H. Cheng, C.B. Jones: "A Logic Covering Undefinedness in Program Proofs." *Acta Informatica* 21 (1984), 251-269.

[2] Hoare and He: "The weakest prespecification", *Information Processing Letters*, 24 (1987), 127-132, North-Holland.

[3] Hoare, Hayes, He, Morgan, Roscoe, Sanders, Sorensen, Spivey, and Sufrin: "Laws of Programming", *Communications of the ACM*, 30 (1987), 8, p. 672-686.

[4] O.-J. Dahl, O. Owe: "Formal Development with ABEL, In *VDM'91: Formal Software Development Methods*, LNCS 552, p. 320-362, Springer Verlag, 1991.

[5] O. Owe: *Partial Logics Reconsidered*. To appear in *Formal Aspects of Computing*.