Fusion and Simultaneous Execution in the Refinement Calculus

Ralph-Johan Back* Michael Butler†

August 21, 1997

Abstract

In the refinement calculus, program statements are modelled as predicate transformers. A product operator for predicate transformers was introduced by Martin [18] and Naumann [25] using category theoretic considerations. In this paper, we look more closely at the refinement-oriented properties of this operator and at its applications. We also generalise the definition of the product operator to form what we call a fusion operator. Together, the fusion and product operators provide us with algebraic ways of composing program statements in the refinement calculus in order to model effects such as conjunction of specifications, simultaneous execution, and embedding of smaller programs into larger contexts.

1 Introduction

Dijkstra introduced weakest-precondition predicate transformers as a means of verifying total correctness properties of sequential programs [9]. The *refinement calculus* of Back [2], Morgan [21], and Morris [23] builds on this, regarding specifications and programs uniformly as predicate transformers. In the refinement calculus, the required behaviour of the program is specified as an abstract, possibly non-executable, program which is then refined by a series of correctness-preserving transformations into an efficient, executable program. The notion of correctness-preserving transformation is modelled by a refinement relation between programs which is transitive, thus supporting stepwise refinement, and is monotonic w.r.t. program constructors, thus supporting piecewise refinement. All transformation laws are derived from properties of predicate transformers.

The refinement calculus provides sequential composition and various choice and assignment operators that are generalisations of Dijkstra's operators, and the applications of these operators are well-known. However, the applications of an operator representing simultaneous execution of program statements are less well developed in the refinement calculus. Such an operator was introduced by Martin [18] and by Naumann [25] using category theoretic considerations. This *product*[1] operator combines predicate transformers by forming the cartesian product of their state spaces. In this paper, we examine the product operator, looking at various distributivity and refinement preserving properties of the operator. We show that the operator can be used to model simultaneous execution and to extend the state spaces of

*Department of Computer Science, Åbo Akademi, Finland, `backrj@abo.fi`.

†Department of Electronics and Computer Science, University of Southampton, United Kingdom, `M.J.Butler@ecs.soton.ac.uk`.

[1] The direction of arrows in [18] and [25] are the reverse of what we use, so our product operator corresponds to their co-product operator.

statements so they can be more easily matched with other statements. These features turn out to be important for embedding programs in larger environments, and for composing programs as used, for example, in the action system formalism [4].

The existing choice operators in the refinement calculus can be used to model disjunction of program specifications, but not for conjunction of specifications. We generalise the definition of the product operator slightly to form what we call a *fusion* operator. This operator can be used to model conjunction of specifications. Furthermore, the product operator turns out to be a special case of the fusion operator thus simplifying some of the proof effort.

We use the higher-order logic formalisation of the refinement calculus of Back and von Wright [6]. Here, a refinement logic is built up in a layered fashion, starting with booleans, which are then lifted to predicates, and then to predicate transformers using pointwise extension. If possible, we reason about predicate transformers purely at the predicate transformer level, but sometimes it is necessary to go to the predicate, and boolean levels. By having all three levels, we have the freedom to do this. There are examples of proof at all three levels in the appendix.

To simplify the development of the operators and their algebraic properties, we make the minimal assumptions on the structure of state spaces. In particular, we don't explicitly model program variables in the state. To make the theory useful for practical program development, we introduce a simple layer of syntactic sugaring that allows us to easily model program variables as cartesian components of the state.

The paper is organised as follows. Section 2 introduces the refinement calculus basics. Although this section is long, the material is straightforward and standard and is useful for a proper understanding of the later material. Section 3 introduces the fusion operator, the product operator and the derived product operator showing that the product operator is a special case of the fusion operator. Section 4 shows how the fusion operator models conjunction of specifications and compares our operator with some alternative approaches. Section 5 shows how the product operators model simultaneous execution. Section 6 looks at the notion of *least data refinement*, which provides a general mechanism for calculating data refinements of commands. It is shown how this notion applies to the product operators. Section 7 looks at some further uses of the product operator such as embedding commands in larger program contexts and composing action systems for modelling concurrent systems. These applications were our initial motivation for studying the product and fusion operators.

Some of the results here have appeared previously in an earlier paper [3]. However, this paper contains important new material, in particular, a comprehensive treatment of program variables, comparisons of our definitions with some alternative definitions and results on the least data refinement of products.

2 Refinement Calculus Basics

We work with *terms* of higher-order logic as described, for example, in [11]. A term in higher-order logic is an expression in a typed lambda calculus possibly having free variables. We write $t[x := e]$ for the term t with all free occurrences of variable x replaced by term e. The type of a term can be atomic, e.g., **Bool**, a function from type to type, $T_1 \to T_2$, or the cartesian product of types, $T_1 \times \cdots \times T_n$.

Predicate Lattice

A *predicate* over a set of states Σ is a boolean function $p : \Sigma \to \mathbf{Bool}$ which assigns a truth value to each state. The set of predicates on Σ is denoted $\mathcal{P}\,\Sigma$:

$$\mathcal{P}\,\Sigma \quad \widehat{=} \quad \Sigma \to \mathbf{Bool}.$$

We define the *entailment ordering* on predicates by pointwise extension: for $p, q : \mathcal{P}\,\Sigma$,

$$p \leq q \quad \widehat{=} \quad (\forall \sigma : \Sigma \cdot p\ \sigma\ \Rightarrow\ q\ \sigma).$$

The identically false predicate is denoted $\perp$, and the identically true predicate is denoted $\top$. Negation, conjunction, and disjunction of (similarly-typed) predicates are defined by pointwise extension, so that, e.g.,

$$(p \wedge q)\ \sigma \quad \widehat{=} \quad (p\ \sigma\ \wedge\ q\ \sigma).$$

For state space Σ, the set of predicates $\mathcal{P}\,\Sigma$ is a complete lattice under the entailment ordering. Conjunction and disjunction represent meet and join respectively, while $\top$ and $\perp$ represent top and bottom respectively.

Relational Lattice

A *relation* from Σ to Γ is a function $P : \Sigma \to \mathcal{P}\,\Gamma$ that maps each state σ to a predicate on Γ. We write

$$\Sigma \leftrightarrow \Gamma \quad \widehat{=} \quad \Sigma \to \mathcal{P}\,\Gamma.$$

This view of relations is isomorphic to viewing them as predicates on the cartesian space $\Sigma \times \Gamma$. The domain and range of relation P are denoted $dom\,P$ and $ran\,P$ respectively. Conjunction and disjunction of relations are defined pointwise, so that, e.g., $(P \wedge Q)\ \sigma\ \widehat{=}\ (P\ \sigma) \wedge (Q\ \sigma)$. The *inclusion ordering* on relations is defined by pointwise extension: for $P, Q : \Sigma \leftrightarrow \Gamma$,

$$P \leq Q \quad \widehat{=} \quad (\forall \sigma : \Sigma \cdot P\ \sigma\ \leq\ Q\ \sigma).$$

For state spaces Σ and Γ, the set of relations $\Sigma \leftrightarrow \Gamma$ is a complete lattice under the inclusion ordering. We write $P; Q : \Sigma \leftrightarrow \Lambda$ for the relational composition of $P : \Sigma \leftrightarrow \Gamma$ and $Q : \Gamma \leftrightarrow \Lambda$, and P^{-1} for the inverse relation. For function $f : \Sigma \to \Gamma$, we write f^{-1} for the relation $(\lambda \gamma \cdot \lambda \sigma \cdot \gamma = f \sigma) : \Gamma \leftrightarrow \Sigma$.

Predicate Transformer Lattice

A *predicate transformer* is a function $S : \mathcal{P}\,\Gamma \to \mathcal{P}\,\Sigma$ from predicates to predicates. We write,

$$\Sigma \mapsto \Gamma \quad \widehat{=} \quad \mathcal{P}\,\Gamma\ \to\ \mathcal{P}\,\Sigma.$$

Note the reversal of Γ and Σ: program statements in Dijkstra's calculus are identified with weakest-precondition predicate transformers that map a postcondition $q : \mathcal{P}\,\Gamma$ to the weakest precondition $p : \mathcal{P}\,\Sigma$ such that the program is guaranteed to terminate in a final state satisfying q whenever the initial state satisfies p. For program statement $S : \Sigma \mapsto \Gamma$, we say that S has *source* Σ (the initial state space) and *target* Γ (the final state space). Programs need not have identical initial and final state spaces. Note that, in order to aid intuition, we sometimes discuss commands from an operational viewpoint, though our formal reasoning is always in terms of predicate transformers.

The *refinement ordering* on predicate transformers is defined by pointwise extension: for $S, T : \Sigma \mapsto \Gamma$,

$$S \leq T \quad \widehat{=} \quad (\forall q : \mathcal{P}\,\Gamma \cdot S\ q\ \leq\ T\ q).$$

The refinement ordering on predicate transformers models the notion of total-correctness preserving program refinement. A total-correctness specification is typically given as a precondition-postcondition pair $(pre, post)$, and program (i.e., predicate transformer) S satisfies $(pre, post)$ if $pre \leq (S\ post)$. Now, for programs S and T, $S \leq T$ holds if and only if T satisfies any specification satisfied by S. Transitivity of the refinement ordering is inherited from transitivity of implication by pointwise extension

For state spaces Σ and Γ, the set of predicate transformers $\Sigma \mapsto \Gamma$ is a complete lattice under the refinement ordering. The bottom element is the predicate transformer *abort* that maps each postcondition to $\bot$, and the top element is the predicate transformer *magic* that maps each postcondition to $\top$. The *abort* statement is never guaranteed to terminate, while the *magic* statement is *miraculous* since it is always guaranteed to establish any postcondition. A miraculous statement cannot be implemented. For statement S, $halt\ S\ \widehat{=}\ S\ \top$ describes those initial states under which S is guaranteed to terminate, while $gd\ S\ \widehat{=}\ \neg(S\,\bot)$ (called *guard* of S) describes those initial states under which S behaves non-miraculously.

Conjunction and disjunction of (similarly-typed) predicate transformers are defined pointwise, so that, e.g., $(S \wedge T)\ q\ \widehat{=}\ (S\ q) \wedge (T\ q)$. Conjunction of statements models *demonic nondeterministic choice* between executing S and T (i.e., each alternative must establish the postcondition), whereas disjunction models *angelic nondeterministic choice* (i.e., some alternative must establish the postcondition). If $halt\ S = \neg halt\ T$, then $S \vee T$ becomes a deterministic choice, while if $gd\ S = \neg gd\ T$, then $S \wedge T$ becomes a deterministic choice.

Predicate Transformer Category

Sequential composition of program statements is modelled by functional composition of predicate transformers: the sequential composition of $S : \Sigma \mapsto \Gamma$ and $T : \Gamma \mapsto \Lambda$ is $S;T : \Sigma \mapsto \Lambda$, where for $p : \mathcal{P}\,\Lambda$,

$$(S;T)\ p \quad \widehat{=} \quad S\ (T\ p).$$

The program statement $skip_\Sigma$ is modelled by the identity predicate transformer on $\mathcal{P}\,\Sigma$.

Predicate transformers form a category: the category $PTran$ has state spaces as objects and predicate transformers as arrows. If Σ, Γ are state spaces, then each predicate transformer $S : \Sigma \mapsto \Gamma$ is an arrow with source Σ and target Γ. For composition of arrows, we use sequential composition and, as the identity arrow of object Σ, we use $skip_\Sigma$. As required for $PTran$ to be a category, we have

$$\begin{aligned} skip_\Sigma; S \ = \ S \ &= \ S; skip_\Gamma, \\ S;(T;U) \ &= \ (S;T);U. \end{aligned}$$

Embedding Functions, Predicates, and Relations

Given a relation $P : \Sigma \leftrightarrow \Gamma$, the *angelic update statement* $\{P\} : \Sigma \mapsto \Gamma$ and *demonic update statement* $[P] : \Sigma \mapsto \Gamma$ are defined by

$$\begin{aligned} \{P\}\ q\ \sigma \quad &\widehat{=} \quad (\exists \gamma : \Gamma \cdot (P\ \sigma\ \gamma) \wedge (q\ \gamma)) \\ [P]\ q\ \sigma \quad &\widehat{=} \quad (\forall \gamma : \Gamma \cdot (P\ \sigma\ \gamma) \Rightarrow (q\ \gamma)). \end{aligned}$$

When started in a state σ, $\{P\}$ angelically chooses a new state γ such that $P\sigma\gamma$ holds, while $[P]$ demonically chooses a new state γ such that $P\sigma\gamma$ holds. If no such new state exists then $\{P\}$ aborts, while $[P]$ behaves as *magic*. Angelic update is monotonic w.r.t. relational inclusion, while demonic update is anti-monotonic:

$$P \leq Q \quad \Rightarrow \quad \{P\} \leq \{Q\} \;\wedge\; [P] \geq [Q].$$

Sequential composition distributes through both updates:

$$\{P\};\{Q\} \;=\; \{P;Q\} \qquad \text{and} \qquad [P];[Q] \;=\; [P;Q].$$

For predicate $p : \mathcal{P}\,\Sigma$, let $\overline{p} : \Sigma \leftrightarrow \Sigma$ be the corresponding *test relation* for p, i.e., $(\lambda\sigma \cdot \lambda\sigma' \cdot p\ \sigma \ \wedge\ \sigma = \sigma')$. Then we write $\{p\}$ for $\{\overline{p}\}$ and $[p]$ for $[\overline{p}]$. $\{p\}$ models the *assert* statement that behaves as *skip* if p holds, otherwise it aborts. $[p]$ models the *guard* statement that behaves as *skip* if p holds, otherwise it behaves as *magic*.

Given a function $f : \Sigma \rightarrow \Gamma$, the *deterministic update statement* $\langle f \rangle : \Sigma \mapsto \Gamma$ is defined by

$$\langle f \rangle\ q\ \sigma \quad \widehat{=} \quad q\ (f\ \sigma).$$

$\langle f \rangle$ simply changes the state from σ to $f\sigma$. Let $\tilde{f}$ be the the deterministic relation corresponding to function f, i.e., $(\lambda\sigma \cdot \lambda\gamma \cdot f\ \ \sigma\ =\ \gamma)$. Then we have $\{\tilde{f}\}\ =\ \langle f \rangle\ =\ [\tilde{f}]$.

Homomorphic Properties

All predicate transformers S constructed using the operators described above will be *monotonic*, i.e., $p \leq q\ \Rightarrow\ S\ p \leq S\ q$. Predicate transformer S is *bottom homomorphic* if $S \perp = \perp$ (i.e., S never behaves miraculously), and *top homomorphic* if $S\ \top = \top$ (i.e., S always terminates). Predicate transformer S is *conjunctive* if $S\ (\wedge i \in I \cdot q_i) = (\wedge i \in I \cdot S\ q_i)$, for non-empty I. S is *universally conjunctive* if it is conjunctive and top homomorphic. *Disjunctive* and *universally disjunctive* predicate transformers are defined dually. Demonic update statements are universally conjunctive, angelic update statements are universally disjunctive, and deterministic update statements are both. Furthermore, the assert statement is conjunctive while the guard statement is disjunctive. The operators $\wedge$ and ; preserve the conjunctivity of their operands, while $\vee$ and ; preserve the disjunctivity of their operands.

Each of $\wedge$, $\vee$, and ; preserve refinement of monotonic predicate transformers, e.g., $S \leq S'\ \Rightarrow\ S;T \leq S';T$. We consider this to be an essential property since it allows for piecewise refinement of programs.

In Dijkstra's original calculus, all statements were non-miraculous and conjunctive [9].

Normal Form

Conjunctive predicate transformers can be written in a normal form involving an assertion followed by a demonic update: for conjunctive S, there exists a predicate p and relation Q such that

$$S \quad = \quad \{p\};[Q].$$

Thus, if we want to show that any conjunctive S enjoys some property, we can do so by showing that $\{p\};[Q]$ satisfies it for any p, Q.

Dually, disjunctive predicate transformers have a normal form: for disjunctive S, there exists a predicate p and relation Q such that

$$S \quad = \quad [p];\{Q\}.$$

Notation for Specification and Programming

Ordinary program constructs such as conditionals, recursions, and assignments may be modelled using the basic operators presented above. A recursive statement has the form $(\mu X \cdot F\ X)$ (least fixed-point of F) where X ranges over $\Sigma \mapsto \Sigma$ and F is a function from $\Sigma \mapsto \Sigma$ to $\Sigma \mapsto \Sigma$. F should be monotonic (w.r.t. refinement) for its least fixed-point to exist, and this is guaranteed if $F\ X$ is constructed from X using the operators described above. Conditional and loop statements are defined as:

$$\begin{array}{rcl} \mathbf{if}\ p\ \mathbf{then}\ S\ \mathbf{else}\ T\ \mathbf{fi} & \widehat{=} & [p]; S\ \wedge\ [\neg p]; T \\ \mathbf{do}\ p \rightarrow S\ \mathbf{od} & \widehat{=} & (\mu X \cdot \mathbf{if}\ p\ \mathbf{then}\ S; X\ \mathbf{else}\ \mathit{skip}\ \mathbf{fi}). \end{array}$$

Program variables may be modelled as cartesian components of the state space, and a multiple assignment $x, y := d, e$ in a state space with three components, representing program variables x, y, z, may be modelled by the deterministic update:

$$\langle \lambda x, y, z \cdot (d, e, z) \rangle.$$

We introduce a layer of syntactic sugaring to deal with program variables in a more conventional fashion. A program variable has a name and a type and we write $x : T$ for program variable named x with type T. Sometimes we omit the type of a variable if it is not relevant or can be understood from context. We write[2]

$$(\ \mathbf{var}\quad (x_1 : \Sigma_1, \ldots, x_m : \Sigma_m)\ \mapsto\ (y_1 : \Gamma_1, \ldots, y_n : \Gamma_n)\ \cdot\ S\ )$$

to say that statement S reads from program variables $x_1 \ldots x_m$ and writes to program variables $y_1 \ldots y_n$. S then has type

$$(\Sigma_1 \times \cdots \times \Sigma_m) \mapsto (\Gamma_1 \times \cdots \times \Gamma_n).$$

Let u be a tuple of program variables of the form $x_1 : \Sigma_1, \ldots, x_m : \Sigma_m$. We shall use u both as a quantified variable, e.g., $(\lambda u \cdot t)$ and as the term $(x_1, \ldots, x_n)$. Let $u[x_i := e]$ be the tuple u with x_i replaced by expression e and let $u \setminus x_i$ be the tuple u with x_i removed. More generally, we may write $u[v := E]$ and $u \setminus v$, where v is a list of program variables and E is a list of expressions. We write $(\mathbf{var}\ u \cdot S)$ for $(\mathbf{var}\ u \mapsto u \cdot S)$.

We require that **var** distributes through the statement connectives:

$$\begin{array}{rcl} (\mathbf{var}\ u \mapsto v \cdot S \wedge T) & \widehat{=} & (\mathbf{var}\ u \mapsto v \cdot S) \wedge (\mathbf{var}\ u \mapsto v \cdot T) \\ (\mathbf{var}\ u \mapsto v \cdot S \vee T) & \widehat{=} & (\mathbf{var}\ u \mapsto v \cdot S) \vee (\mathbf{var}\ u \mapsto v \cdot T) \\ (\mathbf{var}\ u \cdot S; T) & \widehat{=} & (\mathbf{var}\ u \cdot S); (\mathbf{var}\ u \cdot T). \end{array}$$

Assignment is defined in terms of deterministic update:

$$(\mathbf{var}\ u \cdot v := E) \quad \widehat{=} \quad \langle \lambda u \cdot u[v := E] \rangle.$$

Let b be a boolean term whose free variables may include program variables. The nondeterministic update statement $x := x' \mid b$ assigns some value x' satisfying b to program variable x. This is defined in terms of demonic update:

$$(\mathbf{var}\ u \cdot v := v' \mid b) \quad \widehat{=} \quad [\lambda u \cdot \lambda u' \cdot b \wedge (u' \setminus v' = u \setminus v)],$$

where u' is formed from u by replacing each $x_i : T_i$ in u with $x'_i : T_i$. Assertions and guards may be written using boolean terms as follows:

$$\begin{array}{rcl} (\mathbf{var}\ u \cdot \{b\}) & \widehat{=} & \{\lambda u \cdot b\} \\ (\mathbf{var}\ u \cdot [b]) & \widehat{=} & [\lambda u \cdot b]. \end{array}$$

[2] This is not the same as introducing local variables.

Writing predicates as $(\mathbf{var}\ u \cdot b)$, which is syntactic sugar for $(\lambda u \cdot b)$, we can work in a style that is similar to Dijkstra's weakest-precondition calculus. For example, while Dijkstra has

$$wp(v := E, post) \quad \widehat{=} \quad post[v := E],$$

we have

$$(\mathbf{var}\ u \cdot v := E)(\mathbf{var}\ u \cdot post) \quad = \quad (\mathbf{var}\ u \cdot post[v := E]).$$

Let *pre* and *post* be boolean terms. It is easy to show that

$$(\mathbf{var}\ u \cdot \{pre\};\ v := v' \mid post')$$

is the least statement satisfying the precondition-postcondition pair $(\mathbf{var}\ u \cdot pre)$, $(\mathbf{var}\ u \cdot post)$. Implementing this specification involves constructing a statement S such that $(\mathbf{var}\ u \cdot \{pre\};\ v := v' \mid post') \leq S$. If *post* refers to variables of both v and v', then the specification $(\mathbf{var}\ u \cdot \{pre\};\ v := v' \mid post)$ can relate the before and after states in the manner of B [1], VDM [16], and Z [26] specifications. A statement of the form $(\mathbf{var}\ u \cdot \{pre\}\ v := v' \mid post)$ is called a *specification statement* [20] (we usually omit the sequential composition operator from a specification statement). A sorting algorithm, for example, may be specified by

$$\mathbf{var}\ s : \mathbf{seq}\ T \quad \cdot \quad s := s' \mid (sorted\ s') \wedge (permutation\ s\ s').$$

Rules for the stepwise refinement of specification statements into standard programming constructs may be found in [2, 21, 23].

In the rest of this paper, we use identifiers a, b for boolean terms, p, q for predicates, P, Q for relations, and S, T for predicate transformers.

3 Product and Fusion

We write $\Gamma_1 \times \Gamma_2$ for the cartesian product of types Γ_1 and Γ_2. Given two predicates q_1 and q_2 their product is denoted $q_1 \times q_2$ and is defined as follows:

Definition 1 *For* $q_1 : \mathcal{P}\,\Gamma_1,\ q_2 : \mathcal{P}\,\Gamma_2,\quad q_1 \times q_2$ *is of type* $\mathcal{P}\,(\Gamma_1 \times \Gamma_2)$ *where for* $\gamma_1 : \Gamma_1,\ \ \gamma_2 : \Gamma_2$,

$$(q_1 \times q_2)\ (\gamma_1, \gamma_2) \quad \widehat{=} \quad (q_1\ \gamma_1) \wedge (q_2\ \gamma_2).$$

Note that predicates of the form $q_1 \times q_2$ only form rectangular subsets of $\mathcal{P}\,(\Gamma_1 \times \Gamma_2)$ so that arbitrary $q : \mathcal{P}\,(\Gamma_1 \times \Gamma_2)$ cannot be represented as $q_1 \times q_2$.

Next we are interested in a product operator for arrows. Consider first the product of functions: the product of $f_1 : \Sigma \to \Gamma_1$ and $f_2 : \Sigma \to \Gamma_2$ is denoted $f_1 \triangle f_2$ and has type $\Sigma \to \Gamma_1 \times \Gamma_2$ ($\times$ binds tighter than arrows). It is defined as follows:

Definition 2 $(f_1 \triangle f_2)\ \sigma \ \ \widehat{=}\ \ (f_1\sigma,\ f_2\sigma)$.

Now, consider two predicate transformers which have a common source but distinct targets:

$$S_1 : \Sigma \mapsto \Gamma_1, \qquad S_2 : \Sigma \mapsto \Gamma_2.$$

The product of S_2 and S_2 would have type $\Sigma \mapsto \Gamma_1 \times \Gamma_2$. Since we wish S_1 to establish q_1 and S_2 to establish q_2, we could propose that

$$(S_1 \triangle S_2)\ (q_1 \times q_2) \quad = \quad S_1 q_1\ \wedge\ S_2 q_2.$$

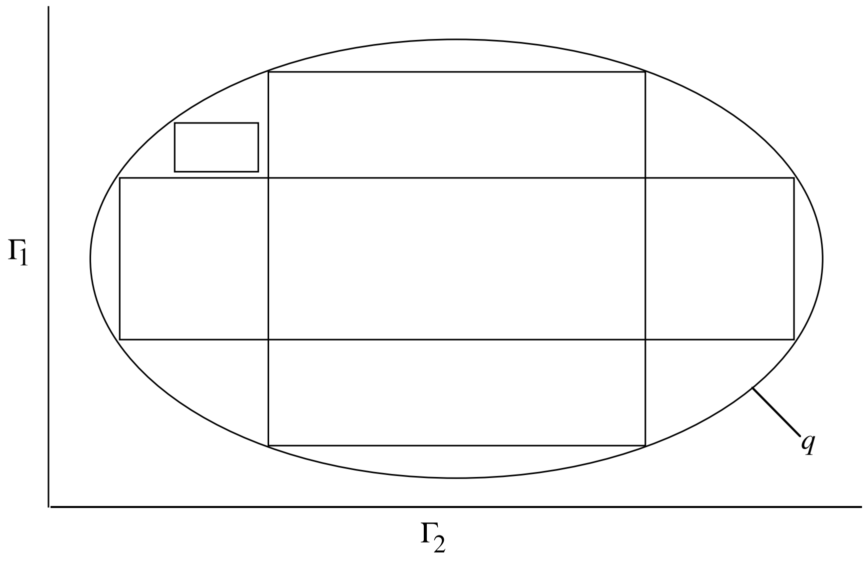

Figure 1: Approximating q with rectangular subsets.

Of course this is not a full definition since, in general, $q : \mathcal{P}(\Gamma_1 \times \Gamma_2)$ cannot be represented by $q_1 \times q_2$. So we approximate q by its rectangular subsets:

$$\begin{aligned}(S_1 \triangle S_2)\ q \quad &= \quad (\exists q_1 : \mathcal{P}\,\Gamma_1;\ q_2 : \mathcal{P}\,\Gamma_2 \mid q_1 \times q_2 \leq q \cdot \\ &\qquad (S_1 \triangle S_2)\ (q_1 \times q_2)).\end{aligned}$$

This concept is illustrated in Fig. 1. The full definition is:

DEFINITION **3** *For* $S_1 : \Sigma \mapsto \Gamma_1,\ \ S_2 : \Sigma \mapsto \Gamma_2,\ \ S_1 \triangle S_2$ *is of type* $\Sigma \mapsto \Gamma_1 \times \Gamma_2$*, where for* $q : \mathcal{P}(\Gamma_1 \times \Gamma_2)$*:*

$$(S_1 \triangle S_2)\ q \quad \widehat{=} \quad (\exists q_1 : \mathcal{P}\,\Gamma_1;\ q_2 : \mathcal{P}\,\Gamma_2 \mid q_1 \times q_2 \leq q \cdot S_1 q_1 \ \wedge\ S_2 q_2).$$

Similar definitions may be found [18] and [25].

Definition 3 may be generalised to define what we call a *fusion* operator that combines commands with common sources and common targets as follows:

DEFINITION **4** *For* $S_1, S_2 : \Sigma \mapsto \Gamma,\ \ S_1 \odot S_2$ *is of type* $\Sigma \mapsto \Gamma$*, where for* $q : \mathcal{P}\,\Gamma$*:*

$$(S_1 \odot S_2)\ q \quad \widehat{=} \quad (\exists q_1, q_2 : \mathcal{P}\,\Gamma \mid q_1 \wedge q_2 \leq q \cdot S_1 q_1 \ \wedge\ S_2 q_2).$$

It is easy to show that the fusion operator is commutative and associative and that it preserves the monotonicity, conjunctivity and disjunctivity of its operands. Fusion also preserves the refinement of its operands. In fact, fusion enjoys a slightly stronger property:

THEOREM **5** *For monotonic predicate transformers,*

$$\{halt\ S_2\}; S_1 \ \leq\ S_1' \quad \Rightarrow \quad (S_1 \odot S_2) \leq (S_1' \odot S_2).$$

This allows the termination condition of one operand to be used as an assumption when refining the other operand

The fusion of two commands only terminates if both operands terminate:

$$halt(S_1 \odot S_2) \quad = \quad halt\ S_1 \ \wedge\ halt\ S_2.$$

Let π_1 and π_2 be the projections from $\Gamma_1 \times \Gamma_2$ to Γ_1 and Γ_2 respectively:

$$\begin{array}{ll} \pi_1 : \Gamma_1 \times \Gamma_2 \to \Gamma_1, & \pi_2 : \Gamma_1 \times \Gamma_2 \to \Gamma_2, \\ \pi_1(\gamma_1, \gamma_2) = \gamma_1, & \pi_2(\gamma_1, \gamma_2) = \gamma_2. \end{array}$$

The inverses of π_1 and π_2 are relations:

$$\pi_1^{-1} : \Gamma_1 \leftrightarrow \Gamma_1 \times \Gamma_2 \qquad \pi_2^{-1} : \Gamma_2 \leftrightarrow \Gamma_1 \times \Gamma_2.$$

Now the demonic update

$$[\pi_1^{-1}] \ : \ \Gamma_1 \mapsto \Gamma_1 \times \Gamma_2,$$

starts in a state $\gamma_1 : \Gamma_1$ and ends in a state $(\gamma_1', \gamma_2') : \Gamma_1 \times \Gamma_2$ such that $\gamma_1 = \gamma_1'$ and γ_2' is chosen nondeterministically.

Given $S_1 : \Sigma \mapsto \Gamma_1$, the predicate transformer

$$S_1; [\pi_1^{-1}] \ : \ \Sigma \mapsto \Gamma_1 \times \Gamma_2$$

is a 'lifted' version of S_1 that behaves as S_1 on the first component of the final state, and nondeterministically writes any value to the second component. Similarly for $S_2; [\pi_2^{-1}]$. Now we have that the product operator ($\triangle$) is a special case of fusion:

THEOREM **6** *For* $S_1 : \Sigma \mapsto \Gamma_1, \ \ S_2 : \Sigma \mapsto \Gamma_2,$

$$S_1 \triangle S_2 \quad = \quad S_1; [\pi_1^{-1}] \ \odot \ S_2; [\pi_2^{-1}].$$

The proof of this theorem is given in the appendix. Because of the theorem, the product operator inherits several important properties directly from the fusion operator including preservation of monotonicity and conjunctivity. Preservation of disjunctivity does not follow since $[\pi_2^{-1}]$ is not disjunctive, but disjunctivity can be proved separately [25]. Most importantly for piecewise refinement, preservation of refinement does follow:

THEOREM **7** *For monotonic predicate transformers,*

$$\{halt\ S_2\}; S_1 \ \leq \ S_1' \quad \Rightarrow \quad (S_1 \triangle S_2) \leq (S_1' \triangle S_2).$$

Although the fusion operator is commutative and associative, the product operator is neither. This is because of the difference in the product structures of the states, i.e., $\Sigma_1 \times \Sigma_2 \neq \Sigma_2 \times \Sigma_1$ and $\Sigma_1 \times (\Sigma_2 \times \Sigma_3) \neq (\Sigma_1 \times \Sigma_2) \times \Sigma_3$. We need to rearrange the states to get equality, e.g.,

$$S_1 \triangle S_2 \quad = \quad (S_2 \triangle S_1); \langle \lambda x, y \cdot (y, x) \rangle.$$

Let ϖ_1 and ϖ_2 be the projections from $\Sigma_1 \times \Sigma_2$ to Σ_1 and Σ_2 respectively. The projections give the following deterministic updates:

$$\langle \varpi_1 \rangle : \Sigma_1 \times \Sigma_2 \mapsto \Sigma_1 \quad \text{ and } \quad \langle \varpi_2 \rangle : \Sigma_1 \times \Sigma_2 \mapsto \Sigma_2.$$

The statement $\langle \varpi_1 \rangle$ starts in a state (σ_1, σ_2) and ends in the state σ_1 simply discarding the second component. The derived product operator combines commands with distinct sources and distinct targets, and is defined as follows:

DEFINITION **8** *For* $S_1 : \Sigma_1 \mapsto \Gamma_1, \ \ S_2 : \Sigma_2 \mapsto \Gamma_2, \ \ S_1 \otimes S_2$ *is of type* $\Sigma_1 \times \Sigma_2 \mapsto \Gamma_1 \times \Gamma_2$*, where*

$$S_1 \otimes S_2 \quad \widehat{=} \quad \langle \varpi_1 \rangle; S_1 \ \triangle \ \langle \varpi_2 \rangle; S_2.$$

We extend the program variable syntax for fusion, product and derived product as follows:

$$\begin{array}{rcl}(\mathbf{var}\ u \mapsto v \cdot S_1 \odot S_2) & \widehat{=} & (\mathbf{var}\ u \mapsto v \cdot S_1)\ \odot\ (\mathbf{var}\ u \mapsto v \cdot S_2) \\ (\mathbf{var}\ u \mapsto (v_1, v_2) \cdot S_1 \mathbin{\triangle} S_2) & \widehat{=} & (\mathbf{var}\ u \mapsto v_1 \cdot S_1)\ \triangle\ (\mathbf{var}\ u \mapsto v_2 \cdot S_2) \\ (\mathbf{var}\ (u_1, u_2) \mapsto (v_1, v_2) \cdot S_1 \otimes S_2) & \widehat{=} & (\mathbf{var}\ u_1 \mapsto v_1 \cdot S_1)\ \otimes\ (\mathbf{var}\ u_2 \mapsto v_2 \cdot S_2).\end{array}$$

Thus with the fusion operator, commands read from and write to the same variables, with the product operator they read from the same variables but write to separate variables and with the derived product operator they also read from separate variables.

4 Conjoining Specifications

Fusion acts as a form of 'co-refinement' operator:

THEOREM 9 *For predicate transformers S_1 and S_2, $S_1 \odot S_2$ solves:*

$$\{halt\ S_2\}; S_1 \ \leq\ X \tag{1}$$

$$\{halt\ S_1\}; S_2 \ \leq\ X, \tag{2}$$

and furthermore, for any conjunctive Y that solves (1) and (2),

$$S_1 \odot S_2 \ \leq\ Y. \tag{3}$$

(See appendix for proof of this theorem.) Thus for conjunctive S_1 and S_2, $S_1 \odot S_2$ is the least conjunctive predicate transformer satisfying (1) and (2). Note that $S_1 \odot S_2$ is not a true refinement of S_1 since it cannot replace S_1, but rather it can replace $\{halt\ S_2\}; S_1$.

Theorem 9 means that the fusion operator can be used to conjoin two specifications to form a specification that refines both specifications within their combined termination condition. This allows us to describe program requirements separately and them combine them. For example, consider the following two requirements:

$$\begin{array}{rcl}R1 & \widehat{=} & (\mathbf{var}\ x : real \cdot\ \ \{x \geq 0\}\ x := x' \mid x'^2 = x\ ) \\ R2 & \widehat{=} & (\mathbf{var}\ x : real \cdot\ \ x := x' \mid x' \geq 0\ ).\end{array}$$

The fusion $R1 \odot R2$ specifies a program that calculates the positive square root of x.

The following theorem provides a simple way of calculating the fusion of two program specifications:

THEOREM 10 *For relations $P, Q : \Sigma \leftrightarrow \Gamma$, and predicates $p, q : \mathcal{P}\,\Sigma$,*

$$\{p\}; [P]\ \odot\ \{q\}; [Q] \ =\ \{p \wedge q\}; [P \wedge Q].$$

Proof of this theorem is given in the appendix.

This clearly illustrates that the effect of the fusion operator is to reduce the (demonic) nondeterminism of the terminating behaviour of both commands. In the case that the intersection of the nondeterminism in both commands is empty, then the fusion behaves magically, e.g.,

$$(\mathbf{var}\ x \cdot\ \ x := x' \mid x' > 0\ \ \odot\ \ x := x' \mid x' < 0\ ) \ =\ magic.$$

In terms of the program variable syntax, Theorem 10 is represented as:

$$\begin{array}{l}\mathbf{var}\ u \cdot\ \ (\{a_1\}\ v := v' \mid b_1)\ \odot\ (\{a_2\}\ v := v' \mid b_2) \\ \quad = \ \mathbf{var}\ u \cdot\ \ \{a_1 \wedge a_2\}\ v := v' \mid b_1 \wedge b_2.\end{array}$$

Alternative Definitions

It is easy to show that the demonic choice of specifications results in disjunction of postconditions, i.e.,

$$\{p_1\};[Q_1] \;\wedge\; \{p_2\};[Q_2] \quad = \quad \{p_1 \wedge p_2\};[Q_1 \vee Q_2].$$

Fusion then is almost a dual of demonic choice. Since all conjunctive commands have a normal form $\{p\};[Q]$, we could define an alternative fusion operator for conjunctive commands which really is a dual of the demonic choice as follows:

$$\{p_1\};[Q_1] \;\oplus\; \{p_2\};[Q_2] \quad \widehat{=} \quad \{p_1 \vee p_2\};[Q_1 \wedge Q_2].$$

This operator results in a true co-refinement, i.e., $S_1 \leq S_1 \oplus S_2$ and not just $\{halt\ S_2\};S_1 \leq S_1 \oplus S_2$. However it is not the least conjunctive co-refinement so that use of this operator could result in specifications being strengthened unnecessarily when being combined. More seriously, this operator is not guaranteed to preserve refinement of its operands. As a counterexample, consider

$$\begin{array}{rcl} S_1 &=& \{x > 0\}\ x := x - 1 \\ S_1' &=& \textbf{if}\ x > 0\ \textbf{then}\ x := x - 1\ \textbf{else}\ x := x + 1\ \textbf{fi} \\ S_2 &=& x := x' \mid x' \neq x. \end{array}$$

It is easy to show that $S_1 \oplus S_2 \not\leq S_1' \oplus S_2$ even though $S_1 \leq S_1'$. These weaknesses rule $\oplus$ out as a useful co-refinement operator.

Leino and Manohar [17] have shown that the following operator does yield the the least conjunctive co-refinement of two specification statements:

$$\{p_1\};[Q_1] \diamond \{p_2\};[Q_2] \quad \widehat{=} \quad \{p_1 \vee p_2\};\ [\ p_1 \Rightarrow Q_1 \;\wedge\; p_2 \Rightarrow Q_2\ ].$$

Here $(p \Rightarrow Q)\sigma\sigma' \;\widehat{=}\; p\sigma \Rightarrow Q\sigma\sigma'$. Using this to define specification conjunction does give an operator that preserves refinement. However, unlike our fusion operator, this operator does not allow the termination condition of one command to be assumed when refining the other.

Morgan [22] has developed an operator $\Box$ on predicate transformers such that $\Box S$ is the least-refined predicate transformer that is both universally conjunctive and refines S. It is easy to show for universally conjunctive S_1 and S_2 (i.e., S_1 and S_2 always terminate), that

$$\Box(S_1 \vee S_2) \quad = \quad S_1 \odot S_2.$$

We say that two specifications $\{p\};[P]$ and $\{q\};[Q]$ are *contradictory* in some initial state σ if σ has a successful outcome in each specification, but does not have a successful outcome in their fusion, i.e., σ holds in p, q, $dom\ P$, and $dom\ Q$, but not in $dom\ (P \wedge Q)$. For those contradictory initial states, the fusion of both specifications behaves miraculously. Ward [29] has defined a conjunction combinator for specification statements where the combination behaves as *abort* when both specifications are contradictory. Thus, in Ward's case, a contradictory combination may be refined by any statement whereas, in our case, it is unimplementable. Furthermore, Ward's combinator is not refinement preserving.

5 Simultaneous Execution

We have already seen the product operator for functions. The product operator for relations is defined as follows:

Definition 11 *For relations $P_1 : \Sigma \leftrightarrow \Gamma_1$, $P_2 : \Sigma \leftrightarrow \Gamma_2$, $P_1 \bigtriangleup P_2$ is a relation of type $\Sigma \leftrightarrow \Gamma_1 \times \Gamma_2$, where*

$$P_1 \bigtriangleup P_2 \quad \widehat{=} \quad P_1 ; \pi_1^{-1} \quad \wedge \quad P_2 ; \pi_2^{-1}.$$

From Theorem 10 (fusion of specification statements) we can then show that the product operator combines with specification statements and deterministic updates in the following manner:

Theorem 12 *For $p_1, p_2 : \mathcal{P}\,\Sigma$, and $P_1 : \Sigma \leftrightarrow \Gamma_1$, $P_2 : \Sigma \leftrightarrow \Gamma_2$,*

$$\{p_1\};[P_1] \;\bigtriangleup\; \{p_2\};[P_2] \quad = \quad \{p_1 \wedge p_2\};[P_1 \bigtriangleup P_2].$$

Theorem 13 *For $f_1 : \Sigma \to \Gamma_1$, $f_2 : \Sigma \to \Gamma_2$,*

$$\langle f_1 \rangle \bigtriangleup \langle f_2 \rangle \quad = \quad \langle f_1 \bigtriangleup f_2 \rangle.$$

Theorem 13 is represented in our program variable syntax as follows:

$$\begin{array}{rl} & (\mathbf{var}\ (v_1, v_2) \cdot \quad v_1 := E_1 \quad \bigtriangleup \quad v_2 := E_2\ ) \\ = & (\mathbf{var}\ (v_1, v_2) \cdot \quad v_1, v_2 := E_1, E_2\ ), \end{array}$$

where E_1 can depend on both v_1 and v_2, and similarly for E_2. Thus, the product operator models simultaneous execution of assignments to separate variables.

With the product operator, both commands have access to the initial values of each others variables. With the derived product operator, we have

$$\begin{array}{rl} & (\mathbf{var}\ (v_1, v_2) \cdot \quad v_1 := E_1 \quad \otimes \quad v_2 := E_2\ ) \\ = & (\mathbf{var}\ (v_1, v_2) \cdot \quad v_1, v_2 := E_1, E_2\ ), \end{array}$$

where E_1 (resp. E_2) is independent of v_2 (resp. v_1).

Theorem 12 is represented as:

$$\begin{array}{rl} & (\mathbf{var}\ (v_1, v_2) \cdot \quad \{a_1\}\ v_1 := v_1' \mid b_1 \quad \bigtriangleup \quad \{a_2\}\ v_2 := v_2' \mid b_2\ ) \\ = & (\mathbf{var}\ (v_1, v_2) \cdot \quad \{a_1 \wedge a_2\}\ v_1, v_2 := v_1', v_2' \mid b_1 \wedge b_2\ ). \end{array}$$

In this case, b_1 is independent of v_2' so that b_1 does not constrain the value assigned to v_2, and similarly for b_2. In the case of the derived product, a_1 and b_1 are also independent of v_2 and vice versa.

When composing commands using the product operator, it can be useful to have some algebraic laws available. For example, we have the following distribution through if-statements:

$$\begin{array}{rl} & (\mathbf{var}\ u \cdot \quad S \;\bigtriangleup\; (\mathbf{if}\ g\ \mathbf{then}\ T_1\ \mathbf{else}\ T_2\ \mathbf{fi})\ ) \\ = & (\mathbf{var}\ u \cdot \quad \mathbf{if}\ g\ \mathbf{then}\ (S \;\bigtriangleup\; T_1)\ \mathbf{else}\ (S \;\bigtriangleup\; T_2)\ \mathbf{fi}\ ). \end{array}$$

In certain cases, a product may be replaced by a sequential composition: if a_2, b_2 are independent of v_1, then

$$\begin{array}{rl} & (\mathbf{var}\ (v_1, v_2) \cdot \quad \{a_1\}\ v_1 := v_1' \mid b_1 \quad \bigtriangleup \quad \{a_2\}\ v_2 := v_2' \mid b_2\ ) \\ = & (\mathbf{var}\ (v_1, v_2) \cdot \quad (\{a_1 \wedge a_2\}\ v_1 := v_1' \mid b_1)\ ;\ (\{a_2\}\ v_2 := v_2' \mid b_2)\ ). \end{array}$$

It is necessary to have a_2 in the first assertion because without it, in a state where the command $\{a_1\}\ v_1 := v_1' \mid b_1$ behaves magically and the command $\{a_2\}\ v_2 := v_2' \mid b_2$ aborts, the product would abort whereas the sequential composition would behave magically (since $magic; S = S$). See [3] for a fuller list of the properties of the operators.

Alternative Definitions

Abrial has defined a parallel operator for the B AMN notation [1] which is equivalent to our derived product. He only works with conjunctive commands and so can define the operator in terms of the relational product on normal forms (similar to our Theorem 12):

$$\{\ p_1 \times p_2\ \};\ [\ Q_1 \otimes Q_2\ ].$$

(Here, $Q_1 \otimes Q_2 \ \ \widehat{=}\ \ \varpi_1; Q_1 \ \triangle\ \varpi_2; Q_2$.) It is arguable that this is an easier way of defining simultaneous execution than our more general product operator. However, with our definition we can also form the product of disjunctive commands which is useful for data refinement as we shall see in the next section.

An alternative way of defining simultaneous execution would be as a special case of the least conjunctive co-refinement operator, in the same way that the product is a special case of fusion:

$$S_1 \diamondsuit' S_2 \qquad \widehat{=} \qquad S_1; [\pi_1^{-1}] \ \ \diamondsuit\ \ S_2; [\pi_2^{-1}].$$

However this would give strange results when combining a terminating command with a nonterminating command, e.g.,

$$(\mathbf{var}\ x, y \cdot \ \ abort \diamondsuit'\ y := y' | b\ ) \quad = \quad (\mathbf{var}\ x, y \cdot \ \ x, y := x', y' | b\ ).$$

The command on the right hand side assigns an arbitrary value to x but it always terminates, thus a nonterminating command could be "fixed" by combining it in parallel with a terminating one!

To properly model a system that aborts on one component of the state and terminates on the other would require a pair of predicate transformers. This would not allow us to freely interchange single specifications with parallel commands as we require when working with the refinement calculus.

6 Data Refinement

The well-known technique of data refinement involves replacing abstract program variables with concrete program variables using an abstraction relation [12]. In the refinement calculus, the abstraction relation is modelled by an abstraction command, and we say that $S : \Sigma \mapsto \Sigma$ is data refined by $S' : \Sigma' \mapsto \Sigma'$ under abstraction command $\alpha : \Sigma' \mapsto \Sigma$ if [1, 10, 24, 27]:

$$\alpha; S \ \leq\ S'; \alpha.$$

For a more comprehensive treatment of data refinement of predicate transformers see [1, 10, 24, 27]. Here we will simply look at how data refinement distributes through the fusion and product operators.

For predicate transformer T, the *right adjoint* of T, denoted T^r, satisfies

$$T; T^r \ \leq\ skip \qquad\qquad skip \ \leq\ T^r; T.$$

T has a right adjoint if and only if T is universally disjunctive [27]. In this case, T^r is universally conjunctive. For relation P, it can be shown that

$$\{P\}^r = [P^{-1}].$$

So-called *forward* data refinement corresponds to the case where the abstraction command α is universally disjunctive, in which case, α will have right adjoint α^r.

Furthermore α and α^r will have normal forms $\{P\}$ and $[P^{-1}]$ respectively. It is easily shown that $\alpha; S; \alpha^r$ is the least forward data-refinement of S, i.e.,

$$\begin{array}{c} \alpha; S \ \leq \ (\alpha; S; \alpha^r); \alpha, \ \text{and} \\ \alpha; S \ \leq \ X; \alpha \quad \Rightarrow \quad \alpha; S; \alpha^r \ \leq \ X. \end{array}$$

Thus, given S and α, we calculate a data refinement of S by refining $\alpha; S; \alpha^r$. Laws for distributing forward data refinement through statement structures may be found in [10, 27]. For example,

$$\alpha; S; T; \alpha^r \quad \leq \quad (\alpha; S; \alpha^r); (\alpha; T; \alpha^r).$$

Laws such as these allow us to calculate data refinements of compound statements by calculating data refinements of the component statements.

Typically, the abstraction command is represented by a boolean term I relating the abstract and concrete variables. In terms of our program variable syntax, the least data refinement of $(\mathbf{var}\ a \cdot S)$ is given by:

$$\{\mathbf{var}\ c \mapsto a \cdot I\}; (\mathbf{var}\ a \cdot S); [\mathbf{var}\ a \mapsto c \cdot I].$$

Note that $\{\mathbf{var}\ c \mapsto a \cdot I\}^r = [\mathbf{var}\ a \mapsto c \cdot I]$.

We are interested in how α and α^r distribute from the left and from the right in $\alpha; (S_1 \odot S_2); \alpha^r$. We do have sub-distributivity from the left:

Lemma 14 *For disjunctive α,*

$$\alpha; (S_1 \odot S_2) \quad \leq \quad (\alpha; S_1) \odot (\alpha; S_2).$$

There is a similar sub-distributivity law for the product operator.

In considering distribution of α^r from the right, we shall restrict our attention to the case where S_1 and S_2 are conjunctive. Since S_1 and S_2 are conjunctive and α^r is universally conjunctive, $(S_1 \odot S_2); \alpha^r$ has the normal form

$$\begin{array}{ll} & (\ \{p_1\}; [P_1] \quad \odot \quad \{p_2\}; [P_2]\ )\ ;\ [Q] \\ = & \{p_1 \wedge p_2\}\ ;\ [\ (P_1 \wedge P_2)\ ]; [Q] \\ = & \{p_1 \wedge p_2\}\ ;\ [\ (P_1 \wedge P_2); Q\ ]. \end{array}$$

We cannot proceed further since, at the level of relations, we do not have the required sub-distributivity, i.e.,

$$(P_1 \wedge P_2); Q \quad \not\geq \quad (P_1; Q) \wedge (P_2; Q).$$

This is because the conjunction on the right hand side has no influence on intermediate states. If Q is of the form $Q_1 \otimes Q_2$ then we do get the following distributivity for relational product:

Lemma 15 *For $P_1 : \Sigma \leftrightarrow \Gamma_1, P_2 : \Sigma \leftrightarrow \Gamma_2,\ Q_1 : \Gamma_1 \leftrightarrow \Phi_1, Q_2 : \Gamma_2 \leftrightarrow \Phi_2$,*

$$(P_1 \bigtriangleup P_2); (Q_1 \otimes Q_2) \quad = \quad (P_1; Q_1) \bigtriangleup (P_2; Q_2).$$

Proof of this lemma is given in the appendix.

This leads to the following theorem about distribution of the least data refinement through the product operators:

Theorem 16 *Assume α has the form $\{P_1 \otimes P_2\}$. Let $\alpha_1 = \{P_1\}$, $\alpha_2 = \{P_2\}$. Then for conjunctive S_1, S_2,*

$$\begin{array}{rcl} \alpha; (S_1 \bigtriangleup S_2); \alpha^r & \leq & (\alpha; S_1; \alpha_1^r) \ \bigtriangleup \ (\alpha; S_2; \alpha_2^r) \\ \alpha; (T_1 \otimes T_2); \alpha^r & \leq & (\alpha_1; T_1; \alpha_1^r) \ \otimes \ (\alpha_2; T_2; \alpha_2^r). \end{array}$$

Proof of this theorem is given in the appendix.

This theorem shows how the least data refinement can be pushed through both product operators. In terms of our program variable syntax, the abstraction commands here may be represented by:

$$\begin{aligned}\alpha &= \{\mathbf{var}\ (c_1, c_2) \mapsto (a_1, a_2) \cdot I_1 \wedge I_2\} \\ \alpha_1 &= \{\mathbf{var}\ c_1 \mapsto a_1 \cdot I_1\} \\ \alpha_2 &= \{\mathbf{var}\ c_2 \mapsto a_2 \cdot I_2\},\end{aligned}$$

where I_1 is independent of a_2, c_2 and I_2 is independent of a_1, c_1.

In Theorem 16, we have relied on the fact that the product operator can be used to compose both disjunctive and conjunctive commands. A definition of product that only worked with conjunctive commands would not have allowed us to work with data refinement in this way.

Note that it is still possible to work with data refinement using only a product operator for conjunctive commands as Abrial shows [1]. He derives an equivalent first order formulation of data refinement (i.e., without quantification over predicates) which allow him to show that if S_1 is data refined by S_2' under abstraction relation R_1 (written $S_1 \leq_{R_1} S_1'$) and $S_2 \leq_R S_2'$, then

$$S_1 \otimes S_2 \quad \leq_{R_1 \otimes R_2} \quad S_1' \otimes S_2'.$$

However this approach does not allow us to work with the least data refinement of a product. In [28], von Wright shows that the least data refinement of a conjunctive command can sometimes be disjunctive. He gives an example command whose only conjunctive data refinement is infeasible (i.e., *magic*) but which does have a disjunctive data refinement that in some contexts can be refined by a feasible conjunctive command. Thus the least data refinement approach provides a richer environment in which to reason and this is supported by our product operator.

Backward data refinement corresponds to the case where α is universally conjunctive. However, universally conjunctive α does not sub-distribute from the left, nor does universally disjunctive α^r sub-distribute from the right, so we cannot distribute backwards data refinement through the fusion and product operators.

7 Other Applications of Product

Embedding

Suppose we want to embed a statement $(\mathbf{var}\ v \cdot S)$ in a context in which a larger list of variables u is being operated on. We require that the variables $u \setminus v$ should remain unchanged. This is achieved using the product operator as follows:

$$\begin{aligned}\mathbf{embed}\ u \cdot (\mathbf{var}\ v \cdot S) \quad \widehat{=} \quad & (\mathbf{var}\ u \mapsto (v, \overline{v}) \cdot u := u); \\ & ((\mathbf{var}\ v \cdot S)\ \triangle\ (\mathbf{var}\ \overline{v} \cdot skip)); \\ & (\mathbf{var}\ (v, \overline{v}) \mapsto u \cdot u := u) \\ & \qquad\qquad \mathit{where}\ \ \overline{v} = u \setminus v.\end{aligned}$$

The statements $(\mathbf{var}\ u \mapsto (v, \overline{v}) \cdot u := u)$ and $(\mathbf{var}\ (v, \overline{v}) \mapsto u \cdot u := u)$ simply rearrange the ordering of program variables. It can be shown that this embedding preserves refinement, i.e., if $S \leq S'$, then

$$(\mathbf{embed}\ u \cdot S) \quad \leq \quad (\mathbf{embed}\ u \cdot S').$$

Furthermore, although the product operator is only commutative and associative up to isomorphism, it is fully commutative and associative within an embedding:

$$\begin{aligned}(\textbf{embed } u \cdot S_1 \bigtriangleup S_2) &= (\textbf{embed } u \cdot S_2 \bigtriangleup S_1)\\ (\textbf{embed } u \cdot (S_1 \bigtriangleup S_2) \bigtriangleup S_3) &= (\textbf{embed } u \cdot S_1 \bigtriangleup (S_2 \bigtriangleup S_3)).\end{aligned}$$

Embedding is important for procedure calling. We model a procedure by simply associating a statement with a procedure name, e.g.,

$$Proc \quad \widehat{=} \quad (\textbf{var } v \cdot S).$$

Calling a procedure is then modelled by embedding the statement associated with the procedure name in the appropriate place:

$$T; (\textbf{var } u \cdot Proc); U \quad \widehat{=} \quad T; (\textbf{embed } u \cdot S); U.$$

Embedding allows us to use the same procedure in different program variable contexts.

Extension and Modification

Given a statement $(\textbf{var } v \cdot S_1)$, we can use the product operator to extend it's functionality with a statement S_2:

$$\begin{aligned}&(\textbf{var } v \cdot S_1) \ \textbf{extn} \ (\textbf{var } (v, w) \mapsto w \cdot S_2)\\ &\quad \widehat{=} \quad ((\textbf{var } (v, w) \mapsto v \cdot v := v); (\textbf{var } v \cdot S_1)) \ \bigtriangleup \ (\textbf{var } (v, w) \mapsto w \cdot S_2).\end{aligned}$$

Here, S_2 may add extra program variables which it writes to, as well as being able to read the variables of S. Extension of S_1 by S_2 is sometimes known as superposition. As an example, we have:

$$\begin{aligned}&(\textbf{var } x \cdot x := e) \quad \textbf{extn} \quad (\textbf{var } (x, y) \mapsto y \cdot y := x + y)\\ &= \qquad (\textbf{var } (x, y) \cdot x, y := e, x + y).\end{aligned}$$

The product operator can be used to modify an existing statement. Suppose we wish to modify $(\textbf{var } u \cdot S_1)$ so that the value it writes to variables v is determined by $(\textbf{var } u \mapsto v \cdot S_2)$ instead. This is achieved as follows:

$$\begin{aligned}&(\textbf{var } u \cdot S_1) \ \textbf{mod} \ (\textbf{var } u \mapsto v \cdot S_2)\\ &\quad \widehat{=} \quad (((\textbf{var } u \cdot S_1); (\textbf{var } u \mapsto \overline{v} \cdot \overline{v} := \overline{v})) \ \bigtriangleup \ (\textbf{var } u \mapsto v \cdot S_2));\\ &\qquad\quad (\textbf{var } (\overline{v}, v) \mapsto u \cdot u := u)\\ &\qquad\qquad\qquad where \ \ \overline{v} = u \setminus v.\end{aligned}$$

For example,

$$\begin{aligned}&\quad (\textbf{var } (x, y) \cdot x, y := d, e) \quad \textbf{mod} \quad (\textbf{var } (x, y) \mapsto x \cdot x := f)\\ &= \quad (\textbf{var } (x, y) \cdot x, y := f, e).\end{aligned}$$

Both extension and modification preserve refinement of their operands. In [3], we describe a summation operator, a categorical dual to the product operator which is shown to model late-binding of procedures. This operator in combination with extension and modification can be used to model inheritance in object oriented programming [3, 19].

$$
\mathcal{M} \;\widehat{=}\; \left(\begin{array}{ll}
\textbf{var } m : \textbf{Int} & \\
\textbf{initially } m := 0 & \\
\textbf{action } a : & m = 0 \longrightarrow m := 1 \\
\textbf{action } c : & m = 1 \longrightarrow m := 0
\end{array}\right)
$$

$$
\mathcal{N} \;\widehat{=}\; \left(\begin{array}{ll}
\textbf{var } n : \textbf{Int} & \\
\textbf{initially } n := 0 & \\
\textbf{action } b : & n = 0 \longrightarrow n := 1 \\
\textbf{action } c : & n = 1 \longrightarrow n := 0
\end{array}\right)
$$

Figure 2: Example action systems

Composing Action Systems

The *action system* formalism of Back and Kurki-Suonio [4] uses predicate transformers to model parallel programs. An action system consists of a state space determined by some program variables, an initialisation predicate transformer, and a set of action predicate transformers. Execution of an action system proceeds by firstly executing the initialisation, then, repeatedly, executing an enabled action (an action A is enabled when $gd\ A$ holds). Typically, an action is written in the form $(\textbf{var}\ u \cdot g \longrightarrow S)$, where g is a boolean term, and

$$(\textbf{var}\ u \cdot g \longrightarrow S) \quad \widehat{=} \quad (\textbf{var}\ u \cdot [g]; S).$$

In the case that S is always enabled, i.e., $gd\ S = \top$, then

$$gd\ (\textbf{var}\ u \cdot g \longrightarrow S) \quad = \quad (\textbf{var}\ u \cdot g).$$

Figure 2 gives two examples of such action systems.

Two action systems may be composed in parallel by forming the union of their program variables, composing their initialisations such that they are executed simultaneously, and forming the union of their actions. Embedding is used to embed the actions in the composite state space. Conventionally, the initialisations are demonic updates $[I_1]$, $[I_2]$, and their composition is simply $[I_1 \wedge I_2]$. The product operator provides a way of composing more general initialisations achieving the same effect.

In [7], a correspondence between action systems and Hoare's CSP [13] is described. Based on CSP parallel composition, a version of parallel composition of action systems is introduced in which commonly labelled actions from the respective action systems are composed such that they are executed simultaneously. This represents synchronised updating of states, and can be modelled using the product operator. To combine the action systems $\mathcal{M}$ and $\mathcal{N}$ of Figure 2, the action labelled c from both $\mathcal{M}$ and $\mathcal{N}$ must be synchronised:

$$(m = 1 \longrightarrow m := 0) \quad \triangle \quad (n = 1 \longrightarrow n := 0) \quad = \quad m = 1 \wedge n = 1 \longrightarrow m, n := 0, 0.$$

The synchronised parallel composition of action systems $\mathcal{M}$ and $\mathcal{N}$ is then:

$$
\mathcal{M} \parallel \mathcal{N} \;=\; \left(\begin{array}{ll}
\textbf{var } m, n : \textbf{Int} & \\
\textbf{initially } m, n := 0, 0 & \\
\textbf{action } a : & m = 0 \longrightarrow m := 1 \\
\textbf{action } b : & n = 0 \longrightarrow n := 1 \\
\textbf{action } c : & m = 1 \wedge n = 1 \longrightarrow m, n := 0, 0
\end{array}\right)
$$

Superposition refinement of action systems is described in [5], where superposition on individual actions is described in terms of sequential composition. Our extension operator could be used instead.

8 Conclusions

We have investigated the fusion and product operators for predicate transformers showing that they satisfy a variety of algebraic laws and have several important applications. These algebraic laws describe properties such as how the operators preserve refinement and distribute through existing constructors of the refinement calculus. The operands of the fusion operator may write to the same final state while those of the product operators write to separate final states; this means that the fusion operator may introduce miraculousness while the product operator never does, i.e., $\triangle$ preserves $\perp$-homomorphism, while $\odot$ does not.

Our fusion operator is useful as an operator for conjoining specifications that, most importantly, is refinement preserving. It compares favourably with some alternative definitions since it may be applied to disjunctive as well as conjunctive commands and it allows the termination condition of one command to be assumed when refining the other. Furthermore, we found to easier to prove certain properties of the fusion operator (e.g., Theorem 10) than of the product operator. Thus the fusion operator served as a useful vehicle for developing properties of the product operators.

We showed that the product operator models simultaneous execution of statements and that, combined with projection updates, it can be used to embed small programs (e.g., procedures) into larger contexts. Simultaneous execution and embedding are also important for composing action systems. Of course, the idea of using categorical products as models of simultaneous execution in programming languages is not new (see, for example, [15]).

Some of the theoretical results for the product operator may be found in [25] and [18], in particular, the preservation of junctivity. We have generalised these definitions to include the fusion operator. Also our approach is different: we develop a product operator for predicates and use properties of this when reasoning about predicate transformers, whereas Martin [18] uses a categorical construction that promotes products and co-products in the category of functions to the category of relations and then to the category of predicate transformers, and Naumann [25] takes a more direct approach working within the category of predicate transformers. The lifting of the operators to the program variable level, and the theorems about the distribution of the fusion and product operators with specification statements (e.g., Theorem 10) are ours. We believe that the extension and modification operators are new in the theory of predicate transformers as is the fusion operator.

Our derived product agrees with Abrial's parallel operator [1] on conjunctive commands. However, unlike Abrial's, our operator can also be applied to disjunctive commands and in Section 6 we showed that this allowed us easily to work with the least data refinement of products which provides a rich environment for reasoning about data refinement.

In [3], we describe a summation operator, a dual to the product operator. This operator is shown to model late-binding of procedures. It is also shown to be a special case of the existing choice operators of the refinement calculus; to that extent it is not an extension of the existing calculus, unlike the fusion operator.

Hoogendijk and Backhouse [14] have investigated products of relations and shown them to satisfy similar distributivity properties as our operator on predicate transformers. However, with a relation one has to choose between whether its domain represents termination or guardedness, and whether nondeterminism should be demonic or angelic, whereas all of these possibilities may be modelled with predicate transformers, i.e., relation P can be embedded into the appropriate predicate transformer lattice in at least four different ways:

$$[P], \qquad \{dom\ P\};[P], \qquad \{P\}, \qquad [dom\ P];\{P\}.$$

The higher order logic we use is equivalent in expressive power to set theory. For example, predicates could be modelled as sets and predicate transformers as functions from sets to sets. Indeed this is the approach taken in [1, 25]. We prefer to work with higher order logic since it ties in with an on-going effort [8] that we have been involved in on providing tool support for program refinement using the HOL theorem-proving system [11].

Traditionally, reasoning about weakest precondition formulae tends to be carried out at the predicate and predicate transformer levels without ever introducing explicit state to work at the boolean level. However, we could only carry out certain proofs at the boolean level, e.g., Theorem 10, so having all three levels gave us more freedom. Although we made minimal assumptions about the structure of the state, we are able to deal with program variables in the conventional fashion by introducing a simple layer of syntactic sugaring.

A Proofs

LEMMA **17** *For* $p_1 : \mathcal{P}\,\Gamma_1,\ p_2 : \mathcal{P}\,\Gamma_2,\qquad \langle\pi_1\rangle p_1 \ \wedge\ \langle\pi_2\rangle p_2 \ =\ p_1 \times p_2.$

LEMMA **18** *For projections* π_1, π_2,

$$\begin{array}{lclclcl} [\pi_1^{-1}];\langle\pi_1\rangle & = & skip, & \qquad & \langle\pi_1\rangle;[\pi_1^{-1}] & \le & skip,\\ [\pi_2^{-1}];\langle\pi_2\rangle & = & skip, & \qquad & \langle\pi_2\rangle;[\pi_2^{-1}] & \le & skip. \end{array}$$

LEMMA **19** *For* $q_1, q_2 : \mathcal{P}\,\Sigma_1 \times \Sigma_2,\qquad [\pi_1^{-1}]\ q_1 \ \times\ [\pi_2^{-1}]\ q_2 \ \le\ q_1 \wedge q_2.$

THEOREM **6** *For* $S_1 : \Sigma \mapsto \Gamma_1,\ \ S_2 : \Sigma \mapsto \Gamma_2$,

$$S_1 \triangle S_2 \ = \ S_1;[\pi_1^{-1}] \ \odot\ S_2;[\pi_2^{-1}].$$

Proof: For $q : \mathcal{P}\,(\Gamma_1 \times \Gamma_2)$,

$$\begin{array}{ll}
 & (S_1;[\pi_1^{-1}] \ \odot\ S_2;[\pi_2^{-1}])\ q\\
= & (\exists q_1, q_2 : \mathcal{P}\,(\Gamma_1 \times \Gamma_2) \mid q_1 \wedge q_2 \le q \cdot (S_1;[\pi_1^{-1}]\ q_1) \wedge (S_2;[\pi_2^{-1}]\ q_2))\\
\le & \qquad \{\text{Lemma 19}\}\\
 & (\exists q_1, q_2 : \mathcal{P}\,(\Gamma_1 \times \Gamma_2) \mid\ [\pi_1^{-1}]q_1 \times [\pi_2^{-1}]q_2 \le q \cdot (S_1;[\pi_1^{-1}]\ q_1) \wedge (S_2;[\pi_2^{-1}]\ q_2))\\
\le & \qquad \{\text{take } q_1' = [\pi_1^{-1}]q_1, \quad q_2' = [\pi_2^{-1}]q_2\}\\
 & (\exists q_1' : \mathcal{P}\,\Gamma_1;\ q_2' : \mathcal{P}\,\Gamma_2 \mid q_1' \times q_2' \le q \cdot S_1 q_1' \ \wedge\ S_2 q_2')\\
= & (S_1 \triangle S_2)\ q.
\end{array}$$

$$\begin{array}{ll}
 & (S_1 \triangle S_2)\ q\\
= & (\exists q_1 : \mathcal{P}\,\Gamma_1;\ q_2 : \mathcal{P}\,\Gamma_2 \mid q_1 \times q_2 \le q \cdot S_1 q_1 \ \wedge\ S_2 q_2)\\
= & \qquad \{\text{Lemma 17}\}\\
 & (\exists q_1 : \mathcal{P}\,\Gamma_1;\ q_2 : \mathcal{P}\,\Gamma_2 \mid\ \langle\pi_1\rangle q_1 \wedge \langle\pi_2\rangle q_2 \le q \cdot S_1 q_1 \ \wedge\ S_2 q_2)\\
= & \qquad \{\text{Lemma 18}\}\\
 & (\exists q_1 : \mathcal{P}\,\Gamma_1;\ q_2 : \mathcal{P}\,\Gamma_2 \mid\ \langle\pi_1\rangle q_1 \wedge \langle\pi_2\rangle q_2 \le q \cdot\\
 & \qquad (S_1;[\pi_1^{-1}];\langle\pi_1\rangle\ q_1) \wedge (S_2;[\pi_2^{-1}];\langle\pi_2\rangle\ q_2))\\
\le & \qquad \{\text{take } q_1' = \langle\pi_1\rangle q_1, \quad q_2' = \langle\pi_2\rangle q_2\}\\
 & (\exists q_1', q_2' : \mathcal{P}\,(\Gamma_1 \times \Gamma_2) \mid q_1' \wedge q_2' \le q \cdot (S_1;[\pi_1^{-1}]\ q_1') \wedge (S_2;[\pi_2^{-1}]\ q_2'))\\
= & (S_1;[\pi_1^{-1}] \ \odot\ S_2;[\pi_2^{-1}])\ q.
\end{array}$$

□

THEOREM **9** *For predicate transformers S_1 and S_2, $S_1 \odot S_2$ solves:*

$$\begin{aligned}\{halt\ S_2\}; S_1 &\leq X && (4)\\ \{halt\ S_1\}; S_2 &\leq X, && (5)\end{aligned}$$

and furthermore, for any conjunctive Y that solves (1) and (2),

$$S_1 \odot S_2 \quad \leq \quad Y. \tag{6}$$

Proof: Firstly, $S_1 \odot S_2$ solves (4):

$$\begin{aligned}(S_1 \odot S_2)\ q &= (\exists q_1, q_2 : \mathcal{P}\,\Gamma \mid q_1 \wedge q_2 \leq q \cdot S_1 q_1 \ \wedge\ S_2 q_2)\\ &\geq S_1 q \ \wedge\ S_2 \top\\ &= \{S_2 \top\}; S_1\ q.\end{aligned}$$

Similarly, $S_1 \odot S_2$ solves (5). Finally, $S_1 \odot S_2$ satisfies (6):

$$\begin{aligned}(S_1 \odot S_2)\ q &= (\exists q_1, q_2 : \mathcal{P}\,\Gamma \mid q_1 \wedge q_2 \leq q \cdot S_1 q_1 \ \wedge\ S_2 q_2)\\ &\leq \qquad \{Y \text{ satisfies (4) and (5)}\}\\ &\quad (\exists q_1, q_2 : \mathcal{P}\,\Gamma \mid q_1 \wedge q_2 \leq q \cdot Y q_1 \ \wedge\ Y q_2)\\ &= \qquad \{Y \text{ conjunctive}\}\\ &\quad (\exists q_1, q_2 : \mathcal{P}\,\Gamma \mid q_1 \wedge q_2 \leq q \cdot Y(q_1 \wedge q_2))\\ &\leq \qquad \{Y \text{ monotonic}\}\\ &\quad Y\ q.\end{aligned}$$

□

THEOREM **10** *For relations $P, Q : \Sigma \leftrightarrow \Gamma$, and predicates $p, q : \mathcal{P}\,\Sigma$,*

$$\{p\}; [P]\ \odot\ \{q\}; [Q] \quad = \quad \{p \wedge q\}; [P \wedge Q].$$

Proof: It is easy to show that $\{p \wedge q\}; [P \wedge Q]$ solves (4) and (5), thus by Theorem 9 $(\{p\}; [P]\ \odot\ \{q\}; [Q]) \ \leq\ \{p \wedge q\}; [P \wedge Q]$. To show the reverse, we have for $r : \mathcal{P}\,\Gamma,\ \ \sigma : \Sigma$,

$$\begin{aligned}&\{p \wedge q\}; [P \wedge Q]\ r\ \sigma\\ =&\qquad \{\text{by definition}\}\\ &p\ \sigma \wedge q\ \sigma \wedge (\forall \sigma' : \Gamma \cdot (P\ \sigma\ \sigma') \wedge (Q\ \sigma\ \sigma') \Rightarrow r\ \sigma')\\ =&\qquad \{\text{pointwise extension}\}\\ &p\ \sigma \wedge q\ \sigma\ \ \wedge ((\lambda \sigma' : \Gamma \cdot P\ \sigma\ \sigma')\ \wedge (\lambda \sigma' : \Gamma \cdot Q\ \sigma\ \sigma')\ \leq\ r)\\ =&\qquad \{ \qquad \{p\}; [P]\ (\lambda \sigma' : \Gamma \cdot P\ \sigma\ \sigma')\ \sigma\\ &\qquad\qquad = p\ \sigma \wedge (\forall \sigma' : \Gamma \cdot (P\ \sigma\ \sigma') \Rightarrow (\lambda \sigma' : \Gamma \cdot P\ \sigma\ \sigma')\ \sigma')\\ &\qquad\qquad = p\ \sigma \wedge (\forall \sigma' : \Gamma \cdot (P\ \sigma\ \sigma') \Rightarrow (P\ \sigma\ \sigma'))\\ &\qquad\qquad = p\ \sigma \qquad\qquad\qquad\qquad \}\\ &\{p\}; [P]\ (\lambda \sigma' : \Gamma \cdot P\ \sigma\ \sigma')\ \sigma\ \ \wedge\\ &\{q\}; [Q]\ (\lambda \sigma' : \Gamma \cdot Q\ \sigma\ \sigma')\ \sigma\ \ \wedge\\ &((\lambda \sigma' : \Gamma \cdot P\ \sigma\ \sigma') \wedge (\lambda \sigma' : \Gamma \cdot Q\ \sigma\ \sigma')\ \leq\ r)\\ \Rightarrow&\qquad \{\text{take } q_1 = (\lambda \sigma' : \Gamma \cdot P\ \sigma\ \sigma'),\ q_2 = (\lambda \sigma' : \Gamma \cdot Q\ \sigma\ \sigma')\ \}\\ &(\exists q_1, q_1 : \mathcal{P}\,\Gamma \cdot (q_1 \wedge q_2\ \leq\ r)\ \wedge\ \{p\}; [P]\ q_1\ \sigma\ \wedge\ \{q\}; [Q]\ q_2\ \sigma)\\ =\ &(\exists q_1, q_1 : \mathcal{P}\,\Gamma \mid q_1 \wedge q_2 \leq r \cdot \{p\}; [P]\ q_1\ \wedge\ \{q\}; [Q]\ q_2)\ \sigma\\ =\ &(\ \{p\}; [P]\ \odot\ \{q\}; [Q]\ )\ r\ \sigma.\end{aligned}$$

□

LEMMA **15** *For* $P_1 : \Sigma \leftrightarrow \Gamma_1, P_2 : \Sigma \leftrightarrow \Gamma_2$, $Q_1 : \Gamma_1 \leftrightarrow \Phi_1, Q_2 : \Gamma_2 \leftrightarrow \Phi_2$,

$$(P_1 \bigtriangleup P_2); (Q_1 \otimes Q_2) \quad = \quad (P_1; Q_1) \bigtriangleup (P_2; Q_2).$$

Proof:

$$\begin{aligned}
& (P_1 \bigtriangleup P_2); (P_1 \otimes Q_2) \ \sigma \ (\phi_1, \phi_2) \\
= \ & (\exists \gamma_1, \gamma_2 \cdot (P_1 \bigtriangleup P_2) \ \sigma \ (\gamma_1, \gamma_2) \ \wedge \ (Q_1 \otimes Q_2) \ (\gamma_1, \gamma_2) \ (\phi_1, \phi_2)) \\
= \ & (\exists \gamma_1, \gamma_2 \cdot P_1 \ \sigma \ \gamma_1 \ \wedge \ P_2 \ \sigma \ \gamma_2 \ \wedge \ Q_1 \ \gamma_1 \ \phi_1 \ \wedge \ Q_2 \ \gamma_2 \ \phi_2) \\
= \ & (\exists \gamma_1 \cdot P_1 \ \sigma \ \gamma_1 \ \wedge \ Q_1 \ \gamma_1 \ \phi_1) \ \wedge \ (\exists \gamma_2 \cdot P_2 \ \sigma \ \gamma_2 \ \wedge \ Q_2 \ \gamma_2 \ \phi_2) \\
= \ & P_1; Q_1 \ \sigma \ \phi_1 \ \wedge \ P_2; Q_2 \ \sigma \ \phi_2 \\
= \ & (P_1; Q_1) \bigtriangleup (P_2; Q_2) \ \sigma \ (\phi_1, \phi_2)
\end{aligned}$$

$\square$

This leads easily to the following lemma:

LEMMA **20** *For conjunctive* S_1, S_2, *universally conjunctive* T_1, T_2,

$$(S_1 \bigtriangleup S_2) \ ; \ (T_1 \otimes T_2) \quad = \quad S_1; T_1 \quad \bigtriangleup \quad S_2; T_2.$$

THEOREM **16** *Assume* α *has the form* $\{P_1 \otimes P_2\}$. *Let* $\alpha_1 = \{P_1\}$, $\alpha_2 = \{P_2\}$. *Then for conjunctive* S_1, S_2,

$$\begin{aligned}
\alpha; (S_1 \bigtriangleup S_2); \alpha^r \ &\leq \ (\alpha; S_1; \alpha_1^r) \ \bigtriangleup \ (\alpha; S_2; \alpha_2^r) \\
\alpha; (T_1 \otimes T_2); \alpha^r \ &\leq \ (\alpha_1; T_1; \alpha_1^r) \ \otimes \ (\alpha_2; T_2; \alpha_2^r).
\end{aligned}$$

Proof: First:

$$\begin{aligned}
\alpha^r \ &= \ \{P_1 \otimes P_2\}^r \\
&= \ [(P_1 \otimes P_2)^{-1}] \\
&= \ [P_1^{-1} \ \otimes \ P_2^{-1}] \\
&= \ [P_1^{-1}] \ \otimes \ [P_2^{-1}] \\
&= \ \alpha_1^r \ \otimes \ \alpha_2^r
\end{aligned}$$

Then:

$$\begin{aligned}
\alpha; (S_1 \bigtriangleup S_2); \alpha^r \ &= \ \alpha; (S_1 \bigtriangleup S_2); \ (\alpha_1^r \ \otimes \ \alpha_2^r) \\
&= \qquad \{\text{Lemma 20, } \alpha_1^r, \alpha_2^r \text{ universally conjunctive}\} \\
& \quad \ \alpha; (S_1; \alpha_1^r \quad \bigtriangleup \quad S_2; \alpha_2^r) \\
&\leq \qquad \{\text{Lemma 14, } \ \alpha \text{ disjunctive}\} \\
& \quad \ \alpha; S_1; \alpha_1^r \quad \bigtriangleup \quad \alpha; S_2; \alpha_2^r
\end{aligned}$$

Next:

$$\begin{aligned}
\alpha; (T_1 \otimes T_2); \alpha^r \ &= \ \alpha; (\langle \pi_1 \rangle; T_1 \quad \bigtriangleup \quad \langle \pi_2 \rangle; T_2); \alpha^r \\
&\leq \qquad \{\text{Above}\} \\
& \quad \ \alpha; \langle \pi_1 \rangle; T_1; \alpha_1^r \quad \bigtriangleup \quad \alpha; \langle \pi_2 \rangle; T_2; \alpha_2^r \\
&\leq \qquad \{\text{Since } \{P_1 \otimes P_2\}; \langle \pi_i \rangle \ \leq \ \langle \pi_i \rangle; \{P_i\}, \ i \in 1..2 \ \} \\
& \quad \ \langle \pi_1 \rangle; \alpha_1; T_1; \alpha_1^r \quad \bigtriangleup \quad \langle \pi_2 \rangle; \alpha_2; T_2; \alpha_2^r \\
&= \ \alpha_1; T_1; \alpha_1^r \quad \otimes \quad \alpha_2; T_2; \alpha_2^r
\end{aligned}$$

$\square$

References

[1] J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] R.J.R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*. Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[3] R.J.R. Back and M.J. Butler. Exploring summation and product operators in the refinement calculus. In B. Möller, editor, *Mathematics of Program Construction, 1995*, volume LNCS 947. Springer–Verlag, 1995.

[4] R.J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[5] R.J.R. Back and K. Sere. Superposition refinement of parallel algorithms. In K.A. Parker and G.A. Rose, editors, *FORTE'91*. North–Holland, 1992.

[6] R.J.R. Back and J. von Wright. Refinement concepts formalised in higher order logic. *Formal Aspects of Comput.*, 5:247–272, 1990.

[7] M.J. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27(2), September 1996.

[8] M.J. Butler, J. Grundy, T. Långbacka, R. Rukšėnas, and J. von Wright. The refinement calculator: proof support for program refinement. In L. Groves and S. Reeves, editors, *Formal Methods Pacific'97*, Springer Series in Discrete Mathematics and Theoretical Computer Science. Springer, 1997.

[9] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[10] P.H.B. Gardiner and C.C. Morgan. Data refinement of predicate transformers. *Theor. Comp. Sci.*, 87:143–162, 1991.

[11] M. Gordon and T. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[12] J. He, C.A.R. Hoare, and J.W. Sanders. Data refinement refined. In *European Symposium on Programming*, volume LNCS 213. Springer–Verlag, 1986.

[13] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice–Hall, 1985.

[14] P.F. Hoogendijk and R.C. Backhouse. Relational programming laws in the tree, list, bag, set hierarchy. *Sci. Comp. Prog.*, 22(1-2):67–105, 1994.

[15] He Jifeng and C.A.R. Hoare. *Categorical Semantics of Programming Languages*. In *Mathematical Foundations of Programming Semantics*, volume LNCS 442. Springer–Verlag, 1990.

[16] C.B. Jones. *Systematic Software Development using VDM – Second Edition*. Prentice–Hall, 1990.

[17] K.R.M. Leino and R. Manohar. *Joining specification statements*. Available from `http://www.research.digital.com/SRC/personal/Rustan_Leino/papers.html`, July 1996.

[18] C.E. Martin. *Preordered Categories and Predicate Transformers*. D.Phil. Thesis, Programming Research Group, Oxford University, 1991.

[19] A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In *FME'97*, 1997.

[20] C.C. Morgan. The specification statement. *ACM Trans. Prog. Lang. and Sys.*, 10(3), 1988.

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

[22] C.C. Morgan. The cuppest capjunctive capping, and Galois. In A.W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare.* Prentice–Hall, 1994.

[23] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comp. Prog.*, 9(3):298–306, 1987.

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

[25] D.A. Naumann. *Two-Categories and Program Structure: Data Types, Refinement Calculi, and Predicate Transformers.* Ph.D. Thesis, University of Texas at Austin, 1992.

[26] J.M. Spivey. *The Z Notation - A Reference Manual.* Prentice–Hall, 1989.

[27] J. von Wright. The lattice of data refinement. *Acta Inf.*, 31(2):105–135, 1994.

[28] J. von Wright. A unified theory of data refinement. Åbo Akademi University, Dept. of Computer Science, February 1994.

[29] N. Ward. Adding specification constructs to the refinement calculus. In *FME'93*, volume LNCS 670. Springer–Verlag, 1993.