# SOFTWARE VERIFICATION RESEARCH CENTRE THE UNIVERSITY OF QUEENSLAND 

Queensland 4072
Australia

## TECHNICAL REPORT

No. 00-30
A Declarative Semantics for Logic
Program Refinement
Ian Hayes* Ray Nickson ${ }^{\dagger}$
Paul Strooper* Robert Colvin*
October 2000

Phone: +61 733651003
Fax: +61 733651533
http://svrc.it.uq.edu.au

Note: Most SVRC technical reports are available via anonymous ftp, from svrc.it.uq.edu.au in the directory /pub/techreports. Abstracts and compressed postscript files are available via http://svrc.it.uq.edu.au

# A Declarative Semantics for Logic Program Refinement 

Ian Hayes* Ray Nickson ${ }^{\dagger}$ Paul Strooper* Robert Colvin*

* School of Computer Science and Electrical Engineering, The University of Queensland, Australia
${ }^{\dagger}$ School of Mathematical and Computing Sciences, Victoria University of Wellington, New Zealand


#### Abstract

The refinement calculus provides a framework for the stepwise development of imperative programs from specifications. This paper presents a semantics for a refinement calculus for deriving logic programs. The calculus contains a wide-spectrum logic programming language, including executable constructs such as sequential conjunction, disjunction, and existential quantification, as well as specifications constructs (general predicates and assumptions) and universal quantification. A semantics is defined for this wide-spectrum language based on executions, which are partial functions from states to states, where a state is represented as a set of bindings. This execution semantics is used to define the meaning of programs and specifications, including parameters and recursion. To complete the calculus, a notion of correctness-preserving refinement over programs in the widespectrum language is defined and a refinement law for introducing recursive procedures is presented.


## 1 Introduction

Our goal is to provide a method for the systematic development of logic programs from specifications. We follow a refinement calculus approach [Bac88], which provides a framework for the stepwise development of imperative programs from specifications. It makes use of a widespectrum language that includes both specification and programming language constructs. This allows a specification to be refined, step by step, to program code within a single language. The programs produced during the intermediate steps of this process may contain specification constructs as components, and hence may not be code suitable for execution.

In this paper, we give a declarative semantics for a refinement calculus for logic programming. The calculus contains a wide-spectrum logic programming language, including executable conjunction, disjunction, and existential quantification, as well as specification constructs (general predicates and assumptions) and universal quantification, which are not in general executable. General predicates allow non-executable properties to be included in specifications. Assumptions represent information about the context in which a program fragment will execute. An implementation is obliged to produce the specified result only if its assumptions are satisfied by the context. The language also supports parametrised procedures and recursion.

| $\langle P\rangle$ | - | specification |
| ---: | :--- | :--- |
| $\{\mathrm{A}\}$ | - | assumption |
| $\left(c_{1} \vee c_{2}\right)$ | - | disjunction |
| $\left(c_{1} \wedge c_{2}\right)$ | - | parallel conjunction |
| $\left(c_{1}, c_{2}\right)$ | - | sequential conjunction |
| $(\exists v \bullet c)$ | - | existential quantification |
| $(\forall v \bullet c)$ | - | universal quantification |
| $i d(T)$ | - | procedure call |

Figure 1: Summary of wide-spectrum language

The semantics of the wide-spectrum language is defined in terms of executions, which are partial functions from initial to final states. A state in turn is represented as a set of bindings, where each binding is a mapping from variables to values. As is traditionally the case with logic programs, we consider only executions where the bindings in the final state are a subset of the bindings in the initial state. To complete the calculus, we define a notion of correctnesspreserving refinement over programs in the wide-spectrum language and a refinement law for introducing recursive procedures.

Section 2 of this paper summarizes the wide-spectrum logic programming language. Section 3 gives the basic definitions necessary for our formal semantics. Section 4 presents the semantics of the base language in terms of executions. Section 5 defines our notion of refinement. Section 6 gives the machinery for dealing with procedures and parameters. Section 7 introduces recursion and in Section 8 we define a refinement rule for introducing recursion and present a small example. Section 9 concludes with a discussion of related work.

Our semantics is described using the notation of the Z specification language [Spi92], with which the reader is assumed to be familiar. Appendix A summarises the main definitions of the paper and illustrates our convention for naming variables. Various properties arising from our semantics, which are required but not proved in the body of the paper, are proved in Appendices B-F.

## 2 Wide-spectrum language

This section presents the wide-spectrum logic programming language [HNS.97], which combines both logic programming language and specification language constructs. It allows constructs that may not be executable, similar to Back's [Bac88] inclusion of specification constructs in Dijkstra's imperative language. This has the benefit of allowing gradual refinement without the need for notational changes during the refinement process. The constructs in the language are specifications, assumptions, propositional operators, quantifiers, and procedure calls. A summary of the language is shown in Figure 11. Below we describe the constructs of the language, and discuss the intuition behind each.

Specifications A specification $\langle P\rangle$, where $P$ is a predicate, represents a set of instantiations of the free variables of the program that satisfy $P$ (we defer detailed discussion of predicates unitl Section 3.3). For example, the specification $\langle X=5 \vee X=6\rangle$ represents the set of
instantiations $\{5,6\}$ for $X$. The specification fail is defined by:

$$
\text { fail }==\langle\text { false }\rangle
$$

It always computes an empty answer set, like Prolog's fail. Note that $\langle$ true $\rangle$ represents the set of all instantiations and therefore has no effect (it is like skip in the traditional refinement calculus).

Assumptions An assumption $\{A\}$, where $A$ is a predicate, expresses a requirement on the context for a program fragment. For example, some programs may require that an integer parameter be non-zero, expressed as $\{X \neq 0\}$. If assumptions about the context are formally expressed, implementations may take advantage of the assumptions, but need not establish (or indeed check) them. If these assumptions do not hold, the program fragment may abort. Aborting includes program behaviour such as nontermination and abnormal termination due to exceptions like division by zero, as well as termination with arbitrary results. We define the (worst possible) program abort by

$$
\text { abort }==\{\text { false }\}
$$

Note that abort is quite different from the program fail, which never aborts, but has an empty solution set. Also, $\{$ true $\}$, like $\langle$ true $\rangle$, has no effect, since every context satisfies it.

Propositional Operators There are two forms of conjunction: a sequential form ( $c_{1}, c_{2}$ ) where $c_{1}$ is evaluated before $c_{2}$; and a parallel version $\left(c_{1} \wedge c_{2}\right)$ where $c_{1}$ and $c_{2}$ are evaluated independently and the intersection of their respective results is formed on completion. The disjunction of two programs $\left(c_{1} \vee c_{2}\right)$ computes the union of the results of the two programs. We overload the symbols ' $\wedge$ ' and ' $V$ ' as both operators on predicates and operators on commands. Because, for example, the meanings of $\langle P \wedge Q\rangle$ and $\langle P\rangle \wedge\langle Q\rangle$ are identical, this does not usually cause confusion.

The following three programs illustrate the behaviour of sequential and parallel conjunction, and show the difference between abortion and failure.

$$
\begin{aligned}
& P 1==\{X \neq 0\},\langle Y=1 / X\rangle \\
& P 2==\langle X \neq 0\rangle,\langle Y=1 / X\rangle \\
& P 3==\langle X \neq 0\rangle \wedge\langle Y=1 / X\rangle
\end{aligned}
$$

If each of the three programs is executed from a state where $X=0, P 1$ is equivalent to abort, while $P 2$ will fail, producing an empty answer set. The behaviour of $P 3$ is also equivalent to abort, because the expression $Y=1 / 0$ is not defined.

Quantifiers The existential quantifier ( $\exists v \bullet c)$ generalises disjunction, computing the union of the results of $c$ for all possible values of $v$. Similarly, the universal quantifier $(\forall v \bullet c)$ computes the intersection of the results of $c$ for all possible values of $v$.

Procedures A procedure definition has the form

$$
i d==v:-c
$$

where $i d$ is an identifier, $v$ is a variable representing the parameter to the procedure, and $c$ is a wide-spectrum program (the motivation for using this somewhat non-standard notation will become clear later). It defines the procedure called $i d$ with a formal parameter $v$ and body $c$. A call on the procedure $i d$ is of the form $i d(T)$, where $T$ is a term: the actual parameter. No
generality is lost in restricting procedures to a single parameter, as multiple parameters may be encoded using compound terms.

Commands We define $C m d$ to be the set of commands in our language, built up from the constructs shown in Figure 11. Note that procedure definitions are not commands in our language; they are dealt with in Section 6.

## 3 Domains

We begin our formal treatment of the semantics by defining the domains over which our semantics of programs are given.

### 3.1 Variables, values, and functors

We have fixed domains of variables (Var), values (Val), and functors (Fun).

$$
[\text { Var, Val, Fun] }
$$

Elements of Var represent the variables that can occur in programs. This includes variables for which a program's answers give values, variables that are bound by universal and existential quantifiers, and variables used in formal parameters. Values are the objects in the universe of discourse, denoted by ground terms. Functors represent the function symbols in our language that are used to construct compound terms. Each functor has an arity defined for it.
arity: Fun $\rightarrow \mathbb{N}$
Atoms are functors of arity zero.
If we restrict our interpretation of ground terms to the Herbrand interpretation, as is typical in logic programming, Val is structured into atoms and compound terms. But we allow more structure than this; Val may be structured according to any algebra, taking into account whatever kind of term equality is appropriate for the application under consideration. For the purposes of this paper, all we assume is that there is a function apply, which models the application of a function to a sequence of values, resulting in a value.

$$
\begin{aligned}
& \text { apply: Fun } \rightarrow(\text { seq } V a l \rightarrow V a l) \\
& \hline \forall f: \text { Fun } \bullet \operatorname{dom}(\text { apply } f) \subseteq\{s: \text { seq } V a l \mid \# s=\operatorname{arity} f\}
\end{aligned}
$$

With this more general interpretation, given a functor $f$ of arity $n$, 'apply $f$ ' may be undefined for some sequences of values of length $n$. We write $\operatorname{def} E(\operatorname{def} P)$ for the predicate that is true precisely when the expression $E$ (predicate $P$ ) is well-defined: that is, when all function applications occuring within it are well-defined. For the Herbrand interpretation, or indeed for any other interpretation in which all functions are total, def $E=\operatorname{def} P=$ true for all expressions $E$ and predicates $P$.

### 3.2 Bindings

A binding is a total function, mapping every variable to a value.

$$
\text { Bnd }==\operatorname{Var} \rightarrow \text { Val }
$$

Each binding corresponds to a single ground answer to a Prolog-like query. The mechanism for representing "unbound" variables is described below.

### 3.3 States and predicates

A predicate is a function from bindings to booleans.

$$
\text { Pred }==\text { Bnd } \rightarrow \mathbb{B}
$$

It corresponds to our usual notion of a predicate with some free variables, which is true or false once provided with a binding for those variables. We write $\bar{P}$ to denote the set of bindings satisfying $P$. For example:

$$
\begin{aligned}
\overline{\text { false }} & =\varnothing \\
\overline{\text { true }} & =\text { Bnd } \\
\overline{X=3} & =\{b: B n d \mid b X=3\} \\
\overline{X<Y} & =\{b: B n d \mid b X<b Y\}
\end{aligned}
$$

A state is a set of bindings.

$$
\text { State }==\mathbb{P} \text { Bnd }
$$

An unbound variable is represented by a possibly infinite state that has one binding to each element of Val. For example, if we suppose that Var contains just the variables $\{X, Y, Z\}$, and $f$ has arity one, the set of solutions to the equation $Y=f(X)$ is represented by the state

$$
\{x, z: \text { Val } \bullet\{X \mapsto x, Y \mapsto \text { apply } f\langle x\rangle, Z \mapsto z\}\}
$$

which contains bindings that explicitly map the unbound variables $X$ and $Z$ to every pair of values $x$ and $z$, with each binding mapping $Y$ to the value apply $f\langle x\rangle$ (assuming apply $f$ is total on singleton sequences).

### 3.4 Terms

A term is a variable, or a functor with a (possibly empty) sequence of terms (the arguments).

$$
\text { Term: }:=\operatorname{var} T\langle\langle\operatorname{Var}\rangle\rangle \mid \text { fun } T\langle\langle F u n \times \operatorname{seq} \text { Term }\rangle\rangle
$$

For any variable $v, \operatorname{var} T(v)$ is a term, and if $f$ is a functor and $t s$ is a sequence of terms, then funT( $f, t s)$ is a term.

A term may have a value when evaluated relative to some binding, or it may be undefined if the term involves the incorrect application of a function. We define a partial function eval that evaluates a term relative to a binding.

```
eval: Term \(\rightarrow(\) Bnd \(\rightarrow\) Val \()\)
    \(\operatorname{eval}(\operatorname{varT} v)=(\lambda b: B n d \bullet b v)\)
    \(\operatorname{eval}(f u n T(f, t s))=\{b: B n d ;\) vs: seq \(V a l \mid\)
        \((\forall i: \operatorname{dom} t s \bullet b \in \operatorname{dom}(\operatorname{eval}(t s(i)))) \wedge\)
        \(v s=\operatorname{map}(\lambda t:\) Term \(\bullet\) eval \(t b)(t s) \wedge\)
        \(v s \in \operatorname{dom}(\) apply \(f\) )
        - \(b \mapsto\) apply \(f v s\}\)
```

To evaluate a compound term $f u n T(f, t s)$ with respect to a binding $b$, all of the terms in the sequence $t s$ must be able to be evaluated with respect to $b$, and the resultant sequence of values $v s$ must be in the domain of apply $f$.

For a term $t$, defined $(t)$ is the set of states for which $t$ is defined for every binding in the state.
defined: Term $\rightarrow \mathbb{P}$ State

$$
\text { defined } t=\{s: \text { State } \mid s \subseteq \operatorname{dom}(\text { eval } t)\}
$$

For a variable $v$, term $t$, and state $s$, assign $v t s$ is the same as state $s$, except that in each binding within $s$ the value of $v$ is replaced by the value of $t$ in that binding. In the following definition, ' $\oplus$ ' stands for function override.

$$
\begin{aligned}
& \text { assign: Var } \rightarrow \text { Term } \rightarrow \text { State } \rightarrow \text { State } \\
& \text { assign } v t=(\lambda s: \text { defined } t \bullet\{b: s \bullet b \oplus\{v \mapsto \text { eval } t b\}\})
\end{aligned}
$$

For some term $t$, free $t$ is the set of free variables in $t$ :

$$
\begin{aligned}
& \text { free: } \operatorname{Term} \rightarrow \mathbb{P} \text { Var } \\
& \text { free }(\operatorname{var} T v)=\{v\} \\
& \text { free }(\text { fun } T(f, t s))=\bigcup\{t: \text { ran } t s \bullet \text { free } t\}
\end{aligned}
$$

Similarly, for a command $c$, the function free $c$ defines the set of free variables in $c$.

## 4 Program execution

### 4.1 Executions

We define the semantics of our language in terms of executions, which are mappings from initial states to final states. The mapping is partial because the program is only well-defined for those initial states that guarantee satisfaction of all the program's assumptions.

Executions satisfy the following healthiness properties:

1. If a command is guaranteed to terminate from an initial state $\bar{P}$ whose bindings all satisfy some predicate $P$, it must also guarantee to terminate from all those initial states $\overline{P^{\prime}}$, where $P^{\prime} \Rightarrow P$. We thus require that any subset $s^{\prime}$ of a set $s$ that is in the domain of an execution $e$, is also in the domain of $e$.

$$
\forall s: \operatorname{dom} e \bullet\left(\forall s^{\prime}: \mathbb{P} s \bullet s^{\prime} \in \operatorname{dom} e\right)
$$

In addition, if a command is guaranteed to terminate from initial state $\bar{P}$ and it is also guaranteed to terminate from initial state $\bar{Q}$, it must terminate from an initial state $\overline{P \vee Q}$. Thus, if all sets in a set of states $s s$ are in the domain of $e$, then their union is also in the domain of $e$.

$$
\forall s s: \mathbb{P}(\operatorname{dom} e) \bullet \bigcup s s \in \operatorname{dom} e
$$

As we show in Appendix B.1, these together are equivalent to the fact that the domain of $e$ is the powerset of the set of all bindings, $b$, such that $\{b\}$ is in the domain of $e$.

$$
\operatorname{dom} e=\mathbb{P}\{b: B n d \mid\{b\} \in \operatorname{dom} e\}
$$

2. Because of the constraining nature of logic programs (command execution cannot decrease "groundedness"), for any state $s$ in the domain of an execution $e$, the set of bindings in the output state $e(s)$ must be a subset of $s$.

$$
\forall s: \operatorname{dom} e \bullet e(s) \subseteq s
$$

3. For a set of bindings $s$, the output set of bindings can be determined by considering the effect of the execution on each singleton binding, and then forming the union of the results.

$$
\forall s: \operatorname{dom} e \bullet e(s)=\bigcup\{b: s \bullet e(\{b\})\}
$$

We thus define:

$$
\begin{align*}
\text { Exec }== & \{e: \text { State } \rightarrow \text { State } \mid \\
& \operatorname{dom} e=\mathbb{P}\{b: \text { Bnd } \mid\{b\} \in \operatorname{dom} e\} \wedge  \tag{1}\\
& (\forall s: \operatorname{dom} e \bullet e(s) \subseteq s) \wedge  \tag{2}\\
& (\forall s: \operatorname{dom} e \bullet e(s)=\bigcup\{b: s \bullet e(\{b\})\})\} \tag{3}
\end{align*}
$$

Note that property (1) implies that $\varnothing \in \operatorname{dom} e$ for all executions $e$. Also, from property (21), $e(\{b\})$ is either $\{b\}$ or $\}$. In Appendix B.2, we show that, provided properties (1) and (2) hold, property (3) is equivalent to any of the following three properties.

$$
\begin{align*}
& (\forall s: \operatorname{dom} e \bullet e(s)=\{b: s \mid e(\{b\}) \neq\{ \})\}  \tag{4}\\
& \left(\forall s: \operatorname{dom} e \bullet \forall s^{\prime}: \mathbb{P} s \bullet e\left(s^{\prime}\right)=e(s) \cap s^{\prime}\right)  \tag{5}\\
& \forall s s: \mathbb{P}(\operatorname{dom} e) \bullet e(\bigcup s s)=\bigcup\{s: s s \bullet e(s)\} \tag{6}
\end{align*}
$$

Property 4 shows that an execution $e$ may be seen as a "filter"; given a state $s$, it either passes or blocks each binding $b$. Property 5 shows that the result of executing a command in a subset $s^{\prime}$ of some state $s$ is consistent with executing the command in state $s$ and restricting the results to those in $s^{\prime}$. This property is similar to the property quoted by Hoare [Hoa00] and attributed to He Jifeng, as one that characterises a pure logic program. For example, Prolog's 'var' does not satisfy the property. Property 6 shows that executions distribute over union, which is used to prove continuity of executions (see Appendix F).

### 4.2 Semantic function for commands

We define the semantics of the commands in our language via a function that takes a command and returns the corresponding execution.
exec: $C m d \rightarrow$ Exec

```
\(\operatorname{exec}(\langle P\rangle)=(\lambda s: \mathbb{P}(\overline{\operatorname{def} P}) \bullet s \cap \bar{P})\)
\(\operatorname{exec}(\{A\})=(\lambda s: \mathbb{P}(\overline{\operatorname{def} A \wedge A}) \bullet s)\)
exec fail \(=\operatorname{exec}\langle\) false \(\rangle=(\lambda s:\) State \(\bullet \varnothing)\)
exec abort \(=\) exec \(\{\) false \(\}=\{\varnothing \mapsto \varnothing\}\)
\(\operatorname{exec}\left(c_{1} \vee c_{2}\right)=\operatorname{exec} c_{1} \cup \operatorname{exec} c_{2}\)
\(\operatorname{exec}\left(c_{1} \wedge c_{2}\right)=\operatorname{exec} c_{1} \cap \operatorname{exec} c_{2}\)
\(\operatorname{exec}\left(c_{1}, c_{2}\right)=\operatorname{exec} c_{1}{ }_{9} \operatorname{exec} c_{2}\)
\(\operatorname{exec}(\exists v \bullet c)=\) exists \(v(\operatorname{exec} c)\)
\(\operatorname{exec}(\forall v \bullet c)=\) forall \(v(\operatorname{exec} c)\)
```

Figure 2: Execution semantics of basic commands

The semantics of the basic commands (excluding procedures, parameters and recursion) is shown in Figure 2. In the remainder of this section, we explain the definitions. In Appendix 0, we show that all executions constructed using the definitions satisfy the healthiness properties of executions.

In Section 6, where we discuss procedures and parameters, we extend the definition of exec with an environment, which maps procedure identifiers to their corresponding executions. For simplicity, we first present the semantics of the basic commands ignoring the environment.

### 4.3 Specifications and assumptions

A specification $\langle P\rangle$ is defined for all states $s$ such that $P$ is defined for all bindings in $s$; the result of executing specification $\langle P\rangle$ consists of those bindings in $s$ that satisfy $P$.

An assumption $\{A\}$ is defined for all states $s$ such that $A$ is defined and $A$ holds for all bindings in $s$; the result of executing assumption $\{A\}$ has no effect (the set of bindings remains unchanged).

The definition for the special-case specification fail is the constant function that returns the empty state, no matter what the initial state is. Hence for any command $c$, including abort,

```
exec(fail,}c)=\operatorname{exec}(\mathrm{ fail )
```

because fail maps any state to the empty state.
The definition for the special-case assumption abort is the function mapping the empty state to the empty state. Hence for any command $c$,

```
exec(abort,},c)=\operatorname{exec(abort)
```

because the domain of abort contains only the empty state, which it maps to the empty state. Note that the empty state is preserved by any command, i.e., for any command $c$, $\operatorname{exec}(c)(\varnothing)=\varnothing$.

### 4.4 Propositional operators

Disjunction and parallel conjunction are defined as pointwise union and intersection of the corresponding executions.

$$
\begin{aligned}
& -ค_{-}: \text {Exec } \times \text { Exec } \rightarrow \text { Exec } \\
& -\cup_{-}: \text {Exec } \times \text { Exec } \rightarrow \text { Exec } \\
& \left(e_{1} \cap e_{2}\right)=\left(\lambda s: \operatorname{dom} e_{1} \cap \operatorname{dom} e_{2} \bullet\left(e_{1} s\right) \cap\left(e_{2} s\right)\right) \\
& \left(e_{1} \cup e_{2}\right)=\left(\lambda s: \operatorname{dom} e_{1} \cap \operatorname{dom} e_{2} \bullet\left(e_{1} s\right) \cup\left(e_{2} s\right)\right)
\end{aligned}
$$

For a conjunction $\left(c_{1} \wedge c_{2}\right)$, if a state $s$ is mapped to $s^{\prime}$ by exec $c_{1}$ and $s$ is mapped to $s^{\prime \prime}$ by exec $c_{2}$, then exec $\left(c_{1} \wedge c_{2}\right)$ maps $s$ to $s^{\prime} \cap s^{\prime \prime}$. Disjunction is similar, but gives union of the resulting states instead of intersection.

Sequential conjunction $\left(c_{1}, c_{2}\right)$, is defined as function composition of the corresponding executions.

$$
\begin{array}{|l}
-\stackrel{\circ}{9} \text {-: Exec } \times \text { Exec } \rightarrow \text { Exec } \\
\\
\left(e_{1} \stackrel{\circ}{9} e_{2}\right)=\left(\lambda s: \operatorname{dom} e_{1} \mid e_{1}(s) \in \operatorname{dom} e_{2} \bullet e_{2}\left(e_{1}(s)\right)\right)
\end{array}
$$

If exec $c_{1}$ maps state $s$ to $s^{\prime}$ and exec $c_{2}$ maps $s^{\prime}$ to $s^{\prime \prime}$, then $\operatorname{exec}\left(c_{1}, c_{2}\right)$ maps $s$ to $s^{\prime \prime}$. If either $s$ is not in the domain of exec $c_{1}$ or $s^{\prime}$ is not in the domain of exec $c_{2}$, then $s$ is not in the domain of $\operatorname{exec}\left(c_{1}, c_{2}\right)$.

### 4.5 Quantifiers

For a variable $v$ and a state $s$, we define the state unbind $v s$ as one whose bindings match those of $s$ in every place except $v$, which is completely unconstrained.

$$
\begin{aligned}
& \text { unbind: Var } \rightarrow \text { State } \rightarrow \text { State } \\
& \hline \text { unbind } v s=\{b: s ; x: \text { Val } \bullet b \oplus\{v \mapsto x\}\}
\end{aligned}
$$

Execution of an existentially quantified command $\exists v \bullet c$ from an initial state $s$ is defined if executing $c$ is defined in the state $s^{\prime}$, which is the same as $s$ except that $v$ is unbound. The resultant state after executing $c$ consists of all those bindings $b$ in $s$ such that there is a value, $x$, for $v$ such that execution of $c$ retains the binding $b \oplus\{v \mapsto x\}$. We thus make the following definition of the existential quantifier for executions.

$$
\begin{aligned}
& \text { exists: Var } \rightarrow \text { Exec } \rightarrow \text { Exec } \\
& \text { exists } v e=(\lambda s: \text { State } \mid \text { unbind } v s \in \operatorname{dom} e \\
& \bullet\{b: s \mid(\exists x: \text { Val } \bullet e(\{b \oplus\{v \mapsto x\}\}) \neq\{ \})\})
\end{aligned}
$$

Universal quantification behaves in a similar fashion to existential quantification, except that for forall $v e$ to retain a binding $b$, execution of $e$ must retain $b \oplus\{v \mapsto x\}$ for all values $x$.

$$
\begin{aligned}
& \text { forall: } \operatorname{Var} \rightarrow \text { Exec } \rightarrow \text { Exec } \\
& \text { forall } v e=(\lambda s: \text { State } \mid \text { unbind } v s \in \operatorname{dom} e \\
& \quad \bullet\{b: s \mid(\forall x: \text { Val } \bullet e(\{b \oplus\{v \mapsto x\}\}) \neq\{ \})\})
\end{aligned}
$$

## 5 Refinement

An execution $e_{1}$ is refined by an execution $e_{2}$ if and only if $e_{2}$ is defined wherever $e_{1}$ is and they agree on their outputs whenever both are defined. This is the usual "definedness" order on partial functions, as used, for example, by Manna [Man74]: it is simply defined by the subset relation of functions viewed as sets of pairs.

$$
\begin{aligned}
& -\sqsubseteq \_: \text {Exec } \leftrightarrow \text { Exec } \\
& \hline e_{1} \sqsubseteq e_{2} \Leftrightarrow e_{1} \subseteq e_{2}
\end{aligned}
$$

For the commands $C m d$ in our language, we define refinement in terms of refinement for the corresponding executions.

$$
\begin{aligned}
& -\sqsubseteq \_: C m d \leftrightarrow C m d \\
& \hline c_{1} \sqsubseteq c_{2} \Leftrightarrow \operatorname{exec} c_{1} \sqsubseteq \operatorname{exec} c_{2}
\end{aligned}
$$

Finally, refinement equivalence ( $\square$ ) is defined for $C m d$ and Exec as refinement in both directions.

$$
\begin{aligned}
& -\sqsubseteq \__{1}: C m d \leftrightarrow C m d \\
& -\sqsubseteq \_: \text {Exec } \leftrightarrow \text { Exec } \\
& c_{1} \sqsubseteq c_{2} \Leftrightarrow c_{1} \sqsubseteq c_{2} \wedge c_{2} \sqsubseteq c_{1}
\end{aligned}
$$

### 5.1 Lattice properties

The refinement relation forms a chain-complete meet semi-lattice over Exec.

- ' $\square$ ' is a partial order because ' $\subseteq$ ' is a partial order on sets;
- meets exist: $e_{1} \sqcap e_{2}=e_{1} \cap e_{2}$;
- there is a unique bottom element, corresponding to the command abort (recall that exec abort $=\{\varnothing \mapsto \varnothing\})$.
Note that joins do not exist in general, because $e_{1} \cup e_{2}$ may not be a function; and there is no top element. For $e_{1} \sqcup e_{2}$ to be defined, we require that $e_{1} \cup e_{2}$ is a function and not simply a relation (i.e., $e_{1}$ returns the same state as $e_{2}$ for the states where both are defined). Even under that condition, the result is not simply $e_{1} \cup e_{2}$, because that could violate condition (1) of executions. Instead, we define $e_{1} \sqcup e_{2}$ by adding to $e_{1} \cup e_{2}$ mappings for all states that consist of bindings for which either $e_{1}$ or $e_{2}$ is defined.

$$
\begin{array}{|l}
-\Pi_{-}: \text {Exec } \times \text { Exec } \rightarrow \text { Exec } \\
-\sqcup \_: \text {Exec } \times \text { Exec } \rightarrow \text { Exec } \\
\hline e_{1} \sqcap e_{2}=e_{1} \cap e_{2} \\
\left(e_{1}, e_{2}\right) \in \operatorname{dom}\left(-\sqcup \_\right) \Leftrightarrow \\
\left.\quad \forall \text { ( } \because \operatorname{dom} e_{1} \cap \operatorname{dom} e_{2} \bullet e_{1}(s)=e_{2}(s)\right) \\
\left(e_{1}, e_{2}\right) \in \operatorname{dom}\left(-\sqcup \_\right) \Rightarrow \\
\quad e_{1} \sqcup e_{2}=\left(\lambda s: \mathbb{P}\left(\cup\left(\operatorname{dom} e_{1} \cup \operatorname{dom} e_{2}\right)\right) \bullet\left\{b: s \mid\left(e_{1} \cup e_{2}\right)(\{b\}) \neq\{ \}\right\}\right)
\end{array}
$$

In Appendix $D$, we show that $\Pi$ and $\sqcup$ (when the latter is well-defined) preserve the healthiness properties for executions.

If $e_{1} \sqsubseteq e_{2}$ then their join $e_{1} \sqcup e_{2}$ is defined (and is $e_{2}$ ). As a result, we can define joins for chains. We first define a chain of executions:

$$
\text { Chain }==\{e c: \mathbb{N} \rightarrow \text { Exec } \mid(\forall i: \mathbb{N} \bullet e c(i) \sqsubseteq e c(i+1))\}
$$

Note that we define every chain as an infinite sequence; a finite chain is simply modelled as an infinite chain in which the last element is repeated infinitely often. If $e c$ is a chain, then $e c(i) \sqcup e c(i+1)$ exists for all $i$ and is equal to $e c(i+1)$. We therefore define the join of a chain as follows.

$$
\begin{array}{|l}
\bigsqcup: \text { Chain } \rightarrow \text { Exec } \\
\hline \sqcup e c=\bigcup(\operatorname{ran} e c)
\end{array}
$$

## 6 Procedures and parameters

To simplify the semantics, we treat procedures, parameters and recursion as separate, though related, concerns (cf. [Mor88]).

### 6.1 Environments

To handle procedure definitions (parameterless, for now), we introduce a given set of procedure identifiers (PIdent) and an environment, which maps procedure identifiers to their corresponding procedure executions.

$$
\begin{aligned}
& {[\text { PIdent }]} \\
& \text { Env }==\text { PIdent } \rightarrow \text { Exec }
\end{aligned}
$$

Hence we change the definition of exec to add an environment parameter.
| exec: Env $\rightarrow$ Cmd $\rightarrow$ Exec
The definitions we have given in Section Th $^{\text {dot depend directly on the environment. The }}$ only change required is to add the environment parameter to the calls on exec for subcomponents, e.g., for an environment $\rho$ :
$\operatorname{exec}(\rho)(S \wedge T)=\operatorname{exec} \rho S \curvearrowright \operatorname{exec} \rho T$

### 6.2 Parametrised commands

To deal with parameters, we introduce the notion of a parametrised command. Given a variable $v$ and a command $c$, the expression $v:-c$ denotes the parametrised command ( $P C m d$ ) that, when provided a term argument $t$, behaves like $c[t / v]$. Non-variable arguments may be encoded using existential quantification and an equality conjunct in the body; multiple arguments may be encoded as a single argument. For example, the Prolog procedure p defined by 'p(Y,a(Z)) :- E' may be encoded as

$$
p==X:-\exists Y, Z \bullet\langle X=\operatorname{pair}(Y, a(Z))\rangle, E
$$

where $X$ is a fresh variable, and the functor pair just constructs the ordered pair of its arguments. Parametrised commands must not have any free variables; in order for $v:-c$ to be well-formed, we require that $c$ has no free variables other than $v$, i.e., free $(c) \subseteq\{v\}$ (any other variables in $c$ must be explicitly quantified).

### 6.3 Parametrised executions

The semantics of parametrised commands will be given by parametrised executions, which are simply functions mapping actual parameter terms to executions.

$$
\text { PExec }==\text { Term } \rightarrow \text { Exec }
$$

We can now deal with parameterised procedures. The only change needed is that the environment maps a procedure identifier to a parametrised execution:

$$
\text { Env }==\text { PIdent } \rightarrow \text { PExec }
$$

Just as we defined exec $c$ to give the meaning of a command $c$, we will now define pexec $p$ to give the meaning of a parametrised command $p$, taking into account the environment as well as the formal parameter.

For an actual parameter term $t$, the execution of a parametrised command $v:-c$ is a function that is defined for all states $s$ for which evaluation of $t$ is defined for all bindings in $s$, and for which $c$ is defined when $v$ is bound to the value of $t$ for each binding in $s$. We determine the result of executing the parametrised command by determining its result for each binding $b$ in $s$. If there is a binding $b^{\prime}$ in the result of executing the parametrised command on binding $b$ with $v$ updated with the corresponding value of the term $t$, then $b$ is in the resultant state. Recall that the result of executing a command on a singleton set is either that singleton set or the empty set.

$$
\begin{aligned}
& \text { pexec: Env } \rightarrow \text { PCmd } \rightarrow \text { PExec } \\
& \text { pexec }(\rho)(v:-c)=(\lambda t: \text { Term } \bullet \\
& \quad(\lambda s: \text { defined } t \mid \text { assign } v t s \in \operatorname{dom}(\operatorname{exec} \rho c) \bullet \\
& \quad\{b: s \mid(\operatorname{exec} \rho c)(\text { assign } v t\{b\}) \neq\{ \}\}))
\end{aligned}
$$

This definition only applies to non-recursive procedures. The definition of pexec for recursive procedures is presented in Section 7.

### 6.4 Refinement

We define refinement between parametrised executions $p_{1}$ and $p_{2}$ by requiring refinement for every possible value of the parameter. We also define refinement equivalence between parametrised executions in the obvious way.

$$
\begin{aligned}
& -\sqsubseteq \__{-} \text {PExec } \leftrightarrow \text { PExec } \\
& -\sqsubseteq \_ \text {PExec } \leftrightarrow \text { PExec } \\
& \hline\left(p_{1} \sqsubseteq p_{2}\right) \Leftrightarrow\left(\forall t: \text { Term } \bullet\left(p_{1} t\right) \sqsubseteq\left(p_{2} t\right)\right) \\
& \left(p_{1} \sqsubseteq p_{2}\right) \Leftrightarrow\left(\forall t: \text { Term } \bullet\left(p_{1} t\right) \sqsubseteq\left(p_{2} t\right)\right)
\end{aligned}
$$

Refinement between parametrised commands $p c_{1}$ and $p c_{2}$ is defined, as expected, in terms of refinement between the corresponding parametrised executions. Strictly speaking, the refinement should itself be parametrised by the environment. For simplicity, in the definitions below, we assume a fixed environment $\rho$.

```
_ \sqsubseteq _: PCmd ↔ PCmd
    \square _: PCmd ↔PCmd
(p\mp@subsup{c}{1}{}\sqsubseteqp\mp@subsup{c}{2}{})\Leftrightarrow\operatorname{pexec}\rhop\mp@subsup{c}{1}{}\sqsubseteq\operatorname{pexec}\rhop\mp@subsup{c}{2}{}
(p\mp@subsup{c}{1}{}\sqsubseteqp\mp@subsup{c}{2}{})\Leftrightarrow\operatorname{pexec}\rhop\mp@subsup{c}{1}{}\sqsubseteq\operatorname{pexec}\rhop\mp@subsup{c}{2}{}
```


### 6.5 Procedure call

A parametrised command may be directly applied to a term; the result is a command, the semantics of which is defined as follows.

$$
\operatorname{exec} \rho((v:-c)(t))=\operatorname{pexec} \rho(v:-c) t
$$

The syntax of a procedure call is $i d(t)$, where $t$ is a term and $i d$ is a procedure identifier. The parametrised command which is the definition of $i d$ in the environment is applied to $t$. If $i d$ is not defined in the environment, the result of a call on id is abort.
exec $\rho i d(t)=$ if $i d \in \operatorname{dom} \rho$ then $(\rho i d t)$ else (exec $\rho$ abort)

### 6.6 Lattice properties

The lattice properties of Exec can be lifted to PExec.

$$
\begin{aligned}
& \text { _ . _ : PExec } \times \text { PExec } \rightarrow \text { PExec } \\
& -\vdash^{\prime}: \text { PExec } \times \text { PExec } \rightarrow \text { PExec } \\
& p_{1} \text { ค } p_{2}=\left(\lambda t: \text { Term } \bullet p_{1} t \sqcap p_{2} t\right) \\
& \left(p_{1}, p_{2}\right) \in \operatorname{dom}\left(-\vdash_{-}\right) \Leftrightarrow\left(\forall t: \operatorname{Term} \bullet\left(p_{1} t, p_{2} t\right) \in \operatorname{dom}\left(-\sqcup_{-}\right)\right) \\
& \left(p_{1}, p_{2}\right) \in \operatorname{dom}\left(\_\vdash^{\prime}\right) \Rightarrow p_{1} \sqcup p_{2}=\left(\lambda t: T e r m \bullet p_{1} t \sqcup p_{2} t\right) \\
& \text { PChain }==\{p: \mathbb{N} \rightarrow \text { PExec } \mid(\forall i: \mathbb{N} \bullet p(i) \sqsubseteq p(i+1))\} \\
& \downarrow: \text { PChain } \rightarrow \text { PExec } \\
& \sqcup p=(\lambda t: \operatorname{Term} \bullet \bigsqcup(\lambda i: \mathbb{N} \bullet p i t))
\end{aligned}
$$

## 7 Recursion

If $i d$ is an identifier and $p$ is a parametrised command, possibly containing instances of $i d$, the recursion block re $i d \bullet p$ er is also a parametrised command. Intuitively, a call on the parametric recursion block re $i d \bullet v:-\ldots i d(t) \ldots$ er is similar to a call on the Prolog procedure defined by id(V) :- ... id(T) ....

### 7.1 Semantics of recursion blocks

A recursion block embeds one or more recursive calls on the block inside a context. Thus, a context is a function from one parametrised command (the recursive call) to another (the entire body of the recursion block):

$$
C t x==P E x e c \rightarrow \text { PExec }
$$

Intuitively, to represent $v:-\operatorname{Con}[P(t)]$, we define the context $\mathcal{C}: C t x$ such that $\mathcal{C}(P)=$ $v:-\operatorname{Con}[P(t)]$. In this example, $\mathcal{C}$ embeds a call on $P$ in the context given by $C o n$, and also provides $P$ with the parameter $t$. Formally, we define a function that extracts a context from a recursion block.

$$
\begin{aligned}
& \text { context: Env } \rightarrow P C m d \rightarrow C t x \\
& \text { context }(\rho)(\text { re } i d \bullet p c \text { er })=(\lambda p: P E x e c \bullet \operatorname{pexec}(\rho \oplus\{i d \mapsto p\})(p c))
\end{aligned}
$$

This function is partial because it is only defined for $P C m d$ s that are recursion blocks.
To define the semantics of recursion blocks we use a fix-point construction, which is defined for all monotonic contexts (see Knaster-Tarski Theorem [Nel89]). We first define the set of monotonic contexts.

$$
M C t x==\left\{\mathcal{C}: C t x \mid\left(\forall p, p^{\prime}: \text { PExec } \bullet\left(p \sqsubseteq p^{\prime}\right) \Rightarrow\left(\mathcal{C}(p) \sqsubseteq \mathcal{C}\left(p^{\prime}\right)\right)\right)\right\}
$$

This monotonicity property holds for every context $\mathcal{C}$ that can be constructed in our language (see Appendix E).

The least fix-point of a context $\mathcal{C}$ is given by fix $\mathcal{C}$, where

$$
\begin{aligned}
& \text { fix: MCtx } \rightarrow \text { PExec } \\
& (\forall \mathcal{C}: M C t x \bullet \text { fix } \mathcal{C} \sqsubseteq \mathcal{C}(\text { fix } \mathcal{C})) \\
& (\forall \mathcal{C}: M C t x ; p: \text { PExec } \bullet(\mathcal{C}(p) \sqsubseteq p) \Rightarrow(\text { fix } \mathcal{C} \sqsubseteq p))
\end{aligned}
$$

Hence the meaning of a recursion block is the least fix-point of the context corresponding to the recursion block.

```
pexec ( }\rho)(\mathrm{ re id • pcer ) = fix (context ( }\rho)(\mathrm{ re id • pcer ) )
```


### 7.2 Constructing the fix-point

To simplify this and the next section, we use the syntax of parametrised commands to stand for their PExecs and we assume a fixed environment $\rho$, which is augmented with a single recursive definition.

The least defined command is abort. The least defined parametrised command is a parametrised command with abort as its body:

$$
\text { abort }_{1}==v:- \text { abort }
$$

For a recursion based on a monotonic context $\mathcal{C}$, we construct the sequence of programs:

$$
\begin{aligned}
& p c: \text { PChain } \\
& p c=\left(\lambda i: \mathbb{N} \bullet \mathcal{C}^{i}\right. \\
& \text { abort } \left.\left._{1}\right)\right)
\end{aligned}
$$

That is, we have

$$
\begin{aligned}
& p c(0)=\mathcal{C}^{0}\left(\text { abort }_{1}\right)=\text { abort }_{1} \\
& p c(i+1)=\mathcal{C}^{i+1}\left(\text { abort }_{1}\right)=\mathcal{C}(p c(i))
\end{aligned}
$$

The sequence $p c$ forms a chain ordered by $\sqsubseteq$ and has a join $(\downarrow p c)$. By the Limit Theorem [Nel89], $\cdot \downarrow p c=$ fix $\mathcal{C}$, as long as $\mathcal{C}$ is a chain-continuous function:

For a chain $p c$ for which the join $\lfloor p c$ exists, a function $\mathcal{C}$ is chain-continuous provided $\mathcal{C}(\downarrow \mid p c)=\downharpoonright(\lambda i: \mathbb{N} \bullet \mathcal{C}(p c(i)))$.

All the contexts $\mathcal{C}$ that can be constructed in our language are chain-continuous (see Appendix (F).

### 7.3 Example

Consider the recursive, parametric program defined by

$$
\text { nats }==(\text { re } n \bullet X:-\langle X=0\rangle \vee(\exists Y \bullet\langle X=s(Y)\rangle), n(Y) \text { er })
$$

We call the monotonic context $\mathcal{N}$, which in this case is

$$
\mathcal{N}=(\lambda n: \text { PExec } \bullet X:-\langle X=0\rangle \vee(\exists Y \bullet\langle X=s(Y)\rangle), n(Y))
$$

The meaning of nats is given by the join of the chain $n c=\left(\lambda i: \mathbb{N} \bullet \mathcal{N}^{i}\left(\mathbf{a b o r t}_{1}\right)\right)$, in which:

$$
\begin{aligned}
n c(0)= & \mathcal{N}^{0}\left(\mathbf{a b o r t}_{1}\right) \\
= & \mathbf{a b o r t}_{1} \\
n c(1)= & \mathcal{N}^{1}\left(\text { abort }_{1}\right) \\
= & X:-\langle X=0\rangle \vee(\exists Y \bullet\langle X=s(Y)\rangle, \text { abort }) \\
n c(2) & \mathcal{N}^{2}\left(\text { abort }_{1}\right) \\
= & X:-\langle X=0\rangle \vee(\exists Y \bullet\langle X=s(Y)\rangle, \\
& \quad(\langle Y=0\rangle \vee(\exists Z \bullet\langle Y=s(Z)\rangle, \text { abort }))) \\
= & X:-\langle X=0\rangle \vee\langle X=s(0)\rangle \vee(\exists Y \bullet\langle X=s(s(Y))\rangle, \text { abort }) \\
\vdots & \\
\text { pexec nats }= & \text { fix } \mathcal{N} \\
= & \lfloor n c \\
= & X:-\left\langle\exists i \in \mathbb{N} \bullet X=s^{i}(0)\right\rangle \vee\left(\exists Y \bullet\left\langle X=s^{\omega}(Y)\right\rangle, \text { abort }\right)
\end{aligned}
$$

The term $s^{\omega}(Y)$ is an abuse of notation, because our definition of terms in Section 3.4 does not permit such infinitary terms.

To interpret the meaning of exec nats $(X)$, we must consider the set of values Val in our domain of discourse. If we only allow finitely representable values in our domain of discourse, as with standard Herbrand interpretations, then

$$
\text { exec } \operatorname{nats}(X)=(\lambda s: \text { State } \bullet\{b: s \mid b X \in\{0, s(0), \ldots\}\})
$$

This execution can lead to an infinite set of bindings for $X$.
To define the set of finitely representable values $F R V a l$, we first consider any set $V$ that satisfies the fix-point equation

$$
V=\{f: \text { Fun; s: seq } V \mid s \in \operatorname{dom}(\operatorname{apply} f) \bullet \text { apply } f s\}
$$

Such a set $V$ includes the set of all finitely representable values that can be constructed from the set of atoms and applications of function symbols. However, each set $V$ may also contain other values. The set of finitely representable values (FRVal) is then the least fix-point of this equation.

$$
\begin{aligned}
F R V a l & =\bigcap\{V: \mathbb{P} \text { Val } \mid \\
V & =\{f: \text { Fun; s: seq } V \mid s \in \operatorname{dom}(\text { apply } f) \bullet \text { apply } f s\}\}
\end{aligned}
$$

If, on the other hand, we also allow infinitary terms (e.g., by allowing rational trees [Col82], or by taking the standard interpretation of 0 as the number 0 and $s$ as the successor function, with $s^{\omega}(0)$ representing infinity), then

```
exec nats(X)=(\lambdas:State
    | ᄀ(\existsb:s; Y:Term\bulletb X = s}\mp@subsup{}{}{\omega}(Y)
    - {b:s|bX\in{0,s(0),\ldots}})
```

If we take this interpretation and allow $s^{\omega}(0)$ in the universe, then for a state $s$ with $X$ unbound there is a $b \in s$ such that $b X=s^{\omega}(Y)$, and hence $s$ is not in the domain of nats.

In this paper we do not explicitly handle the definition of a set of mutually recursive procedure. Such a set can always be encoded as a single procedure and hence given a semantics via this encoding. For example, a set of mutually recursive procedures

$$
\begin{aligned}
& p_{1}==V_{1}:-C_{1} \\
& \vdots \\
& p_{n}==V_{n}:-C_{n}
\end{aligned}
$$

may be encoded as a single procedure

$$
\begin{aligned}
p==(I, & \left.V_{1}, \ldots, V_{n}\right):- \\
& \langle I=1\rangle, C_{1} \vee \\
& \vdots \\
& \langle I=n\rangle, C_{n}
\end{aligned}
$$

where the parameter $I$ encodes which of the original procedures is being called and the parameter names $V_{1}, \ldots, V_{n}$ are assumed to be distinct. A call of the form $p_{1}(t)$ is then encoded as $p\left(1, t,{ }_{-}, \ldots,-\right)$.

## 8 Recursion introduction

### 8.1 Refinement law

Suppose pc: PCmd is a parametrised command; (_ $\left.\prec{ }^{\prime}\right)$ : Term $\leftrightarrow$ Term is a well-founded relation; and $i d$ is a fresh name. The following law can be used to introduce a recursion block into a refinement.

$$
p c \sqsubseteq(\mathbf{r e} i d \bullet v:-\{\forall y: \operatorname{Term} \bullet y \prec v \Rightarrow p c(y) \sqsubseteq i d(y)\}, p c(v) \mathbf{e r})
$$

## Proof

The theorem of well-founded induction states that, for some property $\Phi$ of terms, and wellfounded order (_ $\prec$ _) defined on those terms:

$$
\begin{equation*}
\frac{\forall x: \operatorname{Term} \bullet(\forall y: \operatorname{Term} \bullet y \prec x \Rightarrow \Phi(y)) \Rightarrow \Phi(x)}{\forall x: \text { Term } \bullet \Phi(x)} \tag{WFI}
\end{equation*}
$$

Allow $A(v)$ to abbreviate $\{\forall y$ : Term $\bullet \prec \prec v \Rightarrow p c(y) \sqsubseteq i d(y)\}$. We want to prove

$$
\begin{equation*}
p c(x) \sqsupseteq(\mathbf{r e} i d \bullet v:-A(v), p c(v) \mathbf{e r})(x) \tag{7}
\end{equation*}
$$

for all terms $x$. Take $\Phi(x)$ in (WFI) to be this property. The inductive hypothesis allows us to assume that

$$
\begin{aligned}
& \forall y: T e r m \bullet y \prec x \Rightarrow \\
& \quad p c(y) \sqsupseteq(\mathbf{r e} i d \bullet v:-A(v), p c(v) \mathbf{e r})(y)
\end{aligned}
$$

Starting with the right-hand side of (7) and expanding the definition of $A(v)$ :

```
    \((\mathbf{r e} i d \bullet v:-\{\forall y: T e r m \bullet y \prec v \Rightarrow p c(y) \sqsubseteq i d(y)\}, p c(v) \mathbf{e r})(x)\)
```

```
    unroll recursion: fix \(\mathcal{C} \sqsupseteq C(\) fix \(\mathcal{C})\)
    ( \(v:-\{\forall y:\) Term \(\bullet y \prec v \Rightarrow\)
    \(p c(y) \sqsubseteq(\mathbf{r e} i d \bullet v:-A(v), p c(v) \mathbf{e r})(y)\}, p c(v))(x)\)
\(\square\) parameter application
    \(\{\forall y: T e r m \bullet y \prec x \Rightarrow p c(y) \sqsubseteq(\mathbf{r e} i d \bullet v:-A(v), p c(v) \mathbf{e r})(y)\}, p c(x)\)
\(\square\) inductive hypothesis; eliminate true assumption
    \(p c(x)\)
```


### 8.2 Example

To demonstrate the refinement law, we derive a recursive implementation of a factorial function from the definition of factorial, which is:

$$
\begin{aligned}
& 0!=1 \\
& (n+1)!=n!\times(n+1), \text { for } n \geq 0
\end{aligned}
$$

The derivation is written using the structured calculational proof term-rewriting style of [BGVW98], in which a mark $\llcorner$ thus $\lrcorner$ indicates a subterm to be rewritten at a higher level of indentation.

Our program can assume $U \in \mathbb{N}$ and must establish $V=U$ !:
$1 \bullet(U, V):-\{U \in \mathbb{N}\},\langle V=U!\rangle$
$\square$ introduce recursion: < is well-founded on $\mathbb{N}$
ref $f(U, V)$ :-
$\left\{\forall U^{\prime}, V^{\prime} \bullet U^{\prime}<U \Rightarrow\left\{U^{\prime} \in \mathbb{N}\right\},\left\langle V^{\prime}=U^{\prime}!\right\rangle \sqsubseteq f\left(U^{\prime}, V^{\prime}\right)\right\}$, $\{U \in \mathbb{N}\}$, $\llcorner\langle V=U!\rangle\lrcorner$
er
Now the refinement continues, but the assumptions prior to the specification may be assumed in the context of refining the specification [CHNS.97]:

Assumption 1: $\forall U^{\prime}, V^{\prime} \bullet U^{\prime}<U \Rightarrow\left\{U^{\prime} \in \mathbb{N}\right\},\left\langle V^{\prime}=U^{\prime}!\right\rangle \sqsubseteq f\left(U^{\prime}, V^{\prime}\right)$
Assumption 2: $U \in \mathbb{N}$
2 - $\langle V=U!\rangle$
$\sqsubseteq \quad[$ case analysis, using Assumption $2(U \in \mathbb{N})]$
$\langle U=0 \vee U>0\rangle,\langle V=U!\rangle$
$\sqsubseteq \quad$ [distributive laws, definition of factorial]

$$
(\langle U=0\rangle,\langle V=1\rangle) \vee(\langle U>0\rangle,\llcorner\langle V=U!\rangle\lrcorner)
$$

Again, we obtain an contextual assumption, this time from the specification $\langle U>0\rangle$ immediately prior to the specification we are refining:

$$
\begin{aligned}
& \text { Assumption 3: } U>0 \\
& 3 \bullet\langle V=U!\rangle \\
& \sqsubseteq \text { [integers, assumptions } 2 \text { and } 3 \text {, definition of factorial] } \\
& \left\langle\exists U^{\prime}, V^{\prime} \bullet U=U^{\prime}+1 \wedge V^{\prime}=U^{\prime}!\wedge V=V^{\prime} \times U\right\rangle \\
& \sqsubseteq \text { [distributive laws] } \\
& \exists U^{\prime}, V^{\prime} \bullet\left\langle U=U^{\prime}+1\right\rangle,\left\langle V^{\prime}=U^{\prime}!\right\rangle,\left\langle V=V^{\prime} \times U\right\rangle \\
& \sqsubseteq \text { [assumption after spec: } U \in \mathbb{N} \wedge U>0 \wedge U=U^{\prime}+1 \Rightarrow U^{\prime} \in \mathbb{N} \text { ] } \\
& \exists U^{\prime}, V^{\prime} \bullet\left\langle U=U^{\prime}+1\right\rangle,\left\llcorner\left\{U^{\prime} \in \mathbb{N}\right\},\left\langle V^{\prime}=U^{\prime}!\right\rangle\right\lrcorner,\left\langle V=V^{\prime} \times U\right\rangle \\
& \text { Assumption 4: } U=U^{\prime}+1 \\
& 4 \bullet\left\{U^{\prime} \in \mathbb{N}\right\},\left\langle V^{\prime}=U^{\prime}!\right\rangle \\
& \left.\sqsubseteq \text { [assumption 1: } U^{\prime}<U \text { by assumptions } 2,3,4\right] \\
& f\left(U^{\prime}, V^{\prime}\right)
\end{aligned}
$$

Putting it all back together and removing assumptions, we get:

```
\sqsubseteq \operatorname { r e f } \bullet ( U , V ) : -
    \langleU=0\rangle,\langleV=1\rangle\vee
    \langleU>0\rangle,(\exists\mp@subsup{U}{}{\prime},\mp@subsup{V}{}{\prime}\bullet\langleU=\mp@subsup{U}{}{\prime}+1\rangle,f(\mp@subsup{U}{}{\prime},\mp@subsup{V}{}{\prime}),\langleV=\mp@subsup{V}{}{\prime}\timesU\rangle)
    er
```

To translate this (informally) into Prolog, we:

- turn the recursion block into an implicitly recursive procedure;
- implement the arithmetic specifications using is;
- express the disjunction using separate clauses;
- make the existential quantification implicit.

The result is:

```
f(U,V) :- U=0, V=1.
f(U,V) :- U>0, U1 is U-1, f(U1,V1), V is V1*U.
```


## 9 Conclusions

We have presented a refinement calculus for logic programming. The calculus contains a widespectrum logic programming language, including both specification and executable constructs. We defined a declarative semantics for this wide-spectrum language based on executions, which are partial functions from states to states. This definition is similar to the definition of predicate transformers used in some imperative refinement, e.g., [BvW98]. As well as specifications and program composition constructs, the semantics defines the meaning of procedures, parameters, and recursion. Finally, there is a formal notion of refinement over programs in the widespectrum language, and a refinement law for recursive procedures.

In earlier work [HNS.97], we defined the meaning of a command $c$ in the wide-spectrum language by a pair of predicates: ok.c is a predicate defining the initial condition under which execution of the command is well-defined (essentially representing the assumptions the command makes about its context), while ef.c is the effect of the command, provided that its assumptions are satisfied. The two semantics are closely related in that, for every command $c$,

$$
\begin{aligned}
& \overline{o k . c}=\bigcup \operatorname{dom}(\operatorname{exec} c) \\
& \overline{o k . c \wedge e f . c}=(\operatorname{exec} c)(\overline{o k . c})
\end{aligned}
$$

However, the earlier paper lacked a rigorous treatment of procedures, parameters, and recursion, and the semantics we use here is chosen to facilitate the presentation of those concepts.

A tool has been developed to support the logic programming refinement calculus [CHNS.97]. However, this tool is based around the semantics presented in [HNS97]. We are currently working on tool support for the refinement calculus based on the new semantics.

Our goal of systematically developing logic programs from specifications also underlies the work of Deville [Dev90]. However, the main difference is that Deville's approach to program development is mostly informal, whereas our approach is fully formal. A second distinction is that Deville's approach concentrates on the development of individual procedures. By using a wide-spectrum language, our approach blurs the distinction between a logic description and a logic program. For example, general predicates may appear anywhere within a program, and the refinement rules allow them to be transformed within that context. Similarly, programming language constructs may be used and transformed at any point.

The motivation for the work by Hoare [Hoa(0)] is to come up with unifying theories for logic programming, which is quite different from the motivation for our work. However, the language constructs he considers and the semantics he uses are both very similar to the ones we use.

## References

[Bac88] R. J. R. Back. A calculus of refinements for program derivations. Acta Informatica, 25:593-624, 1988.
[BGvW98] Ralph Back, Jim Grundy, and Joakin von Wright. Structured calculational proof. Formal Aspects of Computing, 9:469-483, 1998.
[BvW98] R. J. Back and J. von Wright. Refinement Calculus: A Systematic Introduction. Springer-Verlag, 1998.
[CHNS97] Robert Colvin, Ian Hayes, Ray Nickson, and Paul Strooper. A tool for logic program refinement. In D. Duke and A. Evans, editors, 2nd BCS-FACS Northern Formal Methods Workshop, Electronic Workshops in Computing. Springer Verlag, 1997. Also available as Technical Report UQ-SVRC-97-32, Software Verification Research Centre, University of Queensland.
[Col82] A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S. A. Tarnlund, editors, Logic Programming, pages 231-251. Academic Press, 1982.
[Dev90] Yves Deville. Logic Programming: Systematic Program Development. AddisonWesley, 1990.
[HNS97] Ian Hayes, Ray Nickson, and Paul Strooper. Refining specifications to logic programs. In J. Gallagher, editor, Logic Program Synthesis and Transformation. Proceedings of the 6th International Workshop, LOPSTR'96, Stockholm, Sweden, August 1996, volume 1207 of Lecture Notes in Computer Science, pages 1-19. Springer Verlag, 1997. Also available as Technical Report UQ-SVRC-96-34, Software Verification Research Centre, University of Queensland.
[Hoa00] Tony Hoare. Unifying theories for logic programming. In John Lloyd, editor, Australian Workshop on Computational Logic, pages 31-56, February 2000.
[Man74] Zohar Manna. Mathematical Theory of Computation. McGraw-Hill, 1974.
[Mor88] Carroll Morgan. Procedures, parameters and abstraction: Separate concerns. Science of Computer Programming, 11:17-27, 1988.
[Nel89] Greg Nelson. A generalization of Dijkstra's calculus. ACM Transactions on Programming Languages and Systems, 11(4):517-561, October 1989.
[Spi92] J. M. Spivey. The Z Notation: A Reference Manual (second edition). Prentice Hall International, 1992.

## A Summary of notation

The following table summarises the important types defined in this paper. We give the basetype definitions of the types here (ignoring restrictions), and the names used (possibly decorated) for values in those types.

The final column gives the section where the full definition may be found.

| Type | Base type | Typical value | Section |
| :---: | :---: | :---: | :---: |
| Cmd | given | $c$ | $\square$ |
| Var | given | $u, v, X, Y, Z$ | 3.11 |
| Val | given | $x, y$ | [3.] |
| Fun | given | $f$ | 3.11 |
| Bnd | Var $\rightarrow$ Val | $b$ | 3.2 |
| State | $\mathbb{P}$ Bnd | $s$ | $3.3]$ |
| Term | $v a r T\langle\langle V a r\rangle\rangle \mid$ fun $T\langle\langle$ Fun $\times$ seq Term $\rangle\rangle$ | $t$ | $3.4]$ |
| Exec | State $\rightarrow$ State | $e$ | 4.11 |
| Chain | $\mathbb{N} \rightarrow$ Exec | ec | 5.15 |
| PIdent | given | id | 5.11 |
| PCmd | given | $p c$ | 5.2 |
| PExec | Term $\rightarrow$ Exec | $p$ | 6.3 |
| Env | PIdent $\rightarrow$ PExec | $\rho$ | 5.3 |
| PChain | $\mathbb{N} \rightarrow$ PExec | $p c$ | 6.6] |
| Ctx | PExec $\rightarrow$ PExec | $\mathcal{C}$ | 7. |
| MCtx | PExec $\rightarrow$ PExec | $\mathcal{C}$ | T.1] |
| FRVal | $\mathbb{P}$ Val | - | 4.3 |

The following table gives the signatures of the important functions defined in the paper, and a reference to the section where the full definition may be found.

| Function | Signature | Section |
| :---: | :---: | :---: |
| arity | Fun $\rightarrow \mathbb{N}$ | [3.] |
| apply | Fun $\rightarrow$ (seq Val $\rightarrow$ Val $)$ | 3.15 |
| eval | Term $\rightarrow$ (Bnd $\rightarrow$ Val $)$ | [3.4 |
| defined | Term $\rightarrow \mathbb{P}$ State | [3.4] |
| assign | Var $\rightarrow$ Term $\rightarrow$ State $\rightarrow$ State | 3.4 |
| free | Term $\rightarrow \mathbb{P}$ Var | 3.4 |
| free | Crmd $\rightarrow \mathbb{P}$ Var | 3.4 |
| exec | Cmd $\rightarrow$ Exec | 4.2 |
| $ค, \cup,{ }_{9}$ | Exec $\times$ Exec $\rightarrow$ Exec | 4.4 |
| unbind | Var $\rightarrow$ State $\rightarrow$ State | 4.5 |
| exists, forall | Var $\rightarrow$ Exec $\rightarrow$ Exec | 4.5 |
| $\sqsubseteq, \sqsupseteq$ | Exec $\leftrightarrow$ Exec | 回 |
| $\sqsubseteq, \square$ | $C m d \leftrightarrow C m d$ | 回 |
| $\square$ | Exec $\times$ Exec $\rightarrow$ Exec | 5.1 |
| $\sqcup$ | Exec $\times$ Exec $\rightarrow$ Exec | 5.1 |
| $\sqcup$ | Chain $\rightarrow$ Exec | 5.1 |
| exec | Env $\rightarrow$ Cmd $\rightarrow$ Exec | 6.1 |
| pexec | Env $\rightarrow$ PCmd $\rightarrow$ PExec | 6.31, 7.11 |
| $\sqsubseteq, \square$ | PExec $\leftrightarrow$ PExec | 6.4 |
| $\sqsubseteq, \square$ | $P C m d \leftrightarrow P C m d$ | 6.4 |
| ค | PExec $\times$ PExec $\rightarrow$ PExec | 6.6 |
| $\stackrel{\rightharpoonup}{\bullet}$ | PExec $\times$ PExec $\rightarrow$ PExec | 6.6 |
| $\downarrow$ | PChain $\rightarrow$ PExec | 5.6 |
| context | Env $\rightarrow$ PCmd $\rightarrow$ Ctx | T.] |
| fix | MCtx $\rightarrow$ PExec | 7. 7 |

## B Definition of Exec

In this appendix we investigate equivalent forms of the definition of Exec. Some of the equivalent forms are used to simplify later proofs. In Section 4.1, we define the set of executions as follows.
Definition B. 1 Program executions

$$
\begin{align*}
\text { Exec }==\{ & e: \text { State } \rightarrow \text { State } \mid \\
& \operatorname{dom} e=\mathbb{P}\{b: \text { Bnd } \mid\{b\} \in \operatorname{dom} e\} \wedge  \tag{8}\\
& (\forall s: \operatorname{dom} e \bullet e(s) \subseteq s) \wedge  \tag{9}\\
& (\forall s: \operatorname{dom} e \bullet e(s)=\bigcup\{b: s \bullet e(\{b\})\})\} \tag{10}
\end{align*}
$$

Exec properties ( $\mathbb{( \square )}$ ) and ( $\mathbb{Q}^{(1)}$ tell us that:

$$
\begin{equation*}
(\forall e: \text { Exec } \bullet\} \in \operatorname{dom} e \wedge e(\})=\{ \}) \tag{11}
\end{equation*}
$$

Applying property ( $\boldsymbol{( 1 )}$ ) to singleton bindings gives us:

$$
\begin{equation*}
(\forall b: B n d \bullet\{b\} \in \operatorname{dom} e \Rightarrow(e(\{b\})=\{ \} \vee e(\{b\})=\{b\}) \tag{12}
\end{equation*}
$$

We also note the equivalence $e(\{b\}) \neq\{ \} \Leftrightarrow e(\{b\})=\{b\}$.

## B. 1 Equivalent forms of Exec property (8)

Exec property (8) can be restated as any of:

$$
\begin{align*}
& \left(\forall s: \operatorname{dom} e \bullet\left(\forall s^{\prime}: \mathbb{P} s \bullet s^{\prime} \in \operatorname{dom} e\right)\right) \wedge(\forall s s: \mathbb{P}(\operatorname{dom} e) \bullet \bigcup s s \in \operatorname{dom} e)  \tag{13}\\
& \operatorname{dom} e=\mathbb{P}(\bigcup \operatorname{dom} e) \tag{14}
\end{align*}
$$

To prove the equivalence we make use of the following lemmas.
Lemma B. 2 For all $s$ in State: Exp and all properties $P$ of bindings:

$$
s \in \mathbb{P}\{b: B n d \mid P(b)\} \Leftrightarrow(\forall b: s \bullet P(b))
$$

Proof:

$$
\begin{aligned}
& s \in \mathbb{P}\{b: B n d \mid P(b)\} \\
\Leftrightarrow & \text { membership in power set } \\
& s \subseteq\{b: B n d \mid P(b)\} \\
\Leftrightarrow & \text { rewrite } \\
& (\forall b: s \bullet b \in\{b: B n d \mid P(b)\}) \\
\Leftrightarrow & \text { simplification } \\
& (\forall b: s \bullet P(b)) \quad \square
\end{aligned}
$$

Lemma B. 3 For all $s$ in State: Exp and $e$ in Exec:

$$
s \in \operatorname{dom} e \Leftrightarrow(\forall b: s \bullet\{b\} \in \operatorname{dom} e)
$$

Proof:

$$
\begin{aligned}
& s \in \operatorname{dom} e \\
\Leftrightarrow & \text { Exec property }(8) \\
& s \in \mathbb{P}\{b: \text { Bnd } \mid\{b\} \in \operatorname{dom} e\} \\
\Leftrightarrow & \text { Lemma B.2 } \\
& (\forall b: s \bullet\{b\} \in \operatorname{dom} e)
\end{aligned}
$$

Theorem B. 4 Equations (8) and (1조) are equivalent.
Proof of $(\mathbb{8}) \Rightarrow(\boxed{\pi})$ : We prove that both conjuncts of ([3]) simplify to true.
Simplifying the first conjunct of ([.3):

$$
\begin{aligned}
& \left(\forall s: \operatorname{dom} e \bullet\left(\forall s^{\prime}: \mathbb{P} s \bullet s^{\prime} \in \operatorname{dom} e\right)\right) \\
\Leftrightarrow & \text { restate } \\
& \left(\forall s, s^{\prime}: \text { State } \bullet s \in \operatorname{dom} e \Rightarrow\left(s^{\prime} \subseteq s \Rightarrow s^{\prime} \in \operatorname{dom} e\right)\right) \\
\Leftrightarrow & \text { Lemma B.3 } \\
& \left(\forall s, s^{\prime}: \text { State } \bullet(\forall b: s \bullet\{b\} \in \operatorname{dom} e) \Rightarrow\left(s^{\prime} \subseteq s \Rightarrow\left(\forall b: s^{\prime} \bullet\{b\} \in \operatorname{dom} e\right)\right)\right) \\
\Leftarrow & \text { definition of subset }
\end{aligned}
$$

Simplifying the second conjunct of ([.3):

```
    (\forallss:\mathbb{P}(\operatorname{dom}e)\bullet\bigcupss\in\operatorname{dom}e)
\Leftrightarrow restate, Lemma B.3
    (\forallss:\mathbb{P State \bullet ss }\in\mathbb{P}(\operatorname{dom}e)=>(\forallb:\bigcupss\bullet{b}\in\operatorname{dom}e))
\Leftrightarrow}\mathrm{ definition of }\mathbb{P
    (\forallss:\mathbb{P State \bullet ss \subseteq dom e=>(\forallb:\bigcupss\bullet{b}\in\operatorname{dom}e))})=\mp@code{\})
\Leftrightarrow definition of subset
    (\forallss:\mathbb{P State \bullet (\foralls:ss\bullet s\in\operatorname{dom}e)}=>(\forallb:\ss\bullet{b}\in\operatorname{dom}e))
\Leftrightarrow Lemma B.3
    (\forallss:\mathbb{P}\mathrm{ State • ( }\foralls:ss\bullet(\forallb:s\bullet{b}\in\operatorname{dom}e))=>(\forallb:\bigcupss\bullet{b}\in\operatorname{dom}e))
\Leftarrow \mp@code { d e f i n i t i o n ~ o f ~ U }
```

Proof of ( $\mathbb{\square}$ ) $\Rightarrow$ (四):
We rewrite ( $\mathbb{8}$ ) using Lemma B.2:

```
    \((\forall s:\) State \(\bullet s \in \operatorname{dom} e \Leftrightarrow s \in \mathbb{P}\{b:\) Bnd \(\mid\{b\} \in \operatorname{dom} e\})\)
\(\Leftrightarrow\) from Lemma B. 2
    \((\forall s\) : State \(\bullet s \in \operatorname{dom} e \Leftrightarrow(\forall b: s \bullet\{b\} \in \operatorname{dom} e))\)
```

We split the equivalence into two implications, and prove that ([3) implies each. Going left to right is trivial since the first conjunct of ([]3) implies

$$
(\forall s: \operatorname{dom} e \bullet(\forall b: s \bullet\{b\} \in \operatorname{dom} e))
$$

Now from the right hand side of the equivalence we know that $\{b: s \bullet\{b\}\} \in \mathbb{P}(\operatorname{dom} e)$. Therefore from the second conjunct of ([3) we know that

$$
\bigcup\{b: s \bullet\{b\}\} \in \operatorname{dom} e
$$

Since $\bigcup\{b: s \bullet\{b\}\}=s$, it follows that $s \in \operatorname{dom} e$.
Theorem B. 5 Equations (B) and (14) are equivalent.
Proof of $([14) \Rightarrow(\mathbb{8})$ :

$$
\operatorname{dom} e
$$

$=$ from (내)
$\mathbb{P}(\bigcup$ dom $e)$
$=$ rewrite as comprehension
$\mathbb{P}\{b: B n d \mid b \in(\bigcup \operatorname{dom} e)\}$
$=$ membership in union
$\mathbb{P}\{b:$ Bnd $\mid\{b\} \in \mathbb{P}(\bigcup$ dom $e)\}$
$=$ from ([4)
$\mathbb{P}\{b: B n d \mid\{b\} \in \operatorname{dom} e\}$
Proof of $(\mathbb{8}) \Rightarrow(14)$ :
Before progressing we note the following property for all sets ss:

$$
\begin{equation*}
\bigcup(\mathbb{P} s s)=s s \tag{15}
\end{equation*}
$$

Proof:

```
    dome
= from ($)
    P}{b:Bnd |{b}\in\operatorname{dom}e
= from ([.5)
    P}(\bigcup(\mathbb{P}{b:Bnd |{b}\in\operatorname{dom}e})
    = from ($)
    P}(\bigcup\mathrm{ dom e)
```


## B. 2 Equivalent forms of Exec property (10)

Exec property (10) can be restated as any of:

$$
\begin{align*}
& (\forall s: \operatorname{dom} e \bullet e(s)=\{b: s \mid e(\{b\}) \neq\{ \}\})  \tag{16}\\
& \left(\forall s: \operatorname{dom} e \bullet\left(\forall s^{\prime}: \mathbb{P} s \bullet e\left(s^{\prime}\right)=e(s) \cap s^{\prime}\right)\right)  \tag{17}\\
& (\forall s s: \mathbb{P}(\operatorname{dom} e) \bullet e(\bigcup s s)=\bigcup\{s: s s \bullet e(s)\} \tag{18}
\end{align*}
$$

We show $(\mathbb{\square}) \Leftrightarrow(\mathbb{\square}) \Leftrightarrow(\mathbb{\square}) \Leftrightarrow(\mathbb{\square})$ ) given that Exec property (9) holds.
Theorem B. 6 Exec property (9) implies the equivalence of (110) and (106).
Proof:

$$
\begin{aligned}
& c \in \bigcup\{b: s \bullet e(\{b\})\} \\
\Leftrightarrow & \text { membership in union } \\
& (\exists b: s \bullet c \in e(\{b\})) \\
\Leftrightarrow & \text { from } \operatorname{Exec} \text { property }(9) \\
& (\exists b: s \bullet c=b \wedge e(\{b\}) \neq\{ \}) \\
\Leftrightarrow & \text { one-point rule } \\
& c \in s \wedge e(\{c\}) \neq\{ \} \\
\Leftrightarrow & \text { rewrite } \\
& c \in\{b: s \mid e(\{b\}) \neq\{ \}\} \quad \square
\end{aligned}
$$

Now we will use ([6) interchangeably with the original definition of Exec property (10) in the following proofs.

Theorem B. 7 Exec property (9) implies the equivalence of (176) and (17).
Proof of $(\mathbb{1 6 )}) \Rightarrow\left([17)\right.$, for $s \in$ dome and $s^{\prime} \in \mathbb{P} s$ :

$$
e(s) \cap s^{\prime}
$$

$=$ from (166)
$\{b: s \mid e(\{b\}) \neq\{ \}\} \cap s^{\prime}$
$=$ from $s^{\prime} \in \mathbb{P} s$
$\left\{b: s^{\prime} \mid e(\{b\}) \neq\{ \}\right\}$
$=$ from (띠)
$e\left(s^{\prime}\right)$
Proof of $(\sqrt{17}) \Rightarrow(\sqrt{16})$ :
Note that, from Exec property (9), we know ([7) implies

$$
\begin{equation*}
(\forall b: s \bullet e(\{b\})=e(s) \cap\{b\}) \tag{19}
\end{equation*}
$$

Assume $s \in \operatorname{dom} e$ :

$$
\begin{aligned}
& \{b: s \mid e(\{b\}) \neq\{ \}\} \\
= & \text { from (ᄄ⿹)}) \\
& \{b: s \mid e(s) \cap\{b\} \neq\{ \}\} \\
= & \text { simplify } \\
& \{b: s \mid b \in e(s)\} \\
= & \text { from Exec property (9) } \\
& e(s) \quad \square
\end{aligned}
$$

Theorem B. 8 Equations (10) and (18) are equivalent.
Proof of $(\mathbb{1 7}) \Rightarrow([\mathbb{1})$ :
This is trivial taking $s s$ to be singleton bindings in $s$.
Proof of $(\mathbb{T D}) \Rightarrow([\mathbb{Z})$, for $s s \in$ dom $e$ :
$e(\bigcup s s)$
= Exec property (10)
$\bigcup\{b: \bigcup s s \bullet e(\{b\})\}$
$=$ expand union
$\bigcup\{s: s s \bullet \bigcup\{b: s \bullet e(\{b\})\}\}$
$=$ Exec property (10)
$\bigcup\{s: s s \bullet e(s)\}$

## C Properties of Execs

In this section we prove that each of our basic commands satisfy the three conditions in Definition B.1. We use the following definitions, with $\left(\operatorname{exec} c_{1}\right)=e$ and $\left(\operatorname{exec} c_{2}\right)=f$ :

```
\(\operatorname{exec}(\langle P\rangle)=(\lambda s: \mathbb{P}(\overline{\operatorname{def} P}) \bullet s \cap \bar{P})\)
\(\operatorname{exec}(\{A\})=(\lambda s: \mathbb{P}(\overline{d e f ~ A \wedge A}) \bullet s)\)
\(\operatorname{exec}\left(c_{1} \vee c_{2}\right)=e \uplus f\)
\(\operatorname{exec}\left(c_{1} \wedge c_{2}\right)=e \cap f\)
\(\operatorname{exec}\left(c_{1}, c_{2}\right)=e_{9}^{\circ} f\)
\(\operatorname{exec}(\exists v \bullet c)=(\lambda s:\) State \(\mid\) unbind \(v s \in \operatorname{dom} e\)
    - \(\{b: s \mid(\exists x:\) Val \(\bullet e(\{b \oplus\{v \mapsto x\}\}) \neq\{ \})\})\)
exec \((\forall v \bullet c)=(\lambda s\) : State \(\mid\) unbind \(v s \in \operatorname{dom} e\)
    - \(\{b: s \mid(\forall x:\) Val \(\bullet e(\{b \oplus\{v \mapsto x\}\}) \neq\{ \})\})\)
```

We also prove the three properties for non-recursive parameterised commands, though to simplify the presentation we include a new definition call. It is similar to the definition of exec for procedure calls, except with an Exec as a parameter rather than a Cmd. The Exec of a procedure call is given by $(\operatorname{pexec}(v:-c))(t))$; in our proofs we instead use call $v$ (exec $c) t$ where

```
call ve t=(\lambdas:\mathbb{P}(\operatorname{dom}(\textrm{eval}t))|\mathrm{ assign v t s f dome}
    - {b:s|e(assign v t{b})}\not={}}
```

We do this so that we do not have to reference the environment, and avoid the step to PExecs. The definition of call may be likened to the definition of exists and forall.

In order to prove the three Exec properties we would prove each of them simultaneously for each construct; for presentation purposes we have chosen to separate the proofs for each Exec property. Hence in the proofs our inductive hypothesis is that all properties of Execs hold for the components.

## C. 1 Proof of Exec domain property for all constructs

For each construct we prove:

$$
\operatorname{dom} e=\mathbb{P}\{b: B n d \mid\{b\} \in \operatorname{dom} e\}
$$

Theorem C. 1 Specifications and assumptions satisfy Exec property (8).
Proof for specifications, using (14) instead of Exec property (§) $($ dom $e=\mathbb{P}(\bigcup$ dome $)$ ):

```
    P}\bigcup(\operatorname{dom}(\operatorname{exec}\langleP\rangle)
= definition
    P \(\mathbb{P}\overline{def.P})
= from ([.0)
    P}\overline{def P
= definition
    dom(exec}\langleP\rangle
```

The proof for assumptions is similar.
We require the following lemma:

## Lemma C. 2

$$
(\forall A, B: \text { State } \bullet \mathbb{P}(A \cap B)=(\mathbb{P} A) \cap(\mathbb{P} B))
$$

Proof:

$$
\begin{aligned}
& s \in \mathbb{P}(A \cap B) \\
\Leftrightarrow & \text { membership in power set } \\
& s \subseteq(A \cap B) \\
\Leftrightarrow & \text { distribute subset } \\
& s \subseteq A \wedge s \subseteq B \\
\Leftrightarrow & \text { membership in power set } \\
& (s \in \mathbb{P} A) \wedge(s \in \mathbb{P} B) \quad \square
\end{aligned}
$$

Theorem C. 3 Parallel conjunction and disjunction satisfy Exec property (§).
Proof for parallel conjunction:

$$
\begin{aligned}
& \mathbb{P}\{b: \text { Bnd } \mid\{b\} \in \operatorname{dom}(e \cap f)\} \\
= & \operatorname{definition} \\
& \mathbb{P}\{b: \text { Bnd } \mid\{b\} \in(\operatorname{dom} e \cap \operatorname{dom} f)\} \\
= & \operatorname{distribution~} \\
& \mathbb{P}\{b: \text { Bnd } \mid\{b\} \in \operatorname{dom} e \wedge\{b\} \in \operatorname{dom} f)\} \\
= & \operatorname{split} \text { into two comprehensions } \\
& \mathbb{P}(\{b: \text { Bnd } \mid\{b\} \in \operatorname{dom} e\} \cap\{b: B n d \mid\{b\} \in \operatorname{dom} f\})
\end{aligned}
$$

```
= Lemma C.2
    P}({b:Bnd |{b}\in\operatorname{dom}e})\cap\mathbb{P}({b:Bnd|{b}\in\operatorname{dom}f}
= inductive hypothesis
    dom}e\cap\operatorname{dom}
= definition
    dom(e\capf)
```

Proof for disjunction：Both parallel conjunction and disjunction have the same domains， hence the above proof holds for disjunction as well．

We require the following lemma
Lemma C． 4 For all $s$ in $\operatorname{dom} e$ ：

$$
(\forall b: s \bullet e(\{b\}) \in \operatorname{dom} f) \Leftrightarrow(\forall b: e(s) \bullet\{b\} \in \operatorname{dom} f)
$$

Proof：

```
    (\forallb:s\bullete({b}) \in\operatorname{dom}f)
\Leftrightarrow case analysis from ([\mathbb{Z})
    (\forallb:s\bullet(b\ine(s)\veeb\not\ine(s))=>e({b})\in\operatorname{dom}f)
\Leftrightarrow distribute implication and quantification
    (\forallb:s\bulletb\ine(s)=>e({b})\in\operatorname{dom}f)\wedge
    (\forallb:s\bulletb\not\ine(s)=>e({b})\in\operatorname{dom}f)
\Leftrightarrow from ([W)}\times
    (\forallb:s\bulletb\ine(s)=>{b}\in\operatorname{dom}f)^
    (\forallb:s\bullete({b})={}=>e({b})\in\operatorname{dom}f)
\Leftrightarrow restate first conjunct from Exec property (9), simplify second from ([⿴囗十|
    (\forallb:e(s)\bullet{b}\in\operatorname{dom}f)
```

Theorem C． 5 Sequential conjunction satisfies Exec property（§）．

## Proof：

```
    \(\operatorname{dom}\left(e_{9} f\right)\)
\(=\) definition
    \(\{s:\) State \(\mid s \in \operatorname{dom} e \wedge e(s) \in \operatorname{dom} f\}\)
\(=\) inductive hyp \(\times 2\)
    \(\{s:\) State \(\mid s \in \mathbb{P}\{b:\) Bnd \(\mid\{b\} \in \operatorname{dom} e\} \wedge e(s) \in \mathbb{P}\{b: B n d \mid\{b\} \in \operatorname{dom} f\}\}\)
\(=\) Lemma B. \(2 \times 2\)
    \(\{s:\) State \(\mid \forall b: s \bullet\{b\} \in \operatorname{dom} e) \wedge(\forall b: e(s) \bullet\{b\} \in \operatorname{dom} f)\}\)
\(=\) Lemma C. 4
    \(\{s:\) State \(\mid \forall b: s \bullet\{b\} \in \operatorname{dom} e) \wedge(\forall b: s \bullet e(\{b\}) \in \operatorname{dom} f)\}\)
\(=\) Join quantifiers
    \(\{s:\) State \(\mid(\forall b: s \bullet\{b\} \in \operatorname{dom} e \wedge e(\{b\}) \in \operatorname{dom} f))\}\)
\(=\) definition
    \(\left\{s:\right.\) State \(\left.\mid\left(\forall b: s \bullet\{b\} \in \operatorname{dom} e{ }_{9} f\right)\right\}\)
    = Lemma B. 2
    \(\mathbb{P}\left\{b: B n d \mid\{b\} \in \operatorname{dom} e_{9} f\right\}\)
```

Theorem C. 6 Both quantifiers satisfy Exec property (§).
Proof for $s \in$ State:

```
    s\in\operatorname{dom}(\operatorname{exec}(\existsv\bulletS))
\Leftrightarrow definition
    unbind vs\in dome
\Leftrightarrow inductive hypothesis; Lemma B.2
    (\forall\mp@subsup{b}{}{\prime}}\mathrm{ : unbind vse{ b
\Leftrightarrow}\mathrm{ definition
    (\forall\mp@subsup{b}{}{\prime}:{b:s,x:Val\bulletb\oplus{v\mapstox}}\bullet{\mp@subsup{b}{}{\prime}}\in\operatorname{dom}e)
\Leftrightarrow}\mathrm{ rewrite
    (\forallb:s\bullet{x:Val\bulletb\oplus{v\mapstox}}\in\operatorname{dom}e)
\Leftrightarrow definition
    (}\forallb:s\bulletunbind v{b}\in\operatorname{dom}e
\Leftrightarrow Lemma B.2
    s\in\mathbb{P}{b:Bnd\bullet{b}\in\operatorname{dom}(\operatorname{exec}(\existsv\bulletS))}
```

The proof for universal quantification is similar.
Theorem C. 7 Parameterised commands satisfy Exec property (8).
Proof for $s \in$ State:

```
    s\in\operatorname{dom(call veet)}
\Leftrightarrow}\mathrm{ definition
    s\in\mathbb{P}(\operatorname{dom}(\mathrm{ eval t)) ^ assign v t sedom e}
definition of eval; definition of assign
    (\forallb:s\bulletb\in\operatorname{dom(eval t))^}
    {b:s\bulletb\oplus{v\mapsto eval t b}}\in\operatorname{dom}e
\Leftrightarrow rewrite; inductive hypothesis
    (\forallb:s\bullet{b}\in\mathbb{P}(\operatorname{dom}(\mathrm{ eval t) ))}\wedge
    (\forallb:s\bullet{b\oplus{v\mapsto eval t b}}\in\operatorname{dom}e)
\Leftrightarrow join quantifiers
    (\forallb:s\bullet{b}\in\mathbb{P}(\operatorname{dom}(\mathrm{ eval }t))\wedge assign vt{b}\in\operatorname{dom}e)
\Leftrightarrow definition
    (\forallb:s\bullet{b}\in\operatorname{dom(call vet)}
& Lemma B.2
    s\in\mathbb{P}{b:\widehat{Bnd}|{b}\in\operatorname{dom}(call vet)}
```


## C. 2 Proof of increase groundedness property for all constructs

For each construct we prove

$$
(\forall s: \operatorname{dom} e \bullet e(s) \subseteq s)
$$

Theorem C. 8 Specifications and assumptions satisfy Exec property (9).
Proof: This follows from inspection.

Theorem C. 9 Parallel conjunction satisfies Exec property (9).
Proof: We know that $s \in \operatorname{dom}(e \cap f) \Rightarrow(s \in \operatorname{dom} e \wedge s \in \operatorname{dom} f)$. Inductively we assume $e(s) \subseteq s \wedge f(s) \subseteq s$, and therefore $e(s) \cap f(s) \subseteq s$.

Theorem C. 10 Disjunction satisfies Exec property (G).
Proof: We know that $s \in \operatorname{dom}(e \cup f) \Rightarrow(s \in \operatorname{dom} e \wedge s \in \operatorname{dom} f)$. Inductively we assume $e(s) \subseteq s \wedge f(s) \subseteq s$, and therefore $e(s) \cup f(s) \subseteq s$.

Theorem C. 11 Sequential conjunction satisfies Exec property (马).
Proof: We know that $s \in \operatorname{dom}\left(e{ }_{9} f\right) \Rightarrow(s \in \operatorname{dom} e \wedge e(s) \in \operatorname{dom} f)$. Inductively we assume $e(s) \subseteq s \wedge f(e(s)) \subseteq e(s)$, and therefore $f(e(s)) \subseteq s$.

Theorem C. 12 Quantifiers and parameterised command satisfy Exec property (9).
Proof: By inspection, the result of each is a restriction on the bindings originally in $s$, therefore the result can only be a subset of $s$.

## C. 3 Proof of range Exec property for all constructs

For each construct we prove, for all $s$ in the domain of $e$,

$$
e(s)=\{b: s \mid e(\{b\}) \neq\{ \}\}
$$

We assume Exec property (8) holds for each construct, and in particular that $(\forall b: s \bullet\{b\} \in$ $\operatorname{dom} e)$.

Theorem C. 13 Specifications satisfy Exec property (19).
Proof:

$$
\begin{aligned}
& \{b: s \mid(\{b\} \cap \bar{P}) \neq\{ \}\} \\
= & \operatorname{simplify} \\
& \{b: s \mid b \in \bar{P}\} \\
= & \operatorname{simplify} \\
& s \cap \bar{P}
\end{aligned}
$$

Theorem C. 14 Assumptions satisfy Exec property (19).
Proof:

$$
\begin{aligned}
& \{b: s \mid\{b\} \neq\{ \}\} \\
= & \text { simplify } \\
& s \quad \square
\end{aligned}
$$

Theorem C. 15 Parallel conjunction satisfies Exec property (10).
Proof:

$$
\begin{aligned}
& \{b: s \mid(e \cap f)(\{b\}) \neq\{ \}\} \\
= & \text { definition } \\
& \{b: s \mid(e(\{b\}) \cap f(\{b\})) \neq\{ \}\} \\
= & \text { manipulation }
\end{aligned}
$$

```
    {b:s|e({b})\not={}\wedgef({b})\not={}}
= split comprehension
    {b:s|e({b})\not={}}\cap{b:s|f({b})\not={}}
= inductive hypothesis
    e(s)\capf(s)
```

Theorem C． 16 Disjunction satisfies Exec property（19）．
Proof：

$$
\begin{aligned}
& \{b: s \mid(e \cup f)(\{b\}) \neq\{ \}\} \\
= & \text { definition } \\
& \{b: s \mid(e(\{b\}) \cup f(\{b\})) \neq\{ \}\} \\
= & \text { union of sets } \\
& \{b: s \mid e(\{b\}) \neq\{ \} \vee f(\{b\}) \neq\{ \}\} \\
= & \text { split comprehension } \\
= & \{b: s \mid e(\{b\}) \neq\{ \}\} \cup\{b: s \mid f(\{b\}) \neq\{ \}\} \\
= & \text { inductive hypothesis } \\
& e(s) \cup f(s) \quad \square
\end{aligned}
$$

Theorem C． 17 Sequential conjunction satisfies Exec property（10）．
Proof：
$\left\{b: s \mid\left(e_{9} f\right)(\{b\}) \neq\{ \}\right\}$
$=$ definition
$\{b: s \mid f(e(\{b\})) \neq\{ \}\}$
$=$ case analysis from（［区）
$\{b: s \mid f(e(\{b\})) \neq\{ \} \wedge$ $(e(\{b\})=\{ \} \vee e(\{b\}) \neq\{ \})\}$
$=$ distribute
$\{b: s \mid(f(e(\{b\})) \neq\{ \} \wedge e(\{b\})=\{ \}) \vee$ $(f(e(\{b\})) \neq\{ \} \wedge e(\{b\}) \neq\{ \})\}$
$=$ simplify
$\{b: s \mid(f(\{ \}) \neq\{ \} \wedge e(\{b\})=\{ \}) \vee$ $(f(\{b\}) \neq\{ \} \wedge e(\{b\}) \neq\{ \})\}$
$=$ from（［⿴囗十）
$\{b: s \mid f(\{b\}) \neq\{ \} \wedge e(\{b\}) \neq\{ \}\}$
$=$ inductive hypothesis on $e$
$\{b: e(s) \mid f(\{b\}) \neq\{ \}\}$
$=$ inductive hypothesis on $f$
$f(e(s))$
Theorem C． 18 Both quantifiers satisfy Exec property（19）．
Proof：
$\{b: s \mid(\operatorname{exec}(\exists v \bullet c)(\{b\})) \neq\{ \}\}$
$=$ definition
$\{b: s \mid\{c:\{b\} \mid(\exists x:$ Val $\bullet e(\{c \oplus\{v \mapsto x\}\}) \neq\{ \})\} \neq\{ \}\}$
$=$ simplification

```
    {b:s|(\existsx:Val\bullete({b\oplus{v\mapstox}})\not={})}
= definition
    (exec(\existsV \bullet c))(s)
```

The proof for universal quantification is similar.
Theorem C. 19 Parameterised command satisfies Exec property (19).
Proof:

$$
\begin{aligned}
& \{b: s \mid(\text { call } v e t)(\{b\}) \neq\{ \}\} \\
= & \text { definition } \\
& \{b: s \mid\{c:\{b\} \mid e(\{c \oplus\{v \mapsto \text { eval } t c\}\}) \neq\{ \})\} \neq\{ \}\} \\
= & \text { simplification } \\
& \{b: s \mid e(\{b \oplus\{v \mapsto \text { eval } t b\}\}) \neq\{ \}\} \\
= & \text { definition } \\
& (\text { call } v e t)(s)
\end{aligned}
$$

## D Lattice properties

## D. 1 Least upper bound of two elements in Exec

From Sect.5.1 we must show that the join of two Execs, when it is defined, returns an Exec.

$$
\left(\forall s: \operatorname{dom} e_{1} \cap \operatorname{dom} e_{2} \bullet e_{1}(s)=e_{2}(s)\right) \Rightarrow e_{1} \sqcup e_{2} \in \text { Exec }
$$

To prove this, we must show that the definition of least upper bound is an Exec, by satisfying the three conditions of Definition [B.].

Theorem D. 1 Least upper bound satisfies Exec property (§).

## Proof:

$\mathbb{P}\left(\bigcup \operatorname{dom}\left(e_{1} \sqcup e_{2}\right)\right)$
$=$ definition
$\mathbb{P}\left(\bigcup\left(\mathbb{P}\left(\bigcup\left(\operatorname{dom} e_{1} \cup \operatorname{dom} e_{2}\right)\right)\right)\right)$
$=$ from (T.5)
$\mathbb{P} \bigcup\left(\operatorname{dom} e_{1} \cup \operatorname{dom} e_{2}\right)$
$=$ definition
$\operatorname{dom}\left(e_{1} \sqcup e_{2}\right)$
Theorem D. 2 Least upper bound satisfies Exec property (9).
Proof: We must show:

$$
\left(e_{1} \sqcup e_{2}\right)(s) \subseteq s
$$

This is trivial from the definition (restriction similar to quantifier definition).
Theorem D. 3 Least upper bound satisfies Exec property (10).
Proof:
From Theorem D.2 and Theorem B.6, we use (16) in place of Exec property (10).

```
    \(\bigcup\left\{b: s \bullet\left(e_{1} \sqcup e_{2}\right)(\{b\})\right\}\)
\(=\) from ([6])
    \(\left\{b: s \mid\left(e_{1} \sqcup e_{2}\right)(\{b\}) \neq\{ \}\right\}\)
\(=\) definition
    \(\left\{b: s \mid\left\{c:\{b\} \mid\left(e_{1} \cup e_{2}\right)(\{c\}) \neq\{ \}\right\} \neq\{ \}\right\}\)
\(=\) simplification
    \(\left\{b: s \mid\left(e_{1} \cup e_{2}\right)(\{b\}) \neq\{ \}\right\}\)
\(=\) definition
\(\left(e_{1} \sqcup e_{2}\right)(s)\)
```


## D. 2 Greatest lower bound of two elements in Exec

From Sect.5.1 we must show that the greatest lower bound of two Execs returns an Exec, i.e., $e_{1} \cap e_{2} \in$ Exec. To prove this, we must satisfy the three conditions of Definition B.1.

Theorem D. 4 Greatest lower bound satisfies Exec property (8).
Proof:

```
    \(s \in \mathbb{P}\left\{b:\right.\) Bnd \(\left.\mid\{b\} \in \operatorname{dom}\left(e_{1} \cap e_{2}\right)\right\}\)
\(\Leftrightarrow\) domain of intersection
    \(s \in \mathbb{P}\left\{b: B n d \mid\{b\} \in \operatorname{dom} e_{1} \wedge\{b\} \in \operatorname{dom} e_{2} \wedge e_{1}(\{b\})=e_{2}(\{b\})\right\}\)
\(\Leftrightarrow\) from Lemma B. 2
    \(\left(\forall b: s \bullet\{b\} \in \operatorname{dom} e_{1} \wedge\{b\} \in \operatorname{dom} e_{2} \wedge e_{1}(\{b\})=e_{2}(\{b\})\right)\)
\(\Leftrightarrow\) Distribute quantification
    \(\left.\left(\forall b: s \bullet\{b\} \in \operatorname{dom} e_{1}\right) \wedge\left(\forall b: s \bullet\{b\} \in \operatorname{dom} e_{2}\right) \wedge\left(\forall b: s \bullet e_{1}(\{b\})=e_{2}(\{b\})\right)\right\}\)
\(\Leftrightarrow\) Lemma B. 2 and simplification
    \(s \in \operatorname{dom} e_{1} \wedge s \in \operatorname{dom} e_{2} \wedge e_{1}(s)=e_{2}(s)\)
\(\Leftrightarrow\) definition
    \(s \in \operatorname{dom}\left(e_{1} \cap e_{2}\right)\)
```

Theorem D. 5 Greatest lower bound satisfies Exec property (9).
Proof: We know

$$
\begin{equation*}
s \in \operatorname{dom}\left(e_{1} \cap e_{2}\right) \Rightarrow\left(e_{1} \cap e_{2}\right)(s)=e_{1}(s) \tag{20}
\end{equation*}
$$

Proof of Theorem D.5, assuming $s \in \operatorname{dom}\left(e_{1} \cap e_{2}\right)$ :
$\left(e_{1} \cap e_{2}\right)(s) \subseteq s$
$\Leftrightarrow$ from ([IT)
$e_{1}(s) \subseteq s$
$\Leftarrow$ inductive hypothesis
Theorem D. 6 Greatest lower bound satisfies Exec property (19).
Proof: Assume $s \in \operatorname{dom}\left(e_{1} \cap e_{2}\right)$.

$$
\bigcup\left\{b: s \bullet\left(e_{1} \cap e_{2}\right)(\{b\})\right\}
$$

$=$ from ( ZD )
$\bigcup\left\{b: s \bullet e_{1}(\{b\})\right\}$
$=$ inductive hypothesis

$$
\begin{aligned}
& e_{1}(s) \\
= & \text { from }(\text { (2П) }) \\
& \left(e_{1} \cap e_{2}\right)(s)
\end{aligned}
$$

## E Monotonicity

In this section we prove monotonicity of the refinement relations for all the constructs in our wide-spectrum language. In each case, we have the premiss that $e \sqsubseteq e^{\prime}$. Since refinement between Execs is defined as subset, we use the following expanded form in our proofs.

$$
\begin{align*}
& \operatorname{dom} e \subseteq \operatorname{dom} e^{\prime} \wedge \\
& \left(\forall s: \operatorname{dom} e \bullet e(s)=e^{\prime}(s)\right) \tag{21}
\end{align*}
$$

For the binary constructs, we make similar assumptions for $f$ and $f^{\prime}$.
Theorem E. 1 Parallel conjunction is monotonic.

$$
\frac{e \subseteq e^{\prime} ; f \subseteq f^{\prime}}{(e \cap f) \subseteq\left(e^{\prime} \cap f^{\prime}\right)}
$$

Proof:

$$
(e \cap f) \subseteq\left(e^{\prime} \cap f^{\prime}\right)
$$

$$
\Leftrightarrow \text { definition of } \cap \text {; subset of functions }
$$

$\left(\operatorname{dom} e \cap \operatorname{dom} f \subseteq \operatorname{dom} e^{\prime} \cap \operatorname{dom} f^{\prime}\right) \wedge$
( $\forall s:(\operatorname{dom} e \cap \operatorname{dom} f) \bullet$
$\left.\left(e(s) \cap f(s)=e^{\prime}(s) \cap f^{\prime}(s)\right)\right)$
$\Leftarrow($ (四 $)$
Theorem E. 2 Disjunction is monotonic.

$$
\frac{e \subseteq e^{\prime} ; f \subseteq f^{\prime}}{(e \uplus f) \subseteq\left(e^{\prime} \uplus f^{\prime}\right)}
$$

Proof:

$$
(e \uplus f) \subseteq\left(e^{\prime} \cup f^{\prime}\right)
$$

$$
\begin{aligned}
\Leftrightarrow & \text { definition of } \cup \text {; subset of functions } \\
& \left(\operatorname{dom} e \cap \operatorname{dom} f \subseteq \operatorname{dom} e^{\prime} \cap \operatorname{dom} f^{\prime}\right) \wedge \\
& (\forall s:(\operatorname{dom} e \cap \operatorname{dom} f) \bullet \\
\Leftarrow & \left.\left(e(s) \cup f(s)=e^{\prime}(s) \cup f^{\prime}(s)\right)\right) \\
\Leftarrow & (\square)
\end{aligned}
$$

Theorem E. 3 Sequential conjunction is monotonic.

$$
\frac{e \subseteq e^{\prime} ; f \subseteq f^{\prime}}{e_{9}^{\circ} f \subseteq e^{\prime}{ }_{9} f^{\prime}}
$$

Proof:

```
    (s, s')\in e g
\Leftrightarrow}\mathrm{ definition
    (\existst\bullet((s,t)\ine)\wedge((t, s')\inf))
=> inductive hypothesis
    (\existst\bullet((s,t)\in\mp@subsup{e}{}{\prime})\wedge((t,\mp@subsup{s}{}{\prime})\in\mp@subsup{f}{}{\prime}))
\Leftrightarrow}\mathrm{ definition
    (s, s') \in e ' }\mp@subsup{}{9}{}\mp@subsup{f}{}{\prime
```

We have $\left(s, s^{\prime}\right) \in e_{9} f \Rightarrow\left(s, s^{\prime}\right) \in e^{\prime}{ }_{9} f^{\prime}$, therefore $e{ }_{9} f \subseteq e^{\prime}{ }_{9} f^{\prime}$.
Theorem E. 4 Both quantifiers are monotonic.

$$
\begin{array}{cc}
e \subseteq e^{\prime} & e \subseteq e^{\prime} \\
{v e \subseteq \text { exists } v e^{\prime}} } & \text { forall } v e \subseteq \text { forall } v e^{\prime}
\end{array}
$$

Proof:
Looking at the domain restrictions on subsets of functions

$$
\begin{aligned}
& \operatorname{dom}(\text { exists } v e) \subseteq \operatorname{dom}\left(\text { exists } v e^{\prime}\right) \\
\Leftrightarrow & \text { definition } \\
& \{s: \text { State } \mid \text { unbind } v s \in \operatorname{dom} e\} \subseteq \\
& \left\{s: \text { State } \mid \text { unbind } v s \in \operatorname{dom} e^{\prime}\right\} \\
\Leftarrow & (\mathbb{\square})
\end{aligned}
$$

Now we also require for all $s$ such that unbind $v s \in \operatorname{dom} e$ :

$$
\begin{aligned}
& (\text { exists } v e)(s)=\left(\text { exists } v e^{\prime}\right)(s) \\
\Leftrightarrow & \text { assuming type of } s \text {, and definition of exists } \\
& \{b: s \mid(\exists x: \text { Val } \bullet(\{b \oplus\{v \mapsto x\}\}) \neq\{ \})\}= \\
& \left\{b: s \mid\left(\exists x: \text { Val } \bullet e^{\prime}(\{b \oplus\{v \mapsto x\}\}) \neq\{ \}\right)\right\}
\end{aligned}
$$

Now $b \in s \wedge$ unbind $v s \in \operatorname{dom} e$ implies $\{b \oplus\{v \mapsto x\}\} \in \operatorname{dom} e$. Therefore $e(\{b \oplus\{v \mapsto$ $x\}\})=e^{\prime}(\{b \oplus\{v \mapsto x\}\})$.

The proof for universal quantification is similar.
Theorem E. 5 Parameterised commands are monotonic.

$$
\frac{e \subseteq e^{\prime}}{\text { call } v e t \subseteq \text { call } v e^{\prime} t}
$$

Looking at the domain restrictions on subsets of functions:

$$
\begin{aligned}
& \operatorname{dom}(\text { call } v e t) \subseteq \operatorname{dom}\left(\text { call } v e^{\prime} t\right) \\
\Leftrightarrow & \text { definition } \\
& \{s: \mathbb{P}(\text { dom eval } t) \mid \text { assign } v \text { t } s \in \operatorname{dom} e\} \subseteq \\
& \left\{s: \mathbb{P}(\text { dom eval } t) \mid \text { assign } v \text { t } s \in \operatorname{dom} e^{\prime}\right\} \\
\Leftarrow & (\mathbb{Z})
\end{aligned}
$$

We also require for all $s$ in $\mathbb{P}($ domeval $t)$ such that assign $v t s \in \operatorname{dom} e$ :

```
    \((\) call \(v e t)(s)=\left(\right.\) call \(\left.v e^{\prime} t\right)(s)\)
\(\Leftrightarrow\) definition
    \(\{b: s \mid e(\) assign \(v t\{b\}) \neq\{ \}\}=\left\{b: s \mid e^{\prime}(\right.\) assign \(\left.v t\{b\}) \neq\{ \}\right\}\)
\(\Leftrightarrow\) rewrite
    \(\left(\forall b: s \bullet e(\right.\) assign \(v t\{b\}) \neq\{ \} \Leftrightarrow e^{\prime}(\) assign \(\left.v t\{b\}) \neq\{ \}\right)\)
\(\Leftarrow\) (Ш1)
```


## F Continuity

Recall the least upper bound of two programs:

$$
\begin{aligned}
&\left(-\sqcup_{-}\right)=\left(\lambda e_{1}, e_{2}:\right. \text { Exec } \\
& \mid\left.\mid \forall s: \operatorname{dom} e_{1} \cap \operatorname{dom} e_{2} \bullet e_{1}(s)=e_{2}(s)\right) \\
& \bullet\left(\lambda s: \mathbb{P}\left(\cup\left(\operatorname{dom} e_{1} \cup \operatorname{dom} e_{2}\right)\right)\right. \\
&\left.\left.\bullet\left\{b: s \mid\left(e_{1} \cup e_{2}\right)(\{b\}) \neq\{ \}\right\}\right)\right)
\end{aligned}
$$

From the definition of $\sqcup_{-}$, for any two Execs such that $e_{1} \sqsubseteq e_{2}, e_{1} \sqcup e_{2}$ is defined, and $e_{1} \sqcup e_{2}=e_{2}$.

Within the context of recursion, the infinite sets of functions form a refinement chain. Hence we have a chain of subsets, i.e.

$$
\text { Chain }==\{e c: \mathbb{N} \rightarrow \text { Exec } \mid \forall i: \mathbb{N} \bullet e c(i) \subseteq e c(i+1)\}
$$

The subset relation guarantees that the union is a function, and hence the least upper bound of a refinement chain, $e c$, is defined as

$$
\bigsqcup e c=\bigcup\{i: \mathbb{N} \bullet e c(i)\}
$$

If $e c$ is a Chain, then for all monotonic contexts $\mathcal{C},(\lambda i: \mathbb{N} \bullet \mathcal{C}(e c(i)))$ is also a Chain.

## Continuity of operators

We wish to prove for all contexts $\mathcal{C}$ that

$$
\mathcal{C}(\bigsqcup e c)=\bigsqcup(\lambda i: \mathbb{N} \bullet \mathcal{C}(e c(i)))
$$

Since all contexts are monotonic, and therefore form a Chain, we may use $\bigcup$ as the least upper bound, i.e.,

$$
\mathcal{C}\left(\bigsqcup^{-c}\right)=\bigcup\{i: \mathbb{N} \bullet \mathcal{C}(e c(i))\}
$$

In our proofs we use the following properties for all $e c \in$ Chain:

$$
\begin{equation*}
(\forall j: \mathbb{N} \bullet s \in \operatorname{dom} e c(j) \Rightarrow e c(j)(s)=(\bigsqcup e c)(s)) \tag{22}
\end{equation*}
$$

and

$$
\begin{equation*}
(\exists j: \mathbb{N} \bullet s \in \operatorname{dom} e c(j)) \Leftrightarrow s \in \operatorname{dom}(\bigsqcup e c) \tag{23}
\end{equation*}
$$

Theorem F. 1 Parallel conjunction is continuous.
To prove continuity for the first argument, we need to show

$$
(\bigsqcup e c) \cap e=\bigcup\{i: \mathbb{N} \bullet e c(i) \curvearrowright e\}
$$

Proof:

$$
\begin{aligned}
&\left(s, s^{\prime}\right) \in \bigcup\{i: \mathbb{N} \bullet e c(i) \cap e\} \\
& \Leftrightarrow \text { membership in union } \\
& \exists j: \mathbb{N} \bullet\left(s, s^{\prime}\right) \in(e c(j) \cap e) \\
& \Leftrightarrow \text { definition } \\
& \exists j: \mathbb{N} \bullet s \in \operatorname{dom} e c(j) \wedge s \in \operatorname{dom} e \wedge \\
& s^{\prime}=e c(j)(s) \cap e(s) \\
& \Leftrightarrow \text { from }(\text { (LZ) } \\
& \exists j: \mathbb{N} \bullet s \in \operatorname{dom} e c(j) \wedge s \in \operatorname{dom} e \wedge \\
& s^{\prime}=(\bigsqcup e c)(s) \cap e(s) \\
& \Leftrightarrow \text { reduce scope of } j,(\text { (Z.3) } \\
& s \in \operatorname{dom}(\bigsqcup e c) \wedge s \in \operatorname{dom} e \wedge \\
& \quad s^{\prime}=(\bigsqcup e c)(s) \cap e(s) \\
& \Leftrightarrow \operatorname{definition} \\
&\left(s, s^{\prime}\right) \in(\bigsqcup e c) \cap e \quad \square
\end{aligned}
$$

The proof for the second argument is similar.
Theorem F. 2 Disjunction is continuous.
To prove continuity for the first argument, we need to show

$$
(\bigsqcup e c) \cup e=\bigcup\{i: \mathbb{N} \bullet e c(i) \uplus e\}
$$

Proof:

$$
\begin{aligned}
& \left(s, s^{\prime}\right) \in \bigcup\{i: \mathbb{N} \bullet e c(i) \cup e\} \\
\Leftrightarrow & \text { membership in union } \\
& \exists j: \mathbb{N} \bullet\left(s, s^{\prime}\right) \in(e c(j) \uplus e) \\
\Leftrightarrow & \text { definition } \\
& \exists j: \mathbb{N} \bullet s \in \operatorname{dom} e c(j) \wedge s \in \operatorname{dom} e \wedge \\
& s^{\prime}=e c(j)(s) \cup e(s) \\
\Leftrightarrow & \text { from }(\text { (L2) }) \\
& \exists j: \mathbb{N} \bullet s \in \operatorname{dom} e c(j) \wedge s \in \operatorname{dom} e \wedge \\
& s^{\prime}=(\bigsqcup e c)(s) \cup e(s) \\
\Leftrightarrow & \text { reduce scope of } j,(\mathbb{Z} \cdot 3) \\
& s \in \operatorname{dom}(\bigsqcup e c) \wedge s \in \operatorname{dom} e \wedge \\
& s^{\prime}=(\bigsqcup e c)(s) \cup e(s) \\
\Leftrightarrow & \text { definition } \\
& \left(s, s^{\prime}\right) \in(\bigsqcup e c) \cup e \quad \square
\end{aligned}
$$

The proof for the second argument is similar.

Theorem F. 3 Sequential conjunction is continuous in the first argument.
We need to show:

$$
(\bigsqcup e c)_{9} e=\bigcup\left\{i: \mathbb{N} \bullet e c(i)_{9} e\right\}
$$

Proof:

$$
\left(s, s^{\prime}\right) \in \bigcup\left\{i: \mathbb{N} \bullet e c(i)_{g} e\right\}
$$

$\Leftrightarrow$ containment in union
$\exists j: \mathbb{N} \bullet\left(s, s^{\prime}\right) \in\left(e c(j){ }_{9} e\right)$
$\Leftrightarrow$ function composition
$\exists j: \mathbb{N} \bullet \exists t:$ State •
$(s, t) \in e c(j) \wedge\left(t, s^{\prime}\right) \in e$
$\Leftrightarrow$ reduce scope of $j$
$\exists t$ :State •
$(\exists j: \mathbb{N} \bullet(s, t) \in e c(j)) \wedge\left(t, s^{\prime}\right) \in e$
$\Leftrightarrow$ membership in union
$\exists t:$ State $\bullet(s, t) \in(\bigsqcup e c) \wedge\left(t, s^{\prime}\right) \in e$
$\Leftrightarrow$ function composition
$\left(s, s^{\prime}\right) \in(\bigsqcup e c){ }_{9} e$
Theorem F. 4 Sequential conjunction is continuous in the second argument.
We need to show:

$$
e_{9}^{\circ}(\bigsqcup e c)=\bigcup\left\{i: \mathbb{N} \bullet e_{9}^{\circ} e c(i)\right\}
$$

Proof:

$$
\begin{aligned}
&\left(s, s^{\prime}\right) \in \bigcup\left\{i: \mathbb{N} \bullet e_{9}^{\circ} e c(i)\right\} \\
& \Leftrightarrow \text { containment in union } \\
& \exists j: \mathbb{N} \bullet\left(s, s^{\prime}\right) \in\left(e{ }_{9}^{9} e c(j)\right) \\
& \Leftrightarrow \text { function composition } \\
& \exists j: \mathbb{N} \bullet \exists t: \text { State } \bullet \\
&(s, t) \in e \wedge\left(t, s^{\prime}\right) \in e c(j) \\
& \Leftrightarrow \text { reduce scope of } j \\
& \exists t: \text { State } \bullet \\
& \quad(s, t) \in e \wedge\left(\exists j: \mathbb{N} \bullet\left(t, s^{\prime}\right) \in e c(j)\right) \\
& \Leftrightarrow \text { membership in union } \\
& \exists t: \text { State } \bullet \\
&(s, t) \in e \wedge\left(t, s^{\prime}\right) \in(\bigsqcup e c) \\
& \Leftrightarrow \text { function composition } \\
&\left(s, s^{\prime}\right) \in e_{9}^{\circ}(\bigsqcup e c) \quad \square
\end{aligned}
$$

To prove continuity of the quantifiers, we require the following lemma:

## Lemma F. 5

unbind $v s \in \operatorname{dom} e \Rightarrow(\forall b: s \bullet\{b \oplus\{v \mapsto x\}\} \in \operatorname{dom} e)$
Proof:
unbind $v s \in \operatorname{dom} e$

```
\Leftrightarrow Exec property (8) and Lemma B.2
    (}\forallb\mathrm{ : unbind vs }\bullet{b}\in\operatorname{dom}e
\Leftrightarrow}\mathrm{ definition
    (\forallb:{b:s;x:Val\bulletb\oplus{v\mapstox}}\bullet{b}\in\operatorname{dom}e)
\Leftrightarrow simplify
    (\forallb:s\bullet{b\oplus{v\mapstox}}\in\operatorname{dom}e)
```

Theorem F. 6 Both quantifiers are continuous.
We need to show:

```
exists v(\bigsqcupec)=\bigcup{i:\mathbb{N}\bullet exists vec(i)}
```

Proof:

```
    \(\left(s, s^{\prime}\right) \in \bigcup\{i: \mathbb{N} \bullet(\) exists \(v e c(i))\}\)
\(\Leftrightarrow\) membership in union
    \(\exists j: \mathbb{N} \bullet\left(s, s^{\prime}\right) \in\) exists \(v e c(j)\)
\(\Leftrightarrow\) membership in function
    \(\exists j: \mathbb{N} \bullet s \in \operatorname{dom}(\) exists \(v e c(j)) \wedge\)
        \(s^{\prime}=(\) exists \(v e c(j))(s)\)
\(\Leftrightarrow\) definition
    \(\exists j: \mathbb{N} \bullet\) unbind \(v s \in \operatorname{dom} \operatorname{ec}(j) \wedge\)
        \(s^{\prime}=\{b: s \mid(\exists x: \operatorname{Val} \bullet e c(j)(\{b \oplus\{v \mapsto x\}\}) \neq\{ \})\}\)
\(\Leftrightarrow\) from (토.5) and (EZ)
    \(\exists j: \mathbb{N} \bullet\) unbind \(v s \in \operatorname{dom} \operatorname{ec}(j) \wedge\)
            \(s^{\prime}=\{b: s \mid(\exists x: \operatorname{Val} \bullet(\bigsqcup e c)(\{b \oplus\{v \mapsto x\}\}) \neq\{ \})\}\)
\(\Leftrightarrow\) reduce scope of \(j\), ( 2.3 )
    unbind \(v s \in \operatorname{dom} \bigsqcup e c \wedge\)
        \(s^{\prime}=\{b: s \mid(\exists x: V a l \bullet(\bigsqcup e c)(\{b \oplus\{v \mapsto x\}\}) \neq\{ \})\}\)
\(\Leftrightarrow\) definition
    \(\left(s, s^{\prime}\right) \in\) exists \(v(\bigsqcup e c)\)
```

The proof for universal quantification is similar.
To prove continuity of parameterised commands, we use the following lemma:

## Lemma F. 7

```
assign v t s\in dome=>(\forallb:s\bulletassign vt{b}\in\operatorname{dom}e)
```

Proof:
assign $v t s \in \operatorname{dom} e$ $\Leftrightarrow$ Exec property (8) and Lemma B.2
$(\forall b:$ assign $v t s \bullet\{b\} \in \operatorname{dom} e)$ $\Leftrightarrow$ definition
$(\forall b:\{b: s \bullet b \oplus\{v \mapsto$ eval $t b\}\} \bullet\{b\} \in \operatorname{dom} e)$ $\Leftrightarrow$ simplify

```
    (\forallb:s\bullet{b\oplus{v\mapsto\textrm{eval}tb}}\in\operatorname{dom}e)
\Leftrightarrow}\mathrm{ definition
    (\forallb:s\bulletassign vt{b}\in\operatorname{dom}e)
```

Theorem F. 8 Parameterised commands are continuous.
Using the definition of call, we need to show:

```
call v(\ec)t=\bigcup{i:\mathbb{N}\bullet call vec(i)t}
```

Proof:
$\left(s, s^{\prime}\right) \in \bigcup\{i: \mathbb{N} \bullet$ call $v e c(i) t\}$
$\Leftrightarrow$ membership in union
$\exists j: \mathbb{N} \bullet\left(s, s^{\prime}\right) \in$ call $v e c(j) t$
$\Leftrightarrow$ membership in function
$\exists j: \mathbb{N} \bullet s \in \operatorname{dom}($ call $v e c(j) t) \wedge$
$s^{\prime}=($ call $v e c(j) t)(s)$
$\Leftrightarrow$ definition
$\exists j: \mathbb{N} \bullet s \subseteq \operatorname{dom}($ eval $t) \wedge$ assign $v t s \in \operatorname{dom} e c(j) \wedge$
$s^{\prime}=\{b: s \mid e c(j)($ assign $v t\{b\}) \neq\{ \}\}$
$\Leftrightarrow$ from Lemma F. 7 and (Z2)
$\exists j: \mathbb{N} \bullet s \subseteq \operatorname{dom}($ eval $t) \wedge \operatorname{assign} v t s \in \operatorname{dom} e c(j) \wedge$
$s^{\prime}=\{b: s \mid(\bigsqcup e c)($ assign $v t\{b\}) \neq\{ \}\}$
$\Leftrightarrow$ reduce scope of $j$, ([.3])
$s \subseteq \operatorname{dom}($ eval $t) \wedge$ assign $v t s \in \operatorname{dom}(\bigsqcup e c) \wedge$ $s^{\prime}=\{b: s \mid(\bigsqcup e c)($ assign $v t\{b\}) \neq\{ \}\}$
$\Leftrightarrow$ definition
$\left(s, s^{\prime}\right) \in$ call $v(\bigsqcup e c) t$

