

**SOFTWARE VERIFICATION RESEARCH CENTRE**  
**DEPARTMENT OF COMPUTER SCIENCE**  
**THE UNIVERSITY OF QUEENSLAND**

Queensland 4072  
Australia

**TECHNICAL REPORT**

No. 94-33

Frame based typing of  
predicate transformers

Brendan P. Mahony

December 1994

Phone: +61 7 3365 1003

Fax: +61 7 3365 1533

**Note:** Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

# Frame based typing of predicate transformers

Brendan P. Mahony

Department of Computer Science  
University of Queensland

## Abstract

The algebraic relationship between predicates and predicate transformers is considered. A canonical decomposition property of transformers is used to develop a typing mechanism for predicate transformer process models based on the input and output interfaces of the processes. Predicate transformers are typed according to the variables read from or written to by the process described by the predicate transformer. This typing system is demonstrated on a small language for specifying dataflow processes.

## 1 Introduction

The relationship between relations and predicates is integral to one approach to the algebraic analysis of the category of monotonic predicate transformers which has been developed by Martin, de Moore, and Gardiner in [10, 3, 5]. Their work approaches the problem from the relation theoretic viewpoint, Whilst simplifying the algebra this makes some of the technical results less accessible to the refinement calculus practitioner. One important result [5, Theorem 3] provides a simple method of typing predicate transformers in terms of the variables names that appear in their input and output interfaces. The purpose of this paper is to explore the implications of this result in the conventional first-order predicate calculus formulation of the refinement calculus and in doing so provide a useful typing system based on the process interfaces described in predicate transformer based specifications.

In other work this author has suggested that predicate transformers are an appropriate formalism for the development of dataflow-like processes [7, 8, 9]. For this purpose it is important to have access to a notion of typing similar to that described above. The major application to date of the typing techniques described in this paper has been in this application of predicate transformers and

as such it provides a convenient source of motivational material for this paper and as a consequence develops as an interwoven supporting theme throughout.

This introduction concludes with some preliminary remarks about the notations used in this paper. In Section 2 variable based typing of predicates is considered. This serves both to motivate the main result of the paper and to lay some important groundwork for its proof. In Section 3 a variable based typing system for predicate transformers is described. Finally, in Section 4 some application specific specialisations of the general typing system are considered and a suitable typing system for a small specification and refinement language is developed. Some of the more involved and unenlightening proofs are given in Appendix A.

## Preliminaries

The aim of this paper is to investigate the algebraic properties of predicate transformers with a view to supporting a strong typing mechanism based on the variables used and constructed. The Z specification language is used throughout to formalise the concepts discussed, though standard syntactic conventions are relaxed or generalised on occasion in the interests of clarity and brevity. The reader unfamiliar with Z is referred to [6, 15].

In preparation, the existence of a collection of variable names

$$[\text{VAR}]$$

and a collection of value names is assumed.

$$[\text{VAL}]$$

The collection of variable names is partitioned into an infinite collection of *process* variables

$$\left| \begin{array}{l} \text{VAR}_\pi : \mathbb{P} \text{VAR} \\ \hline \exists p : \mathbb{N} \succrightarrow \text{VAR}_\pi \bullet \text{true} \end{array} \right.$$

and a collection of *logical variables* which can contain an infinite number of disjoint copies of the process variables.

$$\left| \begin{array}{l} \text{VAR}_\lambda : \mathbb{P} \text{VAR} \\ \hline \text{VAR}_\lambda \cap \text{VAR}_\pi = \emptyset \\ \exists p : \mathbb{N} \succrightarrow (\text{VAR}_\pi \succrightarrow \text{VAR}_\lambda) \bullet \\ \forall n, m : \mathbb{N} \bullet n \neq m \Rightarrow \text{ran}(p \ n) \cap \text{ran}(p \ m) = \emptyset \end{array} \right.$$

**Notation 1.1** The arrow  $A \succrightarrow B$  describes exactly the one-to-one and onto functions from  $A$  to  $B$ . ♡

The process variables are the first-class objects in the sense that only these may be used to represent system observables when developing a formal specification. The logical variables  $\text{VAR}_\lambda$  are used only when developing the semantics of a formal language and associated formal proofs. They help to prevent such problems as variable capture in expressions with bound variables. The strong assumptions about the numbers of logical variables ensure that it is always possible to find fresh logical variables for this purpose. In particular, unlimited distinct copies of the process variables are available, each copy resulting from a particular *decoration* of the process variables.

In this paper process variables are represented by late alphabet lower case Roman characters,  $u, v$ , etc, and their logical variable decorations by late alphabet upper case Roman characters,  $U, V$ , etc. Sets of variables are distinguished by arrows,  $\vec{u}, \vec{v}$ , etc. Predicate valued variables are represented by lower case Greek characters,  $\phi, \psi$ , etc, except for specification predicates which use early alphabet upper case Roman,  $A, E$ , etc. Sets of predicates are distinguished by using upper case Greek characters. Predicate transformers are represented by middle-alphabet, upper-case Roman characters,  $P, Q$ , etc and sets of predicate transformers by the corresponding calligraphic characters,  $\mathcal{P}, \mathcal{Q}$  etc.

## 2 Predicates

A (first-order, classical) predicate over a collection of variables  $\vec{v} \subseteq \text{VAR}$  is an expression which characterises a collection of allowed assignments of values to its variables, known as *states*. In process description, we make use of predicates to express desired properties of and relationships between process components, which are abstractly represented as the values of some collection of the process variables.

Each predicate expression is rendered meaningful through a (semantic) mapping which assigns each expression a set of states which *satisfy* that predicate expression. We abstract away from the concrete syntax of predicate expressions, identifying the collection of predicate expressions with the power set of states.

### 2.1 The lattice of predicates

A (total) state is a total mapping from variable names to values.

$$\text{STATE} == \text{VAR} \longrightarrow \text{VAL}$$

A predicate is a set of states.

$$\text{PRED} == \mathbb{P} \text{STATE}$$

The set of predicates forms a complete lattice under the set intersection and set union operators, called *conjunction* and *disjunction* respectively when applied to predicates.

$$\begin{array}{|l}
\hline
\bigwedge, \bigvee : \mathbb{P} \text{ PRED} \rightarrow \text{PRED} \\
(- \wedge -), (- \vee -) : \text{PRED} \times \text{PRED} \rightarrow \text{PRED} \\
\hline
\forall \Phi : \mathbb{P} \text{ PRED} \bullet \\
\quad \bigwedge \Phi = \bigcap \Phi \\
\quad \bigvee \Phi = \bigcup \Phi \\
\hline
\forall \phi, \psi : \text{PRED} \bullet \\
\quad \phi \wedge \psi = \phi \cap \psi \\
\quad \phi \vee \psi = \phi \cup \psi
\end{array}$$

The bottom and top elements of the lattice are called *false* and *true* respectively.

$$\begin{array}{l}
\text{false} == \emptyset[\text{STATE}] \\
\text{true} == \text{STATE}
\end{array}$$

The lattice of predicates forms a boolean algebra with the addition of the set complement operator, called *not* when applied to predicates.

$$\begin{array}{|l}
\hline
\neg : \text{PRED} \rightarrow \text{PRED} \\
\hline
\forall \phi : \text{PRED} \bullet \neg \phi = \text{STATE} - \phi
\end{array}$$

Other important operators include implication,

$$\begin{array}{|l}
\hline
- \Rightarrow - : \text{PRED} \times \text{PRED} \rightarrow \text{PRED} \\
\hline
\forall \phi, \psi : \text{PRED} \bullet \phi \Rightarrow \psi = \neg \phi \vee \psi
\end{array}$$

variable replacement,

$$\begin{array}{|l}
\hline
-[- \setminus -] : \text{PRED} \times \text{VAR} \times \text{VAL} \rightarrow \text{PRED} \\
\hline
\forall \phi : \text{PRED}; v : \text{VAR}; d : \text{VAL}; \sigma : \text{STATE} \bullet \\
\quad \sigma \in \phi[v \setminus d] \Leftrightarrow \sigma \oplus \{v \mapsto d\} \in \phi
\end{array}$$

the existential and universal quantifiers,

$$\begin{array}{|l}
\hline
\exists - \bullet -, \forall - \bullet - : \text{VAR} \times \text{PRED} \rightarrow \text{PRED} \\
\hline
\forall v : \text{VAR}; \phi : \text{PRED} \bullet \\
\quad \exists v \bullet \phi = \bigvee \{d : \text{VAL} \bullet \phi[v \setminus d]\} \\
\quad \forall v \bullet \phi = \bigwedge \{d : \text{VAL} \bullet \phi[v \setminus d]\}
\end{array}$$

entailment, and equivalence.

$$\begin{array}{|l}
\hline
- \Rightarrow -, - \equiv - : \text{PRED} \leftrightarrow \text{PRED} \\
\hline
\forall \phi, \psi : \text{PRED} \bullet \\
\quad \phi \Rightarrow \psi \Leftrightarrow \phi \subseteq \psi \wedge \\
\quad \phi \equiv \psi \Leftrightarrow \phi = \psi
\end{array}$$

Entailment is the partial order on predicates induced by their lattice structure.

## 2.2 Partial state predicates

A partial state is a partial function from variable names to variable values. The variables,  $\vec{v}$ , in the domain of the state are called the *frame* variables.

$$\text{STATE}_{\vec{v}} == \{\sigma : \text{VAR} \dashrightarrow \text{VAL} \mid \text{dom } \sigma = \vec{v}\}$$

The partial states of particular interest in the design of process specifications are those which are defined on all the process variables, but only a finite number of logical variables.

$$\text{STATE}_{\pi} == \bigcup \{ \vec{Z} : \text{FVAR}_{\lambda} \bullet \text{STATE}_{\vec{Z} \cup \text{VAR}_{\pi}} \}$$

Collections of partial states are called partial state predicates.

$$\text{pPRED}_{\vec{v}} == \text{P STATE}_{\vec{v}}$$

We can redefine the various logical connectives in the partial state context so that  $\text{pPRED}_{\vec{v}}$  forms a boolean algebra.

For the purposes of process description we are interested in the properties of the process state variables and possibly some finite number of logical variables. Predicates over the process variables  $\text{VAR}_{\pi}$  and at most a finite number of logical variables  $\text{VAR}_{\lambda}$  we call process predicates.

$$\text{PRED}_{\pi} == \bigcup \{ \vec{Z} : \text{FVAR}_{\lambda} \bullet \text{pPRED}_{\vec{Z} \cup \text{VAR}_{\pi}} \}$$

## 2.3 Sub-types of predicates

A predicate is said to be *independent* of a variable if and only if the value of that variable does not affect the evaluation of the predicate.

$$\left| \begin{array}{l} \text{indep} : \text{PRED} \leftrightarrow \text{VAR} \\ \hline \forall \phi : \text{PRED}; v : \text{VAR} \bullet \\ \quad \phi \text{ indep } v \Leftrightarrow \\ \quad \forall \sigma : \text{STATE}; a_1, a_2 : \text{VAL} \bullet \\ \quad \quad \sigma \oplus \{v \mapsto a_1\} \in \phi \Leftrightarrow \sigma \oplus \{v \mapsto a_2\} \in \phi \end{array} \right.$$

A predicate is said to be a (total) predicate *over* a collection of variables if and only if it is independent of all other variables.

$$\text{PRED}_{\vec{v}} == \{\phi : \text{PRED} \mid \forall u : \text{VAR} - \vec{v} \bullet \phi \text{ indep } u\}$$

The space  $\text{PRED}_{\vec{v}}$  is closed under the various logical connectives and forms a boolean algebra for every  $\vec{v}$ .

**Lemma 2.1** *For every  $\vec{v} : \text{PVAR}$ , the lattice of partial state predicates,  $\text{pPRED}_{\vec{v}}$ , is lattice isomorphic to the lattice of total predicates over  $\vec{v}$ ,  $\text{PRED}_{\vec{v}}$ .*

See Appendix A for proof.

Following Lemma 2.1 we make no syntactic distinction between partial and total predicates in the following, which is to say we leave implicit any conversions between total and partial predicate valued expressions.

**Example 2.2 (Basis predicates)** A basis for the predicate lattice is formed by the atomic predicates

$$_ = \text{“} = \text{”} \_ = = \lambda v : \text{VAR}; d : \text{VAL} \bullet \{ \sigma : \text{STATE} \mid \sigma(v) = d \}.$$

It is straightforward [16] to show that any predicate is equivalent to an expression involving only such atomic predicates. In fact, for any predicate  $\phi$ ,

$$\phi \equiv \bigvee_{\sigma \in \phi} \bigwedge \{ v : \text{VAR} \bullet v \text{ “} = \text{”} \sigma(v) \}.$$

Thus the basis predicates together with the various logical operators form a semantically complete, if somewhat impractical, language for expressing predicate properties.

Similar decomposition properties hold for each space of partial state predicates,  $\mathbf{pPRED}_{\vec{v}}$ , so that, by Lemma 2.1, if  $\phi$  is a predicate only over the variables  $\vec{v}$  the range of the inner conjunction can be narrowed. In particular, a basis predicate of the form  $v = d$  is a member of  $\mathbf{PRED}_{\{v\}}$  and hence of  $\mathbf{PRED}_{\vec{v}}$  for all  $\vec{v} : \text{VAR}$  such that  $v \in \vec{v}$ . Having thus developed a variable-based typing mechanism for the basis predicates, it is then a relatively straightforward matter to extend it a typing mechanism on the entire basis-predicate language by developing type inference rules for the various logical connectives. In this context a variable is said to “occur free” in a predicate expression if the type inference system cannot be used to show that the predicate is independent of the variable. In practice this corresponds to the variable appearing in a basis predicate within the compound predicate which is not within the scope of a quantifier.

Similar methods are equally applicable to predicate languages with more sophisticated atomic predicates, provided that the dependence of an atomic predicate on a given variable depends solely on whether its name occurs in the atomic predicate expression.

Below, a similar technique is proposed for use with a predicate transformer expression language. ♡

Lemma 2.1 allows two useful ways of looking at the same class of predicates. From the syntactic point of view it is useful to ignore strong typing considerations and have every predicate expression evaluated on total states. Having to apply explicit type “casting” to predicates when composing them with other predicates is not a practical proposition. The syntactic notion of variables that “occur free” in a predicate is then introduced when strong typing properties are required in the course of reasoning. Lemma 2.1 formalises this mechanism

by identifying the total predicates which depend on a particular collection of variables  $\bar{v}$  ( $\text{PRED}_{\bar{v}}$ ) with the partial state predicates with frame  $\bar{v}$  ( $\text{pPRED}_{\bar{v}}$ ). Moving freely between the two formalisms allows the surplus variable names to be ignored or made use of as required, without recourse to purely syntactic notions such as “free”-dom. The goal of the next section is to reproduce this result in the predicate transformer domain and hence enable similar variable handling techniques for predicate transformers, in particular to offer simple support for the strong interface typing required in the development of data-flow processes.

### 3 Monotonic predicate transformers

The lattice of monotonic predicate transformers forms the mathematical basis of several formal approaches to the development of programs [4, 11, 1]. Elsewhere this author has suggested its use in the development of real-time dataflow processes [7]. This application differs from the sequential program case in many ways, but most important with respect to the application of predicate transformers to the problem, is the need to fix clearly the structure of the interface between the process and its environment. In the program case this interface always consists of all program variables visible within the current scope, some of which may be changed and others of which may not. In a dataflow network the process interface assumes much greater importance. Many useful techniques for describing process networks consist of little more than a specification of the interfaces between network elements. The purpose of this section is to develop a simple typing mechanism for predicate transformers capable of describing and enforcing process interfaces.

**Example 3.1** Morgan [11] allows the following refinement of his specification statements (see Example 3.6 below):

$$\frac{x, y : [pre, post]}{y : [pre, post]}.$$

It is possible to make use of the specification statement to describe dataflow processes, using the frame variables (in the above case  $x$  and  $y$ ) as a formal description of the required outputs of the process. Thus in the dataflow process context the above law would correspond to the removal of the variable  $x$  from the output frame, which is clearly not acceptable as it corresponds to a failure to provide the required process interface. ♡

#### 3.1 The lattice of predicate transformers

The monotonic functions from predicates to predicates are called *predicate transformers*.

$$\text{MTRAN} == \text{PRED} \xrightarrow{m} \text{PRED}$$

**Notation 3.2** The arrow  $A \xrightarrow{m} B$  describes exactly the functions  $f$  from  $A$  to  $B$  such that for all  $a, b : A$ , if  $a \leq_A b$  then  $f(a) \leq_B f(b)$ .  $\heartsuit$

Predicate transformers were suggested as a model for programming languages in [4]. Each program statement is associated with a predicate transformer which maps desired properties of the final system state to assumed properties of the initial state, the so-called *weakest precondition* program semantics. Subsequent work by several authors [1, 13, 14, 16, 10] has served to develop the lattice of monotonic predicate transformers into an important semantic domain for the development of sequential programs. Elsewhere [8, 7] this author has suggested the use of predicate transformers in the development of dataflow processes by way of a weakest *assumption* semantics for dataflow machines.

The lattice structure of the predicate transformer domain is lifted from the lattice structure of the underlying predicate domain.

$$\begin{array}{|l}
 \hline
 \prod, \sqcup : \text{P MTRAN} \rightarrow \text{MTRAN} \\
 \neg \square \neg, \neg \sqcup \neg : \text{MTRAN} \times \text{MTRAN} \rightarrow \text{MTRAN} \\
 \mathbf{abort}, \mathbf{magic} : \text{MTRAN} \\
 \hline
 \forall \phi : \text{PRED} \bullet \\
 \quad \forall \mathcal{P} : \text{P MTRAN} \bullet \\
 \quad \quad (\prod \mathcal{P})(\phi) = \bigwedge \{P : \mathcal{P} \bullet P(\phi)\} \\
 \quad \quad (\sqcup \mathcal{P})(\phi) = \bigvee \{P : \mathcal{P} \bullet P(\phi)\} \\
 \quad \forall P, Q : \text{MTRAN} \bullet \\
 \quad \quad (P \sqcap Q)(\phi) = P(\phi) \wedge Q(\phi) \\
 \quad \quad (P \sqcup Q)(\phi) = P(\phi) \vee Q(\phi) \\
 \quad \mathbf{abort}(\phi) = \text{false} \\
 \quad \mathbf{magic}(\phi) = \text{true}
 \end{array}$$

This lattice structure defines a partial order on predicate transformers which is suitable for modelling the correctness or *refinement* relation between specifications and implementations.

$$\begin{array}{|l}
 \hline
 \_ \sqsubseteq \_ : \text{MTRAN} \leftrightarrow \text{MTRAN} \\
 \hline
 \forall P, Q : \text{MTRAN} \bullet \\
 \quad (P \sqsubseteq Q) \Leftrightarrow (\forall \phi : \text{PRED} \bullet P(\phi) \Rightarrow Q(\phi))
 \end{array}$$

### 3.2 Partial predicate transformers

The monotonic functions between partial predicate domains are called *partial* predicate transformers. The collection of monotonic functions from  $\text{pPRED}_{\vec{v}}$  to  $\text{pPRED}_{\vec{u}}$  is called  $\text{pMTRAN}_{\vec{u} \rightarrow \vec{v}}$ , the reverse ordering reflecting their use in weakest precondition and assumption models. Elements of  $\text{pMTRAN}_{\vec{u} \rightarrow \vec{v}}$  are used to model processes that construct systems with state frame  $\vec{v}$  from systems

with state frame  $\vec{u}$ .

$$\mathbf{pMTRAN}_{\vec{u} \rightarrow \vec{v}} == \mathbf{pPRED}_{\vec{v}} \xrightarrow{m} \mathbf{pPRED}_{\vec{u}}$$

Since, by Lemma 2.1,  $\mathbf{pPRED}_{\vec{v}}$  and  $\mathbf{pPRED}_{\vec{u}}$  are isomorphic to  $\mathbf{PRED}_{\vec{v}}$  and  $\mathbf{PRED}_{\vec{u}}$  respectively, we can identify each member of  $\mathbf{pMTRAN}_{\vec{u} \rightarrow \vec{v}}$  with a collection of total state predicate maplets from  $\mathbf{PRED}_{\vec{v}} \xrightarrow{m} \mathbf{PRED}_{\vec{u}}$ , which is to say a partial function on total state predicates, but it is not immediately clear how such a partial function may be extended to a member of  $\mathbf{MTRAN}$  in a consistent manner.

**Example 3.3** In the dataflow refinement calculus all of the well-formed expressions represent processes, which is to say that they are all mappings from predicates over the process variables to predicates over the process variables.

$$\mathbf{PROC} == \mathbf{pMTRAN}_{\mathbf{VAR}_{\pi} \rightarrow \mathbf{VAR}_{\pi}}$$

However in describing the semantics of certain elements of the refinement calculus (for example local variables) it is necessary to make use of logical variables to protect the user of the calculus from inadvertent naming conflicts. Thus the mappings from process predicates to process predicates becomes of interest. In fact, not all such mappings are of interest, but rather only those which satisfy certain properties making them what is termed *process-like*. Since the intended use of the logical variables is to avoid naming conflicts by holding in safekeeping the value of certain external variables, process models should neither introduce dependencies on logical variables, nor affect in any way the values of the logical variables they act upon. In fact, it is not at this stage of the paper possible to formalise these requirements, so for the moment it is simply noted that the process-like predicate transformers  $\mathbf{MTRAN}_{\pi}$  are a strict subclass of the monotonic functions on process predicates.

$$\mathbf{MTRAN}_{\pi} \subset \mathbf{PRED}_{\pi} \xrightarrow{m} \mathbf{PRED}_{\pi}$$

A formal characterisation of  $\mathbf{MTRAN}_{\pi}$  requires the sub-typing mechanism which is developed below and is presented in Definition 3.14. ♡

### 3.3 Sub-types of predicate transformers

The purpose of this section is to identify a correspondence between spaces of partial predicate transformers  $\mathbf{pMTRAN}_{\vec{u} \rightarrow \vec{v}}$  and classes of process-like predicate transformers analogous to that developed between partial and total predicates in Section 2.3. In this way the domain of process-like predicate transformers becomes able to support the development of processes with strongly specified interfaces, as is true of the partial predicate transformers, but with the advantage that each process has well-defined behaviour in all environments. Thus process-like predicate transformers become a useful all-purpose semantic domain for the development of dataflow processes.

**Example 3.4 (Axiomatic sub-typing)** One approach to typing predicate transformers is to try to extend the predicate notion of variable dependence. Butler [2] defines a predicate transformer  $P$  to be independent of a variable  $x$  if and only if

- whenever a predicate  $\phi$  is independent of  $x$  then so is  $P(\phi)$ , and
- for any  $x$ -predicate  $\psi \in \text{PRED}_x$  and arbitrary predicate  $\phi$ ,

$$P(\phi \vee \psi) \equiv P(\phi) \vee (\psi \wedge P(\text{true})).$$

The first axiom ensures that the predicate transformer does not introduce any dependence on  $x$ , whilst the second says that (as far as is possible) it does not remove (non-magically) any dependence on  $x$ . That is, the variable  $x$  is not involved in the action of  $P$ . ♡

This definition has some drawbacks. It is not entirely analogous to the predicate case as it is not obviously tied to a syntactic criteria for determining such independence. The second axiom appears at first sight to be somewhat arbitrary, primarily because of the lack of syntactic interpretation. More importantly, it ties domain and range predicates to the same state frame where our purpose requires the flexibility of allowing domain and range state frames to be independent of each other. Instead this paper develops criteria which can be strongly associated with syntactic structure. The algebraic results of [10] are used to define a syntax-motivated extension of all partial predicate transformers to process-like predicate transformers in such a way that behaviour of the transformer on the partial state is preserved and that it does not interfere (non-magically) with variables outside its original scope. Syntactic motivations then identify the appropriate subtypes of the process predicate transformer domain.

### 3.4 Embedding predicates in the transformer lattice

In [10] Martin identifies two natural embeddings of the predicate lattice into the predicate transformer lattice.

**Definition 3.5** *Suppose that  $\vec{u}, \vec{v} \subseteq \text{VAR}$  and that  $A \in \text{pPRED}_{\vec{u} \cup \vec{v}}$ .*

*The universal image of  $A$  over  $\vec{v}$  within  $\vec{u}$ ,*

$$\vec{u}[A]_{\vec{v}} == \lambda \phi : \text{pPRED}_{\vec{v}} \bullet (\forall \vec{v} - \vec{u} \bullet A \Rightarrow \phi),$$

*is a member of  $\text{pMTRAN}_{\vec{u} \rightarrow \vec{v}}$ .*

*The existential image of  $A$  over  $\vec{v}$  within  $\vec{u}$ ,*

$$\vec{u}\langle A \rangle_{\vec{v}} == \lambda \phi : \text{pPRED}_{\vec{v}} \bullet (\exists \vec{v} - \vec{u} \bullet A \wedge \phi),$$

*is also a member of  $\text{pMTRAN}_{\vec{u} \rightarrow \vec{v}}$ .*

**Example 3.6 (Specification statements)** Predicate transformer expressions of the form

$$\bar{u}\langle A \rangle_{\bar{x}} \circ \bar{x}[E]_{\bar{v}} = \lambda \phi : \mathbf{pPRED}_{\bar{v}} \bullet \exists \bar{X} \bullet A \wedge (\forall \bar{v} \bullet E \Rightarrow \phi)$$

bear a strong resemblance to the atomic terms of the refinement calculus, specification statements. The specification statement [11] consists of an update frame and two predicates and is defined by

$$\bar{v} : [A, E] = \lambda \phi : \mathbf{PRED}_{\pi} \bullet A \wedge \forall \bar{v} \bullet E \Rightarrow \phi.$$

When restricted to  $\mathbf{PRED}_{\bar{v}}$  this is equivalent to

$$\bar{u}\langle A \rangle_{\emptyset} \circ \emptyset[E]_{\bar{v}} = \lambda \phi : \mathbf{pPRED}_{\bar{v}} \bullet A \wedge \forall \bar{v} \bullet E \Rightarrow \phi.$$

The more general form  $\bar{u}\langle A \rangle_{\bar{x}} \circ \bar{x}[E]_{\bar{v}}$  may be seen as the join of a collection of specification statements parameterised by the allowed values of the variables  $\bar{X}$ . That is, for  $\phi : \mathbf{pPRED}_{\bar{v}}$ ,

$$(\bar{u}\langle A \rangle_{\bar{x}} \circ \bar{x}[E]_{\bar{v}})(\phi) = \left( \bigsqcup_{v : \mathbf{STATE}_{\bar{x}}} \bullet v : [A[\bar{X} \setminus \sigma(\bar{X})], E[\bar{X} \setminus \sigma(\bar{X})]] \right) (\phi).$$

♡

The update frame and the program environment provide useful syntactic criteria for the variable-based categorisation of specification statements and predicate embeddings in general. Moreover, it is easy to develop a natural correspondence between the partial and total predicate views in the case of specification statements and predicate embeddings.

**Example 3.7 (Extending predicate embeddings)** We have already seen in Example 3.6 that the specification statement  $\bar{v} : [A, E]$  may be associated with the partial predicate transformer  $\lambda \phi : \mathbf{pPRED}_{\bar{v}} \bullet A \wedge (\forall \bar{v} \bullet E \Rightarrow \phi)$ . Equally, it is clear how to extend predicate embeddings such as  $\bar{u}\langle A \rangle_{\bar{v}}$  and  $\bar{u}[E]_{\bar{v}}$  to process predicate transformers so as to respect their behaviour on  $\mathbf{PRED}_{\bar{v}}$ .

A suitable extension for  $\bar{u}\langle A \rangle_{\bar{v}}$  is

$$\lambda \phi : \mathbf{PRED}_{\pi} \bullet (\exists \bar{v} - \bar{u} \bullet A \wedge \phi)$$

and for  $\bar{u}[E]_{\bar{v}}$  is

$$\lambda \phi : \mathbf{PRED}_{\pi} \bullet (\forall \bar{v} - \bar{u} \bullet A \Rightarrow \phi)$$

♡

**Example 3.8 (A magical specification)** Consider  $\bar{u}[\mathbf{false}]_{\bar{v}}$ . It is easy to see that for any  $z \notin \bar{v} - \bar{u}$ ,  $d_1, d_2 \in \mathit{VAL}$ ,

$$z = d_1 \Rightarrow \forall \bar{v} - \bar{u} \bullet \mathbf{false} \Rightarrow z = d_2.$$

Thus if

$$\lambda \phi : \text{PRED}_\pi \bullet (\forall \vec{v} - \vec{u} \bullet \text{false} \Rightarrow \phi)$$

is to be the process predicate transformer extension of  $\vec{u}[\text{false}]_{\vec{v}}$ , extensions of magical processes must be able to interfere with the values of environment variables.  $\heartsuit$

In order to construct an extension with these properties an important algebraic result due to Gardiner et al [5] is introduced.

**Proposition 3.9** *For all  $P \in \text{pMTRAN}_{\vec{u} \rightarrow \vec{v}}$ ,  $\vec{u}, \vec{v} : \text{PVAR}_\pi$ , there exists  $\vec{X} : \text{PVAR}_\lambda$ ,  $A : \text{PRED}_{\vec{u} \cup \vec{X}}$ , and  $E : \text{PRED}_{\vec{v} \cup \vec{X}}$ , such that*

$$P = \vec{u}\langle A \rangle_{\vec{X}} \circ \vec{X}[E]_{\vec{v}}.$$

*This result, expressed in a slightly different form, has been proved independently by von Wright [16].*

*That is, every partial<sup>1</sup> process predicate transformer is the join of a collection of specification statements.*

Since specification statements have a quite natural expression at the syntactic level, it is easy to view a specification statement, and indeed the join of a collection of specification statements, as being a member of a wide class of predicate transformer subtypes. That is, the syntactic map

$$\phi \mapsto \exists \vec{X} \bullet A \wedge \forall \vec{v} \bullet E \Rightarrow \phi$$

is applicable to any predicate expression  $\phi$  (in which the  $\vec{X}$  do appear free) and may be viewed as a predicate over any collection of variables including the free variables appearing in the expression. We make use of this property to determine the *natural* extension of a partial predicate transformer to a process predicate transformer.

$$\boxed{\begin{array}{l} \overline{[\vec{u}, \vec{v} : \text{PVAR}_\pi]} \\ \overline{-^e : \text{pMTRAN}_{\vec{u} \rightarrow \vec{v}} \rightarrow \text{MTRAN}_\pi} \\ \forall P : \text{pMTRAN}_{\vec{u} \rightarrow \vec{v}} \bullet \\ \quad \forall \vec{W} : \text{FVAR}_\lambda \bullet \forall \phi : \text{PRED}_{\vec{W} \cup \text{PVAR}_\pi} \bullet \\ \quad \quad \forall \vec{X} : \text{VAR}_\lambda - \vec{W} \bullet \forall A : \text{pPRED}_{\vec{u} \cup \vec{X}}; E : \text{pPRED}_{\vec{v} \cup \vec{X}} \bullet \\ \quad \quad \quad (\forall \psi : \text{pPRED}_{\vec{v}} \bullet P(\psi) \equiv (\vec{u}\langle A \rangle_{\vec{X}} \circ \vec{X}[E]_{\vec{v}})(\psi)) \Rightarrow \\ \quad \quad \quad P^e(\phi) \equiv (\exists \vec{X} \bullet A \wedge \forall \vec{v} \bullet E \Rightarrow \phi) \end{array}}$$

Note that care is taken to avoid the capture of free logical variables.

<sup>1</sup>The result of Gardiner et al actually holds for total predicate transformers as well, but this cannot be expressed in the predicate realm since there are no unused variables in a total predicate transformer.

**Proposition 3.10**  $(-^e)$  is a well-defined lattice-morphism and  $\text{PRED}_{\vec{v}} \triangleleft P^e = P$ .

For proof see Appendix A.

### 3.5 The sub-typing mechanism

This natural extension is used to motivate a typing mechanism on process predicate transformers which reflects their relationship with the naturally typed spaces of partial state predicate transformers.

**Theorem 3.11** For all  $\vec{u}, \vec{v} : \mathbb{P}\text{VAR}_\pi$  the partial predicate transformer space  $\text{pMTRAN}_{\vec{u} \rightarrow \vec{v}}$  is lattice isomorphic to its image in  $\text{MTRAN}_\pi$  under  $(-^e)$ ,

$$\text{MTRAN}_{\vec{u} \rightarrow \vec{v}} = \{P : \text{MTRAN}_{\vec{u} \rightarrow \vec{v}} \bullet P^e\}.$$

For proof see Appendix A.

In light of Theorem 3.11 we identify  $\text{MTRAN}_{\vec{u} \rightarrow \vec{v}}$  with  $\text{pMTRAN}_{\vec{u} \rightarrow \vec{v}}$ , making distinction between the two representations only where it simplifies argument.

**Corollary 3.12** A process transformer  $P \in \text{MTRAN}_\pi$  is a member of  $\text{MTRAN}_{\vec{u} \rightarrow \vec{v}}$  if and only if for all  $\vec{W} : \text{FVAR}_\lambda$  there exists  $\vec{X} : \mathbb{P}\text{VAR}_\lambda - \vec{W}$ ,  $A : \text{PRED}_{\vec{u} \cup \vec{X}}$ , and  $E : \text{PRED}_{\vec{v} \cup \vec{X}}$  such that for  $\phi \in \text{PRED}_{\text{VAR}_\pi \cup \vec{W}}$  over the process variables and the logical variables  $\vec{W}$ ,

$$P(\phi) \equiv (\exists \vec{X} \bullet A \wedge \forall \vec{v} \bullet E \Rightarrow \phi)$$

**Example 3.13 (Butler's axioms revisited)** Suppose that  $P \in \text{MTRAN}_{\vec{u} \rightarrow \vec{v}}$  and that  $x \in \text{VAR}_\pi$  is in neither  $\vec{u}$  nor  $\vec{v}$ , then  $P$  is independent of  $x$  in the sense of Example 3.4. Clearly if  $\phi$  is independent of  $x$  then so is  $P(\phi)$  and for any  $x$ -predicate  $\psi \in \text{PRED}_x$  and process predicate  $\phi \in \text{PRED}_{\text{VAR}_\pi \cup \vec{W}}$ ,

$$\begin{aligned} & P(\phi \vee \psi) \\ & \equiv \exists \vec{X} \bullet A \wedge \forall \vec{v} \bullet E \Rightarrow \phi \vee \psi \\ & \equiv \exists \vec{X} \bullet A \wedge ((\forall \vec{v} \bullet E \Rightarrow \phi) \vee \psi) \\ & \equiv \exists \vec{X} \bullet A \wedge \forall \vec{v} \bullet E \Rightarrow \phi \vee \\ & \quad (\exists \vec{X} \bullet A) \wedge \psi \\ & \equiv P(\phi) \vee (P(\text{true}) \wedge \psi) \end{aligned}$$

for some  $\vec{X} \in \text{VAR}_\lambda - \vec{W}$ ,  $A \in \text{PRED}_{\vec{u} \cup \vec{X}}$ , and  $E \in \text{PRED}_{\vec{v} \cup \vec{X}}$ . ♡

It is now possible to formally identify the class of process-like predicate transformers,  $\text{MTRAN}_\pi$ .

**Definition 3.14** *The process-like predicate transformers are precisely those members of  $\text{PRED}_\pi \xrightarrow{m} \text{PRED}_\pi$  which are the extensions of some process predicate transformer from  $\text{PROC}$ . That is  $P \in \text{MTRAN}_\pi$  if and only if for all  $\vec{W} : \mathbb{F}\text{VAR}_\lambda$  there exists  $\vec{X} : \mathbb{P}\text{VAR}_\lambda - \vec{W}$  and  $A, E : \text{PRED}_{\text{VAR}_\pi \cup \vec{W}}$  such that for  $\phi \in \text{PRED}_{\text{VAR}_\pi \cup \vec{W}}$*

$$P(\phi) \equiv (\exists \vec{X} \bullet A \wedge \forall \text{VAR}_\pi \bullet E \Rightarrow \phi)$$

**Corollary 3.15** *The image of  $\text{PROC}$  under  $(-^e)$  is precisely  $\text{MTRAN}_\pi$  and  $(-^e)$  forms a lattice isomorphism between the two.*

**Proof:** Simple application of Corollary 3.12 and Theorem 3.11.  $\square$

## 4 Application specific subtyping systems

This section introduces some application specific terminology which is useful in the typing analysis of refinement calculi for both sequential programs and dataflow processes.

**Definition 4.1** *Suppose that  $P \in \text{MTRAN}_{\vec{u} \rightarrow \vec{v}}$ .*

*We call the pair  $(\vec{u}, \vec{v})$  the process context,  $\vec{u}$  the environment frame, and  $\vec{v}$  the component frame of  $P$ . Note that these frames are not unique, since in general  $P$  can be viewed within many contexts.*

*If  $z \in \text{VAR}_\pi$  is not a component of  $P$  (ie  $z \notin \vec{v}$ ) we say that  $P$  respects  $z$ .*

*If  $z \in \text{VAR}_\pi$  is both a component of  $P$  and part of the environment of  $P$  (ie  $z \in \vec{v} \cap \vec{u}$ ) we say that  $P$  updates  $z$ .*

*If  $z \in \text{VAR}_\pi$  is a component of  $P$  and is not part of the environment of  $P$  (ie  $z \in \vec{v} - \vec{u}$ ) we say that  $P$  constructs  $z$ .*

*Note that for each transformer  $P$  the collections of updated and constructed variables are dependent on the context being considered.*

The notions of respect, update, and construct suggest more sophisticated modes of categorising the context of predicate transformers.

**Definition 4.2** *If  $P$  is in both  $\text{MTRAN}_{\vec{u} \rightarrow \vec{u}}$  and  $\text{MTRAN}_{\vec{u} \rightarrow \vec{z}}$  for some  $\vec{z} \subseteq \vec{u}$  we say that  $P$  is a  $\vec{z}$ -updater within environment  $\vec{u}$ . We write  $\text{MTRAN}_{\vec{u} \xrightarrow{\ominus} \vec{z}}$  for the collection of all such processes.*

*If  $P$  is in both  $\text{MTRAN}_{\vec{u} \rightarrow \vec{u} \cup \vec{z}}$  and  $\text{MTRAN}_{\vec{u} \rightarrow \vec{z}}$  for some  $\vec{z} \cap \vec{u} = \emptyset$  we say that  $P$  is a  $\vec{z}$ -constructor within environment  $\vec{u}$ . We write  $\text{MTRAN}_{\vec{u} \xrightarrow{\oplus} \vec{z}}$  for the collection of all such processes.*

The notion of updaters has obvious application in the modeling of sequential programs where the purpose of each command is to update the values in a collection of named storage locations. The notion of constructors has application

in the modeling of dataflow processes where the purpose of each process is to construct its output streams as a function of its input streams.

The most natural way to view updaters and constructors is as extensions of partial state transformers from  $\vec{z}$  to  $\vec{u}$  into the larger realm of partial state transformers from  $\vec{u}$  to  $\vec{u}$  and  $\vec{u} \cup \vec{z}$  to  $\vec{u}$  respectively. These distinctions are most illuminating when viewed in the partial state context.

**Proposition 4.3** *A process  $P$  is in  $\text{MTRAN}_{\vec{u} \xrightarrow{\Phi} \vec{z}}$ ,  $\vec{z} \subseteq \vec{u}$ , if and only if there exists  $\vec{X} \subseteq \text{VAR}_\lambda$ ,  $A, E \in \text{PRED}_{\vec{u} \cup \vec{X}}$  such that*

$$P = (\lambda \phi : \text{PRED}_{\vec{u}} \bullet \exists \vec{X} \bullet A \wedge \forall \vec{z} \bullet E \Rightarrow \phi)^e$$

*A process  $P$  is in  $\text{MTRAN}_{\vec{u} \xrightarrow{+} \vec{z}}$ ,  $\vec{u} \cap \vec{z} = \emptyset$ , if and only if there exists  $\vec{X} \subseteq \text{VAR}_\lambda$ ,  $A \in \text{PRED}_{\vec{u} \cup \vec{X}}$ , and  $E \in \text{PRED}_{\vec{u} \cup \vec{z} \cup \vec{X}}$  such that*

$$P = (\lambda \phi : \text{PRED}_{\vec{u} \cup \vec{z}} \bullet \exists \vec{X} \bullet A \wedge \forall \vec{z} \bullet E \Rightarrow \phi)^e$$

**Example 4.4 (Type inference in a dataflow language)** A minimal language for the specification of dataflow processes is described by the following partial BNF-style grammar.

$$\begin{aligned} DF & ::= \text{"+" } Z \text{" :"} \text{" [" } P \text{" ,"} P \text{" ]"} \\ & \quad | \text{" [" } \text{"con"} Z \text{" \bullet"} DF \text{" ]"} \\ & \quad | \text{" [" } \text{"param"} Z \text{" \bullet"} DF \text{" ]"} \\ & \quad | \text{" [" } DF \text{" \setminus"} Z \text{" ]"} \\ & \quad | DF \text{" ||"} DF \\ RP & ::= DF \text{" \sqsubseteq"} DF \\ P & ::= \text{predicate expression} \\ Z & ::= \text{variable declaration} \end{aligned}$$

The classes of constructor predicate transformers provides a natural typing domain for this language. In the following well-definedness and typing inference are portrayed in the vertical natural deduction style. Well-definedness of an expression is indicated by the  $\delta$  operator. A type may only be inferred for an expression if it is well-defined.

A specification statement is well-defined provided that the constructed variables are not mentioned in the first predicate, called the *assumption*. Such a specification statement constructs its frame variables, within the environment described by its assumption.

$$\frac{\begin{array}{l} \vec{u} \cap \vec{z} = \emptyset \\ A \in \text{PRED}_{\vec{u}} \\ E \in \text{PRED}_{\vec{u} \cup \vec{z}} \end{array}}{(+\vec{z} : [A, E]) \in \text{MTRAN}_{\vec{u} \xrightarrow{+} \vec{z}}}$$

Expressions of the form  $[[\mathbf{con} \vec{c} \bullet P]]$  and  $[[\mathbf{param} \vec{c} \bullet P]]$  localise environment components of the enclosed specification. A constant represents an abstraction of the physical environment, whilst a parameter determines a collection of acceptable designs. The effect of the enclosure is to protect the local environment from any interaction with like-named external environment components.

$$\frac{\vec{c} \subseteq \vec{u} \quad P \in \text{MTRAN}_{\vec{u} \rightarrow \vec{z}}}{([[\mathbf{con} \vec{c} \bullet P]]), ([[\mathbf{param} \vec{c} \bullet P]]) \in \text{MTRAN}_{(\vec{u}-\vec{c}) \rightarrow \vec{z}}}$$

Expressions of the form  $[[P \setminus \vec{c}]]$  localise constructed components of the enclosed specification. This allows the introduction of interaction channels between network components. The effect of the enclosure is to protect the local constructions from external interactions.

$$\frac{\vec{c} \subseteq \vec{z} \quad P \in \text{MTRAN}_{\vec{u} \rightarrow \vec{z}}}{([P \setminus \vec{c}]) \in \text{MTRAN}_{\vec{u} \rightarrow (\vec{z}-\vec{c})}}$$

Note that the side-conditions on the local variables in these laws place restrictions on the specification language that could not be expressed without a variable-based typing system.

Expressions of the form  $P \parallel Q$  form an interacting network of processes. Within the network, the constructions of each component process form part of the environment of all other component processes.

$$\frac{\text{disjoint } \langle \vec{u}, \vec{x}, \vec{y} \rangle \quad P \in \text{MTRAN}_{(\vec{u} \cup \vec{y}) \rightarrow \vec{x}} \quad Q \in \text{MTRAN}_{(\vec{u} \cup \vec{x}) \rightarrow \vec{y}}}{(P \parallel Q) \in \text{MTRAN}_{\vec{u} \rightarrow (\vec{x} \cup \vec{y})}}$$

It is always acceptable to place a process in any environment which includes all of the inputs it makes use of, but none of its constructs.

$$\frac{\vec{u} \subseteq \vec{v} \quad \vec{v} \cap \vec{z} = \emptyset \quad P \in \text{MTRAN}_{\vec{u} \rightarrow \vec{z}}}{P \in \text{MTRAN}_{\vec{v} \rightarrow \vec{z}}}$$

Finally, a refinement predicate,  $P \sqsubseteq Q$ , is well defined only when the specifications being compared have the same environments and constructions.

$$\frac{P, Q \in \text{MTRAN}_{\vec{u} \rightarrow \vec{z}}}{\delta(P \sqsubseteq Q)}$$

♡

## 5 Summary

Existing predicate transformer based languages either assume that all variables are always available [13] or that each command acts upon a specific subset of the variable names [16]. In the former approach much valuable information is embodied solely in the syntactic representation and is not readily accessible at the semantic level. The latter approach begs the question of when it is possible to move transparently between variable contexts. This paper builds a bridge between these two approaches by developing a variable based subtyping mechanism on total predicate transformers. A decomposition theorem of Martin et al is used to develop a suitable embedding of the partial state approach within the total state formalism. This embedding induces a variable based typing mechanism on the space of total predicate transformers.

The characterising aspect of dataflow processes is that they are constructive. Dataflow processes accept input streams from their environment and construct output streams in response. Thus in modeling dataflow processes it is natural to make use of  $\vec{z}$ -constructors, where  $\vec{z}$  represents a collection of output channels. This presents something of a problem in the use of the full power of the predicate transformer model. A  $\vec{z}$ -constructor can always be refined by a  $\vec{z}$ -updater (for example itself) and a  $\vec{z}$ -updater can be refined by  $\vec{x}$ -updaters for  $\vec{x} \subseteq \vec{z}$ . However this refinement path is not acceptable in the development of dataflow processes, since it corresponds to the implementation of a specification with a process that does not construct all of the required outputs. The typing method developed in this paper is used to prevent such refinements by requiring that the entire refinement process be conducted within the appropriate domain of constructors.

## Acknowledgments

The work reported in this paper has been supported by Australian Research Council grant number A4913006: *Formal methods for the specification and refinement of time-based systems and processes*.

## References

- [1] R. J. R. Back. A calculus of program derivations. *Acta Informatica*, 25:593–624, 1988.
- [2] M. J. Butler. *A CSP Approach to Action Systems*. PhD thesis, Wolfson College, Oxford University, Michaelmas Term 1992.
- [3] O. de Moor. *Categories, Relations, and Dynamic Programming*. PhD thesis, Oxford University Computing Laboratory, Programming Research Group, 1992. Available as Technical Monograph PRG-98.

- [4] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, New Jersey, 1976.
- [5] P. Gardiner, C. Martin, and O. de Moore. An algebraic construction of predicate transformers. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Proceedings of the second international conference on the Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 100–121, 1993.
- [6] I. J. Hayes, editor. *Specification Case Studies*. Prentice Hall International, second edition, 1993.
- [7] B. P. Mahony. Using the refinement calculus for dataflow processes. Technical Report TR94-32, Software Verification Research Centre, University of Queensland, 1994.
- [8] B. P. Mahony and I. J. Hayes. A case-study in timed refinement: A central heater. In J. M. Morris and R. C. Shaw, editors, *Proceedings of the 4<sup>th</sup> Refinement Workshop*, Workshops in Computing, pages 138–149. Springer-Verlag, 1991.
- [9] B. P. Mahony, C. Millerchip, and I. J. Hayes. A boiler control system: A case study in timed refinement. In *International Invitational Workshop - Design and Review of Software Controlled Safety-Related Systems*, Ottawa, June 1993.
- [10] C. Martin. *Preordered Categories and Predicate Transformers*. PhD thesis, Oxford University Computing Laboratory, Programming Research Group, 1991.
- [11] C. C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science, C.A.R. Hoare series editor. Prentice Hall, second edition, 1994.
- [12] C. C. Morgan and T. Vickers, editors. *On the Refinement Calculus*. Springer-Verlag, 1992.
- [13] C.C. Morgan. The specification statement. *ACM Trans. Prog. Lang. and Sys.*, 10(3), July 1988. Reprinted in [12].
- [14] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.
- [15] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, second edition, 1992.
- [16] J. von Wright. *A Lattice-theoretical Basis for Program Refinement*. PhD thesis, Åbo Akademi, Finland, 1991.

## A Proofs

**Proof:** (of Lemma 2.1)

We map each set of partial states over  $\vec{v}$  into a set of total states by relating each partial state to all the total states which equal it on its domain.

$$\begin{aligned}\gamma &== \lambda s : \text{VAR} \rightarrow \text{VAL} \bullet \{\sigma : \text{STATE} \mid (\text{dom } s) \triangleleft \sigma = s\} \\ \Gamma &== \lambda \phi : \mathbb{P}(\text{VAR} \rightarrow \text{VAL}) \bullet \bigcup \gamma(\phi)\end{aligned}$$

We proceed by demonstrating that  $(\text{pPRED}_{\vec{v}}) \triangleleft \Gamma \triangleright \text{PRED}_{\vec{v}}$  is a bijection.

Consider arbitrary  $\phi \in \text{PRED}_{\vec{v}}$ .

**Case (Onto)**

The set

$$\{\sigma : \phi \bullet \vec{v} \triangleleft \sigma\}$$

is a set of partial states over  $\vec{v}$ . Since  $\phi$  is independent of all variables outside  $\vec{v}$ ,  $\Gamma(\{\sigma : \phi \bullet \vec{v} \triangleleft \sigma\}) = \phi$ .

**Case (One to one)**

Further,  $\{\sigma : \phi \bullet \vec{v} \triangleleft \sigma\}$  is the only set of partial states which  $\Gamma$  maps to  $\phi$ , since  $\gamma$  partitions  $\text{PRED}_{\vec{v}}$ .

$$\forall s_1, s_2 : \text{STATE}_{\vec{v}} \bullet s_1 \neq s_2 \Rightarrow \gamma(s_1) \cap \gamma(s_2) = \emptyset \quad (*)$$

So if  $\psi : \text{pPRED}_{\vec{v}}$  is a set of partial states distinct from  $\{\sigma : \phi \bullet \vec{v} \triangleleft \sigma\}$ , either there is some  $\sigma \in \phi$  such that  $\vec{v} \triangleleft \sigma \notin \psi$  and hence  $\sigma \notin \Gamma(\psi)$  or there is some  $s \in \psi$ , but  $s \notin \{\sigma : \phi \bullet \vec{v} \triangleleft \sigma\}$  so that  $\gamma(s) \cap \bigcup(\{\sigma : \phi \bullet \vec{v} \triangleleft \sigma\}) = \emptyset$  and hence  $\Gamma(\psi) \neq \phi$ .

Finally, it is easy to check that the property (\*) ensures that  $\Gamma$  is a boolean-algebra-morphism, since for every predicate,  $\phi$ , in  $\text{PRED}_{\vec{v}}$ ,  $\gamma(\{\sigma : \phi \bullet \vec{v} \triangleleft \sigma\})$  forms a disjoint partition of  $\phi$ , canonical in the sense that each partition element is minimal within  $\text{PRED}_{\vec{v}}$ .  $\square$

**Proof:** (of Proposition 3.10)

Choose  $P : \text{pMTRAN}_{\vec{u} \rightarrow \vec{v}}$ ,  $\vec{W} : \text{FVAR}_{\lambda}$ , and  $\phi : \text{PRED}_{\vec{W} \cup \text{VAR}_{\pi}}$ . Choose any  $\vec{X} : \text{PVAR}_{\lambda} - \vec{W}$ ,  $A : \text{pPRED}_{\vec{u} \cup \vec{X}}$ , and  $E : \text{pPRED}_{\vec{v} \cup \vec{X}}$ , such that for all  $\psi : \text{pPRED}_{\vec{v}}$

$$P(\psi) \equiv \exists \vec{X} \bullet A \wedge \forall \vec{v} \bullet E \Rightarrow \psi.$$

By Proposition 3.9 such a choice is possible.

We show that the value of

$$\exists \vec{X} \bullet A \wedge \forall \vec{v} \bullet E \Rightarrow \phi$$

does not depend on the choice of  $\vec{X}$ ,  $A$ , or  $E$ , so that  $P^e$  is uniquely defined for  $\phi$ .

$$\begin{aligned}
& (\exists \vec{X} \bullet A \wedge \forall \vec{v} \bullet E \Rightarrow \phi) \\
& \equiv (\exists \vec{X} \bullet A \wedge \forall \vec{v} \bullet E \Rightarrow \bigvee_{\sigma \in \phi} \bigwedge \{v : \text{VAR}_\pi \cup \vec{W} \bullet v = \sigma(v)\}) \\
& \equiv \bigvee_{\sigma \in \phi} (\exists \vec{X} \bullet A \wedge \forall \vec{v} \bullet E \Rightarrow \bigwedge \{v : \text{VAR}_\pi \cup \vec{W} \bullet v = \sigma(v)\}) \\
& \equiv \bigvee_{\sigma \in \phi} (\exists \vec{X} \bullet A \wedge \forall \vec{v} \bullet E \Rightarrow \bigwedge \{v : \vec{v} \bullet v = \sigma(v)\} \wedge \\
& \quad \forall \vec{v} \bullet E \Rightarrow \bigwedge \{v : (\text{VAR}_\pi - \vec{v}) \cup \vec{W} \bullet v = \sigma(v)\}) \\
& \equiv \bigvee_{\sigma \in \phi} (\exists \vec{X} \bullet A \wedge \forall \vec{v} \bullet E \Rightarrow \bigwedge \{v : \vec{v} \bullet v = \sigma(v)\} \wedge \\
& \quad (\exists \vec{v} \bullet E) \Rightarrow \bigwedge \{v : (\text{VAR}_\pi - \vec{v}) \cup \vec{W} \bullet v = \sigma(v)\}) \\
& \equiv \bigvee_{\sigma \in \phi} (\exists \vec{X} \bullet A \wedge \forall \vec{v} \bullet E \Rightarrow \bigwedge \{v : \vec{v} \bullet v = \sigma(v)\} \wedge \\
& \quad \neg (\exists \vec{v} \bullet E) \vee \bigwedge \{v : (\text{VAR}_\pi - \vec{v}) \cup \vec{W} \bullet v = \sigma(v)\}) \\
& \equiv \bigvee_{\sigma \in \phi} (\exists \vec{X} \bullet A \wedge \neg (\exists \vec{v} \bullet E) \wedge \forall \vec{v} \bullet E \Rightarrow \bigwedge \{v : \vec{v} \bullet v = \sigma(v)\} \vee \\
& \quad \exists \vec{X} \bullet A \wedge \forall \vec{v} \bullet E \Rightarrow \bigwedge \{v : \vec{v} \bullet v = \sigma(v)\} \wedge \\
& \quad \bigwedge \{v : (\text{VAR}_\pi - \vec{v}) \cup \vec{W} \bullet v = \sigma(v)\}) \\
& \equiv \bigvee_{\sigma \in \phi} (\exists \vec{X} \bullet A \wedge \neg (\exists \vec{v} \bullet E) \wedge \forall \vec{v} \bullet \text{false} \Rightarrow \bigwedge \{v : \vec{v} \bullet v = \sigma(v)\} \vee \\
& \quad \exists \vec{X} \bullet A \wedge \forall \vec{v} \bullet E \Rightarrow \bigwedge \{v : \vec{v} \bullet v = \sigma(v)\} \wedge \\
& \quad \bigwedge \{v : (\text{VAR}_\pi - \vec{v}) \cup \vec{W} \bullet v = \sigma(v)\}) \\
& \equiv P(\text{false}) \vee \bigvee_{\sigma \in \phi} (P(\bigwedge \{v : \vec{v} \bullet v = \sigma(v)\}) \wedge \\
& \quad \bigwedge \{v : (\text{VAR}_\pi - \vec{v}) \cup \vec{W} \bullet v = \sigma(v)\})
\end{aligned}$$

Clearly the last expression is independent of  $\vec{X}$ ,  $A$ , and  $E$  and the first part of the proposition follows.

The second part of the proposition follows trivially from the allowed choices of  $\vec{X}$ ,  $A$ , and  $E$ .  $\square$

**Proof:** (of Theorem 3.11)

Clearly the mapping  $(\_ )^e$  is onto its image in  $\text{MTRAN}_\pi$ , moreover it is one to one since every extension is equivalent to its kernel predicate transformer when restricted to  $\text{PRED}_{\vec{v}}$ .

It is now only necessary to show that  $(\_ )^e$  is a lattice morphism.

Firstly, observe that  $(\lambda P : \text{MTRAN} \bullet \text{PRED}_{\vec{v}} \triangleleft P)$  is an order preserving right inverse to  $(\_ )^e$ . So if the image of  $(\_ )^e$  is a sublattice of  $\text{MTRAN}$  then  $(\_ )^e$  is a lattice-morphism because it is injective. Thus it is sufficient to show that the image of  $(\_ )^e$  is closed under meet and join.

Choose any (possibly empty) indexed set of image elements

$$\mathcal{P} : I \longrightarrow (\_ )^e(\text{pMTRAN}_{\vec{u} \longrightarrow \vec{v}}),$$

then for each  $\vec{Z} : \mathbb{F} \text{VAR}_\lambda$ ,  $\phi : \text{pPRED}_{\text{VAR}_\pi \cup \vec{Z}}$  there exists families

$$\begin{aligned} \vec{X} &: I \longrightarrow (\mathbb{F} \text{VAR}_\lambda) \text{ (mutually disjoint),} \\ A &: I \longrightarrow \text{pPRED}_{\vec{u} \cup \vec{X}}, \text{ and} \\ E &: I \longrightarrow \text{pPRED}_{\vec{v} \cup \vec{X}} \end{aligned}$$

such that for each  $i \in I$ ,

$$P_i(\phi) \equiv \exists \vec{X}_i \bullet A_i \wedge \forall \vec{v} \bullet E_i \Rightarrow \phi.$$

Both the meet and join of  $\mathcal{P}$  at  $\phi$  can be written in the required format.

$$\begin{aligned} &\bigwedge_{i:I} P_i(\phi) \\ &\equiv \bigwedge_{i:I} \exists \vec{X}_i \bullet A_i \wedge \forall \vec{v} \bullet E_i \Rightarrow \phi \\ &\equiv \exists \bigcup_{i:I} \vec{X}_i \bullet \bigwedge_{i:I} A_i \wedge \forall \vec{v} \bullet E_i \Rightarrow \phi \\ &\equiv \exists \bigcup_{i:I} \vec{X}_i \bullet (\bigwedge_{i:I} A_i) \wedge \forall \vec{v} \bullet (\bigvee_{i:I} E_i) \Rightarrow \phi \end{aligned}$$

$$\begin{aligned} &\bigvee_{i:I} P_i(\phi) \\ &\equiv \bigvee_{i:I} \exists \vec{X}_i \bullet A_i \wedge \forall \vec{v} \bullet E_i \Rightarrow \phi \\ &\equiv \bigvee_{i:I} \exists \vec{X} \bullet A_i[\vec{X}_i \setminus \vec{X}] \wedge \forall \vec{v} \bullet E_i[\vec{X}_i \setminus \vec{X}] \Rightarrow \phi && [\vec{X} \text{ fresh}] \\ &\equiv \bigvee_{i:I} \exists \vec{X}; Z \bullet Z = i \wedge (Z = i \Rightarrow A_i[\vec{X}_i \setminus \vec{X}]) \wedge && [Z \text{ fresh}] \\ &\quad \forall \vec{v} \bullet (Z = i \Rightarrow E_i[\vec{X}_i \setminus \vec{X}]) \Rightarrow \phi \\ &\equiv \bigvee_{i:I} \exists \vec{X}; Z \bullet Z = i \wedge (\bigwedge_{j:I} Z = j \Rightarrow A_j[\vec{X}_i \setminus \vec{X}]) \wedge \\ &\quad \forall \vec{v} \bullet (\bigwedge_{j:I} Z = j \Rightarrow E_j[\vec{X}_i \setminus \vec{X}]) \Rightarrow \phi \\ &\equiv \exists \vec{X}; Z \bullet \bigvee_{i:I} Z = i \wedge (\bigwedge_{j:I} Z = j \Rightarrow A_j[\vec{X}_i \setminus \vec{X}]) \wedge \\ &\quad \forall \vec{v} \bullet (\bigwedge_{j:I} Z = j \Rightarrow E_j[\vec{X}_i \setminus \vec{X}]) \Rightarrow \phi \\ &\equiv \exists \vec{X}; Z \bullet (\bigvee_{i:I} Z = i) \wedge (\bigwedge_{j:I} Z = j \Rightarrow A_j[\vec{X}_i \setminus \vec{X}]) \wedge \\ &\quad \forall \vec{v} \bullet (\bigwedge_{j:I} Z = j \Rightarrow E_j[\vec{X}_i \setminus \vec{X}]) \Rightarrow \phi \\ &\equiv \exists \vec{X}; Z \bullet Z \in I \wedge (\bigwedge_{j:I} Z = j \Rightarrow A_j[\vec{X}_i \setminus \vec{X}]) \wedge \\ &\quad \forall \vec{v} \bullet (\bigwedge_{j:I} Z = j \Rightarrow E_j[\vec{X}_i \setminus \vec{X}]) \Rightarrow \phi \end{aligned}$$

□