# Theory and Use of Conditional Composition

Rajit Manohar
K. Rustan M. Leino*

Department of Computer Science
California Institute of Technology
Pasadena, CA 91125.

July 1, 1994

*"The somewhat debatable role of goto statements in practical programming is reflected in their theoretical properties, in that in the treatment both of their semantics and their correctness we are confronted with difficulties of a nature not previously encountered."* From J. de Bakker. *Mathematical Theory of Program Correctness.* Chapter 10. Prentice-Hall, 1980.

## 0. INTRODUCTION

In this note, we generalize the concept of function composition. We introduce the notion of *conditional* composition of functions, and develop a theory of such compositions. We also introduce a conditional *replace* operator, which can be used to define conditional composition in terms of ordinary function composition.

We show how these concepts can be used in four application areas: program semantics; programming languages; databases; embedded systems. In particular, we show how exceptions can be described in terms of the *IF* statement, and how conditional composition can be used to describe other programming language constructs like a jump statement and a loop exit. Contrary to what the quotation given above suggests, we show that goto statements are not very different from the *IF* statement. We relate the conditional composition operator to concepts from relational algebra in databases, and show how conditional composition can be used to describe an embedded system with various priority levels.

## 1. THEORY OF CONDITIONAL COMPOSITION

We generalize the concept of function composition to *conditional* function composition. To enable us to do so, we first introduce a notation for *tagged collections*, which can be used to represent lists, arrays, or any other labeled collection of objects.

Our proof format is from [4]. We use [ ] for everywhere brackets, which denote universal quantification over the state space, and the infix dot to denote function application.

### 1.0. TAGGED COLLECTIONS

A quantifier $Q$ is defined by a triple $(\star, u, f)$ where $\star$ is an associative and symmetric operator from $D \times D \to D$, $u$ is the unit element of $\star$, and $f$ is a function from $T \to D$ (cf. [6]). A quantified expression, (or quantifier) is an expression of the form:

$\langle Q\ x \mid r.x \triangleright t.x \rangle$

where $x$ is an unordered list of identifier names (dummies), $r.x$ (the range) is a predicate, and $t.x$ (the term) is an expression of some type $T$. When $r.x$ is true everywhere or understood, we omit the range and write the quantification as:

$\langle Q\ x \triangleright t.x \rangle$

Informally, if the set of all values $x$ for which $r.x$ holds is $\{x_0, x_1, \ldots, x_k\}$, then the value of the quantified expression is $u \star f.x_0 \star f.x_1 \star \cdots \star f.x_k$. For example, summation can be written as the quantifier $(+, 0, id)$, where $id$ is the identity function. Another common quantifier is the set constructor, which is the triple $\mathcal{S} = (\cup, \emptyset, \{\})$, where function $\{\}$ is defined by the relation $\{\}.x = \{x\}$. We write the set constructor as $\{x \mid r.x \triangleright t.x\}$ instead of $\langle \mathcal{S}\ x \mid r.x \triangleright t.x \rangle$.

* Supported in part by Air Force Office of Scientific Research grant number 61301.

We use quantification to denote a tagged collection. We represent an element in the tagged collection by $(i\colon x)$, where $i$ is the index of the element, and $x$ is the value tagged by $i$. We use $\emptyset$ to denote an empty collection, which has the property that $\emptyset \sqcup l = l$ for any $l$. Formally, we define $\bigsqcup$ to be the quantifier $(\sqcup, \emptyset, id)$. $\sqcup$ is an operator analogous to the disjoint set union operator, often written $\uplus$. An element $(i\colon x)$ is contained in $l0 \sqcup l1$ just when it is contained in either $l0$ or $l1$.

An indexed collection of objects has an additional property, namely that given an index, the element associated with that index is uniquely defined. Therefore, we restrict $\sqcup$ in the following manner: given $l0$ and $l1$, $l0 \sqcup l1$ is defined if and only if the index sets of $l0$ and $l1$ are disjoint.

We define $(i\colon x)$ to be function $(i \mapsto x)$ which is the function whose domain is the element $i$ and whose range is the element $x$. Therefore, we have the property that $(i\colon x).i = x$ which is a consequence of function application. Notice that with this meaning, the restriction on $\sqcup$ follows from the fact that a function cannot be one-to-many.

Since $\bigsqcup$ is simply a quantifier, we can use the following properties which follow from the general properties of quantification. However, we must be careful since $\sqcup$ imposes certain restrictions on its arguments.

We present some of the axioms from [6]. These axioms are simply instances of a few of the axioms for general quantifiers.

The following two axioms can be used to eliminate quantification from an expression.

**Axiom** (*Empty Range*)

$$\langle \bigsqcup i \mid \mathit{false} \triangleright f.i\colon t.i \rangle = \emptyset \tag{0}$$

**Axiom** (*One-point rule*)

$$\langle \bigsqcup i \mid [i = E] \triangleright f.i\colon t.i \rangle = (f.E\colon t.E) \tag{1}$$

For more general ranges, the following axiom can be used.

**Axiom** (*Range split*)

For mutually exclusive $r$ and $s$,

$$\langle \bigsqcup i \mid r.i \vee s.i \triangleright f.i\colon t.i \rangle = \langle \bigsqcup i \mid r.i \triangleright f.i\colon t.i \rangle \sqcup \langle \bigsqcup i \mid s.i \triangleright f.i\colon t.i \rangle \tag{2}$$

The following axiom can be used to rewrite the term.

**Axiom** (*Transform term*)

If, for all $i$, $r.i \Rightarrow (t.i = z.i) \wedge (f.i = g.i)$, then

$$\langle \bigsqcup i \mid r.i \triangleright f.i\colon t.i \rangle = \langle \bigsqcup i \mid r.i \triangleright g.i\colon z.i \rangle \tag{3}$$

Given a tagged collection, it is important to be able to determine the value associated with a particular tag in the collection. We propose the following axiom for tagged collections.

**Axiom** (*Projection*)

For injective $f$,

$$\langle \bigsqcup i \mid r.i \triangleright f.i\colon t.i \rangle.k = \langle \bigsqcup i \mid r.i \wedge [f.i = k] \triangleright f.i\colon t.i \rangle.k \tag{4}$$

We now prove,

**Theorem** (*Projection*)

If $r$ and $s$ are mutually exclusive and $r.k$ holds,

$$(\langle \bigsqcup i \mid r.i \triangleright i\colon t.i \rangle \sqcup \langle \bigsqcup i \mid s.i \triangleright i\colon z.i \rangle).k = t.k \tag{5}$$

Proof. For this proof, we define function $h.i$ in the following manner:

$$h.i = \begin{cases} t.i & \text{if r.i} \\ z.i & \text{if s.i} \end{cases}$$

Now,

$$\begin{aligned}
& (\langle \bigsqcup i \mid r.i \triangleright i\colon t.i \rangle \sqcup \langle \bigsqcup i \mid s.i \triangleright i\colon z.i \rangle).k \\
= \quad & \{\ (3)\text{: transform term; twice}\ \} \\
& (\langle \bigsqcup i \mid r.i \triangleright i\colon h.i \rangle \sqcup \langle \bigsqcup i \mid s.i \triangleright i\colon h.i \rangle).k
\end{aligned}$$

$$
\begin{array}{ll}
= & \quad \{ \text{ (2): range split; } r,s \text{ are mutually exclusive } \} \\
& \langle \bigsqcup i \mid r.i \vee s.i \rhd i\colon h.i \rangle .k \\
= & \quad \{ \text{ (4): projection } \} \\
& \langle \bigsqcup i \mid (r.i \vee s.i) \wedge [i = k] \rhd i\colon h.i \rangle .k \\
= & \quad \{ \ r.k\text{; (3): transform term } \} \\
& \langle \bigsqcup i \mid [i = k] \rhd i\colon t.i \rangle .k \\
= & \quad \{ \text{ (1): one-point rule } \} \\
& (k\colon t.k).k \\
= & \quad \{ \text{ definition } \} \\
& t.k
\end{array}
$$

□

Using these theorems and axioms, we can manipulate arbitrary tagged collections. For example, the tagged collection $a = \langle \bigsqcup i \mid 0 \leq i \wedge i < n \rhd i\colon i^2 \rangle$ can be considered to be an array of size $n$ that contains the squares of the first $n$ non-negative integers. $a.k$ would be the $k$th element of the array (for $0 \leq k < n$), since:

$$
\begin{array}{ll}
& \langle \bigsqcup i \mid 0 \leq i \wedge i < n \rhd i\colon i^2 \rangle .k \\
= & \quad \{ \text{ (4): projection axiom } \} \\
& \langle \bigsqcup i \mid 0 \leq i \wedge i < n \wedge [i = k] \rhd i\colon i^2 \rangle .k \\
= & \quad \{ \ \ 0 \leq k \wedge k < n \ \ \} \\
& \langle \bigsqcup i \mid [i = k] \rhd i\colon i^2 \rangle .k \\
= & \quad \{ \text{ (1): one-point rule } \} \\
& (k\colon k^2).k \\
= & \quad \{ \text{ definition } \} \\
& k^2
\end{array}
$$

□

We use $Tag.D$ to denote the type of a tagged collection of elements from $D$ with a fixed index set $\mathcal{U}$. Notice that given a fixed index set $\mathcal{U}$, the tagged collections in $Tag.D$ can be viewed as total functions from $\mathcal{U} \rightarrow D$. For a partial function $p$ with range $R \subseteq \mathcal{U}$, we have that

$$p = \langle \bigsqcup i \mid i \in R \rhd i\colon p.i \rangle$$

## 1.1. FUNCTIONS AND COMPOSITION

Functions of type $Tag.D \rightarrow D$ for any domain $D$ can be composed in different ways. We use $f$, $g$, and $h$ to denote such functions. We use $L$ to denote a tagged collection. Note that $L = \langle \bigsqcup i \rhd i\colon L.i \rangle$.

We generalize the notion of function composition as follows. For any predicate $p$ on $\mathcal{U}$, we define $\circ_p$ pronounced "compose $p$" as:

$$(f \circ_p g).L = f.(\langle \bigsqcup i \mid p.i \rhd i\colon g.L \rangle \sqcup \langle \bigsqcup i \mid \neg p.i \rhd i\colon L.i \rangle) \tag{6}$$

In words, $(f \circ_p g)$ applied to a tagged collection $L$ replaces $L.k$ by $g.L$, for all $k$ that satisfy $p$.

**Theorem** (*Associativity*)

If $[q \Rightarrow p]$, then $(f \circ_p g) \circ_q h = f \circ_p (g \circ_q h)$ , i.e. $\circ_p$ and $\circ_q$ are mutually associative if $[q \Rightarrow p]$.(7)

Proof.

$$
\begin{array}{ll}
& ((f \circ_p g) \circ_q h).L \\
= & \quad \{ \text{ (6): def. of } \circ_q \ \} \\
& (f \circ_p g).(\langle \bigsqcup i \mid q.i \rhd i\colon h.L \rangle \sqcup \langle \bigsqcup i \mid \neg q.i \rhd i\colon L.i \rangle) \\
= & \quad \{ \text{ (6): def. of } \circ_p \ \}
\end{array}
$$

$$\begin{array}{ll} & f.(\langle \bigsqcup i \mid p.i \triangleright i : g.(\langle \bigsqcup i \mid q.i \triangleright i : h.L\rangle \sqcup \langle \bigsqcup i \mid \neg q.i \triangleright i : L.i\rangle)\rangle \sqcup \\ & \quad \langle \bigsqcup i \mid \neg p.i \triangleright i : (\langle \bigsqcup i \mid q.i \triangleright i : h.L\rangle \sqcup \langle \bigsqcup i \mid \neg q.i \triangleright i : L.i\rangle).i\rangle) \\ = & \quad \{ \text{ (5): projection, since } \neg p.i \Rightarrow \neg q.i \ \} \\ & f.(\langle \bigsqcup i \mid p.i \triangleright i : g.(\langle \bigsqcup i \mid q.i \triangleright i : h.L\rangle \sqcup \langle \bigsqcup i \mid \neg q.i \triangleright i : L.i\rangle)\rangle \sqcup \\ & \quad \langle \bigsqcup i \mid \neg p.i \triangleright i : L.i\rangle) \\ = & \quad \{ \text{ (6): def. of } \circ_q \ \} \\ & f.(\langle \bigsqcup i \mid p.i \triangleright i : (g \circ_q h).L\rangle \sqcup \langle \bigsqcup i \mid \neg p.i \triangleright i : L.i\rangle) \\ = & \quad \{ \text{ (6): def. of } \circ_p \ \} \\ & (f \circ_p (g \circ_q h)).L \end{array}$$

□

**Theorem** (*Associativity*)

$\circ_p$ is associative. (8)

Proof. Follows from (7) and $[p \Rightarrow p]$.

□

A function of special interest is the *projection* function, defined as

$\restriction_k.L = L.k$ (9)

and pronounced as "project $k$". This function can be used to extract a value associated with a particular tag from a tagged collection.

**Theorem** (*Left identity*)

$\restriction_k$ is left identity of $\circ_p$, for all $k$ satisfying $p$. (10)

Proof.

$$\begin{array}{ll} & (\restriction_k \circ_p f).L \\ = & \quad \{ \text{ (6): def. of } \circ_p \ \} \\ & \restriction_k.(\langle \bigsqcup i \mid p.i \triangleright i : f.L\rangle \sqcup \langle \bigsqcup i \mid \neg p.i \triangleright i : L.i\rangle) \\ = & \quad \{ \text{ (9): def. of } \restriction_k \ \} \\ & (\langle \bigsqcup i \mid p.i \triangleright i : f.L\rangle \sqcup \langle \bigsqcup i \mid \neg p.i \triangleright i : L.i\rangle).k \\ = & \quad \{ \text{ (5): projection, since } p.k \ \} \\ & f.L \end{array}$$

□

It follows that $\circ_p$ has no right identity if $p$ is satisfied by more than one $k$ since the projection functions are distinct.

**Theorem** (*Left zero*)

$\restriction_k$ is left zero of $\circ_p$, for all $k$ satisfying $\neg p$. (11)

Proof.

$$\begin{array}{ll} & (\restriction_k \circ_p f).L \\ = & \quad \{ \text{ (6): def. of } \circ_p \ \} \\ & \restriction_k.(\langle \bigsqcup i \mid p.i \triangleright i : f.L\rangle \sqcup \langle \bigsqcup i \mid \neg p.i \triangleright i : L.i\rangle) \\ = & \quad \{ \text{ (9): def. of } \restriction_k \ \} \\ & (\langle \bigsqcup i \mid p.i \triangleright i : f.L\rangle \sqcup \langle \bigsqcup i \mid \neg p.i \triangleright i : L.i\rangle).k \\ = & \quad \{ \text{ (5): projection, since } \neg p.k \ \} \\ & L.k \\ = & \quad \{ \text{ (9): def. of } \restriction_k \ \} \\ & \restriction_k.L \end{array}$$

□

It follows that $\circ_p$ has no right zero if $\neg p$ holds for more than one $k$ since the projection functions are distinct.

**Theorem**

$(f \circ_p \restriction_k) \circ_q g = f \circ_{p \vee q} g$, for all $k$ satisfying $q$. (12)

Proof.

$$
\begin{array}{ll}
 & ((f \circ_p \restriction_k) \circ_q g).L \\
= & \quad \{ \text{ (6): def. of } \circ_q \} \\
 & (f \circ_p \restriction_k).(\langle \bigsqcup i \mid q.i \triangleright i\colon g.L\rangle \sqcup \langle \bigsqcup i \mid \neg q.i \triangleright i\colon L.i\rangle) \\
= & \quad \{ \text{ (6): def. of } \circ_p \} \\
 & f.(\langle \bigsqcup i \mid p.i \triangleright i\colon \restriction_k.(\langle \bigsqcup i \mid q.i \triangleright i\colon g.L\rangle \sqcup \langle \bigsqcup i \mid \neg q.i \triangleright i\colon L.i\rangle)\rangle \sqcup \\
 & \quad \langle \bigsqcup i \mid \neg p.i \triangleright i\colon (\langle \bigsqcup i \mid q.i \triangleright i\colon g.L\rangle \sqcup \langle \bigsqcup i \mid \neg q.i \triangleright i\colon L.i\rangle).i\rangle) \\
= & \quad \{ \text{ (6): def. of } \circ_q \} \\
 & f.(\langle \bigsqcup i \mid p.i \triangleright i\colon (\restriction_k \circ_q g).L\rangle \sqcup \\
 & \quad \langle \bigsqcup i \mid \neg p.i \triangleright i\colon (\langle \bigsqcup i \mid q.i \triangleright i\colon g.L\rangle \sqcup \langle \bigsqcup i \mid \neg q.i \triangleright i\colon L.i\rangle).i\rangle) \\
= & \quad \{ \text{ (10): left identity, since } q.k \} \\
 & f.(\langle \bigsqcup i \mid p.i \triangleright i\colon g.L\rangle \sqcup \\
 & \quad \langle \bigsqcup i \mid \neg p.i \triangleright i\colon (\langle \bigsqcup i \mid q.i \triangleright i\colon g.L\rangle \sqcup \langle \bigsqcup i \mid \neg q.i \triangleright i\colon L.i\rangle).i\rangle) \\
= & \quad \{ \text{ (2): range split } \} \\
 & f.(\langle \bigsqcup i \mid p.i \triangleright i\colon g.L\rangle \sqcup \\
 & \quad \langle \bigsqcup i \mid \neg p.i \wedge q.i \triangleright i\colon (\langle \bigsqcup i \mid q.i \triangleright i\colon g.L\rangle \sqcup \langle \bigsqcup i \mid \neg q.i \triangleright i\colon L.i\rangle).i\rangle \sqcup \\
 & \quad \langle \bigsqcup i \mid \neg p.i \wedge \neg q.i \triangleright i\colon (\langle \bigsqcup i \mid q.i \triangleright i\colon g.L\rangle \sqcup \langle \bigsqcup i \mid \neg q.i \triangleright i\colon L.i\rangle).i\rangle) \\
= & \quad \{ \text{ (5): projection, twice } \} \\
 & f.(\langle \bigsqcup i \mid p.i \triangleright i\colon g.L\rangle \sqcup \langle \bigsqcup i \mid \neg p.i \wedge q.i \triangleright i\colon g.L\rangle \sqcup \\
 & \quad \langle \bigsqcup i \mid \neg p.i \wedge \neg q.i \triangleright i\colon L.i\rangle) \\
= & \quad \{ \text{ (2): range split } \} \\
 & f.(\langle \bigsqcup i \mid p.i \vee (\neg p.i \wedge q.i) \triangleright i\colon g.L\rangle \sqcup \\
 & \quad \langle \bigsqcup i \mid \neg p.i \wedge \neg q.i \triangleright i\colon L.i\rangle) \\
= & \quad \{ \text{ pred. calc. } \} \\
 & f.(\langle \bigsqcup i \mid (p \vee q).i \triangleright i\colon g.L\rangle \sqcup \\
 & \quad \langle \bigsqcup i \mid \neg(p \vee q).i \triangleright i\colon L.i\rangle) \\
= & \quad \{ \text{ (6): def. of } \circ_{p \vee q} \} \\
 & (f \circ_{p \vee q} g).L
\end{array}
$$

□

## 1.1.0. POINT PREDICATES

Let $[\![k]\!]$ be the predicate that is true only at $x = k$, i.e. $[\![k]\!].x \equiv [x = k]$. $[\![k]\!]$ is a *point predicate* since it is true at exactly one point in $\mathcal{U}$.

**Theorem**

$$
\begin{array}{l}
f \circ_p \restriction_k = f \circ_{p \vee [\![k]\!]} \restriction_k \\
f \circ_p \restriction_k = f \circ_{p \wedge \neg [\![k]\!]} \restriction_k
\end{array}
\qquad (13)
$$

Proof. Note that the two lines given above express the same property. We prove the former.

$$
\begin{array}{ll}
 & (f \circ_{p \vee [\![k]\!]} \restriction_k).L \\
= & \quad \{ \text{ (6): def. of } \circ_{p \vee [\![k]\!]} \text{; (9): def. of } \restriction_k \}
\end{array}
$$

$$
\begin{array}{ll}
& f.(\langle\bigsqcup\, i \mid p.i \vee [\![k]\!].i \triangleright i{:}\, L.k\rangle \sqcup \langle\bigsqcup\, i \mid \neg p.i \wedge \neg[\![k]\!].i \triangleright i{:}\, L.i\rangle) \\
= & \quad \{\ (2)\text{: range split; pred. calc. } \} \\
& f.(\langle\bigsqcup\, i \mid p.i \wedge \neg[\![k]\!].i \triangleright i{:}\, L.k\rangle \sqcup \langle\bigsqcup\, i \mid \neg p.i \wedge \neg[\![k]\!].i \triangleright i{:}\, L.i\rangle \sqcup \langle\bigsqcup\, i \mid [\![k]\!].i \triangleright i{:}\, L.k\rangle) \\
= & \quad \{\ (2)\text{: range split; (3): transform term; pred. calc. } \} \\
& f.(\langle\bigsqcup\, i \mid p.i \wedge \neg[\![k]\!].i \triangleright i{:}\, L.k\rangle \sqcup \langle\bigsqcup\, i \mid \neg p.i \wedge \neg[\![k]\!].i \triangleright i{:}\, L.i\rangle \sqcup \\
& \qquad \langle\bigsqcup\, i \mid [\![k]\!].i \wedge p.i \triangleright i{:}\, L.k\rangle \sqcup \langle\bigsqcup\, i \mid [\![k]\!].i \wedge \neg p.i \triangleright i{:}\, L.i\rangle) \\
= & \quad \{\ \text{pred. calc. } \} \\
& f.(\langle\bigsqcup\, i \mid p.i \triangleright i{:}\, L.k\rangle \sqcup \langle\bigsqcup\, i \mid \neg p.i \triangleright i{:}\, L.i\rangle) \\
= & \quad \{\ (6)\text{: def. of } \circ_p\text{; (9): def. of } \upharpoonright_k\ \} \\
& (f \circ_p \upharpoonright_k).L
\end{array}
$$

□

We now have,

**Theorem**

$$f \circ_{false}\, g = f \tag{14}$$

Proof.

$$
\begin{array}{ll}
& (f \circ_{false}\, g).L \\
= & \quad \{\ (6)\text{: def. of } \circ_{false}\ \} \\
& f.(\langle\bigsqcup\, i \mid \mathit{false}.i \triangleright i{:}\, g.L\rangle \sqcup \langle\bigsqcup\, i \mid \neg \mathit{false}.i \triangleright i{:}\, L.i\rangle) \\
= & \quad \{\ (0)\text{: empty range } \} \\
& f.(\emptyset \sqcup \langle\bigsqcup\, i \triangleright i{:}\, L.i\rangle) \\
= & \quad \{\ \emptyset \text{ is the identity of } \sqcup\ \} \\
& f.L
\end{array}
$$

□

We mentioned earlier that $\circ_p$ had no right identity if $p$ holds for more than one point. However, when $p$ is a point predicate, we have that

**Theorem** (*Right identity*)

$\upharpoonright_k$ is the right identity of $\circ_{[\![k]\!]}$. (15)

Proof.

$$
\begin{array}{ll}
& f \circ_{[\![k]\!]} \upharpoonright_k \\
= & \quad \{\ (13)\text{, since } \mathit{false} \vee [\![k]\!] \equiv [\![k]\!]\ \} \\
& f \circ_{false} \upharpoonright_k \\
= & \quad \{\ (14)\ \} \\
& f
\end{array}
$$

□

We introduce some special notation, $\lfloor\ \rfloor_k$, defined as follows

$$\lfloor f \rfloor_k = f \circ_{true} \upharpoonright_k \tag{16}$$

**Theorem**

$$\lfloor f \rfloor_k = f \circ_{\neg[\![k]\!]} \upharpoonright_k \tag{17}$$

Proof.

$$
\begin{array}{ll}
& f \circ_{\neg[\![k]\!]} \upharpoonright_k \\
= & \quad \{\ (13)\ \} \\
& f \circ_{\neg[\![k]\!] \vee [\![k]\!]} \upharpoonright_k \\
= & \quad \{\ \text{pred. calc. } \}
\end{array}
$$

$$f \circ_{true} \lceil_k$$
$= \quad \{$ (16): def. of $\lfloor\ \rfloor_k$ $\}$
$$\lfloor f \rfloor_k$$

$\square$

**Theorem**

$\lfloor \lceil_k \rfloor_j = \lceil_j$ (18)

Proof.

$$\lfloor \lceil_k \rfloor_j$$
$= \quad \{$ (16): def. of $\lfloor\ \rfloor_j$ $\}$
$$\lceil_k \circ_{true} \lceil_j$$
$= \quad \{$ (10): left identity, since $true.k$, for any $k$ $\}$
$$\lceil_j$$

$\square$

**Theorem**

For any $f$, $\lfloor \lfloor f \rfloor_k \rfloor_j = \lfloor f \rfloor_j$ (19)

Proof.

$$\lfloor \lfloor f \rfloor_k \rfloor_j$$
$= \quad \{$ (16): def. of $\lfloor\ \rfloor_j$ $\}$
$$\lfloor f \rfloor_k \circ_{true} \lceil_j$$
$= \quad \{$ (16): def. of $\lfloor\ \rfloor_k$; (8): associativity $\}$
$$f \circ_{true} \lceil_k \circ_{true} \lceil_j$$
$= \quad \{$ (10): left identity, since $true.k$, for any $k$ $\}$
$$f \circ_{true} \lceil_j$$
$= \quad \{$ (16): def. of $\lfloor\ \rfloor_j$ $\}$
$$\lfloor f \rfloor_j$$

$\square$

**Theorem**

$\lfloor\ \rfloor_k$ is idempotent (20)

Proof. Follows from (19). $\square$

**Theorem**

$\lfloor f \circ_{true} g \rfloor_k = \lfloor f \rfloor_k \circ_{true} \lfloor g \rfloor_k$ (21)

Proof.

$$\lfloor f \circ_{true} g \rfloor_k$$
$= \quad \{$ (16): def. of $\lfloor\ \rfloor_k$; (8): associativity $\}$
$$f \circ_{true} g \circ_{true} \lceil_k$$
$= \quad \{$ (10): left identity, since $true.k$ for any $k$ $\}$
$$f \circ_{true} \lceil_k \circ_{true} g \circ_{true} \lceil_k$$
$= \quad \{$ (16): def. of $\lfloor\ \rfloor_k$, twice; (8): associativity $\}$
$$\lfloor f \rfloor_k \circ_{true} \lfloor g \rfloor_k$$

$\square$

With these operators, we cannot define $(f \circ_{p \wedge q} g)$ using $\circ_p$ and $\circ_q$. However, if we had a way to define $\circ_{\neg p}$ in terms of $\circ_p$, then we could obtain $p \wedge q$ from $p \vee q$. We do not know how to do this, but instead introduce a new operator $\bullet_p$, defined by

$$\bullet_p = \circ_{\neg p} \quad . \tag{22}$$

**Theorem** (*dual of (12)*)
$(f \bullet_p \restriction_k) \bullet_q g = f \bullet_{p \wedge q} g$, for all $k$ satisfying $\neg q$. (23)

Proof.

$$
\begin{array}{ll}
 & (f \bullet_p \restriction_k) \bullet_q g \\
= & \quad \{ \text{ (22): def. of } \bullet \text{, twice } \} \\
 & (f \circ_{\neg p} \restriction_k) \circ_{\neg q} g \\
= & \quad \{ \text{ (12), with } p, q := \neg p, \neg q \text{, since } \neg q.k \text{ } \} \\
 & f \circ_{\neg p \vee \neg q} g \\
= & \quad \{ \text{ De Morgan } \} \\
 & f \circ_{\neg (p \wedge q)} g \\
= & \quad \{ \text{ (22): def. of } \bullet_{p \wedge q} \text{ } \} \\
 & f \bullet_{p \wedge q} g
\end{array}
$$

□

We have additional properties of $\bullet_p$ which follow from those of $\circ_p$.

**Theorem** (*Left zero*)
$\restriction_k$ is left zero of $\bullet_p$ for all $k$ satisfying $p$. (24)

Proof. Follows from (11), (22). □

**Theorem** (*Left identity*)
$\restriction_k$ is left identity of $\bullet_p$ for all $k$ satisfying $\neg p$. (25)

Proof. Follows from (10), (22). □

**Theorem** (*Right identity*)
$\restriction_k$ is the right identity of $\bullet_{\neg [\![k]\!]}$ (26)

Proof. Follows from (15), (22). □

**Theorem**

$$
\begin{array}{l}
f \bullet_p \restriction_k = f \bullet_{p \vee [\![k]\!]} \restriction_k \\
f \bullet_p \restriction_k = f \bullet_{p \wedge \neg [\![k]\!]} \restriction_k
\end{array} \tag{27}
$$

Proof. Follow from (13), (22). □

Since $\restriction_k$ is the right identity of $\circ_{[\![k]\!]}$, we examine the effect of $\bullet_{[\![k]\!]} \restriction_k$.

**Theorem**
$f \bullet_{[\![k]\!]} \restriction_k = f \circ_{true} \restriction_k$ (28)

Proof. Follows from (16), (17), (22). □

We explore additional properties of $\circ_p$ for monotonic functions. We assume that there exists some partial ordering $\sqsubseteq$ defined on the elements of $D$. We extend this order to $Tag.D$ by the following definition:

$$\langle \bigsqcup\ i \mid p.i \rhd f.i{:}\ t.i \rangle \sqsubseteq \langle \bigsqcup\ i \mid p.i \rhd f.i{:}\ u.i \rangle \equiv \langle \forall\ i \mid p.i \rhd t.i \sqsubseteq u.i \rangle \quad \text{for injective } f. \tag{29}$$

This ordering is monotonic in each member of $Tag.D$. Note that this order is defined only if $f$ is injective. We now have the following theorems:

**Theorem** (*Monotonicity*)
$f \circ_p g$ is monotonic in $f$. (30)

Proof.

$$
\begin{array}{ll}
 & (f \circ_p g).L \sqsubseteq (f' \circ_p g).L \\
= & \quad \{ \text{ (6): def. of } \circ_p \text{, twice } \} \\
 & f.(\langle \bigsqcup\ i \mid p.i \rhd i{:}\ g.L \rangle \sqcup \langle \bigsqcup\ i \mid \neg p.i \rhd i{:}\ L.i \rangle) \sqsubseteq f'.(\langle \bigsqcup\ i \mid p.i \rhd i{:}\ g.L \rangle \sqcup \langle \bigsqcup\ i \mid \neg p.i \rhd i{:}\ L.i \rangle)
\end{array}
$$

$$\Leftarrow \quad \{\ \}$$
$$f \sqsubseteq f'$$

□

**Theorem** (*Monotonicity*)
$f \circ_p g$ is monotonic in $g$, for monotonic $f$.

Proof.

$$(f \circ_p g).L \sqsubseteq (f \circ_p g').L$$
$$= \quad \{ (6)\text{: def. of } \circ_p\text{, twice} \}$$
$$f.(\langle \bigsqcup i \mid p.i \triangleright i\colon g.L\rangle \sqcup \langle \bigsqcup i \mid \neg p.i \triangleright i\colon L.i\rangle) \sqsubseteq f.(\langle \bigsqcup i \mid p.i \triangleright i\colon g'.L\rangle \sqcup \langle \bigsqcup i \mid \neg p.i \triangleright i\colon L.i\rangle)$$
$$\Leftarrow \quad \{ f \text{ is monotonic} \}$$
$$(\langle \bigsqcup i \mid p.i \triangleright i\colon g.L\rangle \sqcup \langle \bigsqcup i \mid \neg p.i \triangleright i\colon L.i\rangle) \sqsubseteq (\langle \bigsqcup i \mid p.i \triangleright i\colon g'.L\rangle \sqcup \langle \bigsqcup i \mid \neg p.i \triangleright i\colon L.i\rangle)$$
$$= \quad \{ (29)\text{: } \sqsubseteq \text{ on } Tag.D \}$$
$$\langle \bigsqcup i \mid p.i \triangleright i\colon g.L\rangle \sqsubseteq \langle \bigsqcup i \mid p.i \triangleright i\colon g'.L\rangle$$
$$= \quad \{ (29)\text{: } \sqsubseteq \text{ on } Tag.D \}$$
$$\langle \forall\, i \mid p.i \triangleright g.L \sqsubseteq g'.L\rangle$$
$$\Leftarrow \quad \{ \text{calculus} \}$$
$$g.L \sqsubseteq g'.L$$
$$\Leftarrow \quad \{\ \}$$
$$g \sqsubseteq g'$$

□

### 1.2. EXTENSION TO OTHER OPERATORS

So far we have examined the effect of extending $\circ$ to $\circ_p$. We now demonstrate that we can define $\circ_p$ in terms of $\circ$, given an additional operator. This operator can then be used to extend other operators to tagged collections.

To be able to define $\circ_p$ in terms of $\circ$, notice that the $\circ_p$ operator performs a selective substitution. Motivated by this, we define $\blacktriangleleft p \triangleright$ —a conditional *replace* operator— and its dual $\triangleleft p \blacktriangleright$ as follows:

$$\langle \bigsqcup i \triangleright i\colon s.i\rangle \blacktriangleleft p \triangleright \langle \bigsqcup i \triangleright i\colon t.i\rangle = \langle \bigsqcup i \mid \neg p.i \triangleright i\colon s.i\rangle \sqcup \langle \bigsqcup i \mid p.i \triangleright i\colon t.i\rangle \tag{31}$$
$$\langle \bigsqcup i \triangleright i\colon s.i\rangle \triangleleft p \blacktriangleright \langle \bigsqcup i \triangleright i\colon t.i\rangle = \langle \bigsqcup i \mid p.i \triangleright i\colon s.i\rangle \sqcup \langle \bigsqcup i \mid \neg p.i \triangleright i\colon t.i\rangle \tag{32}$$

We can define $\circ_p$ in terms of $\circ$ using the $\blacktriangleleft p \triangleright$ operator. To do so, we define $\mathcal{C}.u$ to be the tagged collection $\langle \bigsqcup i \triangleright i\colon u\rangle$. We now have:

$$(f \circ_p g).L = f \circ (L \blacktriangleleft p \triangleright \mathcal{C}.(g.L)) \tag{33}$$
$$(f \bullet_p g).L = f \circ (L \triangleleft p \blacktriangleright \mathcal{C}.(g.L)) \tag{34}$$

Using this definition of $\blacktriangleleft p \triangleright$, we can extend an arbitrary binary operator *op* having a unit element $u$. Let **op** denote the pointwise extension of *op* to total functions from $\mathcal{U} \rightarrow D$. We can write these total functions as tagged collections. We can define $\mathbf{op}_p$ as follows:

$$X\ \mathbf{op}_p\ Y = X\ \mathbf{op}\ (Y \blacktriangleleft p \triangleright \mathcal{C}.u) \tag{35}$$

Notice that with **op** replaced by $\circ$, this definition differs from the one used for conditional composition. We will see $\mathbf{op}_p$ again in section 4.

## 2. APPLICATION AREA: PROGRAM SEMANTICS

We describe a program semantics with an additional conditional composition operator $\triangleleft_p$, where $p$ is a predicate on $\mathcal{U}$. Imposing a structure on the predicates used for the weakest precondition semantics allows us to identify the composition operators presented earlier with the composition of statements. We follow the path of [7], [8], [9] and [11] which first describe an operational semantics in terms of traces and then derive a weakest precondition semantics from it.

## 2.0. TRACE SEMANTICS

We will use the semantics of statements presented in [8], [11], [5]. We use $\mathcal{X}$ to denote the (nonempty) state space. Variables that refer to the state are implicitly assumed to be from $\mathcal{X}$. Traces are nonempty sequences of states. $fin.t$ is true just when $t$ is a trace of finite length. $inf.t$ is the negation of $fin.t$. Concatenation of traces is denoted by juxtaposition. Using these notational conventions, we have the following trace semantics for *skip* and assignment.

$$skip = \{x \triangleright x\} \tag{36}$$

$$(v := E) = \{x \triangleright x\, x[v := E]\} \tag{37}$$

For sequential composition, we have [8], [11]

$$S; T = \{s, x, t \mid sx \in S \wedge xt \in T \wedge fin.s \triangleright sxt\} \cup \{s \mid s \in S \wedge inf.s \triangleright s\} \tag{38}$$

We assume that there exists a total function $idx: \mathcal{X} \to \mathcal{U}$. This function defines a partition on $\mathcal{X}$. We introduce a conditional composition operator $\lhd_p$ (pronounce "try", as in triangle) —where $p$ is a predicate on $\mathcal{U}$— that has the following operational meaning. We write $S \lhd_p T$ for the statement whose execution consists of: executing $S$; if execution thereof terminates in a state $x$ satisfying $(p \circ idx)$, then executing $T$ as well; if $S$ does not terminate, or $S$ terminates in a state $x$ satisfying $\neg(p \circ idx)$, then not executing $T$. The trace semantics can be described by

$$\begin{aligned} S \lhd_p T = \{&s, x, t \mid sx \in S \wedge xt \in T \wedge fin.s \wedge (p \circ idx).x \triangleright sxt\} \cup \\ &\{s \mid s \in S \wedge (inf.s \vee \neg(p \circ idx).(last.s)) \triangleright s\} \end{aligned} \tag{39}$$

Analogous to $\bullet_p$, we define $\blacktriangleleft_p$ to be $\lhd_{\neg p}$.

**Theorem**

$$; = \lhd_{true} \tag{40}$$

Proof. Follows from the definition of $\lhd_{true}$. □

**Theorem** (*Identity of ;*)

*skip* is the left and right identity of ; (41)

Proof. Can be taken without modification from [8],[11]. □

The following are some theorems about semicolon.

**Theorem** (From [5], [8], [11].)

; is associative. (42)

; is universally $\cup$-distributive in its left argument. (43)

; is positively $\cup$-distributive in its right argument. (44)

We can use the definition of repetition and selection from [5], [8], or [11] without modification since the trace semantics for semicolon has not been changed. We use the definition of the *IF* statement given in [5]. Since we have defined all our earlier semantics in terms of trace sets, we rewrite the definition presented in [5] in the following manner. For any predicate $b$, we define $b?$ as follows:

$$b? = \{x \mid x \in \mathcal{X} \wedge b.x \triangleright x\} \tag{45}$$

Using this definition of $b?$, we define the *IF* statement from [5] as:

$$\mathbf{if}\langle [\!]\ i \triangleright b_i \to S_i\rangle \mathbf{fi} = \langle \cup\ i \triangleright b_i?; S_i\rangle \cup \langle \forall\ i \triangleright \neg b_i\rangle?; E \tag{46}$$

where $E$ is defined as the set of all eternal traces. If we know that $\langle \exists\ i \triangleright b_i\rangle$ holds, then we can simplify (46) to:

$$\mathbf{if}\langle [\!]\ i \triangleright b_i \to S_i\rangle \mathbf{fi} = \langle \cup\ i \triangleright b_i?; S_i\rangle \tag{47}$$

Using these trace semantics, we have the following theorem:

**Theorem** (*Conditional Composition*)

$$S \lhd_p T = S;\ \mathbf{if}\ (p \circ idx) \to T\, [\!]\, \neg(p \circ idx) \to skip\ \mathbf{fi} \tag{48}$$

Proof. For this proof, we use $IF = \mathbf{if}\ (p \circ idx) \to T\, [\!]\, \neg(p \circ idx) \to skip\ \mathbf{fi}$

$S; IF$

$$
\begin{array}{ll}
= & \{ \text{ (38): def. of ; } \} \\
& \{s, x, t \mid sx \in S \wedge xt \in \mathit{IF} \wedge \mathit{fin}.s \triangleright sxt\} \cup \{s \mid s \in S \wedge \mathit{inf}.s \triangleright s\} \\
= & \{ \text{ (46): def. of } \mathit{IF}\text{; (47) } \} \\
& \{s, x, t \mid sx \in S \wedge xt \in ((p \circ idx)?; T) \cup (\neg(p \circ idx)?; \mathit{skip}) \wedge \mathit{fin}.s \triangleright sxt\} \cup \\
& \{s \mid s \in S \wedge \mathit{inf}.s \triangleright s\} \\
= & \{ \text{ range split } \} \\
& \{s, x, t \mid sx \in S \wedge xt \in ((p \circ idx)?; T) \wedge \mathit{fin}.s \triangleright sxt\} \cup \\
& \{s, x, t \mid sx \in S \wedge xt \in (\neg(p \circ idx)?; \mathit{skip}) \wedge \mathit{fin}.s \triangleright sxt\} \cup \{s \mid s \in S \wedge \mathit{inf}.s \triangleright s\} \\
= & \{ \text{ (45): def. of } b? \text{ twice } \} \\
& \{s, x, t \mid sx \in S \wedge xt \in T \wedge \mathit{fin}.s \wedge (p \circ idx).x \triangleright sxt\} \cup \\
& \{s, x, t \mid sx \in S \wedge xt \in \mathit{skip} \wedge \mathit{fin}.s \wedge \neg(p \circ idx).x \triangleright sxt\} \cup \{s \mid s \in S \wedge \mathit{inf}.s \triangleright s\} \\
= & \{ \text{ (36): def. of } \mathit{skip}\text{; } sx := s \text{ } \} \\
& \{s, x, t \mid sx \in S \wedge xt \in T \wedge \mathit{fin}.s \wedge (p \circ idx).x \triangleright sxt\} \cup \\
& \{s \mid s \in S \wedge \mathit{fin}.s \wedge \neg(p \circ idx).(\mathit{last}.s) \triangleright s\} \cup \{s \mid s \in S \wedge \mathit{inf}.s \triangleright s\} \\
= & \{ \text{ range split } \} \\
& \{s, x, t \mid sx \in S \wedge xt \in T \wedge \mathit{fin}.s \wedge (p \circ idx).x \triangleright sxt\} \cup \\
& \{s \mid s \in S \wedge (\mathit{inf}.s \vee (\mathit{fin}.s \wedge \neg(p \circ idx).(\mathit{last}.s))) \triangleright s\} \\
= & \{ \text{ absorption, since } \mathit{fin}.s \wedge \mathit{inf}.s = \mathit{false} \text{ } \} \\
& \{s, x, t \mid sx \in S \wedge xt \in T \wedge \mathit{fin}.s \wedge (p \circ idx).x \triangleright sxt\} \cup \\
& \{s \mid s \in S \wedge (\mathit{inf}.s \vee \neg(p \circ idx).(\mathit{last}.s)) \triangleright s\} \\
= & \{ \text{ (39): def. of } \triangleleft_p \text{ } \} \\
& S \triangleleft_p T
\end{array}
$$

□

This theorem demonstrates how the semantics of the *IF* statement can be used to describe conditional composition.

**Theorem**

$\triangleleft_p$ is associative (49)

Proof. Follows from (48) and (42), (44), (46). □

**Theorem**

$\triangleleft_p$ is universally $\cup$-distributive in its left argument. (50)

Proof. Follows from (48) and (43), (46). □

**Theorem**

$\triangleleft_p$ is positively $\cup$-distributive in its right argument.

Proof. Follows from (48) and (42), (44), (46). □

## 2.1. WEAKEST PRECONDITIONS

We define function $wp.S.Q$ to be the weakest condition on the initial state such that: execution of $S$ terminates; on termination $Q$ holds [4]. We examine the weakest precondition semantics for conditional composition. We use the usual $wp$ semantics from [4] using the definition of $wp$ presented in [8], [11] in terms of traces.

$$wp.S.Q.x = \langle \forall\ t \mid \mathit{first}.t = x \wedge t \in S \triangleright \mathit{fin}.t \wedge Q.(\mathit{last}.t)\rangle \tag{51}$$

We confine our attention to predicates $Q$ that are *partitioned predicates*. Let $t = \langle \bigsqcup\ i \mid i \in \mathcal{U} \triangleright i{:}\ t.i\rangle$ be a tagged collection of predicates. We associate partition $i$ with predicate $t.i$. We define the partitioned predicate $T$ represented in terms of $t$ to be

$$[T \equiv \langle \forall\ i \triangleright ([\![i]\!] \circ idx) \Rightarrow t.i \rangle] \tag{52}$$

From now on, we use tagged collections and predicates interchangeably. Using this definition of a predicate, we can explore properties of the $wp$ and $\triangleleft_p$. We first examine some additional properties that relate $\blacktriangleleft p \triangleright$, $\mathcal{C}$, and partitioned predicates.

**Theorem**

For a partitioned predicate $Q$,

$$[Q \equiv \mathcal{C}.Q] \tag{53}$$

Proof.

$$\begin{array}{ll}
 & \mathcal{C}.Q \\
= & \quad \{ \text{ def. of } \mathcal{C} \ \} \\
 & \langle \bigsqcup\ i \triangleright i \colon Q \rangle \\
= & \quad \{ \ (52) \ \} \\
 & \langle \forall\ i \triangleright ([\![i]\!] \circ idx) \Rightarrow Q \rangle \\
= & \quad \{ \ (52) \ \} \\
 & \langle \forall\ i \triangleright ([\![i]\!] \circ idx) \Rightarrow \langle \forall\ j \triangleright ([\![j]\!] \circ idx) \Rightarrow Q.j \rangle \rangle \\
= & \quad \{ \Rightarrow \text{ over } \forall \ \} \\
 & \langle \forall\ i \triangleright \langle \forall\ j \triangleright ([\![i]\!] \circ idx) \Rightarrow (([\![j]\!] \circ idx) \Rightarrow Q.j) \rangle \rangle \\
= & \quad \{ \text{ interchange of dummies } \} \\
 & \langle \forall\ j \triangleright \langle \forall\ i \triangleright ([\![i]\!] \circ idx) \Rightarrow (([\![j]\!] \circ idx) \Rightarrow Q.j) \rangle \rangle \\
= & \quad \{ \text{ pred. calc. } \} \\
 & \langle \forall\ j \triangleright \langle \exists\ i \triangleright [\![i]\!] \circ idx \rangle \Rightarrow (([\![j]\!] \circ idx) \Rightarrow Q.j) \rangle \\
= & \quad \{ \ idx \text{ is total } \} \\
 & \langle \forall\ j \triangleright ([\![j]\!] \circ idx) \Rightarrow Q.j \rangle \\
= & \quad \{ \ (52) \ \} \\
 & Q
\end{array}$$

□

From (53), we can see that the set of partitioned predicates is the same as the set of ordinary predicates. Therefore, a partitioned predicate is simply a predicate that has some structure.

**Theorem**

$$[(p \circ idx) \equiv \langle \exists\ i \mid p.i \triangleright [\![i]\!] \circ idx \rangle] \tag{54}$$

Proof.

$$\begin{array}{ll}
 & \langle \exists\ i \mid p.i \triangleright ([\![i]\!] \circ idx).x \rangle \\
= & \quad \{ \text{ trading } \} \\
 & \langle \exists\ i \mid ([\![i]\!] \circ idx).x \triangleright p.i \rangle \\
= & \quad \{ \text{ def. of } [\![i]\!] \ \} \\
 & \langle \exists\ i \mid [idx.x = i] \triangleright p.i \rangle \\
= & \quad \{ \text{ one-point rule } \} \\
 & p.(idx.x) \\
= & \quad \{ \text{ def. of } \circ \ \} \\
 & (p \circ idx).x
\end{array}$$

□

We now examine the effect of $\blacktriangleleft p \triangleright$ on partitioned predicates.

**Theorem**

$$[(Q\ \blacktriangleleft p \triangleright\ R) \equiv (\neg(p \circ idx) \Rightarrow Q) \wedge ((p \circ idx) \Rightarrow R)] \tag{55}$$

Proof.

$(\neg(p \circ idx) \Rightarrow Q) \wedge ((p \circ idx) \Rightarrow R)$

$=$ $\{$ (54) twice $\}$

$(\langle \exists\ i \mid \neg p.i \triangleright [\![i]\!] \circ idx \rangle \Rightarrow Q) \wedge (\langle \exists\ i \mid p.i \triangleright [\![i]\!] \circ idx \rangle \Rightarrow R)$

$=$ $\{$ pred. calc. $\}$

$\langle \forall\ i \mid \neg p.i \triangleright ([\![i]\!] \circ idx) \Rightarrow Q \rangle \wedge \langle \forall\ i \mid p.i \triangleright ([\![i]\!] \circ idx) \Rightarrow R \rangle$

$=$ $\{$ (31): def. of $\blacktriangleleft p \triangleright$; (52) $\}$

$\langle \bigsqcup\ i \triangleright i \colon Q \rangle \ \blacktriangleleft p \triangleright\ \langle \bigsqcup\ i \triangleright i \colon R \rangle$

$=$ $\{$ def. of $\mathcal{C}$, twice $\}$

$\mathcal{C}.Q\ \blacktriangleleft p \triangleright\ \mathcal{C}.R$

$=$ $\{$ (53) twice $\}$

$Q\ \blacktriangleleft p \triangleright\ R$

$\square$

We now examine some properties of $wp$. Since the traces of semicolon, *skip*, and assignment are the same as those in [8] and [11], all their results still hold. In particular, we have

**Theorem** (From [8], [11].)

$[wp.skip.Q \equiv Q]$ (56)

$[wp.(v := E).Q \equiv Q[v := E]]$ (57)

$[wp.(S;T).Q \equiv wp.S.(wp.T.Q)]$ (58)

We now examine the weakest precondition semantics for conditional composition.

**Theorem**

$[wp.(S \lhd_p T).Q \equiv wp.S.(Q\ \blacktriangleleft p \triangleright\ wp.T.Q)]$ (59)

Proof.

$wp.(S \lhd_p T).Q$

$=$ $\{$ (48): $\lhd_p$ as $IF$ $\}$

$wp.(S; \mathbf{if}\ (p \circ idx) \rightarrow T \,[\!]\, \neg(p \circ idx) \rightarrow skip\ \mathbf{fi}).Q$

$=$ $\{$ (58): $wp$ of ; $\}$

$wp.S.(wp.\mathbf{if}\ (p \circ idx) \rightarrow T \,[\!]\, \neg(p \circ idx) \rightarrow skip\ \mathbf{fi}.Q)$

$=$ $\{$ $wp$ of $IF$ (from [4], [8], [11]) $\}$

$wp.S.(((p \circ idx) \vee \neg(p \circ idx)) \wedge ((p \circ idx) \Rightarrow wp.T.Q) \wedge (\neg(p \circ idx) \Rightarrow wp.skip.Q))$

$=$ $\{$ $[X \vee \neg X]$; (56) $\}$

$wp.S.(((p \circ idx) \Rightarrow wp.T.Q) \wedge (\neg(p \circ idx) \Rightarrow Q))$

$=$ $\{$ (55) $\}$

$wp.S.(Q\ \blacktriangleleft p \triangleright\ wp.T.Q)$

$\square$

We now relate $\lhd_p$ to the composition operator $\circ_p$.

**Theorem**

$[wp.(S \lhd_p T) \equiv wp.S \circ_p wp.T]$ (60)

Proof.

$wp.(S \lhd_p T).Q$

$=$ $\{$ (59): $wp$ of $\lhd_p$ $\}$

$wp.S.(Q\ \blacktriangleleft p \triangleright\ wp.T.Q)$

$=$ $\{$ (53) $\}$

$$
\begin{array}{ll}
 & wp.S.(Q \blacktriangleleft p \triangleright \mathcal{C}.(wp.T.Q)) \\
= & \quad \{ \text{ (33): def. of } \circ_p \text{ using } \blacktriangleleft p \triangleright \} \\
 & (wp.S \circ_p wp.T).Q
\end{array}
$$

□

We conclude this section by identifying a program with its weakest precondition. We have the following correspondences:

$skip = id$

$S \triangleleft_p T = S \circ_p T$, or

$\triangleleft_p = \circ_p$

$\blacktriangleleft_p = \bullet_p$

# 3. APPLICATION AREA: PROGRAMMING LANGUAGES

The $\triangleleft_p$ operator presented above can be used in various ways. In this section, we suggest a disciplined approach to the introduction of such an operator into a programming language.

We then demonstrate how one can use this operator to describe the semantics of exceptions [7], loop exits, and structured jumps [1].

## 3.0. SEMANTICS OF PARTITIONS

We describe how the semantics of conditional composition can be used to describe the semantics of a program given a partition of the state space. We identify an element of the partition with an element from the index set $\mathcal{U}$.

> *Restriction.* Each element of the partition of the state space has the same cardinality. Furthermore, there exists a well-defined mapping from one element of the partition to another. (*End of Restriction.*)

### 3.0.0. PARTITIONS

We assume that there exist functions $\lfloor\ \rfloor_k$ for $k \in \mathcal{U}$, which map states to states. These functions are assumed to satisfy the following two properties:

$$\langle \forall\ x, k \triangleright idx.\lfloor x \rfloor_k = k \rangle \tag{61}$$

$$\langle \forall\ x \triangleright \langle \forall\ k, j \triangleright \lfloor \lfloor x \rfloor_k \rfloor_j = \lfloor x \rfloor_j \rangle\rangle \tag{62}$$

Properties (61) and (62) guarantee that the partitions are of equal size. $[\![i]\!] \circ idx$ can be used to determine if the current state belongs to element $i$ from the partition. The restriction given above implies that there exist functions $\lfloor\ \rfloor_k$ satisfying (61) and (62).

### 3.0.1. PROGRAM STATEMENTS

Notice that the functions $\lfloor\ \rfloor_k$ define a method by which one can switch from one element of the partition to another. To enable us to do this in the framework of a programming language, we introduce a new statement $switch_k$, defined by

$$switch_k = \{x \triangleright x\ \lfloor x \rfloor_k\} \tag{63}$$

> *Restriction.* For a programming language implementing such a semantics, *switch* is the only statement by which a transition can be made from one element of the partition to another. (*End of Restriction.*)

**Theorem** (*Left zero*)

$switch_k$ is the left zero of $\triangleleft_p$, for all $k$ satisfying $\neg p$. (64)

Proof.

$$
\begin{array}{ll}
 & switch_k \triangleleft_p T \\
= & \quad \{ \text{ (39): def. of } \triangleleft_p \}
\end{array}
$$

$\{s, x, t \mid sx \in switch_k \wedge xt \in T \wedge fin.s \wedge (p \circ idx).x \triangleright sxt\} \cup$

$\{s \mid s \in switch_k \wedge (inf.s \vee \neg(p \circ idx).(last.s)) \triangleright s\}$

$=$ { (63): def. of $switch$, since $\neg p.k$ }

$\{s \mid s \in switch_k \triangleright s\}$

$=$ { def. of set }

$switch_k$

$\square$

**Theorem**

$$wp.switch_k.Q.x = Q.\lfloor x \rfloor_k. \qquad (65)$$

Proof.

$wp.switch_k.Q.x$

$=$ { (51): def. of $wp$ }

$\langle \forall\ t \mid first.t = x \wedge t \in switch_k \triangleright fin.t \wedge Q.(last.t)\rangle$

$=$ { (63): def. of $switch_k$ }

$\langle \forall\ t \mid t = x \triangleright Q.\lfloor t \rfloor_k\rangle$

$=$ { one-point rule }

$Q.\lfloor x \rfloor_k$

$\square$

*Restriction.* Earlier, we had used tagged collections as predicates. We now impose one additional restriction on the predicates used in the tagged collection. If $Q$ is the tagged collection $\langle \bigsqcup\ k \triangleright k\colon q.k\rangle$, then

$$\langle \forall\ i, j, k, x \triangleright q.k.\lfloor x \rfloor_i = q.k.\lfloor x \rfloor_j\rangle \qquad (66)$$

In other words, $Q.k.x$ does not depend on the element of the partition to which $x$ belongs. (*End of Restriction.*)

**Theorem**

$$wp.switch_k = \upharpoonright_k \qquad (67)$$

Proof.

$wp.switch_k.Q.x$

$=$ { (65) }

$Q.\lfloor x \rfloor_k$

$=$ { (52): tagged collection as a predicate }

$Q.k.\lfloor x \rfloor_k$

$=$ { (66) }

$Q.k.x$

$=$ { (9): def. of $\upharpoonright_k$ }

$(\upharpoonright_k.Q).x$

$\square$

Once again identifying a program with its $wp$, we have

$switch_k = \upharpoonright_k$

Note that the properties of functions discussed in section 1 do not always carry over to the trace semantics, although they are maintained by the weakest precondition semantics. This can be expected, since trace semantics are more concrete than $wp$ semantics which only refer to initial and final states of the computation. We can derive additional properties using the theorems from section 1. For instance, we can prove (from (10)) that

$switch_k \lhd_p S = S, \quad \text{if } p.k$

The semantics of partitioned state spaces can be used in various ways. Defining the partition of the state space in different ways can result in semantics for different language constructs. The sections below describe three such partitions of the state space. They use the following partitions:

1. For the semantics of exceptions presented in [7], a boolean coordinate $oc$ is introduced into the state which is used to partition the state space into two subspaces.
2. To extend the semantics presented in [7] to handle more than one exception, we let the $oc$ coordinate be integer valued.
3. Structured jumps use a partition in which an extra coordinate *label* is used to identify the label of the block to which a branch is to be executed.

## 3.1. SEMANTICS OF EXCEPTIONS—NORMAL AND EXCEPTIONAL STATES

In this section, we relate the semantics presented in [7] to those presented here. $\unlhd$ is used to denote the $\lhd$ operator from [7]. Note that in [7], there is an underlying assumption that all programs begin in normal states. The functions considered in [7] are from $D \times D \to D$, which is a special case of the generalized compositions presented here with $\mathcal{U} = \{0, 1\}$.

### 3.1.0. FUNCTIONS OF TWO ARGUMENTS

We have the following correspondence between functions and compositions presented in [7] and those explored in this note. The correspondence follows from the fact that the normal state is identified with $0 \in \mathcal{U}$, and the exceptional state is identified with 1.

| Exceptions | Partitions | Exceptions | Partitions |
|---|---|---|---|
| $\rangle\!\!\circ\!\!\langle$ | $\circ_{false}$ | $L$ | $\upharpoonleft_0$ |
| $\langle\circ$ | $\circ_{[\![0]\!]}$ | $R$ | $\upharpoonleft_1$ |
| $\circ\rangle$ | $\circ_{[\![1]\!]}$ | $\lfloor x \rfloor$ | $\lfloor x \rfloor_0$ |
| $\langle\circ\rangle$ | $\circ_{true}$ | $\lceil x \rceil$ | $\lfloor x \rfloor_1$ |

We have that

$$idx.x = \begin{cases} 0 & \text{if } x.oc = \bot \\ 1 & \text{if } x.oc = \top \end{cases}$$

$\lfloor x \rfloor_0 = x[oc := \bot]$

$\lfloor x \rfloor_1 = x[oc := \top]$

The theorems presented in [7] follow from those presented here. In addition, we have the following correspondence:

$nor = [\![0]\!] \circ idx$

$exc = [\![1]\!] \circ idx$

### 3.1.1. TRACE SEMANTICS

The program constructs presented in [7] are different from those presented here. In particular, execution always begins in the normal state, and the try operator lowers the exception before executing the exception handler. We have the following correspondence between the semantics presented in [7] and in this note.

| Exceptions | Partitions |
|---|---|
| $S; T$ | $S \lhd_{[\![0]\!]} T$ |
| $S \unlhd T$ | $S \lhd_{[\![1]\!]} (switch_0; T)$ |
| *raise* | $switch_1$ |
| *skip* | $\{x \rhd \lfloor x \rfloor_0\}$ |

Notice that the normal and exceptional states defined in [7] are not symmetric, a fact reflected in the table given above.

**Theorem**

$S \trianglelefteq T = S \lhd_{[\![1]\!]} (switch_0; T)$

Proof.

$$
\begin{array}{ll}
& S \lhd_{[\![1]\!]} (switch_0; T) \\
= & \quad \{ \text{ (39): def. of } \lhd_{[\![1]\!]} \} \\
& \{s, x, t \mid sx \in S \wedge xt \in (switch_0; T) \wedge ([\![1]\!] \circ idx).x \wedge fin.s \rhd sxt\} \cup \\
& \{s \mid s \in S \wedge (inf.s \vee \neg([\![1]\!] \circ idx).(last.s)) \rhd s\} \\
= & \quad \{ \neg[\![1]\!] \equiv [\![0]\!] \} \\
& \{s, x, t \mid sx \in S \wedge xt \in (switch_0; T) \wedge ([\![1]\!] \circ idx).x \wedge fin.s \rhd sxt\} \cup \\
& \{s \mid s \in S \wedge (inf.s \vee ([\![0]\!] \circ idx).(last.s)) \rhd s\} \\
= & \quad \{ \text{ (38): def. of ; since } switch_0 \text{ contains only finite traces of length} > 1 \} \\
& \{s, t, u, x, y \mid sx \in S \wedge xty \in switch_0 \wedge yu \in T \wedge fin.s \wedge ([\![1]\!] \circ idx).x \rhd sxtyu\} \cup \\
& \{s \mid s \in S \wedge (inf.s \vee ([\![0]\!] \circ idx).(last.s)) \rhd s\} \\
= & \quad \{ \text{ (63): def. of } switch_0;\ t, y := \epsilon, \lfloor x \rfloor_0 \} \\
& \{s, u, x \mid sx \in S \wedge \lfloor x \rfloor_0 u \in T \wedge fin.s \wedge ([\![1]\!] \circ idx).x \rhd sx\lfloor x \rfloor_0 u\} \cup \\
& \{s \mid s \in S \wedge (inf.s \vee ([\![0]\!] \circ idx).(last.s)) \rhd s\} \\
= & \quad \{ \text{ switch to notation in [7] } \} \\
& \{s, u, x \mid sx \in S \wedge \lfloor x \rfloor u \in T \wedge fin.s \wedge exc.x \rhd sx\lfloor x \rfloor u\} \cup \\
& \{s \mid s \in S \wedge (inf.s \vee nor.(last.s)) \rhd s\} \\
= & \quad \{ \text{ from [7] } \} \\
& S \trianglelefteq T
\end{array}
$$

□

### 3.1.2. WEAKEST PRECONDITIONS

The semantics differ in the definition of the weakest preconditions. The reason for this difference is that [7] only considers programs that execute statements in the normal state.

[7] defines the weakest precondition of a statement $S$ given postcondition $Q$ to be the condition that guarantees that: the program terminates in a normal state; on termination $Q$ holds. As opposed to this, we define the weakest precondition to be the condition that guarantees that: the program terminates; on termination $Q$ holds. Also, we do not have to apply $wp.S$ to a pair as in [7], but to a single (partitioned) predicate.

## 3.2. MORE THAN ONE EXCEPTIONAL STATE

If we extend the semantics in [7] to $n$ exceptions, and we define $\trianglelefteq_k$ to be $\lhd_{[\![k]\!]}$, then we can use similar correspondences between the constructs presented in this note and [7] to obtain a semantics for programs with more than one exceptional state.

## 3.3. LOOP EXITS

Modula-3 is an example of a programming language with exceptions [10]. A loop exit in Modula-3 is considered to be the raising of an exception which causes control to be passed outside the loop. With the semantics described in this note, $switch_k$ can be used to exit from a loop, and $wp$-semantics may be used to determine the condition which holds on termination of the loop. We use **loop** $S$ **end** to mean an infinite repetition of statement $S$, composed using $\blacktriangleleft_{[\![k]\!]}$; that is, $S \blacktriangleleft_{[\![k]\!]} S \blacktriangleleft_{[\![k]\!]} \cdots$

(**loop** $S$ **end**) $\lhd_{[\![k]\!]} E$

In the program given above, $S$ is a statement, possibly containing $switch_k$ statements, in which substatements are composed using $\blacktriangleleft_{[\![k]\!]}$.

Multiple exits can be defined using a single exception handler which can handle more than one type of exception using a structure similar to the one shown above. If there are two possible exits, say $k$ and $l$,

$switch_k$ or $switch_l$ can be used to exit the loop. The semicolon used to define the loop now becomes $\blacktriangleleft_{[\![k]\!]\vee[\![l]\!]}$.

The exit exception handler can be refined to handle the two different exists separately as follows: **(loop** $S$ **end)** $\lhd_{[\![k]\!]\vee[\![l]\!]}$ $((skip \lhd_{[\![k]\!]} E0) \lhd_{[\![l]\!]} E1)$

### 3.4. STRUCTURED JUMPS

Another application of the semantics presented here is to define the meaning of a branch statement (cf. [1], [2]). We describe how a structured branching technique can be implemented with two different statement labels. For two labels, we let $\mathcal{U} = \{A, B, N, E\}$.

Let $A$ and $B$ be the branch labels. Let $S0$, $S1$ be the programs which are executed when control is transferred to labels $A$ and $B$ respectively. $switch_A$, $switch_B$ are statements which transfer control from one statement block to another directly. The program shown below implements such a branching scheme using the composition operators described in this note. We assume that the program starts executing $S0$ initially. $E$ is used to terminate execution, and the semicolon shown below is defined to be $\lhd_{[\![N]\!]}$. Substatements (if any) of $S0$ and $S1$ are composed using $\lhd_{[\![N]\!]}$. The semicolon defining the loop is $\blacktriangleleft_{[\![E]\!]}$. If we interpret $A \lhd_p B \lhd_q C$ to mean $(A \lhd_p B) \lhd_q C$, then the program can be written as:

$$
\begin{array}{l}
(switch_A \lhd_{[\![A]\!]\vee[\![B]\!]} \textbf{ loop } skip \\
\qquad\qquad\qquad \lhd_{[\![A]\!]} (switch_N; S0; switch_E) \\
\qquad\qquad\qquad \lhd_{[\![B]\!]} (switch_N; S1; switch_E) \\
\qquad\quad \textbf{end} \lhd_{[\![E]\!]} switch_N)
\end{array}
$$

### 3.5. DISCUSSION

In this section, the semantics of partitions was introduced by imposing restrictions on programming language constructs, and introducing the *switch* statement. It was shown how partitions could be used to give a semantics for exceptions, loop exits, and structured jumps. Notice that theorem (48) suggests an implementation of the semantics of partitions using the *IF* statement. By showing how this operator can be used to describe exceptions, we have also demonstrated that a programming language with exceptions would have to check if an exception had been raised after the completion of every statement. However, the suggested restrictions on programming languages —namely, that *switch* is the only statement that can change the current partition— would enable a large number of these checks to be removed, since statements that raise or lower exceptions could be determined syntactically.

The presented trace semantics was preceded by two attempts. As a first attempt, the trace semantics presented in [7] was extended to allow *raise* to be the left and right identity of $\unlhd$. This led to our next attempt in which we tried to formulate a symmetric semantics for exceptions. *lower* was introduced as the counterpart of *raise*. In this note we have presented a semantics which handles any partition of the state space uniformly.

The results from [7] served as the foundation for the semantics presented here. Other early references to semantics in particular semantics of exception handling are [3] and the unpublished [1]. The formerly mentioned of these references provides a good discussion of how exception handling can simplify the structure of certain programs. It gives separate predicate transformers for normal and exceptional outcomes, whereas we do not distinguish between the two. In [1], an arbitrary number of outcomes is considered and a structured branching scheme is discussed. In [7], $wep.S$ is applied to a pair of predicates, whereas we only have one (partitioned) predicate.

## 4. APPLICATION AREA: DATABASES

For a tagged collection $\langle \bigsqcup i \triangleright i\colon V.i \rangle$, we may, in the realm of databases, consider each $i\colon V.i$ pair a *row*, whose *key* is $i$. Then we may want to apply operation $op$ to only some of the rows, discriminating based on the key. For the selection of the rows, we use a predicate over the index set, usually denoted $p$.

We show some applications of $\blacktriangleleft p \triangleright$ to relational databases [12]. We may think of $Tag.D$ as the type of a keyed relation, in which some keys may have no value. In an actual database, that would be represented by the absence of a row. Here, however, we need *some* value associated with each key. Rather than just introducing a special value to indicate an empty row, we let the the tagged collection be over *sets* of values of the rows in the relation that is modeled. Hence, we choose $D$ to be some power set. This also allows a key to have associated with it a set of size greater than 1. If in a tagged collection, every key is associated with a set of size at most 1, then we say the tagged collection (or relation) to be *uniquely keyed*. In the sequel, we will let "$p$-rows" refer to those rows specified by $p$.

We now have that

$A \cup_p B$

is, in terms of the modeled relations, $A$ with the $p$-rows of $B$ added. Similarly,

$A \cap_p B$

is $A$ in which each $p$-row has been reduced so as to only include those values also found in the corresponding row of $B$.

The *replace* operation is written

$A \blacktriangleleft p \triangleright B$ .

The value of this expression is $A$ with the $p$-rows replaced by the $p$-rows of $B$. In other words, it is $A$ less its $p$-rows, plus the $p$-rows of $B$.

Also, if $A$ and $B$ are uniquely keyed relations and $+$ is some binary operation defined over the values, we can define $+$ over sets of values as

$$\begin{array}{lcl} \{a\} + \{b\} & = & \{a+b\} \\ \{a\} + \{\} & = & \{a\} \\ \{\} + \{b\} & = & \{b\} \\ \{\} + \{\} & = & \{\} \end{array}$$

With that definition, $A +_p B$ yields $A$ in which the values of the $p$-rows are increased by the corresponding values in $B$.

## 5. APPLICATION AREA: EMBEDDED SYSTEMS

Consider an embedded system which has various priority levels numbered from 0 to $N-1$. Level 0 represents the normal level of operation. All other levels represent an emergency situation which must be handled before normal execution can resume. The levels 1 to $(N-1)$ represent different degrees of the same emergency. After handling the emergency at level $k$, the system can lower the emergency to some other level. The state space is extended with coordinate $lev$ which indicates the current emergency level. We assume that there is some external process that can increase $lev$ at any time. This external process will inform the system of any emergency that occurs by raising $lev$.

We assume that there exists an atomic $lower(m,n)$ operation which can be described as:

$lower(m,n) \equiv \mathbf{if}\ lev \leq m \rightarrow lev := n \,[\!]\, lev > m \rightarrow skip\ \mathbf{fi}$

We need atomicity of this operation since the predicate $lev \leq m$ is not monotonic in $lev$. To enable execution to resume in the middle of a routine that handles an emergency, we add the following $N$ coordinates $pc[0 \ldots N-1]$. In addition, each routine has certain critical sections which cannot be interrupted no matter what the emergency level is. To handle these, we use the semicolon which is defined to be $\triangleleft_{true}$. Initially, all the newly introduced coordinates are zero. We can describe the system as follows:

$SYS \equiv \mathbf{loop}\ skip$

$\triangleleft_{lev=0}\ S_0$

$\triangleleft_{lev=0}\ S_1$

...

$\triangleleft_{lev=N-1}\ S_{N-1}$

**end**

where $S_k$ is defined as:

$S_k \equiv \mathbf{loop}\ skip$

$\triangleleft_{lev=k \wedge pc[k]=0}\ S_{k,0}; pc[k] := 1$

$\triangleleft_{lev=k \wedge pc[k]=1}\ S_{k,1}; pc[k] := 2$

...

$\triangleleft_{lev=k \wedge pc[k]=M-1}\ S_{k,M-1}; pc[k] := 0$

**end**

The semicolon that defines the loop for $SYS$ is $\triangleleft_{true}$. The semicolon that defines the loop for $S_k$ is $\triangleleft_{lev=k}$. Statements $S_{k,j}$ are actions composed with the semicolon. Since semicolon has been defined to be $\triangleleft_{true}$, these actions are not interruptible.

## 6. CONCLUSION

In this note, tagged collections and their compositions were introduced. These collections can be viewed as quantified expressions and can be used to manipulate lists and functions using the existing properties of quantification and one additional axiom. These tagged collections were used to introduce the theory of conditional composition of functions, and the conditional replace operator.

These concepts were used to describe the semantics of programming languages, and to introduce new programming language constructs. We described the semantics of partitioned state spaces, which was then used to describe the semantics of exceptions, loop exits, and structured jumps. The conditional replace operator was used to describe operations in relational databases. Conditional composition was used to describe the operation of an embedded system with priority levels.

## References

[1] R.-J.R. Back and M. Karttunen. A predicate transformer semantics for statements with multiple exits. University of Helsinki, unpublished MS, 1983.

[2] J. de Bakker. *Mathematical Theory of Program Correctness.* Chapter 10. Prentice-Hall, 1980.

[3] F. Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, 10:163-174, 1984.

[4] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.

[5] R.M. Dijkstra. Operational Semantics: correction & embellishment. Internal Note rutger 16, 1993.

[6] H.P. Hofstee, and K.R.M. Leino. Class notes, CS284: *Reasoning about Program Correctness: Sequential Programs.* California Institute of Technology, Spring term 1993-94.

[7] K.R.M. Leino and J.L.A. van de Snepscheut. Semantics of Exceptions. To appear, IFIP transactions 1994

[8] J.J. Lukkien. An operational semantics for the guarded command language. In R.S. Bird, C.C. Morgan, and J.C.P. Woodcock, editors, *Mathematics of Program Construction*, number 669 in Lecture Notes in Computer Science, pp 233-249. Springer-Verlag, 1993.

[9] J.J. Lukkien. *Parallel Program Design and Generalized Weakest Preconditions.* PhD thesis, Groningen University, 1991. Also, Caltech technical report CS TR 90-16.

[10] G. Nelson, editor. *Systems Programming with Modula-3.* Prentice-Hall, 1991.

[11] J.L.A. van de Snepscheut. On Lattice Theory and Program Semantics. Caltech technical report CS TR 93-19.

[12] J.D. Ullman. *Principles of Database Systems*, Computer Science Press, 1982.