Making Program Logics Intelligible

John C. Reynolds
Carnegie Mellon University

Lovelace Lecture — June 8, 2011

Research partially supported by National Science Foundation Grant CCF-0916808
Dedication

To the British computer scientists who taught me so much when I was young.

Those who are gone:

    Christopher Strachey  Peter Landin  Robin Milner

and those who continue to instruct me:

    Tony Hoare  Rod Burstall  Alan Robinson
Introduction

To verify program specifications, rather than generic safety properties, it will be necessary to integrate verification into the process of programming.

Program proving is unlike theorem proving in mathematics - mathematical conjectures may give no hint as to how they could be proved, but programs are written by programmers, who must understand informally why their programs work. The job of verification is not to explore some immense search space, but to formalize the programmer’s intuitions until any faults are revealed.
This requires specifications and proofs that are succinct and intelligible - which in turn require logics that go beyond predicate calculus (the assembly language of program proving). In this talk, I will recount and illustrate several steps, old and new, towards this goal.
The Distance between Traditional Logic and Programming

Particularly when describing arrays, program specifications are typically full of inequalities and set definitions using inequalities. For example, a diagram that a programmer might write, e.g.,

\[
X: \begin{array}{ccccc}
  & & & & \\
a & c & e & d & b
\end{array}
\]

might be expressed by:

\[
a \leq c \leq e \leq d \leq b \quad \text{and} \quad \text{dom } X = \{ i \mid a < i < b \}.
\]
Partition Diagrams

The obvious difficulty with diagrams such as

\[ \begin{array}{cccc} a & c & e & d & b \\ \end{array} \]

is that expressions tend to migrate slightly:

\[ \begin{array}{cccc} a & c & e & d & b \\ \end{array} \]

This fertile source of program errors can be avoided by capturing the expressions inside the boxes:

\[ \begin{array}{cccc} a & c & e & d & b \\ \end{array} \]

We call such a diagram a *partition diagram*. 
Interval Diagrams

An *interval diagram* is an annotated box denoting a finite consecutive set of integers. In particular

\[
\begin{align*}
\begin{array}{c}
\hline
a \\
\hline
a \quad b \\
\hline
a \quad b \\
\hline
a
\end{array}
\end{align*}
\]

denotes the set of integers \( i \) such that

\[
\begin{align*}
& a < i \leq b \\
& a \leq i \leq b \\
& a < i < b \\
& a \leq i < b \\
& i = a.
\end{align*}
\]
Partition Diagrams

A diagram of the form

\[ a_0 \quad a_1 \quad a_2 \quad \cdots \quad a_n \]

is called a partition diagram. The intervals denoted by

\[ a_0 \quad a_1, \quad a_1 \quad a_2, \quad \cdots, \quad a_{n-1} \quad a_n \]

are called the component intervals of the partition diagram, and the interval denoted by \( a_0 \quad a_n \) is called the total interval of the partition diagram.

A partition diagram is an assertion that its component intervals are a partition of its total interval, i.e., that the component intervals are disjoint and their union is the total interval.
Some Abbreviations

Just as with interval diagrams, we may write $|a|$ instead of $a - 1$ and $a$ instead of $a - a$ within partition diagrams. For example,

\[
\begin{array}{ccc}
  a & k & b \\
\end{array}
\]

has the same meaning as

\[
\begin{array}{ccc}
  a & k & k & b \\
\end{array}
\]

which has the same meaning as

\[
\begin{array}{ccc}
  a - 1 & k - 1 & k & b \\
\end{array}
\]
Summing an Array (Annotated Specification)

\[
\begin{align*}
\{ [a, b] \subseteq \text{dom } X \} \\
\text{newvar } k \text{ in } (k := a - 1; s := 0; \\
\{ \text{inv: } [a, k, b] \land s = \sum_{i \in [a, k]} X(i) \}) \\
\text{while } k < b \text{ do (} \\
\quad k := k + 1; \\
\quad \{ [a, k, b] \land s = \sum_{i \in [a, k]} X(i) \} \\
\quad s := s + X(k)) \\
\{ s = \sum_{i \in [a, b]} X(i) \}
\end{align*}
\]
Array Values as Functions

We regard an array as a variable whose value is a function, whose domain is an interval. For example,

\[
\text{real array } X(1:10) \ldots
\]
declares \(X\) to be an array whose values have the domain \([1, 10]\).

When \(S \subseteq \text{dom } X\), we write \(X\lceil S\) for the restriction of \(X\) to \(S\), which is the function such that

\[
\text{dom}(X\lceil S) = S \quad \forall i \in S. \ (X\lceil S)i = X_i.
\]
We also write \( \{X\} \) for the *image* of \( X \), i.e., the set

\[
X = \{ X_i \mid i \in \text{dom } X \}.
\]

For example, suppose \( sq \) is the function with domain \([0 \ 5]\) that maps each number between 0 and 5 into its square. Then

\[
\{sq\} = \{0, 1, 4, 9, 16, 25\} \quad \{sq\_2 \ 4\} = \{4, 9, 16\}.
\]
Pointwise Extension of a Relation

Suppose $\rho$ is a binary relation between values. Then the pointwise extension of $\rho$, written $\rho^*$, is the relation between sets of such values such that

$$S \rho^* T \text{ iff } \forall x \in S, y \in T. x \rho y.$$

For example,

$$\{2, 3\} \leq^* \{3, 4\} \quad \{2, 3\} \neq^* \{4, 5\}$$

are both true, but

$$\{2, 3\} <^* \{3, 4\} \quad \{2, 3\} =^* \{2, 3\} \quad \{2, 3\} \neq^* \{2, 3\}$$

are all false.
We also abbreviate

\[
\{x\} \rho^* T \text{ by } x \rho^* T \\
S \rho^* \{y\} \text{ by } S \rho^* y.
\]
Ordered Arrays

We define the assertion

$$\text{ord}_\rho X \text{ iff } \forall i, j \in \text{dom } X. \ i < j \text{ implies } X(i) \rho X(j).$$

For example, $\text{ord}_{\leq} X$ asserts that $X$ is ordered in nonstrict increasing order.
Binary Search

The following is the invariant of a simple while-loop program for binary search:

\[
\begin{align*}
&\begin{array}{c}
a \ b \\
\end{array} \subseteq \text{dom } X \land \text{ord}_\leq(X_{\begin{array}{c}a \ b \\
\end{array}}) \land \\
&\begin{array}{c}
a \ c \ d \ b \\
\end{array} \land \\
\{X_{\begin{array}{c}a \ c \\
\end{array}}\} \leq^* y \land y <^* \{X_{\begin{array}{c}d \ b \\
\end{array}}\}.
\end{align*}
\]

Compare this with

\[
\begin{align*}
&\forall i. \ a \leq i \leq b \implies i \in \text{dom } X) \land \\
&\forall i, j. \ a \leq i < j \leq b \implies X(i) \leq X(j) \land \\
&a - 1 \leq c - 1 \leq d \leq b \land \\
&(\forall i. \ a \leq i < c \implies X(i) \leq y) \land \\
&(\forall i. \ d < i \leq b \implies y < X(i)).
\end{align*}
\]
Partition: The Invariants

\[ I = m_i n \land \{ A \} m_i \leq^* r \land m_j n \land r \leq^* \{ A \} j n \land (i \leq j \text{ implies } (\exists p. i p n \land r \leq A(p)) \land (\exists q. m q j \land A(q) \leq r)) \]

\[ I_1 = m_i n \land \{ A \} m_i \leq^* r \land (\exists p. i p n \land r \leq A(p)) \]

\[ I_2 = m_j n \land r \leq^* \{ A \} j n \land (\exists q. m q j \land A(q) \leq r) \]

\[ I' = m_i n \land \{ A \} m_i \leq^* r \land m_j n \land r \leq^* \{ A \} j n \land (i + 1 \leq j - 1 \text{ implies } (\exists p. i p n \land r \leq A(p)) \land (\exists q. m q j \land A(q) \leq r)) \]
Partition: The Procedure

let Partition(A, i, j ; m, n) =
{ \text{[m n] } \subseteq \text{dom } A \land \text{[m n]} }

newvar r in ( 
\text{r} := A((m + n) \div 2) ; i := m ; j := n ;
{\text{inv: I}}

while \text{i} \leq \text{j} do ( 
{I_1 \land I_2}

while \text{A(i)} < \text{r} do \text{i} := \text{i} + 1 ;
while \text{r} < \text{A(j)} do \text{j} := \text{j} - 1 ;
{I_1 \land r \leq A(i) \land I_2 \land A(j) \leq r}

if \text{i} \leq \text{j} then ( 
newvar t in (t := A(i) ; A(i) := A(j) ; A(j) := t) ;
{I_1 \land A(i) \leq r \land I_2 \land r \leq A(j) \land i \leq j}
{I'}

i := i + 1 ; j := j - 1 )))

{ \text{[m j i n]} \land \{A\text{[m i]} \leq \ast \{A\text{[j n]}} \}
Rearrangement

We write \( X \sim Y \), and say that \( X \) is a rearrangement of \( Y \), when there is a bijection (i.e., a one-to-one correspondence) \( B \) from \( \text{dom } X \) to \( \text{dom } Y \) such that

\[
\forall i \in \text{dom } X. \ X(i) = Y(B(i)).
\]

Suppose \( S \subseteq \text{dom } X = \text{dom } Y \). We write \( X \sim_S Y \), and say that \( X \) is a rearrangement restricted to \( S \) of \( Y \), when \( X \) is a rearrangement of \( Y \) and, for all \( i \in (\text{dom } X) - S \), \( X(i) = Y(i) \).
Partition Revisited

let Partition\((A, i, j ; m, n) = \{
\begin{array}{ll}
m & n \\
m & n
\end{array}
\subseteq \text{dom} A \land \begin{array}{ll}
m & n \\
m & n
\end{array} \land A = A_0 \}
\)

newvar \(r\) in \(
\begin{array}{ll}
r : = A((m + n) \div 2) ;
i : = m ;
j : = n ;
\end{array}
\)
\{inv: I \land A\sim \begin{array}{ll}
m & n
\end{array} A_0 \}

while \(i \leq j\) do \(
\begin{array}{ll}
\{I_1 \land I_2 \land A\sim \begin{array}{ll}
m & n
\end{array} A_0 \}
\end{array}
\)
while \(A(i) < r\) do \(i : = i + 1 ;\)
while \(r < A(j)\) do \(j : = j - 1 ;\)
\{I_1 \land r \leq A(i) \land I_2 \land A(j) \leq r \land A\sim \begin{array}{ll}
m & n
\end{array} A_0 \}
if \(i \leq j\) then \(
\begin{array}{ll}
\text{newvar} \ t \ \text{in} \ (t : = A(i) ;
A(i) : = A(j) ;
A(j) : = t) ;
\end{array}
\)
\{I_1 \land A(i) \leq r \land I_2 \land r \leq A(j) \land i \leq j \land A\sim \begin{array}{ll}
m & n
\end{array} A_0 \}
\{I' \land A\sim \begin{array}{ll}
m & n
\end{array} A_0 \}
\)
i : = i + 1 ;
j : = j - 1 \))\))
\{\begin{array}{ll}
m & j \\
i & n
\end{array} \land \{A\} \begin{array}{ll}
m & i
\end{array} \leq^* \{A\} \begin{array}{ll}
j & n
\end{array} \land A\sim \begin{array}{ll}
m & n
\end{array} A_0 \}
Quicksort

\text{letrec } \text{Quicksort}(A; \ m, n) = \begin{cases} \{m \ n \subseteq \text{dom } A \land A = A_0\} \quad \text{if } m < n \text{ then newvar } i, j \text{ in } \begin{cases} \{m \ n \subseteq \text{dom } A \land m \ n \} \\ \text{Partition}(A, i, j; m, n); \begin{cases} m \ j \ i \ n \land \{A|m|i\} \leq^* \{A|j|n\} \land A \sim m n A_0 \} \text{ Quicksort}(A; m, j); \begin{cases} m \ j \ i \ n \land \text{ord}_{\leq A} m j \land \{A|m|i\} \leq^* \{A|j|n\} \land A \sim m n A_0 \} \text{ Quicksort}(A; i, n) \begin{cases} m \ j \ i \ n \land \text{ord}_{\leq A} m j \land \text{ord}_{\leq A} i n \land \{A|m|i\} \leq^* \{A|j|n\} \land A \sim m n A_0 \} \end{cases} \end{cases} \end{cases} \end{cases} \end{cases} \end{cases} \text{ Quicksort}(A; i, n) \begin{cases} m \ j \ i \ n \land \text{ord}_{\leq A} m j \land \text{ord}_{\leq A} i n \land \{A|m|i\} \leq^* \{A|j|n\} \land A \sim m n A_0 \} \end{cases} \begin{cases} \text{ord}_{\leq A} m n \land A \sim m n A_0 \} \end{cases} 


Realignment

We write $X \simeq Y$, and say that $X$ is a realignment of $Y$, when there is a monotonic bijection $B$ from $\text{dom } X$ to $\text{dom } Y$ such that

$$\forall i \in \text{dom } X. \ X(i) = Y(B(i)).$$

When $X$ and $Y$ are array values, i.e., functions on intervals, $X \simeq Y$ holds only when $X$ is a “shift” of $Y$.

But when $X$ and $Y$ are functions whose domains can be arbitrary finite sets of integers — which we might call “lacy array values” — things become more interesting.
Eliminating Zeroes

\[ \{ \mathbf{a b} \subseteq X \land X = X_0 \} \]

newvar \( d \) in \( (c := a ; d := a) ; \)

\{ inv: \[ \mathbf{a c d b} \] \land \\
\[ X \mathbf{a c} \simeq X_0 \] (\[ \mathbf{a d} \cap \{ i | X(i) \neq 0 \} \]) \land \\
\[ X \mathbf{d b} = X_0 \] \}

while \( d \leq b \) do

if \( X(d) = 0 \) then \( d := d + 1 \)

else \( (X(c) := X(d) ; c := c + 1 ; d := d + 1) \) \}

\{ \[ \mathbf{a c b} \] \land X \mathbf{a c} \simeq X_0 \] (\[ \mathbf{a b} \cap \{ i | X(i) \neq 0 \} \]) \}
Separation Logic: An Example

Suppose

\[
\text{list } \epsilon i \overset{\text{def}}{=} i = \text{nil}
\]

\[
\text{list}(a \cdot \alpha) i \overset{\text{def}}{=} \exists j. \ i \leftarrow a, j \land \text{list } \alpha j
\]

and consider the program

\[
LREV \overset{\text{def}}{=} j := \text{nil};
\]

\[
\text{while } i \neq \text{nil} \text{ do (} k := [i + 1] ; [i + 1] := j ; j := i ; i := k).\]

To prove \(\{\text{list } \alpha i\} \ LREV \ {\text{list } \alpha^\dagger j}\), the invariant

\[
\exists \alpha, \beta. \ \text{list } \alpha i \land \text{list } \beta j \land \alpha^\dagger_0 = \alpha^\dagger \cdot \beta
\]

is inadequate.
An adequate invariant (in Hoare logic):

\[(\exists \alpha, \beta. \text{list } \alpha \ i \land \text{list } \beta \ j \land \alpha^{\dagger}_0 = \alpha^{\dagger} \cdot \beta)\]
\[\land (\forall k. \text{reachable}(i, k) \land \text{reachable}(j, k) \Rightarrow k = \text{nil}).\]

An adequate invariant (in separation logic):

\[\exists \alpha, \beta. (\text{list } \alpha \ i \ast \text{list } \beta \ j) \land \alpha^{\dagger}_0 = \alpha^{\dagger} \cdot \beta.\]

where \ast is the \textit{separating conjunction}.\]
Enriching the Concept of State

In separation logic, the *state* consists of two components:

— A *store*, which maps variables into their values,

— A *heap*, which maps addresses (which are values) into their values.
The Separating Conjunction

The assertion $p_1 \ast p_2$ holds for a heap $h$ when $h$ can be partitioned into two disjoint subheaps $h_1$ and $h_2$ such that $p_1$ holds in $h_1$ and $p_2$ holds in $h_2$.

The Points-To Relation

The assertion $\ell \mapsto e$ holds when the heap consists of a single cell mapping the address $\ell$ into the value of $e$. 
Some Inference Rules

• Frame Rule (O’Hearn)

\[
\frac{\{p\} \ c \ \{q\}}{\{p \ast r\} \ c \ \{q \ast r\}},
\]

where the free variables of \( r \) are not modified by \( c \).

• Existential Quantification

\[
\frac{\{p\} \ c \ \{q\}}{\{\exists v. \ p\} \ c \ \{\exists v. \ q\}},
\]

where \( v \) does not occur free in \( c \).
Concurrent Composition (O’Hearn)

\[
\begin{array}{c}
\{p_1\} c_1 \{q_1\} \quad \{p_2\} c_2 \{q_2\} \\
\{p_1 \ast p_2\} c_1 \parallel c_2 \{q_1 \ast q_2\},
\end{array}
\]

where the free variables of \( p_1, c_1, \) and \( q_1 \) are not modified by \( c_2, \) and vice-versa.
The Iterated Separating Conjunction

Suppose $S$ is a finite set. Then the assertion $\bigodot_{i \in S} P(i)$ holds when the heap can be partitioned into an family $h_i$ of heaplets indexed by $i$, such that, for all $i \in S$, $P(i)$ holds in $h_i$.

For example, $\bigodot_{i \in S} \ell + i \mapsto A(i)$ describes an array with base address $\ell$, whose subscripts range over $S$, and whose value is the function $A$. 
More Definitions

When \( S \) is a finite set of integers,

\[
\ell \mapsto^S A \equiv S \subseteq \text{dom } A \land \bigcirc_{i \in S} \ell + i \mapsto A(i)
\]

\[
A \sim^S A' \equiv (A|S) \sim (A'|S).
\]
Quicksort Revisited: Some Assumptions

\{ \ell \mapsto mn \ A_0 \}\}

Quicksort( ; \ell, m, n)

\{ \exists A. \ell \mapsto mn A \land A \sim mn A_0 \land \text{ord}_{\leq}(A[m n]) \}\}

\{ \ell \mapsto mn A_0 \land m n \}\}

Partition(i, j ; \ell, m, n)

\{ \exists A. \ell \mapsto mn A \land A \sim mn A_0 \land mj in \land

\{ A[m i] \leq^* \{ A[j n] \}\}}
Quickstart Revisited: The Procedure Body

\{\ell \mapsto \boxed{mn} \ A_0\}

if \(m < n\) then newvar \(i, j\) in (\(\ell \mapsto \boxed{mn} \ A_0 \land \boxed{m} \boxed{n}\) )

Partition\((i, j ; \ell, m, n)\)

\{\exists A_1. \ell \mapsto \boxed{mn} \ A_1 \land A_1 \sim \boxed{mn} \ A_0 \land \boxed{m} \boxed{j} \boxed{i} \boxed{n}\}\land
\{A_1] \boxed{m} \boxed{i}\} \leq^* \{A_1] \boxed{j} \boxed{n}\}\}

\begin{cases}
\{\ell \mapsto \boxed{mj} \ A_1\} \quad & \exists A_1 \quad \text{Quicksort}(\ell, m, j)\\
\{\exists A_2. \ell \mapsto \boxed{mj} \ A_2 \land A_2 \sim \boxed{mj} \ A_1 \land \text{ord}_\leq(A_2] \boxed{mj}\}\}
\end{cases}

\begin{cases}
\{\ell \mapsto \boxed{in} \ A_1\} \quad & \exists A_1 \quad \text{Quicksort}(\ell, i, n)\\
\{\exists A_3. \ell \mapsto \boxed{in} \ A_3 \land A_3 \sim \boxed{in} \ A_1 \land \text{ord}_\leq(A_3] \boxed{in}\}\}
\end{cases}

\left\{\begin{array}{l}
\exists A_1\\
\end{array}\right\}

\{\exists A'. \ell \mapsto \boxed{mn} \ A' \land A' \sim \boxed{mn} \ A_1 \land A_1 \sim \boxed{mn} \ A_0 \land \boxed{m} \boxed{j} \boxed{i} \boxed{n}\}\land
\{A_1] \boxed{m} \boxed{i}\} \leq^* \{A_1] \boxed{j} \boxed{n}\}\land
\text{ord}_\leq(A'] \boxed{mj}\) \land \text{ord}_\leq(A'] \boxed{in}\}

\{\exists A'. \ell \mapsto \boxed{mn} \ A' \land A' \sim \boxed{mn} \ A_0 \land \text{ord}_\leq(A'] \boxed{mn}\}

where \(A' = (A_2] \boxed{mj}\) \cup (A_1] \boxed{j} \boxed{i}\) \cup (A_3] \boxed{i} \boxed{n}\).
What’s Next

— Permissions

— Lacy Arrays