Programming Research Group

THE WEAKEST SPECIFUNCTION

Yifeng Chen and J. W. Sanders

PRG-TR-08-00

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

The Weakest Specifunction

Yifeng Chen and J. W. Sanders

Programming Research Group, OUCL,
Parks Road, Oxford OX1 3QD, UK
{yc, jeff}@comlab.ox.ac.uk

November 2, 2000

Abstract

This paper introduces a calculus of weakest specification for supporting reuse of established components in deriving a design (in the sense of formal methods). The *weakest specifunction* generalises the notions of weakest prespecification and weakest parallel environment; but instead of calculating the weakest required component of a target specification, it calculates the weakest specification *function* whose value refines the target when applied to an established component. In particular it overcomes the restriction of those other calculii to taking merely one required component at a time.

A general theory of specifunctions is developed and applied to a new weakest-design calculus, which involves two required components: the *par-seq specifunction* places one established component in parallel with one required component and the result in sequence with another required component to meet a given specification. The weakest par-seq specifunction is of importance in any derivation calculus for a language combining parallel and sequential composition. A complete theory for it is presented here in the context of a refinement calculus for an abstraction of PRAM and BSP programming, and the resulting calculus is applied to the derivation of a distributed algorithm for greatest common divisor.

1 Introduction

One of the important techniques to have arisen in the quest to ensure that software engineering becomes more efficient is *reuse* of *established components*. Such components include: legacy code which must be revised module by module; part of a design being produced by a cooperating development team; a simple abstracted version of a complex design; a smaller instance of the desired design (recursion); as well as the usual 'off the shelf' component implemented previously. However the software engineering technique of formal methods has been slow to provide support for reuse. In this paper we propose such a formal method. In the context of a refinement calculus it provides notation and laws to support the derivation of a program from its specification, given certain components which are to be incorporated in the final product. In particular it leads to a general framework for compositional refinement.

The weakest-specfunction calculus extends several previous 'weakest' calculii. The general technique may be motivated by analogy with algebra. An equation in unknown $x:X$ specifies its solutions whilst providing no explicit indication of how to identify them. To that extent it is analogous to a software specification, which describes what its implementations achieve without indicating their structure. Solution of an equation requires inversion of the algebraic operations involved in its definition. But not every operation, of a given type, has an inverse. For example, amongst integers multiplication does not have an inverse.

However the notion of integer division provides a *weak inverse* to integer multiplication. It is readily defined:

$$q \mathbin{\mathsf{div}} p \;\hat{=}\; \sqcup \{n:\mathbb{N} \mid n \times p \leq q\}, \tag{1}$$

where $\sqcup$ denotes maximum. Evidently the set on the right-hand side is nonempty and finite unless $p = 0$ in which case $q \mathbin{\mathsf{div}} p$ is not (and ought not to be) defined.

Definition (1) is much easier to use than the standard definition based on the Euclidean algorithm. The simple general theory of Galois connections yields from it the following equivalent, which makes explicit that integer division is a weak inverse of $\times$

$$x \leq q \mathbin{\mathsf{div}} p \;\equiv\; x \times p \leq q. \tag{2}$$

Most of the desired laws of integer division follow in an entirely straightforward manner from either (1) or (2). We mention just the following three laws, for which analogues in the context of program derivation will appear in section 2. Weak inverse is exact in this sense:

$$(q \times p) \; \mathsf{div} \; p = q; \tag{3}$$

repeated division simplifies

$$(q \; \mathsf{div} \; p) \; \mathsf{div} \; r = q \; \mathsf{div} \; (p \times r); \tag{4}$$

and addition in the numerator distributes in this sense:

$$(q \; \mathsf{div} \; p) + (r \; \mathsf{div} \; p) \leq (q+r) \; \mathsf{div} \; p. \tag{5}$$

Such laws may then be used to assist in solution of 'specifications' involving integer variables and multiplication. Thus even in integer arithmetic there is value to be gained by considering the weak inverses of the various operations and by using the simple theory of Galois connections to construct a calculus. In this paper we shall use that approach in the calculus of program derivation. There it is nondeterminism which gives rise to 'weakness' and it is the specification and program combinators which are inverted.

Before leaving elementary mathematics we make one further observation. Formula (1) provides an example in which integers q and p are known but the two integers d and r are unknown; however either of d and r determines the other. In some situations, particularly in program derivation, we need to solve for several independent unknowns. Here is an analogous example from applied mathematics.

In a constraint-satisfaction problem it may be required to determine several unknown coefficients c_i $(0 \leq i < n)$ of some design $f = f(x, c_0, \cdots, c_{n-1})$ in order to satisfy a constraint

$$q \leq f(p, c_0, \cdots, c_{n-1}) \, .$$

(See, for example, [9] chapter 9 for a range of examples. There the more typical formulation $-q \leq -|g(p) - \sum_{0 \leq i < n} \; c_i p^i|$ is used. We have used a form which corresponds directly with programming; compare the inequality above with expression (7).) Each set of coefficients determines a particular function. Instead of directly dealing with $f(p)$, we first study the coefficients of the minimum function that satisfies the constraint. Such a minimum may well not exist; however if it does, its corresponding design enjoys some important properties:

1. it sets a theoretical minimum for any design satisfying the constraint, so that any other design satisfying the constraint must dominate, pointwise, this minimum function;

2. if this minimum design can be calculated using mathematical laws, it becomes a new starting point for development of an even better solution allowing maximum freedom for further improvement;

3. a slight modification or improvement of p still keeps $f(p)$ as low as possible because the function is a pointwise minimum.

The crux of this paper is the application of weak inversion to program specifications involving several unknowns. In particular given specification Q and established component P we shall wish to find components A and B for which $(P\|A) \fatsemi B$ refines Q.

The concept of 'weakest calculation' in computing owes its origins to Dijkstra's *weakest preconditions* [7]. There a program is characterised as the predicate transformer which maps a postcondition (i.e. predicate over final states) to the weakest precondition (i.e. predicate over initial states) certain to be converted by action of the program to the given postcondition. More precisely, for each program P in Dijkstra's guarded-command language, $wp.P$ denotes its predicate-transformer semantics: for any postcondition q, $(wp.P)(q)$ denotes the weakest precondition from which execution of P is certain to terminate in a state satisfying q. In terms of disjunction of predicates and Hoare logic $p\,\{P\}\,q$ its definition bears some resemblance to (1):

$$(wp.P)(q) \;\hat{=}\; \vee\,\{p \mid p\,\{P\}\,q\}.$$

By replacing each postcondition and precondition by a binary predicate (of initial and final states) and by using such predicates to represent not just code but also specifications, C.A.R. Hoare and J. He generalised Dijkstra's weakest precondition calculus to a calculus of *weakest prespecifications* [11]. Refinement of specifications corresponds to removal of nondeterminism. In terms of the combinator $\sqcap$ for nondeterminism it is defined:

$$P \sqsubseteq Q \;\hat{=}\; P \sqcap Q = P. \tag{6}$$

In those terms the weakest prespecification of P in Q is the weakest (i.e. most nondeterministic, most unrefined) specification X for which X followed by P meets Q:

$$Q/P \;\hat{=}\; \sqcap\,\{X \mid Q \sqsubseteq X \fatsemi P\}.$$

The result is a weak inverse for sequential composition on the right. There is a similar definition, and calculus, for sequential composition on the left; it is called *weakest postspecification*. One of the consequences of the results of the present paper is that X has been liberated to lie in any fixed position in such a composition of components.

Weakest prespecification and postspecification provide a weak inverse for sequential composition. That approach was extended to parallel composition by Lai and Sanders [14] who studied the *weakest environment*: the weakest process which placed in parallel with an established component P meets specification Q:

$$Q /\!\!/ P \;\hat{=}\; \sqcap \{X \mid Q \sqsubseteq X \| P\}.$$

That was done in a model without sequential composition, and few laws were provided for its calculus, several of them with complex assumptions. Nonetheless their approach was applied to Valiant's BSP [18] by Lecomber [15]. In the present paper we remove many of the restrictions of [14] and show that our results are still capable of application to a general PRAM/BSP-style of programming.

The theory of Galois connections shows that in general any operator which distributes nondeterminism has a weak inverse on which can be based a calculus. In the cases of weakest prespecification, weakest postspecification and weakest environment, we may view the unknown component as an argument of the operator and the known component as a parameter. Then all conform to the following definition for appropriate choices of operator f_P:

$$\sqcap\{X \mid Q \sqsubseteq f_P(X)\}. \tag{7}$$

However that observation cannot be used as the basis for a useful theory. For, as noticed in [12], it is not at all easy to ensure that an operator distributes nondeterminism; it fails, for example, for a structure with two or more unknown components. More realistic is to suppose that we have a single known component whose environment includes more than one unknown component. In this paper the *weakest specifunction* is introduced and a general theory established. Thus definition (7) is replaced by

$$\sqcap\{f \mid Q \sqsubseteq f(P)\} \tag{8}$$

in which P is an established component and the function f is unknown.

The technique of weakest specifunction is well suited, for example, to the derivation of programs in the refinement calculus of a language exhibiting both sequential and parallel composition, so in this paper we concentrate on that context. Indeed for much of this paper we shall be occupied with the *par-seq specifunction*, in which specification Q and established component P are known whilst X and Y are unknown components for which the parallel composition of P and X, followed by Y, refines Q:

$$Q \sqsubseteq (\lambda E \cdot (E \| X) \fatsemi Y)(P).$$

The context will be the language *Logs* of [6] but of course the approach is general: the general technique is of use in deriving programs in any programming language.

Whilst generalising each of those constructs, weakest specifunction raises them to the level of functions between designs. As observed, that removes the difficulty noted in [12] of extending previous approaches to designs with more than one required component, which limitation has been partly responsible for the restricted use of the technique to date. The weakest specifunction promotes the introduction of various new weakest calculii; and enables us to study the general properties shared by all such calculii.

The weakest specifunction calculates, as the name implies, the *weakest* function certain to complete a family of refinements using an established component. We are interested in *weakest* designs because they allow the implementor the greatest subsequent implementation freedom whilst still ensuring correctness. In particular they diminish the chance of producing so-called *miraculous* or *infeasible* designs: those which cannot be refined to code.

Weakest specifunction is defined in section 2 and studied there and in section 3 where the important par-seq calculus is investigated. It can be thought of as a specification calculus for the popular PRAM paradigm [8] and Valiant's more portable BSP [18] (in which processes act sequentially but communicate asynchronously with each other using global variables; after each *superstep* all communications must have been synchronised). *Logs* abstracts BSP's communication details and combines sequential and parallel constructs. In section 4 the new calculus is applied to derive a program for the greatest common divisor of two integers. Further laws and more involved examples, including some which appear to be new, appear in [5].

We begin by introducing the refinement structure in which our calculus is defined.

2 Weakest specifunctions

2.1 Specification spaces

Program derivation produces a sequence of refinements spanning specification to implementation. Each 'design' in the sequence is in general a hybrid of specification and code. The space of such entities we call 'specification space' and of it require only that it be closed under arbitrary common refinements and their opposite. However we follow tradition and take as primitive the refinement ordering.

Definition 1 *A* specification space *is a complete lattice* $S = (\mathsf{S}, \sqsubseteq)$. *Thus with respect to the greatest lower bound* (glb), $\sqcap$, *and least upper bound* (lub), $\sqcup$, *of* $\sqsubseteq$ *it is a lattice in which* every *subset has a glb and lub. In particular, it has a maximum element* $\top$ *and minimum element* $\bot$.

Examples, important for program derivation, are as follows.

1. The specification space of all predicates on state space W, $Pred(W)$, consists of all Boolean-valued functions on W with the pointwise ordering lifted from that on the Booleans $\mathbb{B}$: for $\phi, \psi : Pred(W)$,

   $$\phi \sqsubseteq \psi \;\hat{=}\; \forall w : W \cdot \phi(w) \geq \psi(w).$$

   Reversal of the order there corresponds to the intuition that the constant predicate *true* is underspecification and so is the weakest predicate $\bot$, whilst the constant predicate *false* is overspecification and is the strongest predicate $\top$. Thus as a specification the minimum imposes no constraint and the maximum is inconsistent or overconstraint. Glb is conjunction and lub is disjunction.

   That space is of great importance in specification, and therefore also as a space in which are embedded many pragmatic semantic models.

2. Refinement calculii for sequential languages tend to focus on (extensions of) Dijkstra's guarded-command language. For signature (i.e. tuple of program variables) $w : W$, it has given rise to two popular specification spaces. Perhaps foremost is the space of all monotonic predicate

transformers over W (i.e. monotonic functions on the space of predicates on W) endowed with the pointwise-implication order [2]

$$t \sqsubseteq u \;\hat{=}\; \forall \phi : Pred(W) \cdot t(\phi) \leq u(\psi).$$

That provides a rich space for derivations to code by extending the guarded-command language to contain also assertions and coercions and by being closed under sequential composition, demonic choice (glb), angelic choice (lub) and dual. This space therefore provides the natural semantics of the sequential refinement calculus [16] which reveals further constructs like the specification statement, logical constants, local blocks and procedure invocation.

An alternative semantics is provided by the binary-relation (i.e. state-before/state-after) model [13], consisting of all relations on W with the inclusion ordering (i.e. 'at least as deterministic, as relations')

$$r \sqsubseteq s \;\hat{=}\; r \supseteq s.$$

Although both model the guarded-command language, their laws differ (for example in distributing sequential composition over the two kinds of choice). Observe that the order on relations is consistent with that on predicates: its top is $\{\,\}$ and its bottom $W \times W$.

In this paper we shall not be committed to one model or the other and simply let $(Seq, \sqsubseteq)$ denote a specification space for sequential programming. Its laws will, not surprisingly, determine is properties.

3. Refinement calculii for parallel programming, as far as they have been explored, focus on the various process algebras of which there are several. For example the process algebra CSP (for Communicating Sequential Processes) [10] has been provided with a range of semantic models each yielding a specification space. However the primary model is that in which each process is represented by its failures and divergences over some universe X, with the *at least as deterministic* order ([13], section 8.2). It contains CSP code [10] and is closed under sequential composition, external choice, hiding, nondeterministic choice and parallel composition. Other models of CSP are described in [17].

   In this paper we simply let $(Par, \sqsubseteq)$ denote a specification space for parallel programming.

4. For more-loosely coupled parallel processes, the specification space of Bulk-Synchronous Processes over a given signature with the 'at least as deterministic and at least as stable' order, contains BSP code [18] and also the same kinds of miraculous or infeasible specification constructs as above [15, 3].

   A second, slightly more abstract, model is provided by the specification space $Logs(w)$ of logic for global synchrony having signature w [6]. It is in the context of $Logs(w)$ that we exploit the theory expounded in this paper. A summary of the language is given in section 3.

Further examples of specification spaces appear, for example, in [13].

2.2 Specifunctions

A derivation step in program derivation corresponds to a transformation of specification space. It is such transformations that we are interested in studying.

Definition 2 *A* specifunction *is a function f between specification spaces which satisfies, for any subset E of the domain,*

$$f(\sqcap E) = \sqcap\{f(e) \mid e \in E\}.$$

In particular f is monotone. The set of all specifunctions from specification space $S = (\mathsf{S}, \sqsubseteq_S)$ to specification space $T = (\mathsf{T}, \sqsubseteq_T)$ is denoted $\mathcal{S}(S,T)$. Order between specifunctions is defined pointwise: if $f, g : \mathcal{S}(S,T)$ then

$$f \sqsubseteq g \;\hat{=}\; \forall s : \mathsf{S} \cdot f(s) \sqsubseteq_T g(s).$$

For example for specification spaces S and T as above, the specifunction in $\mathcal{S}(S,T)$ which transforms any specification in S to the greatest specification $\top$ in T equals the constant specifunction $(\lambda E : \mathsf{S} \cdot \top)$.

In the following we shall be lazy and suppress the carrier spaces, referring to them by the name of the specification space itself. Thus for example the previous λ-expression will be written henceforth as $(\lambda E : S \cdot \top)$.

It is elementary to show that with that ordering the space of all specifunctions itself becomes a specification space.

Proposition 1 *For any specification spaces S and T, the space $(\mathcal{S}(S,T), \sqsubseteq)$ of all specifunctions from S to T is partially ordered (pointwise) and forms a complete lattice in which glb and lub are pointwise.*

Note that glb and lub in a space of specifunctions are not in general the pointwise glb and lub; a counterexample is given in 2 below. With that warning, we continue to use the same symbols $\sqsubseteq$, $\sqcap$ and $\sqcup$.

Proposition 2 *In a complete lattice of specifunctions, if all pointwise glb's (or pointwise lub's) belong to the lattice, they form the glb's (or lub's) of the lattice.*

Important specifunctions, and sets of specifunctions, include the following.

1. The primary specification combinator is conjunction. To it corresponds the specifunction defined on specification space $Pred(W)$, for $\psi : W \to \mathbb{B}$, by

   $$f_\psi(\xi) \ \hat{=} \ \xi \wedge \psi.$$

   Observe that in $Pred(W)$, by idempotence of conjunction,

   $$f_\psi \sqcap f_\phi = f_{\psi \vee \phi}.$$

   So in this case the glb of specifunctions is expressed in terms of (the glb of) specifications.

   Note that the useful specification construct of negation is not a specifunction since it is not monotonic; indeed for that reason its use in hierarchical refinement is severely limited.

2. On a specification space Seq closed under sequential composition three specifunctions are of interest: composition by established components on the left, right, and on both sides:

   $$\begin{array}{l} (\lambda E : Seq \cdot A \mathbin{;} E) \\ (\lambda E : Seq \cdot E \mathbin{;} A) \\ (\lambda E : Seq \cdot A \mathbin{;} E \mathbin{;} B). \end{array}$$

   They are all specifunctions if non-deterministic choice distributes through sequential composition. The corresponding sets of specifunctions are

   $$\begin{array}{rcl} \mathcal{P}re(Seq) & \hat{=} & \{(\lambda E : Seq \cdot A \mathbin{;} E) \mid A \in Seq\} \\ \mathcal{P}ost(Seq) & \hat{=} & \{(\lambda E : Seq \cdot E \mathbin{;} A) \mid A \in Seq\} \\ \mathcal{S}w(Seq) & \hat{=} & \{(\lambda E : Seq \cdot A \mathbin{;} E \mathbin{;} B) \mid A, B \in Seq\}. \end{array}$$

Since $\mathcal{P}ost(Seq)$ is a complete lattice, it contains all lub's. But the lub of $\lambda E \cdot E \,;\, A$ and $\lambda E \cdot E \,;\, B$ is $\lambda E \cdot E \,;\, (A \sqcup B)$ which need not (for instance in the relational model) equal $\lambda E \cdot ((E \,;\, A) \sqcup (E \,;\, B))$.

3. On a specification space Par closed under parallel composition, the specifunction corresponding to parallel composition with an established context or environment has specifunction set

$$\mathcal{P}ar(Par) \;\; \hat{=} \;\; \{(\lambda E : Par \cdot E \,\|\, A) \mid A \in Par\}.$$

4. For any specification spaces S and T, the largest set of specifunctions from S to T equals $\mathcal{S}(S,T)$.

A specifunction acts on a set of specifunctions as follows.

Definition 3 *The action of specifunction $f : \mathcal{S}(U,T)$ on specifunction set $\mathcal{G} \subseteq \mathcal{S}(S,U)$ is $f \circ \mathcal{G} \;\hat{=}\; \{f \circ g \mid g \in \mathcal{G}\}$.*

The composition of two specifunction sets $\mathcal{F} \subseteq \mathcal{S}(U,T)$ and $\mathcal{G} \subseteq \mathcal{S}(S,U)$ is $\mathcal{F} \circ \mathcal{G} \;\hat{=}\; \{f \circ g \mid f \in \mathcal{F},\, g \in \mathcal{G}\}$.

A specifunction set $\mathcal{F}$ is idempotent if $\mathcal{F} \circ \mathcal{F} = \mathcal{F}$.

For example $\mathcal{P}re$, $\mathcal{P}ost$, $\mathcal{S}w$, $\mathcal{P}ar$ and $\mathcal{S}(S,T)$ are all idempotent.

2.3 Weakest specifunctions

For the purpose of making a derivation step, it is the *weakest* design fitting the context which offers the largest number of options for further development. That leads us to the concept of the 'weakest specifunction' in a given set of specifunctions.

Definition 4 *Given specification spaces S and T, a set of specifunctions $\mathcal{F} \subseteq \mathcal{S}(S,T)$, a target specification $Q : T$, and an established component $P : S$, the* weakest specifunction, *$Q \div_{\mathcal{F}} P$, is defined to be the glb in $\mathcal{S}(S,T)$ of all specifunctions in $\mathcal{F}$ refining Q when they are applied to P*

$$Q \div_{\mathcal{F}} P \;\; \hat{=} \;\; \sqcap \{f : \mathcal{F} \mid Q \sqsubseteq f(P)\}.$$

In order to make calculation meaningful, the weakest specifunction must be a member of $\mathcal{F}$. That leads us to the definition:

Definition 5 *A set $\mathcal{F} \subseteq \mathcal{S}(S,T)$ of specifunctions is called* closed *if, for any non-empty subset $\mathcal{G} \subseteq \mathcal{F}$, the pointwise glb of $\mathcal{G}$ exists in $\mathcal{F}$.*

Thus for a closed specifunction space, if there is any specifunction refining the target specification when it is applied to the established component, the weakest one (which exists in $\mathcal{S}(S,T)$) lies in $\mathcal{F}$.

By way of example we have the following:

Theorem 1 *If operator $\sqcap$ of Seq is distributed by sequential composition (the case, for example, for the relational semantics of the guarded-command language) then the sets of specifunctions $\mathcal{P}re$ and $\mathcal{P}ost$ are closed; similarly $\mathcal{P}ar$ is closed if $\sqcap$ is distributed by parallel composition in Par (as is the case, for example, for the failures-divergences semantics of CSP).*

Proof. Consider first the case of $\mathcal{P}ar$. For any non-empty subset $\mathcal{G}$ of $\mathcal{P}ar$ there is a subset $\mathcal{A}$ of Par for which

$$\mathcal{G} \;\hat{=}\; \{(\lambda E\!:\!Par \cdot E \| A) \mid A \in \mathcal{A}\}.$$

Then the glb is $\sqcap \mathcal{G} = (\lambda E : Par \cdot E \| \sqcap \mathcal{A})$, a specifunction in $\mathcal{P}ar$ which is therefore closed.

Similar reasoning applies to $\mathcal{P}re$ and $\mathcal{P}ost$. $\square$

Some important sets of specifunctions are not closed; one is provided by the next result.

Theorem 2 *For specification space $T(w)$ of predicate transformers, the specifunction set $\mathcal{S}w$ is not closed.*

Proof. One of the simplest counterexamples seems to be the following.

Let $x : \mathbb{B}$ be the solitary variable, of Boolean type, and set f_1 and f_2 to be the following two specifunctions in $\mathcal{S}w$

$$\begin{array}{rcl}
f_1 & \hat{=} & (\lambda E\!:\!T \cdot A_1 \fatsemi E \fatsemi B_1) \\
f_2 & \hat{=} & (\lambda E\!:\!T \cdot A_2 \fatsemi E \fatsemi B_2) \\
A_1 & \hat{=} & \textbf{if } x \textbf{ then } \top \textbf{ else } (x := \mathit{false}) \\
B_1 & \hat{=} & \top \\
A_2 & \hat{=} & \textbf{if } x \textbf{ then } \top \textbf{ else } (x := \mathit{true}) \\
B_2 & \hat{=} & \textbf{if } x \textbf{ then } \top \textbf{ else } (x := \mathit{false}).
\end{array}$$

An exhaustive search has shown that there is no specifunction $(\lambda E\!:\!T \cdot A \fatsemi E \fatsemi B)$ which, at any $E\!:\!T$, equals the disjunction of f_1 and f_2. Thus $\mathcal{S}w$ is not closed. $\square$

Next we need a condition that ensures *some* solution exists, in order that being closed may yield a weakest solution.

Definition 6 *A set of specifunctions $\mathcal{F} \subseteq \mathcal{S}(S,T)$ is said to be* topped *if it contains the maximum specifunction, $(\lambda E\!:\!S \cdot \top_T)$, of $\mathcal{S}(S,T)$.*

In particular if a set of specifunctions from S to T is topped then for any target specification $Q\!:\!T$ and any initial specification $P\!:\!S$, the set contains at least one specifunction which when applied to P yields a refinement of Q.

The next result determines which of our examples are topped.

Theorem 3 *If specification space Seq satisfies $\forall E : Seq \cdot \top \mathbin{;} E = \top$ then the specifunction sets $\mathcal{P}re$ and $\mathcal{S}w$ are topped. The specifunction set $\mathcal{P}ost$ is* not *topped if Seq satisfies $\forall E\!:\!Seq \cdot (\perp \mathbin{;} E) = \perp \neq \top$.*

Specification space Par satisfies $\forall E\!:\!Par \cdot E \| \top = \top$ iff specifunction set $\mathcal{P}ar$ is topped.

Proof. In the first two cases the maximum specifunction

$$(\lambda E\!:\!Seq \cdot \top) \;\equiv\; (\lambda E\!:\!Seq \cdot \top \mathbin{;} E) \;\equiv\; (\lambda E\!:\!Seq \cdot \top \mathbin{;} E \mathbin{;} B)$$

belongs to the set of specifunctions. In the third any specifunction $f : \mathcal{P}ost$ is strict and so $f(P)$ does not refine Q unless $Q = \perp$. □

However evidently adding the maximum of $\mathcal{S}(S,T)$ to any set $\mathcal{F}$ of specifunctions yields a topped result.

The sets of specifunctions which are closed and topped are those upon which we rely for a useful calculus of program derivation.

Definition 7 *A set $\mathcal{F} \subseteq \mathcal{S}(S,T)$ of specifunctions is called a* specifunction space *if it is closed and topped. The set of all specifunction spaces from S to T is denoted* $\textsc{Spec}(S,T)$.

Since a closed and topped set of specifunctions is a complete lattice (with the ordering provided by definition 2), it is also a specification space. However recall that although we use the same notation, the glb and lub in $\mathcal{F}$ need not be expressible in terms of those of the underlying specification spaces. Indeed the lub (in specifunction space $\mathcal{P}ost$) of the specifunctions $\lambda E \cdot (E \mathbin{;} A)$ and $\lambda E \cdot (E \mathbin{;} B)$ is equal, pointwise, to the specifunction $\lambda E \cdot E \mathbin{;} (A \sqcup B)$; however in general

$$\lambda E \cdot (E \mathbin{;} A) \sqcup (E \mathbin{;} B)$$

cannot be expressed in the form $\lambda E \cdot (E \mathbin{;} C)$ and so the lub in $\mathcal{P}ost$ does not equal the pointwise lub.

In standard languages, $\mathcal{P}re$ and $\mathcal{P}ar$ are specifunction spaces by virtue of the laws above, whilst $\mathcal{S}w$ is not. The important space $\mathcal{P}ost$ becomes a specifunction space by having the maximum of $\mathcal{S}(S,T)$ added to it.

The characterisation of the weakest specifunction in terms of a Galois connection lifts to functions a standard result; it is important as a means of manipulating the definition.

Law 1 *For all* $f:\mathcal{F}$, $Q \sqsubseteq f(P)$ *iff* $Q \div_{\mathcal{F}} P \sqsubseteq f$.

From that result, the basic properties of weakest specifunctions follow readily. The first deals with a nondeterministic established component; the second with a conjoined specification; and the third provides the sense in which the weakest specifunction is immune from refinement of the established component.

Law 2 $Q \div_{\mathcal{F}} (P \sqcap R) = (Q \div_{\mathcal{F}} P) \sqcup (Q \div_{\mathcal{F}} R)$

Law 3 $(P \sqcup Q) \div_{\mathcal{F}} R = (P \div_{\mathcal{F}} R) \sqcup (Q \div_{\mathcal{F}} R)$

Law 4 *If* $P \sqsubseteq R$ *then* $(Q \div_{\mathcal{F}} P)(R) = (Q \div_{\mathcal{F}} R)(R)$.

2.4 Comparing specifunction spaces

Given specifunction spaces $\mathcal{F}, \mathcal{G} : \textsc{Spec}(S,T)$, for the purposes of program derivation $\mathcal{G}$ is at least as good as $\mathcal{F}$ if, for each $P:S$ and $Q:T$, the *least* specifunction $Q \div_{\mathcal{G}} P$ of $\mathcal{G}$ — for that is the one used in derivation — refines the least specifunction $Q \div_{\mathcal{F}} P$ of $\mathcal{F}$:

Definition 8 *The set* $\textsc{Spec}(S,T)$ *of specifunction spaces from* S *to* T *is pre-ordered by*

$$\mathcal{F} \sqsubseteq \mathcal{G} \ \ \hat{=}\ \ \forall P:S \setminus \{\top_S\} \cdot \forall Q:T \setminus \{\top_T\} \cdot \ Q \div_{\mathcal{F}} P \ \sqsubseteq \ Q \div_{\mathcal{G}} P.$$

We shall thus compare specifunction spaces by looking only at their performance at *non-magic* points.

That provides a pre-order on $\textsc{Spec}(S,T)$ with respect to which $\textsc{Spec}(S,T)$ itself becomes a complete lattice. Thus:

Theorem 4 *If S and T are specification spaces then* $(\textsc{Spec}(S,T), \sqsubseteq)$ *is a specification space.*

A trivial sufficient condition for ordering of specifunction spaces is containment:

Proposition 3 *If $\mathcal{F} \supseteq \mathcal{G}$ then $\mathcal{F} \sqsubseteq \mathcal{G}$.*

Thus $\mathcal{F} \sqsubseteq \mathcal{G}$ means that $\div_{\mathcal{F}}$ is never a worse approximation than $\div_{\mathcal{G}}$. For example the specifunction set

$$\mathcal{C}onst \ \hat{=} \ \{(\lambda E\!:\!S \cdot A) \mid A \in T\}$$

is a specification space which always contains the best approximation: for any pair P, Q

$$(Q \div_{\mathcal{C}onst} P)(P) \ = \ (\lambda E\!:\!S \cdot Q)(P) \ = \ Q.$$

Hence we have:

Proposition 4 *$\mathcal{C}onst$ is a specifunction space and is the minimum element of* $(\textsc{Spec}(S,T), \sqsubseteq)$.

2.5 Miraculous specifunction spaces

The equivalence generated by the pre-order on specifunction spaces captures equivalence from the viewpoint of derivation.

Definition 9 *Let $\cong$ denote the equivalence generated by the pre-order on specifunction spaces.*

Since the specifunction space $\mathcal{C}onst$ contains the best approximation for any specifunction space it must be equivalent to the space of all specifunctions: $\mathcal{C}onst \cong \mathcal{S}(S,T)$.

However such specifunction spaces can be *too* helpful in program derivations. They play the part here of miracles in the refinement calculii for the guarded-command language: for any target specification Q, there is always a specifunction $(\lambda E\!:\!\mathsf{S} \cdot Q)$, independent of E, which refines Q.

Definition 10 *Specification space $\mathcal{F}$ is* miraculous *iff $\mathcal{F} \cong \mathcal{S}(S,T)$.*

Any useful specifunction space must be designed to avoid being miraculous; otherwise an established component P cannot play any role in the weakest specifunction refinement of target specification Q. From our reasoning above we have:

Proposition 5 *Specification space* $\mathcal{C}onst$ *is miraculous.*

Another example is provided by the specification space formed by specifunctions which sandwich their argument in a local block. Let us write declaration of local variable r in block A (in which r is not free) as usual as

$$\mathbf{var}\ r \cdot A.$$

If E is a program and r is a variable not free in E then E_{+r} denotes (c.f. [13]) the program whose signature equals that of E extended to include r, and which behaves like E but with variable r unchanged. Now we can define

$$\mathcal{S}wp(Seq) \ \hat{=}\ (\{(\lambda E : Seq(x) \cdot \mathbf{var}\ r \cdot A \,⨾\, E_{+r} \,⨾\, B) \mid A, B \in Seq(x{+}r)\}, \sqsubseteq).$$

There $x{+}r$ denotes the signature consisting of the two variables x and r.

Theorem 5 $\mathcal{S}wp$ *is miraculous.*

Proof. Let $\mathcal{F}$ be the subset of $\mathcal{S}wp$ defined by

$$\mathcal{F} \ \hat{=}\ \{f_{A,B} \mid A \in Seq(x),\ B \in Seq(x{+}r{+}s{+}t)\},$$

where

$$\begin{aligned} f_{A,B} \ \hat{=}\ (\lambda E : Seq(x) \cdot \mathbf{var}\ & r, s, t \cdot \\ & s := x \,⨾\, A_{+r+s+t} \,⨾\, t := x ⨾ \\ & E_{+r+s+t} ⨾ \\ & B). \end{aligned}$$

We reason that $\mathcal{F}$ is a specifunction space. Firstly it is closed. If $\mathcal{G}$ is any non-empty subset of $\mathcal{F}$, for its elements $g \,\hat{=}\, g_{A,B}$ we write A as A_g and B as B_g. Then the glb of $\mathcal{G}$ is the specifunction

$$\begin{aligned} \sqcap\mathcal{G} \ \hat{=}\ (\lambda E : Seq(x) \ \cdot \mathbf{var}\ & r, s, t \cdot \\ & s := x \,⨾\, (\sqcap_{g\in\mathcal{G}}\ A_g)_{+r+s+t} \,⨾\, t := x ⨾ \\ & E_{+r+s+t} ⨾ \\ & \sqcap_{g\in\mathcal{G}}\ (\ A_g[x, x' := s, t]\ )_{+x} \,⨾\, B_g) \end{aligned}$$

where the coercion $(x = t)_\top$ was defined in section 3.1. Thus $\sqcap \mathcal{G} \in \mathcal{F}$ and $\mathcal{F}$ is closed.

Next $\mathcal{F}$ is topped, because

$$(\lambda E\colon Seq(x) \cdot \top_{Seq(x)}) \;=\; (\lambda E\colon Seq(x) \cdot \mathbf{var}\ r\ \cdot \top_{Seq(x+r)} \mathbin{;} E_{+r} \mathbin{;} B)$$

is a specifunction in $\mathcal{F}$. Thus, being both closed and topped, $\mathcal{F}$ is a specifunction space.

Now we argue that $\mathcal{S}wp = \mathcal{F}$. For an arbitrary specifunction $s_{C,D} : \mathcal{S}wp$, with $C, D \in Seq(x{+}r)$, writing

$$s_{C,D} = (\lambda E\colon Seq(x) \cdot \mathbf{var}\ r\ \cdot C \mathbin{;} E_{+r} \mathbin{;} D),$$

we have

$$\begin{aligned} s_{C,D} \;\hat{=}\; (\lambda E\colon Seq(x) \cdot (\,\mathbf{var}\ r, s, t \cdot \\ & s := x \mathbin{;} (\,\mathbf{var}\ r\ \cdot C\,)_{+r+s+t} \mathbin{;} t := x \mathbin{;} \\ & E_{+r+s+t} \mathbin{;} \\ & (\,C[x, x' := s, t]\,)_{+x} \mathbin{;} D_{+s+t}) \end{aligned}$$

Thus $s_{C,D} \in \mathcal{F}$. Since $s_{C,D}$ was arbitrary, we have $\mathcal{F} = \mathcal{S}wp$ as claimed.

Finally for any non-magic $Q, P : Seq(x)$, we have

$$const_Q(P) \;=\; \mathbf{var}\ r\ \cdot (r' = x \mathbin{;} P_{+r} \mathbin{;} x := r \mathbin{;} Q_{+r})\,.$$

Since Q and P were arbitrary, $\mathcal{S}wp$ is miraculous. □

2.6 A general framework for refinement

The formal-methods top-down view of program derivation is to model specifications as mathematical objects in a refinement order and to provide laws for the transformation of an original (abstract) specification to more detailed designs until finally the stage of an executable program is reached.

However we may not be able to formulate a full refinement at the early stages of program development. Instead, we may start from an abstract specification of a component and incrementally build up a more detailed full refinement. Such program refinement combines both 'top-down' and 'bottom-up' views, and corresponds to the complementary programming activities in practice. Such an approach may be termed *compositional refinement.*

The calculus of weakest specfunction can be used to formalise compositional refinement in which we start from an established component P and try to build a more complicated component Q from P. The following law decomposes such program development activity into two parts by introducing an intermediate component R. The first sub-task then becomes building R from P, and the second sub-task becomes building Q from R.

Law 5 *If* $\mathcal{H} \supseteq \mathcal{F} \circ \mathcal{G}$, *then* $Q \div_{\mathcal{H}} P \;\sqsubseteq\; (Q \div_{\mathcal{F}} R) \circ (R \div_{\mathcal{G}} P)$.

This law generalises the 'sequential composition' law in [16] where an intermediate condition, instead of an intermediate specification, is introduced. Various ways of adding new components are allowed by choice of designs of $\mathcal{F}$ and $\mathcal{G}$.

If $\mathcal{F} \circ \mathcal{G}$ is also a specifunction space, the refinement becomes an equation. This leads to the observation that if R is properly chosen in the decomposition, it becomes completely 'safe' in the sense that no feasibility is lost.

Law 6 $Q \div_{\mathcal{F}\circ\mathcal{G}} P \;=\; \sqcap_R (Q \div_{\mathcal{F}} R) \circ (R \div_{\mathcal{G}} P)$.

Proof. Law 5 establishes the '$\sqsubseteq$' direction.

Conversely, suppose $\mathcal{F} : \textsc{Spec}(T, U)$ and $\mathcal{G} : \textsc{Spec}(S, T)$ so that $\mathcal{F} \circ \mathcal{G} : \textsc{Spec}(S, U)$. For any $P : S$ and $Q : U$ suppose that $f : \mathcal{F}$ and $g : \mathcal{G}$ satisfy $f \circ g = Q \div_{\mathcal{F}\circ\mathcal{G}} P$. Then:

$$
\begin{array}{ll}
\sqcap_{R \in T} (Q \div_{\mathcal{F}} R) \circ (R \div_{\mathcal{G}} P) & \\
\sqsubseteq & \text{taking } g(P) : T \text{ for } R \\
(Q \div_{\mathcal{F}} g(P)) \circ (g(P) \div_{\mathcal{G}} P) & \\
\sqsubseteq & \text{any } g \in \mathcal{G} \text{ satisfies } g(P) \sqsubseteq g(P) \\
(Q \div_{\mathcal{F}} g(P)) \circ g & \\
\sqsubseteq & \text{choice of } f \text{ and definition of glb} \\
f \circ g & \\
= & \text{choice of } f \text{ and } g \\
Q \div_{\mathcal{F}\circ\mathcal{G}} P. & \square
\end{array}
$$

For less-shallow results, indeed for laws concerning weakest specifunctions which are useful in program derivation, we must assume more about the underlying specification spaces. In general, application of the theory involves four steps:

1. definition of the underlying semantic lattice

2. design of a useful set of specifunctions
3. proof that the set of specifunctions forms a nonmiraculous specifunction space
4. discovery of enough laws to make weakest specifunction calculation practical.

In the remainder of the paper we continue the study of specifunctions in the important case of a language which contains both sequential and parallel composition. Since derivations are calculations we follow the usual approach of choosing, in tackling the first step above, a predicate-based semantics.

3 Weakest specifunctions for *Logs*

In this section we use the language of global synchrony, *Logs* [6], as a vehicle to demonstrate use of weakest specifunctions. After providing laws for calculating the weakest specifunctions of the spaces $\mathcal{P}re$, $\mathcal{P}ost$ and $\mathcal{P}ar$ of section 2.2, we concentrate on a new specifunction which combines those behaviours: the par-seq specifunction.

3.1 *Logs* summarised

Global synchronisation is one of the features of PRAM/BSP programming. However its benefits come at a cost: global *puts* and *gets* are expensive operations and so must be deployed carefully. In deriving PRAM/BSP programs it is thus convenient to pass through a level of abstraction in which global operations have been abstracted and all that remains is the trace of intermediate states achieved at the step (or synchronisation point) boundaries. For an example of the use of that abstraction see the treatment of the dining philosophers in [6]; for its relationship to BSP code see [5].

From such a point of view a computation is described as a predicate whose observables consist of an initial state $\overleftarrow{w}$, a sequence $\overline{w}$ of intermediate states and a final state $\overrightarrow{w}$. If the computation terminates then $\overline{w}$ is finite; otherwise — for instance in a reactive computation — $\overline{w}$ is infinite and $\overrightarrow{w}$ is unconstrained. When convenient we write ws for the state sequence $\overleftarrow{w}+\overline{w}+\overrightarrow{w}$. A

$\langle\, q \,\rangle_n$	n-step specification
$\mathbf{var}\ x \cdot P$	variable declaration
$P \mathbin{;} Q$	sequencing
$P \sqcap Q$	(demonic) nondeterminism
$P \| Q$	parallelism.

Figure 1: Basic commands of language *Logs*

summary of pertinent notation is:

$\overleftarrow{w}$	initial state
$\overline{w}$	sequence of intermediate states
$\overrightarrow{w}$	final state, if $\overline{w}$ is finite
ws	sequence $\overleftarrow{w} + \overline{w} + \overrightarrow{w}$
$ws_{<n}$	sequence $\overleftarrow{w} + w_0 + \ \cdots\ + w_{n-1}$
$ws_{i,j}$	$w_i + \ \cdots\ + w_{j-1} + \overrightarrow{w}$.

Any input and output are thought of, as usual, as being incorporated in state. Thus q describes how computation progresses by recording the step-by-step relationship between the states.

If predicate q describes an n-step computation, for $n : \mathbb{N} \cup \{\infty\}$, we write $\langle\, q \,\rangle_n$. Thus $\langle\, q \,\rangle_0$ means that computation q contains no synchronisations, whilst $\langle\, q \,\rangle_1$ means that it contains exactly one synchronisation and $\langle\, q \,\rangle_\infty$ describes a reactive computation with no final state. Substitution of e for w in predicate p is denoted $p[w := e]$ and a pair of multiple substitutions is written $p[v, w := e, f]$.

Glb corresonds to nondeterminism as in the guarded-command language and so refinement is defined by equivalence (6). However now lub corresponds to parallel composition.

The syntax of language *Logs* is given in figure 1. Derived commands are given in figure 2 and explained as follows. No-op II (or skip) is a 0-step assignment which leaves state unchanged. The most nondeterministic computation $\perp$ (or abort) could take any number of steps, behaving arbitrarily at each. The most refined computation $\top$ (or magic) achieves the strongest specification *false* over all observables. A coercion of predicate b over states is skip if the predicate holds but abort otherwise. An assumption of predicate

$I\!I$	$\langle\, \overleftarrow{w} = \overrightarrow{w} \,\rangle_0$	skip
$\bot$	$\sqcap_{n \leq \infty} \langle\, \mathit{true} \,\rangle_n$	abort
$\top$	false	magic
$b_\top$	$\langle\, b[w := \overleftarrow{w}] \,\rangle_0$	assumption
$\mathbf{if}\ []_i\ b_i \to P_i\ \mathbf{fi}$	$\sqcap_i\ (b_i)_\top \mathbin{;} P_i$	conditional
$b_\bot$	$\mathbf{if}\ b \to I\!I\ []\ \neg b \to \bot\ \mathbf{fi}$	coercion
$\mathbf{do}\ []_i\ b_i \to P_i\ \mathbf{od}$	$\sqcap_n (\mathbf{if}\ []_i\ b_i \to P_i\ \mathbf{fi})^n \mathbin{;} (\neg \vee_i\ b_i)_\top$	iteration
$[\![\, q \,]\!]$	$\sqcap_{0 \leq n < \infty} \langle\, q \,\rangle_n$	liveness property
$\vert\, q \,\vert$	$\sqcap_{0 \leq n \leq \infty} \langle\, q \,\rangle_0^n$	safety property

Figure 2: The derived commands of language *Logs*, by syntax, semantics and name.

b is skip if the predicate holds but magic otherwise. A guarded command $b \to P$ behaves like P if b holds but otherwise aborts. Conditionals and iterations of guarded commands are defined as usual, the latter in terms of recursion in the standard way:

$$\mu X \cdot \mathbf{if}\ []_i b_i \to P_i \mathbin{;} X\ [] \neg \vee_i\ b_i \to I\!I\ \mathbf{fi}$$

or in terms of iteration

$$\begin{aligned} P^0 &\mathrel{\hat=} I\!I \\ P^{n+1} &\mathrel{\hat=} P \mathbin{;} P^n \end{aligned}$$

as recorded in figure 2.

Abstraction from *puts* and *gets* places emphasis on temporal properties. Suppose that q is an n-step computation. The liveness property generated by q is the property which ensures that q eventually terminates: it is established after some finite number of steps. The safety property of q is the property which ensures that q remains invariant throughout its intermediate steps.

Computations with different numbers of steps when placed in parallel abort. However the parallel composition of two n-step computations is again an n-step computation described by the conjunction of the predicates describing the two computations. In general a *Logs* command may consist of an infinite number of terms:

$$\sqcap_{0 \leq i \leq \infty} \langle\, q_i \,\rangle_i.$$

There, by the law just mentioned for parallel compositions, it can be assumed that for each n there is exactly one predicate q_n taking n steps. Of course it could be that $q_i = \mathit{false}$. In fact that is normal form for arbitrary *Logs* commands and any finite *Logs* command may be reduced to $\sqcap_{n \leq N} \langle\, q_n \,\rangle_n$.

A complete set of laws, and reduction to normal form, is given in [6]. We use that result, in the case of finite computation, in section 3.6. However note that here we have avoided introducing the dual of a computation, although it arises implicitly in section 3.6.

3.2 Previous specifunction spaces for *Logs*

The specifunction spaces $\mathcal{P}re$ and $\mathcal{P}ost$ are defined by substituting *Logs* for the specification space *Seq* of section 2.2. Similarly $\mathcal{P}ar$ is defined by replacing specification space *Par* with *Logs*. We adopt the convention that the (implicit) signature is w.

Two simple laws, for 1-step computations, are immediate. In stating them we repeat the standard notation [11] for weakest pre and post specifications.

Law 7

$$\begin{array}{rcl}
\langle\, q \,\rangle_1 \div_{\mathcal{P}re} \langle\, p \,\rangle_0 & \hat{=} & \langle\, q \,\rangle_1 \,/\, \langle\, p \,\rangle_0 \\
& = & (\lambda E : \mathit{Logs} \cdot \langle\, \forall \overline{w} \cdot \;\; p[\overleftarrow{w}, \overrightarrow{w} := \overrightarrow{w}, \overline{w}] \\
& & \qquad\qquad\qquad\qquad \Rightarrow \\
& & \qquad\qquad\qquad\qquad q[ws := \overleftarrow{w} + w + \overline{w}] \,\rangle_1 \,\mathbin{;}\, E) \\
\langle\, q \,\rangle_1 \div_{\mathcal{P}ost} \langle\, p \,\rangle_1 & \hat{=} & \langle\, q \,\rangle_1 \setminus \langle\, p \,\rangle_1 \\
& = & (\lambda E : \mathit{Logs} \cdot E \mathbin{;} (\forall w \forall \overline{w} \cdot \;\; p[ws := \overline{w} + w + \overleftarrow{w}] \\
& & \qquad\qquad\qquad\qquad\qquad \Rightarrow \\
& & \qquad\qquad\qquad\qquad\qquad q[ws := \overline{w} + w + \overrightarrow{w}]))
\end{array}$$

In the analogous law for parallelism involving two 1-step computations, we repeat the notation [14] for the weakest parallel environment.

Law 8

$$\begin{array}{rcl}
\langle\, q \,\rangle_1 \div_{\mathcal{P}ar} \langle\, p \,\rangle_1 & \hat{=} & \langle\, q \,\rangle_1 /\!\!/ \langle\, p \,\rangle_1 \\
& = & (\lambda E : \mathit{Logs} \cdot E \parallel (\,\langle\, p \Rightarrow q \,\rangle_1 \sqcap (\sqcap_{n \neq 1} \langle\, \mathit{true} \,\rangle_n)\,))
\end{array}$$

Evidently when the numbers of steps are not equal the weakest specifunction becomes the established component itself.

3.3 The par-seq specifunction space

We see from the discussion above the significance, for derivation in a parallel language like *Logs*, of incorporating *simultaneously* both sequential and parallel refinement. We now show how to achieve it.

We start, incautiously, by observing that one obvious specifunction set exhibiting both sequential and parallel components is

$$\mathcal{P}ost \circ \mathcal{P}ar \ \hat{=} \ \{(\lambda E\!:\!Logs \cdot (E \parallel A) \fatsemi B) \mid A, B \in Logs\}.$$

Specifunctions in that space are conveniently written using a patently two-dimensional notation:

Definition 11 *For any $A, B\!:\!Logs$,*

$$\frac{\;|\,A}{B} \ \hat{=} \ (\lambda E\!:\!Logs \cdot (E \parallel A) \fatsemi B).$$

The result is readily confirmed to be a specifunction. In it A is called the parallel partner *and B the* sequent*.*

However the specifunction set $\mathcal{P}ost \circ \mathcal{P}ar$ is not closed; proof of that fact resembles the proof of theorem 2 and is omitted.

Theorem 6 *$\mathcal{P}ost \circ \mathcal{P}ar$ is not closed.*

To resolve that difficulty we introduce a temporary variable by applying the specifunction:

$$(\lambda E\!:\!Logs \cdot \mathbf{var}\ h\ \cdot E)$$

to $\mathcal{P}ost(w{+}h) \circ \mathcal{P}ar(w{+}h)$. With it we define a set of specifunctions which turns out to be a specifunction space.

Definition 12

$$\begin{aligned}\mathcal{P}arseq \ &\hat{=} \ \{\mathbf{var}\ h\ \cdot \frac{\;|\,A}{B} \ \mid \ A, B \in Logs(w{+}h)\} \\ &= \ \{\lambda E\!:\!\mathcal{L}ogs \cdot (\mathbf{var}\ h\ \cdot (E \parallel A) \fatsemi B) \ \mid \\ &\qquad\qquad A, B \in Logs(w{+}h)\}\,.\end{aligned}$$

Again each of those functions is readily shown to be a specifunction. For convenience we henceforth write $Q \div_{\mathcal{P}arseq} P$ as $Q \div P$.

Proof that $\mathcal{P}arseq$ is closed uses the fundamental computational notion of accumulation. To state it we define a restricted form of par-seq specifunction whose use will endure far beyond this theorem.

Definition 13 *For $A : Logs(w)$ and $B : Logs(w{+}h)$, the par-seq specifunction is defined as follows:*

$$\frac{\Diamond \,\big|\, A}{B} \;\hat{=}\; \textbf{var } h \; \frac{\big|\, A \| H}{B}$$

where h is a fresh variable of sequence type used by computation H, defined below, to accumulate the intermediate states of A. Assume that A is an n-step computation. Recalling the shorthand $ws_{<n}$ from section 3.1, we define H to ensure that its final value for h records the history of the n-step computation A:

$$H \;\hat{=}\; \sqcap_{n \in \mathbb{N}} \langle\, \overrightarrow{h} = ws_{<n} \,\rangle_n .$$

Thus the final value of h is a sequence of length two greater than the number of steps in A.

Theorem 7 *For any $A, B : \mathcal{L}ogs$ the function $\dfrac{\Diamond \,\big|\, A}{B}$ is indeed a specifunction and it acts on the non-miraculous specifunction space $\mathcal{P}arseq$.*

Proof. To establish that $\dfrac{\Diamond \,\big|\, A}{B}$ is a specifunction we must show that for any subset $\mathcal{E}$ of $\mathcal{P}arseq$,

$$\frac{\Diamond \,\big|\, A}{B}\,(\sqcap \mathcal{E}\,) \;=\; \sqcap \left\{ \frac{\Diamond \,\big|\, A}{B}\,(E) \mid E \in \mathcal{E} \right\} .$$

But

$$\frac{\Diamond \,\big|\, A}{B}\,(\sqcap \mathcal{E})$$

$=$ definition of the par-seq function

$\textbf{var } h \cdot (A \,\|\, H \,\|\, \sqcap \mathcal{E}) \,⨾\, B$

$=$ law of *Logs*

$\textbf{var } h \cdot (\sqcap\{(A \,\|\, H \,\|\, E) \;\mid\; E \in \mathcal{E}\}) \,⨾\, B$

$=$ law of *Logs*

$\mathbf{var}\ h\ \cdot \sqcap\{(A \parallel H \parallel E) \mathbin{;} B \ \mid\ E \in \mathcal{E}\}$

$=$ law of *Logs*

$\sqcap\{\mathbf{var}\ h\ \cdot (A \parallel H \parallel E) \mathbin{;} B \ \mid\ E \in \mathcal{E}\}$

$=$ definition of the par-seq function

$$\sqcap \left\{ \frac{\Diamond \mid A}{B} (E) \ \mid\ E \in \mathcal{E} \right\} .$$

Next, to establish that $\mathcal{P}arseq$ is a specifunction space, consider the set of specifunctions

$$\mathcal{D} \ \hat{=}\ \left\{ \frac{\Diamond \mid A}{B} \ \middle|\ A \in Logs(w), B \in Logs(w{+}h) \right\} .$$

Firstly $\mathcal{D}$ is closed since for any non-empty subset $\mathcal{G}$ of $\mathcal{D}$, say

$$\mathcal{G} \ \hat{=}\ \left\{ \frac{\Diamond \mid A_i}{B_i} \ \mid i \in I \right\},$$

calculation shows that

$$\sqcap\mathcal{G} \ =\ \frac{\Diamond \mid \sqcap\{A_i \mid i \in I\}}{\mathbf{if}\ []_i\ A_i[ws := h] \rightarrow B_i\ \mathbf{fi}} .$$

Secondly $\mathcal{D}$ in fact equals $\mathcal{P}arseq$, because any specifunction $\left(\mathbf{var}\ h\ \frac{\mid A}{B}\right)$ has an equal specifunction in $\mathcal{D}$, namely

$$\frac{\Diamond \mid \mathbf{var}\ h\ \cdot A}{C \mathbin{;} B}$$

where C is a zero-step *Logs* computation specified by remembering that variable h itself has an initial value, intermediate states as it records those of A, as well as the final value mentioned above:

$$C \ \hat{=}\ \langle\, \exists hs_{<m} \cdot A[ws := \overleftarrow{h}\,] \wedge \overleftarrow{w} = \overrightarrow{w}\, \rangle_0 .$$

The idea is to delay (or hide) the calculation of $\overrightarrow{h}$ in A and at the same time use $\overrightarrow{h}$ to record the global state at each step. From the information recorded in sequence $\overleftarrow{h}$ by H, program C can restore A's original $\overrightarrow{h}$.

The fact that $\mathcal{P}arseq$ is closed now follows immediately. In fact $\mathcal{P}arseq$ is also topped (let A be $\top$) and hence is a specifunction space.

To complete the proof we observe that if $n < m$ and $q \neq \mathit{false}$ (by assumption of comparison of specifunction spaces) then

$$(\langle\, q\,\rangle_n \div \langle\, p\,\rangle_m)(\langle\, p\,\rangle_m) = \top \neq \langle\, q\,\rangle_n\,. \qquad \square$$

Example Let us calculate, by way of example, $\top \div P$ where P is a finite computation in normal form. We claim

$$\top \div \sqcap_i \langle\, p_i\,\rangle_i \quad = \quad \frac{\diamond \,\big|\, \sqcap_i \langle\, \neg p_i\,\rangle_i}{\bot}\,.$$

Firstly we verify that the right-hand side is indeed a solution $f : \mathcal{L}ogs$ of the refinement $f(\sqcap_i \langle\, p_i\,\rangle_i) \sqsupseteq \top$ and secondly we prove that it is the weakest such refinement. For the first we calculate:

$\dfrac{\diamond \,\big|\, \sqcap_i \langle\, \neg p_i\,\rangle_i}{\bot}\,(\sqcap_i \langle\, p_i\,\rangle_i)$

$=$ definition of the par-seq specifunction

$\mathbf{var}\ h \cdot (H \,\|\, \sqcap_i \langle\, \neg p_i\,\rangle_i \,\|\, \sqcap_i \langle\, p_i\,\rangle_i) \mathbin{;} \bot$

$=$ law of *Logs*

$\mathbf{var}\ h \cdot (H \,\|\, \sqcap_i \langle\, \mathit{false}\,\rangle_i) \mathbin{;} \bot$

$=$ law of *Logs*

$\mathbf{var}\ h \cdot \top \mathbin{;} \bot$

$=$ law of *Logs*

$\top\,.$

Secondly we establish that if $\dfrac{\diamond \,|\, A}{B}\,(\sqcap_i \langle\, p_i\,\rangle_i) \sqsupseteq \top$ then for all $E : \mathit{Logs}$,

$$\frac{\diamond \,|\, A}{B}\,(E) \quad \sqsupseteq \quad \frac{\diamond \,\big|\, \sqcap_i \langle\, \neg p_i\,\rangle_i}{\bot}\,(E)\,.$$

The technique we use prefigures that used in proving law 15. For any $E : \mathit{Logs}$, if $E \sqsupseteq P$ then

$$\mathit{LHS} \ \sqsupseteq\ \frac{\diamond \,|\, A}{B}\,(P) \ =\ \top \ =\ \mathit{RHS}\,,$$

whilst if $E \,\|\, P = \top$ we observe that $E \sqsupseteq \sqcap_i \langle\, \neg p_i\,\rangle_i$ and hence

$$\frac{\Diamond \mid A}{B}\,(E)$$

$\sqsupseteq$ monotonicity

$$\frac{\Diamond \mid \bot}{B}\,(E)$$

$=$ observation above

$$\frac{\Diamond \mid \sqcap_i \langle \neg p_i \rangle_i}{B}\,(E)$$

$\sqsupseteq$ monotonicity

$$\frac{\Diamond \mid \sqcap_i \langle \neg p_i \rangle_i}{\bot}\,(E)\,.$$

Otherwise let $E_1 \ \hat{=}\ E \| \sqcap_i \langle p_i \rangle_i$ and $E_2 \ \hat{=}\ E \| \sqcap_i \langle \neg p_i \rangle_i$ and apply the reasoning in the proof to follow of law 15. □

The following theorem confirms the intuition that $\div_{\mathcal{P}arseq}$ is at least as good as both $\div_{\mathcal{P}ost}$ and $\div_{\mathcal{P}ar}$.

Theorem 8

1. $\mathcal{P}arseq \sqsubseteq \mathcal{P}ost$.

2. $\mathcal{P}arseq \sqsubseteq \mathcal{P}ar$.

Proof. For the first we observe that each specifunction $(\lambda E : Logs \cdot E \,;\, A)$ can be also written as $(\lambda E : Logs \cdot \mathbf{var}\ h \cdot (E \| H \| \bot) \,;\, A)$.

For the second, each specifunction $(\lambda E : Logs \cdot E \parallel A)$ can be written as $(\lambda E : Logs \cdot \mathbf{var}\ h \cdot (E \| H \| A) \,;\, II)$. □

Theorem 9 *The specifunction spaces $\mathcal{P}re$, $\mathcal{P}ost$, $\mathcal{P}ar$ and $\mathcal{L}ogs$ are idempotent.*

Proof. Only the idempotence of $\mathcal{L}ogs$ is non-trivial. But

$\mathcal{L}ogs$

$\supseteq$ $II \in \mathcal{L}ogs$

$\mathcal{L}ogs \circ \mathcal{L}ogs$

$\supseteq$

$$\left(\mathbf{var}\ h \frac{\rfloor \perp}{II}\right) \circ \mathcal{L}ogs$$

$=$ definitions

$\mathcal{L}ogs.$ $\square$

3.4 Temporal laws for $\mathcal{L}ogs$

The laws of this and the next section are generally useful and indeed will find application in the example of section 4. Each is a special case of law 15 to be proved in section 3.6.

Recall that $\mid q \mid$ denotes the safety property consisting of a one-step *Logs* command q repeated arbitrarily many times; it may, of course, not terminate. For such commands there is a simple law:

Law 9 $\quad \mid q \mid \div \mid p \mid \;=\; \dfrac{\rfloor \mid p \Rightarrow q \mid}{\mid q \mid}.$

A program that eventually terminates satisfies a liveness property $[\![q]\!]$ in which predicate q relates initial and final states. Recalling the simplified notation $P \setminus Q$ for $P \div_{\mathcal{P}ost} Q$ we have:

Law 10 $\quad [\![q]\!] \div [\![p]\!] \;\sqsubseteq\; \dfrac{\rfloor [\![r]\!]}{[\![q \setminus (p \wedge r)]\!]}$. *If $p \wedge r$ is a deterministic binary relation (i.e. a total function) then the refinement becomes equality.*

The following law shows how to split a computation into a conjunction of safety and liveness properties:

Law 11 *If q is idempotent (i.e. $q \fatsemi q = q$) then*

$$(\mid q \mid \wedge [\![r]\!]) \;\div\; (\mid p \mid \wedge [\![s]\!]) \;\sqsubseteq\; \frac{\rfloor \mid p \Rightarrow q \mid}{\mid q \mid \wedge [\![r \setminus (s \wedge q)]\!]}.$$

3.5 Elementary laws for $\mathcal{L}ogs$

The laws of this section help to motivate law 15, proved in the next section, of which they are a special case.

We start from the simple case in which commands $Q = \langle q \rangle_1$ and $P = \langle p \rangle_1$ are both one-step commands of *Logs*. The first law yields the parallel partner when the sequent is trivial (and follows by law 8).

Law 12 *If* $q \Rightarrow p$ *then* $\langle q \rangle_1 \div \langle p \rangle_1 \sqsubseteq \dfrac{|\langle p \Rightarrow q \rangle_1}{\langle \overleftarrow{w} = \overrightarrow{w} \rangle_0}$.

The second law yields the sequent when the parallel partner is evident (employing the notation for variable h from definition 13):

Law 13 *If* $p \Rightarrow \exists \overrightarrow{w} \cdot q$ *then*

$$\langle q \rangle_1 \div \langle p \rangle_1 \sqsubseteq \textbf{var } h \frac{|\langle ws = \overrightarrow{h} \rangle_1}{\langle p[ws := \overleftarrow{h} + \overleftarrow{w}] \Rightarrow q[ws := \overleftarrow{h} + \overrightarrow{w}] \rangle_0} .$$

The next law shows how to decompose a par-seq specifunction of one-step commands into two parts to which the previous two laws apply.

Law 14 $\langle q \rangle_1 \div \langle p \rangle_1 \sqsubseteq (\langle q \rangle_1 \div \langle p \wedge \exists \overrightarrow{w} \cdot q \rangle_1) \circ (\langle p \wedge \exists \overrightarrow{w} \cdot q \rangle_1 \div \langle p \rangle_1)$.

3.6 A complete law for $\mathcal{L}ogs$

For any target *Logs* specification Q and established component P, we need to be able to identify *Logs* commands $par = par(Q, P)$ (the parallel partner) and $seq = seq(Q, P)$ (the sequent) for which

$$Q \div P = \frac{\Diamond \,|\, par}{seq} .$$

In this section we provide a single law to achieve that, and show in theorem 10 that it does so.

Given two *Logs* specifications P and Q, we write them in normal form: $P = \sqcap_i \langle p_i \rangle_i$ and $Q = \sqcap_i \langle q_i \rangle_i$. As already noted in section 3.1 either p_i or q_i may be the predicate *false*.

The following law calculates the weakest par-seq specifunction of two specifications Q and P in normal form, provided that the target Q is not magic. Indeed that case, not useful in practice, has already been covered by the example in section 3.3.

Recall the shorthand $ws_{i,j}$ from section 3.1.

Law 15 *If* $P = \sqcap_i \langle p_i \rangle_i$ *and* $Q = \sqcap_i \langle q_i \rangle_i$ *then* $Q \div P = \dfrac{\Diamond \,|\, A}{(b)_\perp \,⨾\, B}$ *where*

$$
\begin{array}{rcl}
A & \hat{=} & \sqcap_i \langle\, p_i \Rightarrow r_i \,\rangle_i \\
r_i & \hat{=} & \vee_{j \geq i} r_{ij} \\
r_{ij} & \hat{=} & \exists ws_{i,j} \cdot q_j \\
b & \hat{=} & \vee_i (p_i \wedge r_i)[ws_{<i}, \overrightarrow{w} := h, w]
\end{array}
$$

$$
\begin{array}{rcl}
B & \hat{=} & \textbf{if } []_i (\# = i) \to B_i \textbf{ fi} \\
B_i & \hat{=} & \textbf{if } []_{j \geq i} (r_{ij}[ws_{<j} := h]) \to B_{ij} \textbf{ fi} \\
B_{ij} & \hat{=} & \left\langle\, q_j[ws_{<i}, ws_{i,j} := \overleftarrow{h}, ws_{0,j-i}] \,\right\rangle_{j-i} .
\end{array}
$$

For convenience, we write

$$
f^* \;\hat{=}\; Q \div P \quad \text{and} \quad f \;\hat{=}\; \frac{\Diamond \mid A}{(b)_\perp \,;\, B} .
$$

Lemma 1 $\quad f^* \sqsubseteq f$.

Proof. Since evidently $f : \mathcal{L}ogs$ it suffices to check $Q \sqsubseteq f(P)$; for f^* is the weakest such f and so the desired refinement holds. We reason as follows:

$$
\begin{array}{ll}
f(P) & \\
= & \text{definitions of } f,\ P \text{ and } A \\
\textbf{var } h \cdot (\sqcap_i \langle\, p_i \,\rangle_i \parallel \sqcap_i \langle\, p_i \Rightarrow r_i \,\rangle_i \parallel H) \,;\, (b)_\perp \,;\, B & \\
= & \text{logic and law of } \mathit{Logs} \\
\textbf{var } h \cdot (\sqcap_i \langle\, p_i \wedge r_i \,\rangle_i \parallel H) \,;\, (b)_\perp \,;\, B & \\
= & \text{law of } \mathit{Logs} \\
\sqcap_i \textbf{ var } h \cdot (\langle\, p_i \wedge r_i \,\rangle_i \parallel H) \,;\, (b)_\perp \,;\, B & \\
= & \text{definition of } H \\
\sqcap_i \textbf{ var } h \cdot \left\langle\, p_i \wedge r_i \wedge \overrightarrow{h} = ws_{<i} \,\right\rangle_i \,;\, B & \\
= & \text{definition of } B_i \\
\sqcap_i \textbf{ var } h \cdot \left\langle\, p_i \wedge r_i \wedge \overrightarrow{h} = ws_{<i} \,\right\rangle_i \,;\, B_i & \\
= & \text{logic and definition of } B_{ij} \\
\sqcap_i \textbf{ var } h \cdot \sqcap_{j \geq i} \left\langle\, p_i \wedge r_i \wedge \overrightarrow{h} = ws_{<i} \,\right\rangle_i \,;\, (r_{ij}[ws_{<i} := h])_\top \,;\, B_{ij} & \\
\sqsupseteq & \text{logic}
\end{array}
$$

$$\sqcap_i \sqcap_{j\geq i} \ \mathbf{var}\ h \cdot \left\langle r_{ij} \wedge \overrightarrow{h} = ws_{<i} \right\rangle_i \fatsemi \left\langle q_j[ws_{<i}, ws_{i,j} := \overleftarrow{h}, ws_{0,j-i}] \right\rangle_{j-i}$$

$\sqsupseteq$ logic

$\sqcap_i \sqcap_{j\geq i} \ \mathbf{var}\ h \cdot \langle q_j \rangle_j$

$=$ logic

$\mathbf{var}\ h \cdot Q$

$=$ law of *Logs*

$Q\,.$ □

Lemma 2 *If $f : \mathcal{L}ogs$ and $E : Logs$ then $f(E) \sqsupseteq E \fatsemi \bot$. In particular if $E = \sqcap_i \langle e_i \rangle_i$ then $f(E) \sqsupseteq \sqcap_i \sqcap_{j\geq i} \langle \exists \overrightarrow{w} \cdot e_i \rangle_j$.*

Proof. For any $E : Logs$ we choose to reason:

$f(E)$

$=$ definition of f

$$\frac{\Diamond \mid A}{(b)_\bot \fatsemi B}\,(E)$$

$=$ definition of the par-seq specifunction

$\mathbf{var}\ h \cdot (E \parallel A \parallel H) \fatsemi (b)_\bot \fatsemi B$

$\sqsupseteq$ monotonicity of the *Logs* operators and laws of *Logs*

$E \fatsemi \bot\,.$

Finally it remains to observe that in the case $E = \sqcap_i \langle e_i \rangle_i$ we have

$$E \fatsemi \bot \ = \ \sqcap_i \sqcap_{j\geq i} \langle \exists \overrightarrow{w} \cdot e_i \rangle_j\ .$$ □

Lemma 3 *If $E : Logs$ with $E \sqsupseteq P$ then*

$$Q \sqsubseteq f(E) \sqsubseteq f^*(E)\,.$$

Proof. Consider an arbitrary specifunction $f' : \mathcal{L}ogs$ for which $f'(P) \sqsupseteq Q$. For the first refinement we reason:

$$
\begin{array}{ll}
f'(E) & \\
\sqsupseteq & \text{assumption on } E \text{ and monotonicity of } f' \\
f'(P) & \\
\sqsupseteq & \text{assumption on } f' \\
Q &
\end{array}
$$

from which the result follows by taking $f' \;\widehat{=}\; f$.

Furthermore by Lemma 2, $f'(E) \sqsupseteq (E \,{}_{9}^{\circ}\, \bot)$ and since parallel composition is the lub in *Logs*, we conclude $f'(E) \sqsupseteq (E \,{}_{9}^{\circ}\, \bot) \,\|\, Q$.

For the second refinement we reason

$$
\begin{array}{ll}
f(E) & \\
= & \text{definition of } f \\
\mathbf{var}\ h \cdot (E \,\|\, A \,\|\, H) \,{}_{9}^{\circ}\, (b)_{\bot} \,{}_{9}^{\circ}\, B & \\
= & \| \text{ is lub in } \mathit{Logs} \\
\sqcap_i\, \mathbf{var}\ h \cdot \left\langle\, e_i \wedge (p_i \Rightarrow r_i) \wedge \overrightarrow{h} = ws_{<i} \,\right\rangle_i \,{}_{9}^{\circ}\, (b)_{\bot} \,{}_{9}^{\circ}\, B & \\
= & \text{reasoning as in the proof of lemma 1 in this and the next few steps} \\
\sqcap_i\, \mathbf{var}\ h \cdot \left\langle\, e_i \wedge r_i \wedge \overrightarrow{h} = ws_{<i} \,\right\rangle_i \,{}_{9}^{\circ}\, B_i & \\
= & \\
\sqcap_i \sqcap_{j \geq i}\ \mathbf{var}\ h \cdot \left\langle\, e_i \wedge r_i \wedge \overrightarrow{h} = ws_{<i} \,\right\rangle_i \,{}_{9}^{\circ}\, (r_{ij}[ws_{<i} := h])_{\top} \,{}_{9}^{\circ}\, B_{ij} & \\
= & \\
\sqcap_i \sqcap_{j \geq i}\ \mathbf{var}\ h \cdot \left\langle\, e_i \wedge r_{ij} \wedge \overrightarrow{h} = ws_{<i} \,\right\rangle_i \,{}_{9}^{\circ}\, B_{ij} & \\
= & \\
\sqcap_i \sqcap_{j \geq i}\ \langle\, \exists \overrightarrow{w} \cdot e_i \wedge q_j \,\rangle_j & \\
= & \text{law of } \mathit{Logs} \\
\sqcap_i \sqcap_{j \geq i}\ \langle\, \exists \overrightarrow{w} \cdot e_i \,\rangle_j \ \|\ \sqcap_j \langle\, q_j \,\rangle_j & \\
= & \text{logic and definitions of } E \text{ and } Q \\
(E \,{}_{9}^{\circ}\, \bot) \,\|\, Q & \\
\sqsupseteq & \text{laws of } \mathit{Logs} \\
Q\,. &
\end{array}
$$

Thus $f(E) \sqsubseteq f'(E)$. Since f' was arbitrary, we deduce $f(E) \sqsubseteq f^*(E)$. $\square$

Lemma 4 *If $E : Logs$ with $E \| P = \top$ then $f(E) \sqsubseteq f^*(E)$.*

Proof.

$$
\begin{array}{ll}
f(E) & \\
= & \text{definition of } f \\
\mathbf{var}\ h \cdot (E \,\|\, A \,\|\, H) \,;\, (b)_\perp \,;\, B & \\
= & E\|P = \top \text{ implies } E \sqsupseteq \sqcap_i \langle \neg p_i \rangle_i \sqsupseteq A \\
\mathbf{var}\ h \cdot (E \,\|\, H) \,;\, (b)_\perp \,;\, B & \\
= & b \Rightarrow \vee_i p_i, \text{ law of } logs \text{ and } b \text{ falsified by the `record' of } E \\
\mathbf{var}\ h \cdot (E \,\|\, H) \,;\, \perp \,;\, B & \\
= & B \neq \top \text{ because } Q \neq \top \\
\mathbf{var}\ h \cdot (E \,\|\, H) \,;\, \perp & \\
= & \text{definition of } \mathbf{var} \\
E \,;\, \perp & \\
\sqsubseteq & \text{lemma 2} \\
f^*(E)\,. & \square
\end{array}
$$

Proof of Law 15. We must show $f^* = f$. Since $f : \mathcal{L}ogs$ with $f(P) \sqsupseteq Q$, by lemma 3, $f^* \sqsubseteq f$.

Conversely, for $P = \sqcap_i \langle p_i \rangle_i$ write $\sim P = \sqcap_i \langle \neg p_i \rangle_i$ and decompose any $E : Logs$, into two parts:

$$
\begin{array}{lcl}
E_1 & \hat{=} & E \,\|\, P \\
E_2 & \hat{=} & E \| \sim P\,.
\end{array}
$$

Then

$$
\begin{array}{l}
E = E \,\|\, \perp = E \,\|\, (P \sqcap \sim P) = (E \,\|\, P) \sqcap (E \| \sim P) = E_1 \sqcap E_2 \\
E_1 = E \,\|\, P \sqsupseteq P \\
E_2 \,\|\, P = E \,\|\, \sim P \,\|\, P = E \,\|\, \top = \top\,.
\end{array}
$$

Now we reason

$$
\begin{array}{ll}
f(E) & \\
= & \text{above} \\
f(E_1 \sqcap E_2) & \\
= & \text{each specifunction is distributive} \\
f(E_1) \sqcap f(E_2) & \\
\sqsubseteq & \text{lemmas 3 and 4 using properties of } E_1 \text{ and } E_2 \text{ above} \\
f^*(E_1) \sqcap f^*(E_2) & \\
= & \text{distributivity of specifunction } f^* \\
f^*(E_1 \sqcap E_2) & \\
= & \text{property of } E_1 \text{ and } E_2 \text{ above} \\
f^*(E)\,. &
\end{array}
$$

Thus $f \sqsubseteq f^*$ as required. □

That single law suffices to obtain the result we seek.

Theorem 10 [*Completeness of Logs laws*] *The weakest par-seq specifunction of any pair of finite Logs commands can be calculated using* **law 15**.

Proof. Any computation of *Logs* can be reduced to normal form using the laws of *Logs*, as shown in [6]. □

4 Example: greatest-common divisor

In this section we indicate use of the foregoing calculus by deriving a *Logs* algorithm for the greatest common divisor (gcd) of a distributed array of positive integers.

4.1 Specification

Suppose that each of p processors contains a single positive integer $A[i],\ 0 \leq i < p$. It is required to find a *Logs* algorithm to calculate the gcd of the array $A[0, p)$.

We begin with a pre-post specification which requires the computation to terminate in a finite number of steps with the output $\overrightarrow{d}$ equal to the gcd of all numbers from the initial array $\overleftarrow{A}$.

$$\textsc{Spec}\ \mathbf{1} \quad \hat{=} \quad \Big\|\Big[\ \overrightarrow{d} = \mathsf{gcd}(\overleftarrow{A})\ \Big]\Big\|$$

4.2 Strengthening the specification

The well-known Euclidean algorithm for the case $p = 2$ calculates the gcd of a, b by iterating

$$\begin{array}{ll} a := (a \ \mathsf{mod}\ b) & \text{if} \quad a > b \\ b := (b \ \mathsf{mod}\ a) & \text{if} \quad b > a. \end{array}$$

But for the case $p > 2$ caution must be taken. Indeed in the following plausible program, fairness of the non-deterministic choice in the iteration is not guaranteed; thus neither is termination!

$$\|_{i \le p} \ \ \mathbf{do} \ []_{j \ne i} \, A[i] > A[j] \rightarrow A[i] := A[i] \ \mathsf{mod}\ A[j] \ \mathbf{od}$$

Since termination is a kind of liveness property law 11, which deals with safety and liveness simultaneously, will be used to avoid that error.

We retain the strategy of reducing the numbers step by step until one of them becomes the global gcd. However we do not need to elaborate the reduction algorithm: any algorithm can be used if it is safe in the sense that the global gcd is not changed during the reduction. To ensure that, we conjoin a safety condition to the specification.

$$\textsc{Spec}\ \mathbf{2} \quad \hat{=} \quad \left| \mathsf{gcd}(\overleftarrow{A}) = \mathsf{gcd}(\overrightarrow{A}) \right| \wedge \left\| \left[\ \overrightarrow{d} = \mathsf{gcd}(\overleftarrow{A}) \right] \right\|$$

As a matter of routine logic we have:

Proposition 6 $\quad \textsc{Spec}\,1 \ \sqsubseteq \ \textsc{Spec}\,2$

4.3 Identifying an established component

Now we must decide how to identify an established component. The simplest method for a 'homogeneous' problem is to imagine that the specification has already been established on a 'smaller scale'. ('Weakening a constant to a variable' is an example of that method; so is recursion.) For then if we can calculate the rest of a refinement using the weakest specifunction technique, that calculation can be repeated until a program results.

In the present case we think of the target specification as being in parameterised form, so that $\textsc{Spec}\,2$ is regained by taking the extreme value $j = p{-}1$ of the parameter.

That leads us to define, for $0 < j < p$,

$$\textsc{Spec}\ \mathbf{3} \ \hat{=}\ \lfloor s_j \rfloor \wedge [\![t_j]\!] \quad \textit{where}$$

$$s_j \ \hat{=}\ \mathsf{gcd}(\overleftarrow{A}[0,j)) = \mathsf{gcd}(\overrightarrow{A}[0,j))$$

$$t_j \ \hat{=}\ \overrightarrow{d} = \mathsf{gcd}(\overleftarrow{A}[0,j)).$$

In order to make explicit the dependence on j of specification $\textsc{Spec}\,3$ we write, for $0 < j < p$,

$$Q_j \ \hat{=}\ \textsc{Spec}\,3.$$

Thus Q_j computes the gcd of the array $A[0,j)$.

Using compact notation for index decrement, $j^- \ \hat{=}\ j-1$, we can now calculate the weakest specifunction between Q_j and Q_{j^-}. For $0 < j < p$ we seek commands $\textsc{Spec}\,4$ and $\textsc{Spec}\,5$ for which

$$Q_j \div Q_{j^-} \ \sqsubseteq\ \frac{\mid \textsc{Spec}\,4}{\textsc{Spec}\,5}\ .$$

The laws of section 3 lead us to take

Proposition 7 *For* $0 < j < p$,

$$\textsc{Spec}\ \mathbf{4} \ =\ \lfloor s_{j^-} \Rightarrow s_j \rfloor$$

$$\textsc{Spec}\ \mathbf{5} \ =\ \lfloor s_j \rfloor \wedge [\![t_j \setminus (t_{j^-} \wedge s_j)]\!]\ .$$

Proof. Routine verification confirms that each s_j is idempotent.

Also by law 2, the parallel partner is a safety property and the sequent contains both safety and liveness properties. □

4.4 Refining the parallel partner

There is a variety of ways to refine the parallel partner. In the extreme case we can leave $A[j]$ unchanged:

$$(s_{j^-} \Rightarrow s_j) \Leftarrow \mathrm{Id}_j$$

where $\mathrm{Id}_j \ \hat{=}\ \overleftarrow{A}[j] = \overrightarrow{A}[j]$.

However a more interesting specification always reduces $A[j]$ 'safely':

$$\begin{array}{l} (s_{j^-} \Rightarrow s_j) \\ \Leftarrow \\ \vee_{k<j} \left(\overleftarrow{A}[j] > \overleftarrow{A}[k] \ \wedge\ \overrightarrow{A}[j] = \overleftarrow{A}[j] \ \mathsf{mod}\ \overleftarrow{A}[k] \right) \ \vee\ \mathrm{Id}_j . \end{array} \tag{9}$$

That prompts us to introduce a conditional guarded command to refine the parallel partner to code. For $0 < j < p$,

$$
\textsc{Spec}\ \mathbf{6} \ \hat{=} \ \left| \begin{array}{lll} \textbf{if} & & \\ []_{k<j} \ \overleftarrow{A}[j] > \overleftarrow{A}[k] & \rightarrow & \overrightarrow{A}[j] = \overleftarrow{A}[j] \ \mathsf{mod} \ \overleftarrow{A}[k] \\ [] \quad \mathit{true} & \rightarrow & \mathrm{Id}_j \\ \textbf{fi} & & \end{array} \right|
$$

To make explicit its dependence on j we write, for use in the next section,

$$P_j \ \hat{=} \ \textsc{Spec}\,6.$$

The informal reasoning above provides a sketch proof of:

Proposition 8 *For* $0 < j < p$, $\textsc{Spec}\,4 \ \sqsubseteq \ \textsc{Spec}\,6$.

Proof. Direct from implication (9). □

4.5 Refining the sequent

The sequent must terminate in a state in which d is the gcd of all numbers in $A[0, j)$. Before the start of the sequent, the established component Q_{j^-} has already terminated in a state in which d is the gcd of all numbers in $A[0, j^-)$. The safety property s_j ensures that the gcd of $A[0, j)$ has been kept unchanged since the beginning of the whole computation. Thus the final output of d can be derived from that of the established component:

$$t_j \setminus (t_{j^-} \wedge s_j) \ \Leftarrow \ \overrightarrow{d} = \mathsf{gcd}(\overleftarrow{d}, \overleftarrow{A}[j]).$$

Thus for $0 < j < p$ the sequent is refined by:

$$\textsc{Spec}\ \mathbf{7} \ \hat{=} \ \left| \mathsf{gcd}(\overleftarrow{A}[0, j)) = \mathsf{gcd}(\overrightarrow{A}[0, j)) \right| \wedge \left\| \left[\overrightarrow{d} = \mathsf{gcd}(\overleftarrow{d}, \overleftarrow{A}[j]) \right] \right\| .$$

Now the gcd of the two numbers d and $A[j]$ can be calculated using the standard Euclidean algorithm. Since the sequent keeps $A[0, j^-)$ unchanged, the safety requirement is met. Thus for $0 < j < p$ we define

$$\textsc{Spec}\ \mathbf{8} \ \hat{=} \ \mathit{II}_1 \parallel \cdots \parallel \mathit{II}_{j^-} \parallel S_j \quad \textit{where}$$

$$\mathit{II}_k \ \hat{=} \ |\, \mathrm{Id}_k \,|$$

$$
S_j \ \hat{=} \ \begin{array}{lll} \textbf{do} & & \\ \quad d > A[j] & \rightarrow & \langle (\overrightarrow{d} = \overleftarrow{d} \ \mathsf{mod} \ \overleftarrow{A}[j]) \wedge \mathrm{Id}_j \rangle_1 \\ [] \ d < A[j] & \rightarrow & \langle (\overrightarrow{A}[j] = \overleftarrow{A}[j] \ \mathsf{mod} \ \overleftarrow{d}) \wedge \overleftarrow{d} = \overrightarrow{d} \rangle_1 \\ \textbf{od} & & \end{array}
$$

$$\left\|\left[\ \overrightarrow{d} = \mathsf{gcd}(\overleftarrow{A})\ \right]\right\| \;\sqsubseteq\; \begin{array}{l}\textbf{do for } j := 0 \textbf{ to } p^{-} \;\rightarrow \\ \quad \mathit{II}_0 \parallel \cdots \parallel \mathit{II}_{j^-} \parallel S_j \parallel P_{j^+} \parallel \;\cdots\; \parallel P_{p^-} \\ \textbf{od}\end{array}$$

Figure 3: The result of the derivation in section 4

Proposition 9 *For* $0 < j < p$, $\quad \text{Spec}\,5 \;\sqsubseteq\; \text{Spec}\,7 \;\sqsubseteq\; \text{Spec}\,8$.

Proof. The former follows by the dicussion at the beginning of this subsection.

For the latter, to ensure that the gcd of $A[0, j)$ remains unchanged, one simple implementation is just to keep each element in the part $[0, j)$ of the array A unchanged. However we take for granted the standard verification involving the binary gcd. □

Proposition 10 *For* $0 < j < p$, $\quad Q_j \div Q_{j^-} \;\sqsubseteq\; \dfrac{\mid P_j}{\mathit{II}_1 \parallel \cdots \parallel \mathit{II}_{j^-} \parallel S_j}$.

Proof. Routine expansion. □

4.6 Collecting code

The weakest specifunction calculation is repeated until it reaches Q_0, which is implemented directly by a sequent of 0 steps:

$$S_0 \;\hat{=}\; \left(\overrightarrow{d} = \overrightarrow{A}[0] = \overleftarrow{A}[0]\right).$$

These calculations form a compositional refinement according to law 5 and theorem 9:

$$(Q_{p^-} \div Q_0) \;\sqsubseteq\; (Q_{p^-} \div Q_{p^{--}}) \circ \cdots \circ (Q_1 \div Q_0).$$

Our program is obtained by combining those final specifunctions

$$\text{Spec}\,\mathbf{9} \quad \hat{=} \quad \begin{array}{l}\textbf{do for } j := 0 \textbf{ to } p^{-} \;\rightarrow \\ \quad \mathit{II}_0 \parallel \cdots \parallel \mathit{II}_{j^-} \parallel S_j \parallel P_{j^+} \parallel \;\cdots\; \parallel P_{p^-} \\ \textbf{od}\end{array}$$

where $j^+ \;\hat{=}\; j{+}1$.

Its correctness follows from the results established:

Theorem 11 $\quad \textsc{Spec}\,1 \;\sqsubseteq\; \textsc{Spec}\,9$.

Weakest specifunction calculation has successfully decomposed the design of a parallel algorithm into two parts. In the first part, the main concern is to approach the final solution, safely and efficiently, without necessarily guaranteeing progress; the second part of the algorithm starts from a state already close to the final solution and then forces further progress leading to termination. Repetition of such decomposition produces a flat, parallel algorithm.

This algorithm can readily be modified to work in a distributed environment exhibiting efficient but unreliable communication for the first part of the algorithm and less efficient but reliable communication for the second.

5 Conclusion

In this paper we have introduced the concept of the weakest specifunction and provided its general theory. We have indicated the potential of that concept by providing a complete law for a two-parameter specifunction, the par-seq specifunction, which combines parallel and sequential combinators and so lies beyond the scope of previous theories. We have applied the resulting calculus to derive, by factoring it into safety and liveness, a distributed algorithm for the greatest common divisor of a set of distributed integers.

The technique presented here differs from program derivation in a refinement calculus where laws are provided for correct refinement but the implementor is left entirely to his own choice of which law to apply at each instance. The weakest specifunction technique, like the weakest specification techniques from which it arose, enables the weakest component complementing an established component to be *calculated*.

Raising the level of consideration from specifications to functions between specifications seems to afford a substantial increase in applicability of weakest specification calculii and hence to provide formal support for the reuse of established components in software engineering. The details of calculation are still more complicated than we would like; we therefore look towards automated assistance for more general use of this technique.

References

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

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

[3] Y. Chen, Reactive-BSP semantics. Technical Report, Oxford University Computing Laboratory, 1999.

[4] Y. Chen, Specification for reactive bulk-synchronous programming. *Proc. 8th Euromicro Workshop on Parallel and Distributed Processing*: 190-196, IEEE Computer Society Press, 2000.

[5] Y. Chen, *Formal Methods for Bulk-Synchronous Programming*. DPhil. thesis, Oxford University Computing Laboratory, to appear 2000.

[6] Y. Chen and J.W. Sanders, Logic of global synchrony. Technical Report, Oxford University Computing Laboratory, 2000.

[7] E. W. Dijkstra, Guarded commands, non-determinacy and the formal derivation of programs. *Communications of the ACM*, **18**:453-457, 1975.

[8] S. Fortune and J. Wyllie, Parallelism in random access machines. *Proc. 10th Annual ACM Symp. on Theory of Computing*: 114-118, 1978.

[9] F. B. Hildebrand, *Introduction to Numerical Analysis*. Tata McGraw-Hill, 1978.

[10] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall series in Computer Science, 1985.

[11] C. A. R. Hoare and J. He, The weakest prespecification I,II. *Fundamenta Informatica*, **9**:51-84, 217-252, 1986.

[12] C. A. R. Hoare et al. Laws of programming. *Communications of the ACM*, **30**(8):672-686, 1987.

[13] C. A. R. Hoare and J. He, *Unifying Theories of Programming*. Prentice-Hall series in Computer Science, 1998.

[14] L. Lai and J. W. Sanders, A weakest-environment calculus for communicating processes. *Proc. 4th Nordic Transputer Conference: Parallel Programming and Applications*, ed. P. Fritzon and L. Finmo, 381-395, IOS Press, Ohmsha, 1995.

[15] D. S. Lecomber, *Methods of BSP Programming.* Oxford University Computing Laboratory DPhil. thesis, 1998.

[16] C. C. Morgan, *Programming from Specifications*, second edition. Prentice-Hall series in Computer Science, 1994.

[17] A. W. Roscoe, *The Theory and Practice of Concurrency.* Prentice-Hall series in Computer Science, 1997.

[18] L. G. Valiant, A bridging model for parallel computation. *Communications of the ACM*, **33**(8):103-111, 1990.