A Primer for Program Composition Notation

K. Mani Chandy
Stephen Taylor

CRPC-TR90056
June, 1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

A Primer for Program Composition Notation

K. Mani Chandy, Stephen Taylor *
California Institute of Technology

20 June 1990

Abstract

This primer describes a notation for program composition. Program composition is putting programs together to get larger ones. PCN (Program Composition Notation) is a programming language that allows programmers to compose programs so that composed programs execute efficiently on uniprocessors, shared-memory multiprocessors, or distributed-memory multicomputers. In particular, PCN is intended to execute efficiently on multicomputers, each node of which is a shared-memory multiprocessor.

The programs that are put together using PCN can be in PCN itself or in C or in Fortran. Later implementations of PCN will allow composition of programs in notations in addition to C and Fortran.

PCN is implemented on a variety of sequential and concurrent architectures including UNIX workstations, Symult 2010, Intel iPSC, BBN Butterfly, Encore Multimax and Sequent Symmetry.

Several programming examples are presented in the primer. The examples are presented with methods for reasoning about the correctness of PCN programs.

*Supported by NSF, AFOSR and ONR

1 Overview

PCN is based on UNITY [3], a theory and a notation for concurrent programming, and on STRAND[6], a notation derived from concurrent logic programming[11]. Composition in PCN is motivated by, but is different from, composition in CSP [8] The motivation for PCN and a comparison of PCN with other notations is found in [4]. A programming environment and the run-time support system for PCN are described in [2] and [7], respectively.

1.1 New Concepts

PCN has a three concepts that are not in languages such as C or Fortran. Next, these concepts are discussed very briefly and informally. Readers may want to skim through sections describing familiar material so as to spend more time on the new concepts.

1. **Mutables and Permanents:** PCN has two kinds of variables: mutables and permanents. Mutables are variables as in C. Permanents are different from variables in C; values of permanents can be algebraic formulae (such as $y+z$), the initial value of a permanent is a special symbol indicating that it is undefined, and a permanent is defined at most once. For most programmers, the concept of mutable variables is familiar and the concept of permanents is new.

2. **Composition Operators:** A program in PCN is a program heading (program name and arguments), a declaration of types of mutables and a block. A block is an elementary block or a composed block. An elementary block is an assignment statement (similar to assignments in C or Fortran), or a definition statement that defines permanents, or a call to a program written in PCN, C, or Fortran. Later implementations will allow composition of programs in other languages. A composed block is a composition operator followed by a list of blocks or guarded-blocks (a block preceded by a boolean expression); the composition operator specifies how the blocks are to be put together.

   The only things programmers can do in PCN are:

- *put blocks together using composition operators, or*
- *define elementary blocks.*

PCN has four composition operators: sequential composition, choice composition, parallel composition and interleaved composition. In addition, programmers can define new composition operators in terms of the basic four.

Sequential composition of a list of blocks executes the blocks in sequence, just as in C or Fortran. Choice composition is an extension of **if-then-else** and guarded commands. In a parallel composition of a list of blocks, all blocks in the list are executed in parallel, and the parallel composition completes execution when all its constituent blocks complete execution. In interleaved composition, statements of constituent blocks are executed sequentially, but in an interleaved fashion. For most programmers, sequential and choice composition are familiar concepts, while parallel and interleaved composition are new.

The central concept of PCN is that of composition — putting blocks together — and once that is mastered all forms of composition are equally easy.

3. **Tuples:** PCN has a data type called a tuple which is a (pointer to a) sequence of items between braces '{' and '}'. Tuples play the role of pointers in PCN. Linear lists, circular lists, trees and other such linked structures are constructed using tuples.

Tuples are a minor modification to pointers and structs in C; therefore, programmers will understand the concept readily. The concepts of mutables, permanents and composition are different from concepts found in conventional notations, and therefore, readers should focus attention on these ideas.

1.2 Highlighting Examples, Syntax and Operation

Most of this primer consists of examples; a large number are found in the *Simple Programming Examples* section. There are times when readers will want to study examples carefully and there are other times when readers will want

to skip examples. To help identify examples, an example is placed between lines as in:

Examples of tuples

The empty tuple: {}.

Most examples are on odd-numbered pages, and are therefore on right-hand side pages with most text on left-hand side pages.

For ease in indentification, syntax is placed between lines as in:

---

tuple :: $\{\prec term \succ\} \mid \ldots$

The operation of statements in PCN are described in terms of operations in familiar languages such as C. Operational descriptions of PCN statements are placed between lines as in:

---

repeat skip until rhs is reducible;
assign the reduced value of rhs to m.

---

2 Syntax

The syntax is given in BNF. All nonterminal symbols are in italics, and all terminal symbols are in plain type. The notation $\prec$ *su* $\succ$, where *su* is a syntactic unit, represents a list of zero or more instances of the syntactic unit, with multiple instances separated by commas. The notation $\prec$ *su* $\succ^{(1)}$ is a list of one or more instances of *su* separated by commas. The notation $\langle su \rangle$ denotes an optional syntactic unit *su*.

Symbols in this document translate to symbols found on standard keyboards as in C:
$\stackrel{?}{=}$ translates to ?=, $\leq$ to <=, $\neq$ to !=,$\rightarrow$ to ->, and $\geq$ to >=.

Variable names, macros, file inclusion and comments are as in C.

A variable name is a sequence of characters where the first character is a letter, and a character is a letter or a digit. A letter is an upper case or lower case letter of the alphabet, or it is the symbol "_". An upper case letter is different from a lower case letter.

Macros and file inclusion are discussed in the section called *Compilation and Modules.*

A comment begins with /* and ends with */ as in C.

3 Data Types

3.1 Conventional Data Types

Conventional data types, such as in C, are also data types in PCN:

1. **char** for character,
2. **int** for integer,
3. **float** for single-precision floating point number, and
4. **double** for double-precision floating point number.

Qualifiers for int in C (such as short, long, and unsigned) are not available in PCN.

PCN has **arrays** of these data types. Arrays in PCN and C are treated in the same way.

Strings in PCN are treated in exactly the same way as in C. A string S is an array A of char, where the characters of S are $A[i]$, in increasing order of i starting with $i = 0$ and ending with $i = k$, where k is the smallest index such that $A[k+1]$ is the *null character* `\0`. If $A[0]$ is the null character, S is the empty string. A constant string can be denoted by placing the characters of the string between quotes; for example `"PCN"` is a string consisting of the three characters: P, C and N. The empty string is `" "`.

In this document, a **number** is an integer or a single-precision floating point value or a double-precision floating point value. We define a **simple-value** as a number or a character or an array of numbers or a string.

PCN does not have structures (or 'structs') as C does. However, PCN has **tuples** [6], which are discussed next.

3.2 Tuples and Lists

A tuple is a pointer to a possibly empty sequence of **terms**, where a term is a simple-value, an expression, or a tuple. Expressions in PCN have the same syntax as arithmetic expressions in C, except that the only operators in PCN

Examples of tuples

The empty tuple: $\{\}$.

A 1-tuple: $\{\{x\}\}$, where the single element of the tuple is itself a 1-tuple, $\{x\}$.

A 2-tuple: $\{"msg", 3\}$.

Examples of lists

The empty list: $\{\}$

A single-element list: $\{d, \{\}\}$ is a list containing a single element, d.

A four-element list: $\{a, \{b, \{c, \{d, \{\}\}\}\}\}$ is a list containing the sequence of 4 elements, a, b, c and d, in that order.

are $+$, $-$, $*$ and $/$. A tuple is represented in a program as a sequence of terms between braces — '{' and '}' — where terms are separated by commas.

In most of this manual we treat a tuple as the sequence of terms that it points to, rather than as a pointer. For instance, we shall refer to the tuple $\{1,2\}$ rather than the *pointer* to $\{1,2\}$.

A list is a special case of tuple. A list is:

1. The empty tuple, $\{\}$, or
2. A 2-tuple, $\{a,b\}$, where the second element of the tuple, b, is a list.

PCN has a more succinct notation for lists: a list consisting of a sequence of zero or more elements can be represented by the sequence of elements between the enclosing brackets'[' and ']'. Also, for brevity, we can employ the notation, $[L_1, L_2, \ldots, L_k \mid z]$, to represent the tuple, $\{L_1, \{L_2, \{\ldots\{L_k, z\}\ldots\}\}\}$.

For convenience in dealing with program-calls, the notation $h(x_0, \ldots, x_k)$, where h is an identifier and $x_0, \ldots, x_k$ are tuple-elements, denotes the tuple $\{"h", x_0, \ldots, x_k\}$.

Syntax of Tuples A tuple has the following syntax:

---

tuple :: $\{\prec$ *term* $\succ\}$ |
$[\prec$ *term* $\succ]$ |
$[\prec$ *term* $\succ^{(1)}$ '|' *term*] |
identifier$(\prec$ *term* $\succ)$

term :: *expression* | *tuple*

Examples of list notation

The empty list: The empty list is [ ]; hence [ ] = { }.

A single-element list: $[d]$ is a list containing a single element, d; hence $[d] = \{d, \{\}\}$.

A four-element list: $[a, b, c, d]$ is a list containing the sequence of 4 elements, a, b, c and d, in that order; hence $[a, b, c, d] = \{a, \{b, \{c, \{d, \{\}\}\}\}\}$.

Catenation of lists: A list, y, consisting of a sequence of values, u, v, w, followed by another list, z, is represented by: $[u, v, w \mid z]$; thus, if $z = [b, c, d]$, and $y = [u, v, w \mid z]$, then $y = [u, v, w, b, c, d]$.

More examples of tuples

$g(x)$ and $\{"g", x\}$ denote the same tuple.

$f(x, y)$ and $\{"f", x, y\}$ denote the same tuple.

The *sizeof* Function PCN includes a function *sizeof* which has a single argument and returns:

- the length of its argument in characters, where its argument is treated as a string, if its argument is a string or an array of char,
- the number of elements in the argument if its argument is a tuple or an array of numbers, and
- 1 (one) if its argument is a single number or character.

The argument of *sizeof* must be a variable. Note the difference between the *sizeof* functions in C and in PCN: in C the function returns the size of its argument in bytes, whereas in PCN the function returns the number of elements.

Elements of a tuple are referenced in the same way as elements of an array in C: $t[i]$ is element i of a tuple t, for $0 \leq i < sizeof(t)$.

Examples of sizeof function

Arrays of numbers:
/* declare u to be an array of 10 integers */
int u[10];
sizeof(u) is 10.

A single number:
/* declare i to be an integer */
int i;
sizeof(i) is 1.

A tuple: Let z be the tuple $\{x, y\}$; then *sizeof*(z) is 2.

A string:
/* declare D to be an array of 10 chars */
char D[10];
let D[0] = "P", and let D[1] = \0 (i.e., D[1] is the null character of C) then *sizeof*(D) is 1.

4 Variables

4.1 Values of Variables

At each point in a computation, a variable has precisely one **value**. The value of a variable is a term. (Recall that a term is an expression or a tuple.) The value of a variable in PCN can be an expression such as $y+z$. In C, execution of the assignment $x = y + z$ causes the value of x to become the value of y plus the value of z, and thus the values of variables x, y and z in C are always numbers; the execution of the assignment in C does not make the value of x become the formula $y + z$. Indeed, in most notations, values of variables are numbers or characters, but not expressions. Variables in PCN can have expressions as values which allows for a degree of symbolic computation in addition to the usual numeric computation of C and Fortran.

Notation for Value of a Variable In C, $x = 2$ denotes that x has value 2. We need additional notation to denote the value of a variable in PCN as illustrated by the following example. In PCN, $u = x + y + z$ does not necessarily imply that the value of u is the expression $x + y + z$, because it is possible that the value of u is the expression $v + z$, and the value of v is the expression $x + y$. To avoid this ambiguity, we denote the value of a variable x by $value(x)$. Instead of writing, $x = 2$, to denote that x has value 2, we shall say that $value(x)$ is 2. Thus, *value* is a function that maps from variables to terms.

At any point in a computation, we can substitute the value of x for x. Therefore, if u has value $v + z$ and v has value $x + y$ at some point in a computation, then $u = v + z$ and $v = x + y$ and hence, $u = (x + y) + z$ at that point. If u and v are undefined, we are not permitted to conclude anything about the relationship between u and v; in particular we cannot conclude that $u = v$.

Classes of Variables A variable in PCN is either a **mutable** or a **permanent**. Informally, a mutable is similar to variables in C and a permanent is a variable that is assigned at most once, as in logic programming.

Examples of Values of Variables

Possible values of a variable x are presented next.

`Simple-Value`: *value*(x) is 2.

`Expression as Value`: *value*(x) is $y + z$.

`Expression as Value`: *value*(x) is $u + v * (y + z)$.

`Tuple as Value`: *value*(x) is $\{y\}$.

`Tuple as Value`: *value*(x) is $\{2,"A", y, z\}$.

4.2 Mutables

The type of a mutable is declared in programs in which it is used, its initial value is an arbitrary value of its declared type, and its value can be changed arbitrarily often during a computation by execution of assignment statements that assign values to it. Type declarations are as in C. A mutable type is a C type (i.e., char, int, float or double) or it is a **tuple**.

The elements of a mutable tuple are variables. The declaration of a tuple is identical to other declarations in C, and the word `tuple` is used as the type name. Arrays of mutable tuples are permissible. The value of a variable, declared to be of type tuple, is a tuple (and is not an address of a tuple).

4.3 Permanents

A permanent is different from variables used in C. A permanent is either **undefined** or **defined**. A permanent is defined *at most once* in a computation. A permanent is defined to be a term. A permanent is defined by executing a **definition statement** in which the permanent appears on the left-hand side; definition statements are described later.

The value of an undefined permanent is undefined. The value of a defined permanent is its definition. The value of a permanent does not reference mutables.

Permanents are not declared.

4.4 Review of Differences between Permanents and Mutables

Declaration Mutables are declared. Permanents are not declared.

Initial Value The initial value of a mutable is an arbitrary value of its declared type. The initial value of a permanent is a special symbol indicating that it is undefined.

Expressions as Values The value of a mutable cannot be an expression. The value of a permanent can be an expression.

Tuples as Values The elements of a mutable tuple can be mutables or permanents. The value of a permanent can be a tuple; however, no element of a permanent tuple can be a mutable.

Changes in Values The value of a mutable can be changed arbitrarily many times. The value of a permanent can be changed at most once, from undefined to a defined value. Once a permanent is defined, its value remains unchanged forever thereafter.

5 Programs

A program consisists of a heading followed by a declaration section followed by a block. The heading is the program name and a list of formal parameters, as in C. In PCN all parameters are passed by reference, unlike in in C where parameters can be passed by value. The syntax of a declararation section is identical to that in C. The scope of a variable is the program in which it appears: all variables that appear in a program are either formal parameters or local variables of the program. All mutables referenced in a program are declared in the declaration section of the program; permanents are not declared.

Local variables in PCN are local to the program in which they are declared, whereas local variables in C can be declared to be local to *blocks* within programs. Also, C allows programs to access variables that are not formal parameters or local variables of the program, whereas PCN does not.

The dimensions of a local array of a program can change from one call of the program to the next, unlike in C where some of the dimensions of a multidimensional local array must remain unchanged in all calls.

The syntax of a block is:

---

block	::	*elementary-block* | *composed-block*
elementary-block	::	*definition-statement* | *assignment-statement* | *program-call*

Composed blocks are discussed later; the next few sections discuss each of the forms of elementary blocks.

An Example of Formal Parameters and Local Variables

$p(sum, x)$
int sum, v;
$\{?\ x \stackrel{?}{=} [m \mid xs]\ \rightarrow$
$\quad\quad \{;\ v := m,\ sum := sum + v,\ p(sum, xs)\}$
$\}$

The operators in this example are not important here; only the heading and the declarations are relevant. The first line is the heading for a program with name, p, and two formal parameters, sum and x. The second line declares sum and v to be integer. Therefore, sum and v are mutable. Since the types of x, xs and m are not declared, they are permanent. Since xs, m and v are not formal parameters, they are local variables of program p.

6 Definition statements

A definition statement has the following syntax:

---

definition-statement :: *permanent* = *term*

The right-hand side of a definition cannot be a mutable tuple. The execution of the definition statement $x = rhs$ completes, and at completion, *value*(x) is rhs' where rhs' is obtained by substituting *value*(v) for each mutable v in rhs. Mutables do not appear in rhs', and hence the value of a permanent does not name mutables.

The value of an expression in PCN can be an array. For example if mutable m is declared to be an array of integers, then the value of the expression 'm' is an array. Hence, upon completion of the execution of the statement $x = m$, the value of x is an array.

Arithmetic operators in PCN are identical to those in C, and hence their operands are numbers, not arrays of numbers. So the expression '$m+1$', where m is an array, is incorrect.

Examples of Definition Statements Without Mutables

In these examples, z is a permanent.

`number`: Execution of the definition statement
$z = 3.0$
terminates with
$value(z) = 3.0$.

`string`: Execution of the definition statement,
$z =$ "abc "
terminates with
$value(z) =$ "abc".

Examples of Definition Statements With Mutables

In these examples, z and y are permanents, and m is a mutable with value 2 at the point in the computation at which the definition statements are executed.

`expression`: Execution of the definition statement

$$z = y + m + 5$$

terminates with

$$value(z) = y + 2 + 5.$$

`tuple`: Execution of the definition statement

$$z = \{y, \{m\}, 5\}$$

terminates with

$$value(z) = \{y, \{2\}, 5\}.$$

`tuple`: Execution of the definition statement

$$z = [m, "b"]$$

terminates with

$$value(z) = [2, "b"].$$

Examples of Definition Statements With Arrays

If A is a 2-element integer array with $A[0] = 1$, and $A[1] = 2$, then the definition statement, $z = A$ terminates with z defined as a 2-element integer array with $value(z[0]) = 1$, and $value(z[1]) = 2$.

7 Reducibility

The meaning of an assignment is based on the concept of reducibility. For a term e, the *reduced value of* e is a simple-value or tuple x, where $x = e$. A term e *is reducible at a point* t *in a computation* if and only if a reduced value of e can be computed at t, i.e.,

1. e is a simple-value or a tuple, in which case the reduced value of e is e, or

2. $value(e)$ is f and f is reducible, in which case the reduced value of e is the reduced value of f, or

3. e is an expression and all variables in e are reducible, in which case the reduced value of e is computed by substituting the reduced value of v for each variable v in e and evaluating.

7.1 Properties of Reducibility

Value Equals Reduced Value For all variables x, the value of x is equal to its reduced value. This is because the reduced value of x is obtained from the value of x by substitution and expression evaluation.

Unique Reduced Values The reduced value of an element e at a point in a computation is unique. For example, if the reduced value of e is 2 at a point in a computation then the reduced value of e cannot also be 3 at that point.

Mutables Are Reducible A mutable is reducible at all points in computation and the reduced value of a mutable is its value.

Undefined Permanents Are Not Reducible An undefined permanent is not reducible because the value of an undefined permanent is a special symbol indicating that it is undefined, and hence none of the rules of reducibility can be employed to compute a simple value or tuple for it.

Examples of Reducibility

Example 1
Let z be a permanent, and let m be a mutable that is declared to be an integer. If at a point t in a computation, *value*(z) is 1 and *value*(m) is 2, then $z + m$ is reducible and its reduced value is 3, at t.
If $m = 4$ at a later point t', then the reduced value of $z + m$ is 5 at t'.

Example 2
If at a point t in a computation, y is undefined, then y is not reducible at t. If at t, the value of z is $y + 1$, then z is not reducible at t because y is not reducible at t.

Example 3
If at a point t in a computation, *value*(z) is $y + 1$, and *value*(y) is 0, then z is reducible and its reduced value is 1 at t. Furthermore, if z is a permanent, the reduced value of z remains 1 at all points after t.

Defined Permanents A defined permanent may or may not be reducible. For example, if $value(x) = y + z$, where x, y and z are permanents, then x is not reducible if y or z is not reducible. If, however, y and z are reducible with reduced values (say) 1 and 2 respectively, then x is reducible and has reduced value 3.

Once Reducible, Remains Reducible Once a term is reducible it remains reducible forever thereafter. The reasons for this are as follows. Mutables, simple-values and tuples are always reducible. Once a permanent is reducible it remains reducible because its value remains unchanged. Once an expression (that can name mutables and permanents or both) is reducible it remains reducible.

Reduced Values of Permanents The reduced value of a permanent remains unchanged. For example, if the reduced value of z is 2 at some point in a computation, then the reduced value of z remains 2 forever thereafter. Likewise, the reduced value of an expression, that does not name mutables, remains unchanged. For example, if the reduced value of $y + z$ is 3 at some point in a computation (where y and z are permanents), then the reduced value of $y + z$ remains 3 thereafter.

Reduced Values of Mutables The reduced value of a mutable can change; for instance mutable m can have value 2 at some point in a computation and value 3 at a later point. Likewise, the reduced value of an expression that names mutables can change. For example, the value of expression $y + z + m$ can change, where m is a mutable, because m can change value.

More Examples of Reducibility

Example 4
If at a point t in a computation, $value(y) = \{1, z\}$, then y is reducible and its reduced value is $\{1, z\}$ at t. Note that y is reducible even if z is not reducible.

Example 5
If at a point t in a computation, $value(y) = A$, where A is an integer array, then y is reducible, and its reduced value is A at t.

Example 6
If at a point t in a computation, $value(y) = x + 1$, and $value(x) = 2 * y - 2$, then both x and y are nonreducible at t. (Note that from the mathematics of simultaneous equations we can conclude $x = 0$ and $y = 1$, but according to our definitions x and y are not reducible.)

8 Assignment Statements

An assignment-statement is an assignment of either a simple-value or a tuple.

---

assignment-statement :: *simple-assignment* |
tuple-assignment

8.1 Assignment of Simple-Values

The syntax of an assignment statement that assigns a simple-value is:

---

simple-assignment :: *mutable* := *expression*

The execution of the assignment statement $m := rhs$ where rhs is an expression and m is a mutable variable, declared to be one of the types in C, is as follows:

---

repeat skip until rhs is reducible;
assign the reduced value of rhs to m.

A skip is an operation that does nothing, and it is sometimes referred to as a 'no-op.' Therefore, while rhs is not reducible, the program executes operations that do nothing — in other words, the program waits. When rhs becomes reducible, the reduced value of rhs (coerced to be the same type as m) is assigned to m, and the assignment completes. If rhs never becomes

Simple Examples of Assignment

Right-Hand Side Does Not Reference Permanents

int m, i, j, u[10], v[10];
..., $m := i + j$, ..., $u := v$, ...

The assignment $m := i + j$ is executed in the same way as the assignment $m = i + j$ in C: the sum of the values of the integers i and j is assigned to m.
The assignment $u : = v$ makes u become the array v.

Right-Hand Side References Permanents

int m;
..., $m := z$, ...

The assignment, $m := z$, where m is an integer mutable, and z is a permanent, is executed as follows. While z is not reducible, skip. When z becomes reducible, assign its value (coerced to integer) to m. If z never becomes reducible, execution of the assignment does not complete.

reducible the assignment does not complete, and execution of the assignment statement is an infinite number of skips. Note that if *rhs* does not reference permanents, the assignment statement is executed without skips, because *rhs* is reducible; in this case the assignment is executed in the same way as assignments in C.

8.2 Assignment to Mutable Tuples

The syntax of an assignment-statement that assigns a tuple is as follows:

---

tuple-assignment	::	*mutable* := *tuple* |
		mutable := *mutable*

The right-hand side of an assignment to a mutable tuple is a tuple all of whose elements are variables, or it is a mutable that is declared to be a tuple. An assignment to a mutable tuple always completes. (Unlike assignments to variables declared to be one of the C types, the execution of an assignment to a tuple does not skip until the right-hand side becomes reducible; this is because the right-hand side is a tuple and hence is reducible.)

Upon completion of the assignment $m := rhs$, where *rhs* is a tuple, $value(m)$ is *rhs*. The assignment $u := v$, where u and v are mutables declared to be of type tuple, copies the value of v into u, as does the execution of any assignment $u := v$. Since v is a pointer to a tuple, the assignment $u := v$ makes u and v point to the same tuple. Therefore, the assignment makes $value(u)$ become $value(v)$.

Examples of Assignments to Tuples

Right-Hand Side is a Tuple

/* declare m to be a mutable of type tuple */
tuple m;
... $m := \{u, v\}$ ...

This assignment $m := \{u, v\}$ completes, and at completion, *value*(m) is $\{u, v\}$.

9 Program Calls

The syntax of a program call is:

---

program-call :: *program-name*($\prec$ *term* $\succ$) |
'permanent($\prec$ *term* $\succ$) |
mutable

A program call *program-name*($\prec$ *term* $\succ$) is the same as a function call in C, except that all parameter passing is by reference, and the program does not return a value (in the way that a function does). Later, we will describe two modifications to this syntax that allow programmers to specify processors on which called programs are to be spawned, and modules (files) that contain the source texts of the called programs. The essential meaning of program calls does not depend on these modifications; so, we discuss these modifications later in sections called *Architectures, Implementation and Efficiency* and *Compilation and Modules*, respectively.

Later in this section, we consider program calls of the forms *'permanent*($\prec$ *term* $\succ$) and *mutable*.

9.1 Calling C programs from PCN

PCN programs can call C programs. The actual parameters in a call to a C program can be permanents or mutables. All parameter passing in PCN is by reference; hence, the arguments of the C program must be *pointers* to simple-values (i.e., char, int, float or double) or mutable tuples.

A mutable tuple x is assigned $\{Cvar[0], \ldots, Cvar[n-1]\}$ by executing the built-in subroutine *build_data*$(Ctype, n, Cvar, x)$, where $Ctype$ is char, int, float or double, n is an integer, and $Cvar$ is a C variable of type array of $Ctype$ with at least n elements. Similarly, $Cvar$ becomes $x[i]$ by executing *read_data*$(Ctype, n, Cvar, x)$.

Examples of Calls to Programs

Consider the C program:

$q(v, w, x)$
int $*v$, $*w$, $*x$;
{ $*v = *v - *x$; $*w = *w + *x$ }

The execution of the call $q(a, b, z)$, where a and b are mutable integer variables and z is a permanent, is as follows: repeat skip until z is reducible; when z becomes reducible, execute $q(v, w, z')$ where z' is the reduced value of z. Note that even though z is a permanent, and its value cannot be changed by the C program, z is passed by reference and not by value. That is why the type declaration of formal parameter, x, is 'int $*x$', and not 'int x'. Permanent z must reduce to an integer value because the corresponding formal parameter x is a pointer to an integer.

Consider the PCN program:

$p(w, x, y, z)$
$\{\| \; y = w + x, \;\; z = w - x\}$

A call $p(a, b, c, d)$ of program p causes program p to be executed, *even if actual parameters are not reducible.* We will see later that the program completes, and at completion *value*(c) is $a + b$ and *value*(d) is $a - b$, regardless of whether a or b are reducible.

The execution of a C program call is as follows:

---

repeat skip until all actual parameters are reducible;
execute the C program.

We do not specify the behavior of PCN programs that call C programs which continue execution for ever; therefore, programmers must ensure that C programs terminate.

9.2 Calling Fortran Programs from PCN

Fortran programs are called in the same way as C programs. Parameter passing in Fortran is by reference, as it is in PCN.

9.3 Calling PCN Programs from PCN

PCN programs can call PCN programs; the actual parameters of the call can be mutables or permanents. The called program is initiated even if some or all of the actual parameters are not reducible.

There is a difference between the execution of calls to C programs and PCN programs. A called C program is initiated only when all its actual parameters are reducible. A called PCN program is initiated without waiting for all its actual parameters to be reducible. (A strict semantic is used for C calls and a nonstrict semantic for PCN calls.)

Types of actual parameters should be the same as types of corresponding formal parameters in calls to PCN programs. In particular, an actual parameter should be a permanent if and only if the corresponding formal parameter is a permanent.

9.4 Program Names as Actual Parameters of Programs

A program-name can be passed as an argument of a program. To distinguish a *variable* whose value is a program name from a *program name*, the symbol

Example of a Program Name as A Parameter

Next, consider a program *map*, which has a formal parameter *operator* defined as a string that is a name of a program.

```
map(operator, lst, result)
int result;
{? lst ≟ [head | tail] →
        {; 'operator(head, result),
           map(operator, tail, result)
        }
}
```

Passing a Program Name as a Parameter

Calling the preceding program with $map(\text{"}add\text{"}, L, R)$, causes the following block to be executed:

```
{? L ≟ [head | tail] →
      {; add(head, R), map("add", tail, R)}
}
```

' is employed. For example $y(x)$ is a call of program y, whereas $'y(x)$ is a call to a program g, where the reduced value of y is "g".

Recall that $f(x_0, \ldots, x_k)$ represents the tuple $\{"f", x_0, \ldots, x_k\}$. Similarly, $'f(x_0, \ldots, x_k)$ represents the tuple $\{f, x_0, \ldots, x_k\}$.

The program call $'y(x)$ where x is a list of actual parameters and y is a permanent, is executed as follows:

---

repeat skip until y is reducible;
execute $g(x)$ where "g" is the reduced value of y.

9.5 Program Calls as Mutables

A program call $f(arg_0, \ldots, arg_n)$ is represented internally within the PCN compiler as the mutable tuple $\{"f", arg_0, \ldots, arg_n\}$. Let the value of a mutable tuple t be $\{"f", arg_0, \ldots, arg_n\}$ at a point where t is executed; then execution of t is equivalent to the execution of the program call $f(arg_0, \ldots, arg_n)$.

Example of a Program Call in a Mutable Tuple

Program *for*, presented next, is similar to a for-loop. It executes program, *prog*, for *index* ranging from *low* to *high*.

$for(index, low, high, prog)$
int *index*, *low*, *high*;
tuple *prog*;
{; $index := low$, $loop(index, high, prog)$}
}

$loop(index, high, prog)$
int *index*, *high*;
tuple *prog*;
{? $index \leq high \rightarrow$
    {; *prog*, $index := index + 1$, $loop(index, high, prog)$}
}

The call, $for(i, 0, 2, sum(x, i, result))$, causes the following sequence of program calls to be executed:
$sum(x, 0, result), sum(x, 1, result), sum(x, 2, result)$.

The call $for(j, 0, 1, product(j, x, y, val))$ causes the following sequence of program calls to be executed:
$product(0, x, y, val), product(1, x, y, val)$.

10 Composed Blocks

A composed block has the following syntax.

---

composed-block :: $\{;\ \prec block \succ^{(1)}\ \}\ |$
$\{\|\ \prec block \succ^{(1)}\ \}\ |$
$\{[]\ \prec block \succ^{(1)}\ \}\ |$
$\{?\ \prec guard \rightarrow block \succ^{(1)}\ \}$

---

where ';' denotes sequential composition, '||' denotes parallel composition, '[]' denotes interleaved composition and '?' denotes choice composition.

11 Sequential Composition

Let d be the block $\{;\ \ b_1, \ldots, b_k\}$, where $k > 0$. The execution of d is a sequential execution of b_i, in order, from $i = 1$ to $i = k$.

12 Parallel Composition

Let d be the block $\{\|\ \ b_1, \ldots, b_k\}$, where $k > 0$, and for all i where $0 < i \leq k$, b_i is a block. In an execution of d, all blocks b_i are executed in parallel. Block d terminates when the computations of b_i terminate for all i. (A computation of d is a fair interleaving of computations of b_i, for all i, $0 < i \leq k$. Execution is fair in the following sense: For all i, it is always the case that eventually computation of b_i will progress if b_i has not terminated.)

Programmers must ensure that the following condition about shared variables is satisfied.

Restriction on Shared Variables in Parallel Composition *Shared mutables must not change value during parallel composition.*

Example of Sequential Composition

```
p(j, k, x, y)
int j, k, m;
/* Let the value of j be J. */
/* m is a local integer mutable of p */

{;
   m := 2,
   /* value(m) is 2 */

   x = m + 1,
   /* The reduced value of x is 3. */

   k := y + j
   /* value(k) is sum of the */
   /* reduced values of y and j. */
}
```

First m becomes 2, then x is defined as 2+1 (and hence its reduced value is 3), and then, after executing skips until y becomes reducible, x becomes the sum of the values of y and j.

This restriction is equivalent to: In a parallel composition block d defined as $\{\|\ b_1, \ldots, b_k\}$,

for each variable v accessed in distinct blocks b_i and b_j:

1. v remains unchanged during the execution of d, or

2. v is a permanent.

For this purpose, each element of a shared array is treated as separate variable; therefore, blocks composed in parallel can modify a shared array, but each *element* of the array must remain unchanged or be accessed by at most one block. Similarly, each element of a tuple is treated as a separate element.

An important consequence of this restriction is that blocks, composed in parallel, interact with each other in a disciplined manner. Consider a predicate z that references only variables that appear in one of the constituent blocks, say b_i of parallel composition block d. For example, z could be $u = v + 1$, where u and v are variables referenced in block b_i. Suppose we can reason from the text of block b_i (i.e., by considering block b_i in isolation, independent of the blocks that b_i is composed with) that predicate z holds at some point p in b_i. In our reasoning we are not permitted to conclude that permanents are undefined if they are undefined in b_i (because permanents can be defined in blocks that are composed with b_i). Then, no matter what blocks are composed in parallel with b_i, we are assured that our reasoning is valid, and z holds at p.

An equally important consequence of this restriction is that we do not have to be concerned about atomicity in parallel composition.

Choice composition is discussed next, and interleaved composition is discussed later.

Example of Parallel Composition

$\{\|\ p(j,k,x,y),\ y = x + j\}$

(Program p is defined in the previous example.) This parallel composition block obeys our convention about shared variables. The only shared mutable is j which is not modified in the block.
Let the value of j be 1 when this block is executed. The computation of this block terminates, and at termination, the reduced value of x is 3, and y is 4, and k is 5.
A possible computation of the parallel composition block is: y is defined as $x+1$ (assuming $j = 1$), then, in program p: m becomes 2, then x becomes defined as $2 + 1$, and finally k becomes 5. Note that y can become defined before x becomes defined.

13 Choice Composition

A choice composition block is similar to a guarded command [5]. A guard in a choice composition block is a boolean expression or the keyword **default**. At a point in a computation, a boolean expression is:

1. not reducible, or
2. reducible and has value *true*, or
3. reducible and has value *false*.

We shall see later that once a guard is reducible it remains reducible for ever thereafter.

There can be at most one default guard in a choice composition block. A choice block without a default guard is equivalent to a choice block with the addition of the guarded block: default $\rightarrow$ skip. Therefore,we can restrict attention to choice blocks that contain precisely one default guard.

The basic idea about execution of the choice block:

$\{?$ default $\rightarrow b_0, G_1 \rightarrow b_1, \ldots, G_k \rightarrow b_k \}$

is as follows:

1. If all guards are *false* then execute b_0; execution of the choice block terminates when execution of b_0 terminates.
2. If at least one guard is *true* then execute any block b_i where G_i is *true*; execution of the choice block terminates when execution of b_i terminates.
3. Because guards can be nonreducible, there is a third possibility: at least one guard is nonreducible and no guard is *true*. In this case the program repeatedly executes skips until one of the first two conditions holds.

Execution Details Details of executions are only relevant in interleaved composition, which is used rarely. Hence, the reader can skip to the next subsection (which is called *Guards*).

Examples of Choice Blocks

Example 1
Consider the choice-block:

$$\begin{array}{ll}\{? & x \geq 0 \rightarrow y = x + 1, \\ & x \leq 0 \rightarrow y = x - 1 \\ \}\end{array}$$

The execution of this block is as follows. While x is irreducible, skip. When x becomes reducible, if $x > 0$, then only the first guard holds, and hence y is defined as $x+1$; if $x < 0$, then only the second guard holds, and hence y is defined as $x - 1$; if $x = 0$, then both guards hold, and a nondeterministic choice is made to define y either as $x + 1$, or as $x - 1$. Execution of the block terminates after y is defined.

Example 2
Consider the choice-block:

$$\begin{array}{ll}\{? & x \geq 0 \rightarrow y = x + 1 \\ \}\end{array}$$

The execution of this block is as follows. While x is irreducible, skip. When x becomes reducible: if $x \geq 0$ then y is defined as $x + 1$ else y is left unchanged; execution of the block terminates.

A more detailed execution of the choice block described in a C-like notation is as follows. Let S be a set of indices. Initially S is the set of indices i for all i where $1 \leq i \leq k$. Let *done* be a boolean variable that holds when execution of the choice block completes; initially *done* = *false*. While S is nonempty and *done* does not hold, select a member i of set S, where the selection is made nondeterministically and fairly; if G_i is reducible with value *true* then execute b_i and then set *done* to *true*; if G_i is reducible with value *false* then delete i from S. If the while loop completes, then upon completion of the loop, either S is empty or *done* holds. On completion of the while loop, if S is empty then execute b_0 and then set *done* to *true*.

Fairness in selection of members of S is a requirement that if members are selected from S infinitely often then each member that remains in S is selected infinitely often. Equivalently, if some guard G_i becomes reducible, then eventually i is removed from set S (if, when G_i is evaluated, its reduced value is *false*) or some block b_j is executed.

$S := \{i | 1 \leq i \leq k\}$;
done := *false*;
while($S \neq \{\}$ && !*done*){
    nondeterministically and fairly select any i in S;
    if(G_i is reducible){
        if(G_i){b_i; *done* = *true*}
        else $S = S - i$
    }
}
/* end execution of while loop */
/* S is empty or *done* */
if($S = \{\}$){b_0; *done* = *true*}

13.1 Guards

The syntax of a guard is:

Another Example of a Choice Block

Example 3
Consider the choice-block:

$$\{? \quad x \geq 0 \rightarrow y = x + 1,$$
$$\quad z \geq 0 \rightarrow y = z + 1$$
$$\}$$

Repeat skips until both guards are reducible and both have reduced value *false*, or at least one of the guards is reducible and has reduced value *true*. In the former case the choice block terminates without changing the value of any variable. In the latter case, if both guards have reduced value *true* then execute either $y = x + 1$ or $y = z + 1$, if only $x \geq 0$ has reduced value *true* then execute $y = x + 1$, and if only $z \geq 0$ has reduced value *true* then execute $y = z + 1$.

---

guard :: $\prec$ *guard-element* $\succ^{(1)}$ |
default

A guard is either a sequence of one or more guard elements or *default.*

Case 1: If All Guard-Elements are Reducible If all the guard-elements of a guard G are reducible, then G is reducible, and the value of G is a 'conditional-and' of its guard-elements: Each of the guard-elements in the sequence is evaluated in order until all guard-elements in the sequence are evaluated or a guard-element evaluates to *false*; if all guard-elements evaluate to *true* the value of the guard is *true*, otherwise the value of the guard is *false.* The evaluation of a guard is similar to the evaluation, in C, of an expression consisting of the sequence of guard-elements with the logical connective `&&` between guard-elements.

Case 2: At Least One of the Guard-Elements is Irreducible Next, consider the case where at least one of G's guard-elements is irreducible: If all guard-elements before the first nonreducible element of G evaluate to *true*, then G is not reducible; otherwise, $G = false$.

13.2 Guard-Elements

The syntax of a guard-element is:

---

guard-element :: *type-check* | *comparison* |
data-check | *pattern-match*

A Simple Example of Type Check

$\{?\ \ \text{int}(x), x \geq 5 \ \rightarrow\ y = x + 1\ \}$

Consider the case where x is reducible. The guard is evaluated as follows: First evaluate int(x); if x is an integer then evaluate $x \geq 5$. Therefore, if x is a character, the guard evaluates to *false* and evaluation of the guard stops. If x is integer and $x > 5$, then the guard evaluates to *true*, and if x is integer and $x \leq 5$, then the guard evaluates to *false*.

Type Checks The syntax for *type-check* is:

---

type-check :: *type-name*(*permanent*)
type-name :: int | float | double | char | tuple

If x is not reducible, *type-check* $h(x)$ is not reducible. If x is reducible, *type-check* $h(x)$ evaluates to:

1. *true* if the reduced value of x is of type h or is an array of type h,
2. *false* otherwise.

For example, if x is reducible, then int(x) holds if and only if the reduced value of x is an integer or an array of integers.

Comparison The syntax of a comparison is:

---

comparison :: *term equality-test term* |
expression ordering expression
equality-test :: $== \mid \neq$
ordering :: $< \mid \leq \mid > \mid \geq$

A comparison $x\ \alpha\ y$, where α is an ordering, is reducible if and only if both x and y are reducible; the reduced values of x and y must be numbers or characters (but not arrays). Characters and numbers are compared as in C.

A comparison $x\ \alpha\ y$, where α is an equality-test, is reducible only if both x and y are reducible. An equality-test $x == y$, where the reduced values of

Another Example of Type Check and Comparison

$\{?\ \ \text{int}(x), x \geq y\ \rightarrow\ z = x + 1\ \}$

Consider the case where y is irreducible and x is reducible. If the reduced value of x is an integer, then the guard, is not reducible because the first guard-element evaluates to *true* and the second guard-element is not reducible. If the reduced value of x is a character, then the guard evaluates to *false*, because the first guard-element evaluates to *false*.

Next consider the same program except that the order of guard-elements is reversed.

$\{?\ \ x \geq y, \text{int}(x)\ \rightarrow\ z = x + 1\ \}$

As before, consider the case where y is irreducible and x is reducible. As in the last example, if the reduced value of x is an integer, then the guard, is not reducible because the first guard-element is not reducible. If the reduced value of x is a character then the guard is not reducible for the same reason. Note that in the previous example, if x is a character the guard *is* reducible. This example shows that the ordering of guard-elements can make a difference to the computation.

x and y are tuples, is equivalent to the following sequence of equality-tests:

$sizeof(x) == sizeof(y)$, and for all i where $0 \leq i < sizeof(x)$: $x[i] == y[i]$.

Thus, equality tests of tuples are equivalent to sequences of equality-tests without tuples. A comparison $x = y$ is reducible if x and y reduce to simple values. Equality of characters and numbers is as in C.

Inequality is defined as the negation of equality.

Data Check The syntax of a data-check is:

---

data-check :: data(*permanent*)

If x is reducible then data(x) = *true*. If x is not reducible, then data(x) is also not reducible. The value of data(x) is never *false*.

Testing Equality of Tuples

$\{?\ x == y\ \rightarrow\ p(x)\ \}$

In the equality test, x or y can be tuples (and therefore can be lists). If $x = [0,1,2]$, then the equality test succeeds only if y is equal to the same list. If y is $[0,1|z]$, where z is a nonreducible permanent, then $x == y$ is not reducible.

As another example, consider the case where the reduced value of x is z and the reduced value of y is also z. The equality test is reducible if and only if z is reducible. Of course, if z is reducible, the equality test succeeds.

13.3 Pattern Matches

A pattern-match is merely a syntactic convenience for operating on tuples. It has the following syntax:

---

pattern-match :: *variable* $\stackrel{?}{=}$ *pattern*
pattern :: $\{\prec$ *pattern-element* $\succ\}$
pattern-element :: *simple-value* | *permanent* | *pattern*

A pattern-match $x \stackrel{?}{=} pat$ succeeds (i.e., has reduced value *true*) if the reduced value of x is a pattern of the same 'form' as the pattern, *pat*, on the right. For example, if *pat* is $\{u, v\}$, then the match succeeds if the reduced value of x is a tuple of size 2. If a match succeeds, a variable in the pattern become an alias for the corresponding element of the tuple for the remainder of the guard and its associated block. Thus the pattern-match $x \stackrel{?}{=} \{u, v\}$ succeeds if x is the 2-tuple $\{x[0], x[1]\}$, and if the match succeeds then u becomes an alias for $x[0]$, and v becomes an alias for $x[1]$ for the remainder of the guard and its associated block. Next, matches are discussed in more detail.

Evaluation of a Pattern-Match A permanent that appears in a pattern must be undefined when the pattern-match is evaluated. *Mutables cannot appear in patterns.*

A pattern match, $z \stackrel{?}{=} pat$ can be transformed into a sequence of guard-elements without the pattern match by the following syntactic transformation: Replace the pattern match by,

$$tuple(z), (sizeof(z) = sizeof(pat))$$

and for each i, where $0 \leq i < sizeof(z)$, if $pat[i]$ is a

pattern add the pattern-match, $z[i] \stackrel{?}{=} pat[i]$,

Simple Patterns

In the following example, z, hd and tl are permanents that are undefined at the point at which the pattern matches are executed.

$$z \stackrel{?}{=} \{hd, tl\} \quad \rightarrow \{\| \ hd = u, \ v = tl\}$$

is equivalent to:

$$tuple(z), \ sizeof(z) = 2 \quad \rightarrow \{\| \ z[0] = u, \ v = z[1]\}$$

The match $z \stackrel{?}{=} \{hd, tl\}$ succeeds if z is a tuple of the same form as $\{hd, tl\}$, i.e., if z is a 2-tuple. If the match succeeds, then hd is an alias for $z[0]$, and tl is an alias for $z[1]$ in the remainder of the guard and its associated block.

simple-value add the equality-test, $z[i] == pat[i]$,

permanent replace all instances of $pat[i]$ by $z[i]$ in the remainder of the guard and its associated block.

Patterns with Strings and Numbers

$z \stackrel{?}{=} \{v, \{y\}, 2, \text{"}msg\text{"}\} \quad \rightarrow \{\| \ v = y\}$

is equivalent to

$tuple(z), (sizeof(z) = 4),$
$(tuple(z[1]), (sizeof(z[1]) = 1)),$
$(z[2] = 2), (z[3] = \text{"}msg\text{"}) \quad \rightarrow$
$\{\| \ z[0] = z[1][0]\}$

The match succeeds if the reduced value of z is the same form as the pattern, i.e., if z's reduced value is a 4-tuple where $z[3]$ and $z[4]$ are the integer 2 and the string "*msg*", respectively, and where the form of $z[1]$ is the pattern $\{y\}$. If the match succeeds, v and $z[0]$ are aliases of each other, and likewise, y and $z[1][0]$ are aliases of each other.

Patterns with Lists

$z \stackrel{?}{=} [w \mid x] \quad \rightarrow \{; sum := sum + w, \ p(sum, x)\}$

is equivalent to

$tuple(z), sizeof(z) = 2, \rightarrow$
$\{; \ sum := sum + z[0], \ p(sum, z[1])\}$

14 Interleaved Composition

Interleaved composition is rarely used, but it is discussed here for completeness. This section can be skipped by most programmers. Interleaved composition is used only in a few programs, such as the *fair merge*, that most programmers copy from program libraries.

Interleaved composition is the same as parallel composition except that:

1. shared mutables can be changed during execution of an interleaved composition block (whereas shared mutables cannot be changed during execution of a parallel composition block), and

2. all blocks that are composed with interleaved composition are executed on the same processor (whereas blocks composed with parallel composition can be executed on several processors). Therefore, at most one of the blocks composed by interleaved composition can execute at any time, whereas more than one of the blocks composed by parallel composition can execute at the same time.

The computation of an interleaved composition block is an interleaving of the computations of its constituent blocks. This means that a step in the execution of an interleaved composition block is a step of one of its constituent blocks, and execution of the composed block terminates when all its constituent blocks terminate execution.

A sequence s of steps of computation of a block b is said to be **atomic** if in reasoning about b we can assume that no block, other than b, executes between the initiation of the first step of s and completion of the last step of s. If sequence s is atomic then every subsequence of s is atomic as well.

Consider execution of $\{;\ \ m := n,\ l := m\}$. To reason about this block we need to know whether some other block can change the value of m after execution of the first statement, $m := n$, and before execution of the second statement, $l := m$. If sequential composition of the two statements is an atomic action then, in reasoning about the block, we can assume that no other block executes after the initiation of the first statement and completion of the second. The larger the granularity of atomic actions, the easier it is to reason about a block because there are fewer points at which other blocks can interfere.

Fair-Merge Example

Program *fair-merge* has two input arguments, x and y, and an output argument z, where all arguments are permanents. At any point in the computation, an argument is a list or a list concatenated with an undefined permanent. Let:

$$x = [x^1, \ldots, x^i | xs]$$
$$y = [y^1, \ldots, y^j | ys]$$
$$z = [z^1, \ldots, z^k | zs]$$

where xs, ys and zs are undefined or the empty list, and i, j and k are nonnegative integers. Assume that the elements of x and y are distinct; for example, an element could be tagged with the list (x or y) that it is in. The specification of *fair-merge* is:

1. $z^1, \ldots, z^k$ is an interleaving of some prefix of $x^1, \ldots, x^i$ and some prefix of $y^1, \ldots, y^k$. (A prefix is an initial subsequence.)

2. Eventually, $z = [z^1, \ldots, z^k, \ldots, z^n | u]$, where $n \geq k$, and u is an undefined permanent or the empty list, and $z^1, \ldots, z^n$ contains $x^1, \ldots, x^i$ and $y^1, \ldots, y^j$.

We develop *fair-merge* using interleaved composition.

Atomicity A sequence of steps in a computation is atomic, if no step in the sequence is:

1. a program call, or
2. execution of a parallel composition block, or
3. repetition of skips until a term becomes reducible.

Therefore,

1. The execution of a definition statement is an atomic action.
2. The execution of an assignment statement $x := rhs$, where rhs does not reference permanents, is an atomic action.
3. The execution of an assignment statement $x := rhs$, where x is a mutable declared to be one of the C types, and rhs is an expression that names permanents, is:

   repeat skip until rhs is reducible;
   assign the reduced value of rhs to x.

   The assignment of the reduced value of rhs to x is an atomic action.
4. The execution of an assignment statement $x := rhs$, where rhs is a tuple, or a mutable of type tuple, is an atomic action.
5. A call to a C program is executed as follows:

   repeat skip until all arguments of C are reducible;
   execute the C program passing it pointers to mutable arguments and to reduced values of its permanent arguments.

   The execution of the C program, after all its arguments are reducible, is atomic.

A Program that uses Atomicity

Program *copy* has two arguments; the first is a permanent list, v, that is not modified by the program, and the second is a mutable tuple, m. Mutable tuple m has a single element which is a permanent; let the value of m be $\{y\}$ when $copy(v, m)$ is called. Program *copy* copies list v into y.

$copy(v, m)$
tuple m;
$\{?\ \ v \stackrel{?}{=} [u \mid vs],\ m \stackrel{?}{=} \{y\} \rightarrow$
$\quad \{;\ \ y = [u \mid ys],\ m := \{ys\},\ copy(vs, m)\}$
$\}$

The operation of the program is as follows. If v is the empty list, the program terminates. If v is not the empty list, y becomes the head element, u, of v followed by an undefined permanent ys, and then m becomes the tuple $\{ys\}$, and then $copy(vs, m)$ is executed where vs is the tail of list v. Thus m remains a single-element tuple $\{ys\}$ containing the undefined tail of y.
We shall develop *fair-merge* by interleaved composition of $copy(x, m)$ and $copy(y, m)$. Blocks in interleaved composition can modify shared mutables; hence it is permissible for both *copy* programs to modify m.

6. The execution of sequential composition of atomic actions is an atomic action.

7. The evaluation of a guard (see the paragraph on execution of choice blocks) is an atomic action.

8. In the execution of a guarded block $G \rightarrow B$, the evaluation of G (and if the reduced value of G is *true*) followed by an atomic action in B, is atomic.

Atomicity is not relevant in any form of composition other than interleaving composition. In particular, atomicity is irrelevant in parallel composition because shared mutables cannot change value during parallel composition.

Fair-Merge Example

$fair\_merge(x, y, z)$
tuple m;
{; $m := \{z\}$,
{[] $copy(x, m), copy(y, m)$},
{? $m \stackrel{?}{=} \{w\} \rightarrow w = [\,]$ }
}

In program *copy*, evaluation of a guard followed by sequential execution of the following definition statement and assignment statement form a single atomic action that copies the next value of the input list to the output. At every point in the computation, we are guaranteed that computation of each of the *copy* programs will progress if the program has not terminated. Therefore, all values in the input list eventually get into the output list. A *copy* program terminates if and when its entire input list has been copied (or, equivalently, the list remaining to be copied is empty). The interleaved composition block terminates if and when both *copy* programs terminate; at that point, the undefined tail, w, of z is defined to be the empty list.

15 Syntactic Sugar: Composition of Guarded Statements

The syntax that we gave for composed blocks was:

$$
\begin{array}{lll}
\textit{composed-block} :: & \{;\ \ \prec \textit{block} \succ^{(1)}\ \} \mid \\
& \{\|\ \ \prec \textit{block} \succ^{(1)}\ \} \mid \\
& \{[]\ \ \prec \textit{block} \succ^{(1)}\ \} \mid \\
& \{?\ \ \prec \textit{guard} \rightarrow \textit{block} \succ^{(1)}\ \}
\end{array}
$$

Thus parallel and sequential composition compose *blocks* but not *guard* $\rightarrow$ *block*s. For notational convenience we relax this syntax to allow parallel and sequential composition of *guard* $\rightarrow$ *block*s. Each *guard* $\rightarrow$ *block* is transformed into the choice block $\{?\ \textit{guard} \rightarrow \textit{block}\}$. The sugared syntax is:

$$
\begin{array}{lll}
\textit{composed-block} :: & \{;\ \ \prec \textit{unit} \succ^{(1)}\ \} \mid \\
& \{\|\ \ \prec \textit{unit} \succ^{(1)}\ \} \mid \\
& \{[]\ \ \prec \textit{unit} \succ^{(1)}\ \} \mid \\
& \{?\ \ \prec \textit{guard} \rightarrow \textit{block} \succ^{(1)}\ \}
\end{array}
$$

$$\textit{unit} :: \textit{block} \mid \textit{guard} \rightarrow \textit{block}$$

We may find it convenient to think of *all* composition blocks as consisting of a composition operator operating on a list of *guard* $\rightarrow$ *block*, where some *guard*s can be *true*.

Examples of Syntactic Sugar

The following programs, the first with the added sugar, and the second without it, are equivalent:

$$
\begin{array}{l}
f(t,z) \\
\{?\quad t \stackrel{?}{=} \{left, val, right\} \quad \rightarrow \{\|\ f(left,l), f(right,r), \\
\qquad\qquad\qquad\qquad\qquad\quad l \geq r \rightarrow z = 1 + l, \\
\qquad\qquad\qquad\qquad\qquad\quad r \geq l \rightarrow z = 1 + r \\
\qquad\qquad\qquad\qquad\quad \} \\
\}
\end{array}
$$

$$
\begin{array}{l}
g(t,z) \\
\{?\quad t \stackrel{?}{=} \{left, val, right\} \quad \rightarrow \{\|\ g(left,l), g(right,r), \\
\qquad\qquad\qquad\qquad\qquad\quad \{?\ l \geq r \rightarrow z = 1 + l\}, \\
\qquad\qquad\qquad\qquad\qquad\quad \{?\ r \geq l \rightarrow z = 1 + r\} \\
\qquad\qquad\qquad\qquad\quad \} \\
\}
\end{array}
$$

The Variable Name '_' There are a few places where we would like to refer to a variable that we do not wish to use later. For instance, we may want a pattern with 3 elements, but we wish to use only the first of the 3 elements. Instead of coming up with names for the remaining two elements we can use '_' as a name for both elements. Each instance of '_' is treated as the name of a *new* variable.

16 Built-In Programs and I/O

make_tuple and build_data One of the built-in programs provided in PCN is *make_tuple* which has two arguments, x and n where x is a permanent that is defined by *make_tuple* and n is a variable unchanged by the program. The program defines x to be a tuple of n elements, where n is a nonnegative integer, where each element of x is a permanent. (If $n \leq 0$ then x is defined to be the empty tuple.) The elements of tuple x are left undefined by *make_tuple*. For example, *make_tuple*$(x, 2)$ defines x to be a 2-tuple, leaving permanents $x[0]$ and $x[1]$ undefined.

The program call *make_tuple*(x, n) is executed as follows:

---

repeat skip until n is reducible;
coerce n to integer (if necessary);
if$(n \geq 0)$ define x to be a tuple with n elements
else define x to be a tuple with 0 elements.

Program *build_data* is described in subsection *Calling C Programs from PCN* in section *Program Calls*.

I/O The standard libraries of C are used for I/O and for interacting with the operating system.

An Example Using Variable Name '_'

```
p(x, z)
int n;
{?  x ≟ [{"msg", m, _} | xs] →
        {; f(m, n), z = [n | zs], p(xs, zs)},
    x ≟ [{"val", _, m} | xs] →
        {; g(m, n), z = [n | zs], p(xs, zs)}
}
```

This program has an input argument x and an output argument z. The input is a list of 3-element tuples, where the first element of each tuple is the string "*msg*" or the string "*val*". If the first element is the string "*msg*" then the second element of the tuple is used by the program to compute n by executing $f(m, n)$, and then n is placed in list z. If the second element is the string "*val*" then the third element of the tuple is used by the program to compute n by executing $g(m, n)$, and then n is placed in list z.

17 Architectures, Implementation and Efficiency

The speed of execution of a PCN program depends on its implementation. To understand how to develop programs that execute quickly, programmers need to understand something about the implementation of PCN. The implementation may change in future releases, but the central ideas about efficiency are not likely to change significantly.

Uniprocessors, Multiprocessors, and Multicomputers PCN programs run on uniprocessors, multiprocessors or multicomputers. A multiprocessor is a collection of two or more (uni)processors where all processors access a common address space. A multicomputer is a collection of computers where different computers in the collection have disjoint address spaces[10]. A node of a multicomputer (i.e., one of the computers in the collection that forms the multicomputer) can be a uniprocessor, a multiprocessor or a multicomputer. One of the processors in a multiprocessor or a multicomputer is designated the *host* processor; this is the processor with which programmers interact. (The host processor in a uniprocessor is the uniprocessor itself.) A computer with n processors (in addition to the host) has its processors indexed 0 through n, where the host is indexed 0.

Mutables, Permanents and Address Spaces A mutable resides in precisely one address space. The implementation does not make copies of mutables. By contrast, a permanent can have several copies that reside in arbitrarily many address spaces. Since a permanent is either undefined, or defined and unchanging, all copies of a permanent are consistent in the following sense: if two copies of a permanent differ in value then in one of the copies the permanent is undefined. A program does not make use of undefined permanents; it merely waits for the permanent to become defined. Therefore, no problems are created if the value of a permanent is defined in one copy and undefined in another.

Consider a block b in a program p. If b references mutables (declared in p) then b is executed in the address space in which p is initiated. This is because

there is only one copy of a mutable, and all blocks that access a mutable are executed in the address space in which the mutable resides. If b does not reference mutables then b can be executed in any address space; copies of permanents are made in the address space in which b is executed, as needed by b. The greatest degree of concurrent execution is achieved by employing parallel composition in which the blocks composed in parallel do not share mutables; this allows blocks to be spawned on arbitrary address spaces and thus employs concurrency in multicomputers and multiprocessors.

Interleaved Composition All blocks composed by interleaved composition that reference mutables are executed on the *same* processor (and hence on the same address space) to guarantee that at most one of the blocks composed by interleaved composition is executing at any time.

Granularity If the execution time of a block is small, the time required to spawn the block in a remote address space may exceed the time gained from concurrent execution of the block. Therefore, programmers should ensure that block granularity in parallel composition is appropriate for the target architecure.

18 Simple Programming Examples

18.1 Membership in a List

Develop a program *member* with arguments x, m and r, where x is a list, m and r are mutables, and at termination of execution of the program, $r = true$ if and only if m appears in list x. Mutable m is to be left unchanged by *member*. Of course, permanent x must be left unchanged by *member*.

$member(x, m, r)$
int m, r;
$\{?\quad x \stackrel{?}{=} [\,] \qquad\qquad \rightarrow\ r := false,$
$\qquad x \stackrel{?}{=} [v|xs],\ v == m \quad \rightarrow\ r := true,$
$\qquad x \stackrel{?}{=} [v|xs],\ v \neq m \quad \rightarrow\ member(xs, m, r)$
$\}$

Assume that $false = 0$ and $true = 1$, to be consistent with C.

Operation of the Program

1. If x is the empty list, then r becomes *false* and the program terminates.
2. If x is nonempty, let the head of x be v, and let the tail of x be xs; if $v = m$ then r becomes *true* and the program terminates.
3. If x is nonempty and the head of x is not m, then the value of r is set by $member(xs, m, r)$, and $member(x, m, r)$ terminates execution when $member(xs, m, r)$ does.

Reasoning About the Program In many examples we reason about the correctness (and the efficiency) of programs by induction. In this example, we carry out induction on the length of x. (The length of a list is the number of elements in it.)

Base Case: If x is the empty list then r is *false* upon termination because m does not appear in an empty list.

Induction Step: Assume that $member(x, m, r)$ is correct (i.e., it terminates with the correct values of its arguments) for all lists x that have at most k elements, for some $k \geq 0$, and prove that it is correct for lists with $k+1$ elements. If x has $k+1$ elements it has a head element. Let the head of x be v and let the tail of x be xs; then xs has k elements. If $v = m$ then r must be *true* at termination of the program because m is in list x. If $v \neq m$, then at termination r is *true* if and only if m is in xs; by the induction assumption, $member(xs, m, r)$ sets r to *true* if and only if m is in xs.

A Program with Permanents Now consider a program with the same specification, except that m and r are permanents, where m is left unchanged by the program, and r is defined by the program. All we need to do is to remove the declaration of m and r, and replace assignment statements by definition statements.

$member1(x, m, r)$
$\{?\quad x \stackrel{?}{=} [\,] \qquad\qquad \rightarrow\ r = false,$
$\quad\ x \stackrel{?}{=} [v|xs],\ v == m \quad \rightarrow\ r = true,$
$\quad\ x \stackrel{?}{=} [v|xs],\ v \neq m \quad \rightarrow\ member1(xs, m, r)$
$\}$

18.2 Sum all Elements in a List

Develop a program *sum* with arguments x and r, where x is a list of integers and r is a mutable integer. At termination of execution of the program, r is required to be the initial value of r plus the sum of the elements of x. List x is to be left unchanged. For example, if $x = [1, 2, 3]$ and $r = 4$ initially, then $r = 10$ at termination of the program.

$sum(x, r)$
int r;
$\{?\quad x \stackrel{?}{=} [v|xs] \quad \rightarrow \{;\ r := r + v,\ sum(xs, r)\}$
$\}$

Operation of the Program In this program:

1. If x is the empty list the program terminates leaving r unchanged.
2. If x is nonempty, let v be the head of x and let xs be the tail of x; mutable r becomes $r + v$, and then $sum(xs, r)$ is executed.

Reasoning About the Program Let r^i be the initial value of r, and let r^f be the value of r when program *sum* terminates execution. We reason about the program by inducting on the length of x.

Base Case: If x is the empty list, the program terminates and r is left unmodified, and hence $r^f = r^i$, as required.

Induction Step: Assume that program $sum(x, r)$ is correct for all lists x with length at most k, for some $k \geq 0$, and prove that it is correct for lists x with length $k + 1$. If the length of x is $k + 1$ then x is nonempty; let v be the head of x and let xs be the tail of x. List xs has length k. From the induction assumption, $sum(xs, r)$ is correct. Hence $r^f = r^i + v +$ sum of elements in xs, and hence r^f is the sum of r^i and the sum of all elements of x, as required by the specification.

Another Summation Example Next we write a program *total* to define a permanent z as the sum of all the elements of a list x. The difference between this program and the previous one is that z is permanent whereas r is mutable, and furthermore z is to be the sum of the elements of x, whereas the sum of the elements of x was added to r in the previous program.

$total(x, z)$
int r;
$\{;\ \ r := 0,\ sum(x, r),\ z = r\}$

An alternate version *total*1, using only permanents and parallel composition, is given next.

$total1(x, z)$
$\{?\quad x \stackrel{?}{=} [\,] \quad \rightarrow z = 0,$
$\quad\quad x \stackrel{?}{=} [v|xs] \quad \rightarrow \{\| z = zs + v,\ total1(xs, zs)\}$
$\}$

Operation of the Program The operation of this program is as follows.

1. If x is the empty list then permanent z is defined to be 0.
2. If x is nonempty, let v be the head of x and let xs be the tail of x. Define permanent z to be the expression $zs + v$, where zs is defined by $total1(xs, zs)$.

Reasoning About the Program We are obliged to ensure that blocks in parallel composition do not modify shared mutables. This program has no mutables, and so the restriction about shared variables holds vaccuously.

We reason by induction on the length of x, as in the previous example. The reasoning is not given here because it is largely a repetition of the arguments given earlier.

Difference Between Programs Let x be the list $[1, 2]$. At the termination of $total(x, z)$, permanent z is defined to be the number 3. At the termination of $total1(x, z)$, permanent z is defined by the following equations:
$z = 1 + a$, where a is defined by
$a = 2 + b$, where b is defined by
$b = 0$.
If we now execute $m := z$, where m is a mutable, and then print m, we will get the same answer whether we use *total* or $total1$ because the reduced value of z is the same in both cases.

At first glance, sequential programs such as *total* may appear more efficient in memory and time than parallel programs such as $total1$. Consider the following variation in which a parallel implementation can require less time than a sequential implementation.

18.3 Sum Function of Elements in a List

Let x be a list. Define z as follows: z is the sum over all elements of the list of a function g applied to each element. For example, if x is $[1,2]$ and g is the square operation, then $z = 5$.

A sequential program *sigma*, analagous to *sum* in the previous subsection, is given next.

```
sigma(x, r)
int r, w;
{?   x  ?= [v|xs]   → {; f(v, w), r := r + w, sigma(xs, r)}
}
```

Program $f(v,w)$ sets the value of w to be $g(v)$.

Program Operation The operation of this program is as follows. If x is empty the program terminates with r unchanged. If x is not empty, let v be the head of x, and let xs be the tail of x; first compute w, then increment r and then call $sigma(xs, r)$ to sum the remainder of the list.

Program *tote*, given next, is analogous to *total* of the previous subsection.

```
tote(x, z)
int r;
{;   r := 0, sigma(x, r), z = r}
```

A version using parallel composition is given next.

```
totpar(x, z)
int w;
{?   x ?= [ ]       → z = 0,
     x ?= [v|xs]    → {||  {; f(v, w), y = w},
                           z = zs + y,
                           totpar(xs, zs)
                       }
}
```

This program spawns programs $f(v,w)$, for each element v of list x, in parallel. Thus if f takes a long time to execute, and x is a large list, and there are a large number of processors, the parallel version will execute faster than the sequential version, because execution of f for different elements of the list will proceed in parallel.

In program *totpar* we are obliged to ensure that no mutable shared variable in a parallel composition block is modified. There are no mutable shared variables in the parallel composition block, and hence the restriction is satisfied.

Consider an erroneous program in which the parallel composition block of *totpar* is replaced:

```
error_totpar(x, z)
int w;
{?   x ≟? [ ]        → z = 0,
     x ≟? [v|xs]     → {||  f(v, w),
                            z = zs + w,
                            error_totpar(xs, zs)
                       }
}
```

In this program, w is a mutable shared by blocks $f(v,w)$ and $z = zs + w$ in a parallel composition block, and w is modified by $f(v,w)$; this violates the restriction on shared mutables in parallel composition. The reason for the restriction is that program *error_totpar*(x,z) does not specify what value of w is to be used in the definition $z = zs + w$: is it the value of w before, during or after the execution of $f(v,w)$? This problem does not arise in *totpar* because the shared permanent y is used in place of shared mutable w, and y is defined as the correct value of w — the value of w after $f(v,w)$ is executed — by means of the sequential composition block $\{;\ f(v,w),\ y = w\}$.

18.4 Reverse a List

Develop a program *rev* with arguments x, e and b, where x and e are lists that are to be left unchanged by *rev*, and b is to be defined by *rev* to be the list of elements in x, in reverse order, concatenated with e. For example, if $x =$

$["A","B"]$, and $e = ["C","D"]$, then b is to be defined as $["B","A","C","D"]$. (The name b stands for the beginning of the reversed list, and e stands for the end of the reversed list.)

$$
\begin{array}{l}
rev(x, e, b) \\
\{?\quad x \stackrel{?}{=} [\,] \qquad\qquad \rightarrow b = e, \\
\qquad x \stackrel{?}{=} [v \mid xs] \;\rightarrow \{\|\; es = [v \mid e],\; rev(xs, es, b)\} \\
\}
\end{array}
$$

Operation of the Program

1. If x is the empty list, then $b = e$.

2. If x is nonempty, let v be the head of x and let xs be the tail of x. Define b by $rev(xs, es, b)$, where es is defined as v followed by e. For example, if $x = ["A","B"]$, and $e = ["C","D"]$, then v is $"A"$, and xs is $["B"]$. Hence, es is $["A", "C", "D"]$.

Reasoning About the Program We first ensure that mutable shared variables are not modified in parallel composition, and then reason about the program by induction on the length of x. This reasoning is very similar to that given for the previous programs, and is left to the reader.

18.5 Height Of A Binary Tree

Develop a program ht with arguments t and z, where t is a binary tree, and z is to be defined to be the height of the tree. A tree t is either the empty tuple, $\{\,\}$, or a 3-tuple $\{left, val, right\}$, where $left$ and $right$ are the left and right subtrees of t. Both t and z are permanents, and t is to be left unchanged by the program.

$$
\begin{array}{l}
ht(t, z) \\
\{?\quad t \stackrel{?}{=} \{\,\} \qquad\qquad\qquad\quad \rightarrow z = 0, \\
\qquad t \stackrel{?}{=} \{left, val, right\} \;\;\rightarrow \{\|\; ht(left, l), ht(right, r), \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \{?\; l \geq r \rightarrow z = 1 + l,
\end{array}
$$

$r \geq l \rightarrow z = 1 + r$

}

}

}

Operation of the Program

1. If t is the empty tuple, in which case t is the empty tree, its height z is defined to be 0.

2. If t is not the empty tree, then t is a 3-tuple. Let t be the tuple $\{left, val, right\}$. Define permanents l and r by $ht(left, l)$ and $ht(right, r)$, respectively. If l exceeds r then define z as the expression $1 + l$. If r exceeds l then define z as the expression $1 + r$. If $r = l$, then define z either as $1 + l$ or $1 + r$.

Reasoning About the Program We check that mutable shared variables are not modified in parallel composition. We reason about the program by induction on the height of tree t.

Base Case: If t is the empty tree, then its height z is defined (correctly) as 0.

Induction Step: Assume that the program is correct for all trees t with height at most k for some $k \geq 0$, and prove that the program is correct for all trees with height $k + 1$. If t is a tree with height $k + 1$, then t is a tuple of the form $\{left, val, right\}$ where the heights of trees $left$ and $right$ are at most k. By the induction assumption, $ht(left, l)$ and $ht(right, r)$ correctly define l and r to be the heights of the left and right subtrees of t. Hence z is 1 more than the height of the higher subtree.

18.6 Preorder Traversal of a Binary Tree

Develop a program *preorder* with arguments t, b and e, where t is a binary tree, b and e are lists. Binary trees are represented using tuples as in the last

example. Parameters t and e are to be left unchanged by the program. List b is to be the list consisting of the *val* of all nodes of the tree in preorder, concatenated with list e. (A traversal of a tree in preorder visits the root, then the left subtree, and finally the right subtree.)

$$
\begin{array}{l}
preorder(t, b, e) \\
\{? \quad t \stackrel{?}{=} \{\,\} \qquad\qquad\quad \rightarrow b = e, \\
\qquad t \stackrel{?}{=} \{left, val, right\} \quad \rightarrow \{\| \;\; b = [val \mid l], \\
\qquad\qquad\qquad\qquad\qquad\quad preorder(left, l, m), \\
\qquad\qquad\qquad\qquad\qquad\quad preorder(right, m, e) \\
\qquad\qquad\qquad\qquad\quad \} \\
\}
\end{array}
$$

Operation of the Program

1. If t is the empty tree then b is defined as e.
2. If t is not empty, then it is a tuple of the form $\{left, val, right\}$. In parallel, define m by $preorder(right, m, e)$, and l by $preorder(left, l, m)$, and b as *val* followed by l.

Reasoning About the Program First check that shared mutables are not modified in parallel composition.

We prove correctness of the program by induction on the height of tree t.

Base Case: If t is the empty tree, then $b = e$, because there are no nodes to traverse.

Induction Step: Assume that $preorder(t, b, e)$ is correct for trees of height at most k, $k \geq 0$, and show that it is also correct for trees of height $k+1$. Let t be a tree of height $k+1$. Then $t = \{left, val, right\}$, where $left$ and $right$ are trees of height at most k. Define a permanent m by $preorder(right, m, e)$. By the induction assumption, m is the preorder traversal of the right subtree concatenated with e. Define a permanent l by $preorder(left, l, m)$. By the induction assumption, l is the preorder traversal of the left subtree concatenated

with m. Define b as $[val \mid l]$; hence, b is val followed by the preorder traversal of the left subtree of t followed by the preorder traversal of the right subtree of t, and hence b is the preorder traversal of t.

18.7 Quicksort with Copying

In this section we present C.A.R.Hoare's quicksort [9] program, $q0$, that uses lists (of permanents); later, we discuss an *in place* quicksort, $q1$, that uses arrays. (The quicksort algorithm is discussed in most texts on algorithms such as [1].)

In this section, when we refer to a list of numbers we mean a list of permanents that (eventually) reduce to numbers. Program $q0$ has two input variables, x and end, and one output variable, z: variables x and end are permanents that are not defined by the program, and z is a permanent that is defined by the program. All three variables are lists of numbers. The output z is specified to be the list x sorted in increasing order concatenated with list end. For example if $end = [5,4]$ and $x = [2,1]$, then $z = [1,2,5,4]$. If end is the empty list, then z is x sorted in increasing order.

$q0(x, end, z)$
$\{?\quad x \stackrel{?}{=} [\,] \quad \rightarrow z = end,$
$\quad\; x \stackrel{?}{=} [mid \mid xs] \quad \rightarrow \{\| \; part(mid, xs, left, right),$
$\qquad\qquad\qquad\qquad\quad q0(left, [mid \mid r], z),$
$\qquad\qquad\qquad\qquad\quad q0(right, end, r)$
$\qquad\qquad\qquad\qquad \}$
$\}$

Operation of the Program The operation of program $q0$ is as follows. If x is the empty list then z is defined to be end. If x is nonempty, let mid be the first element of x, and let xs be the remaining elements of x. The call $part(mid, xs, left, right)$ defines $left$ to be the list of values of xs that are at most mid, and $right$ to be the list of values of xs that exceed mid. Call $q0(right, end, r)$, thus defining r to be the sorted list of $right$ appended to end. Call $q0(left, [mid \mid r], z)$, thus defining z to be the sorted list of $left$ followed by mid followed by r.

Reasoning About the Program There are no shared mutables in the parallel composition, and hence the restrictions on parallel composition are satisfied. We reason about this program by induction on the length of x. The reasoning is straightforward and is left to the reader.

Next, we discuss program *part*. Program *part* inspects each element of xs in turn, placing elements that are at most *mid* in $left$ and all other elements in *right*.

$part(mid, xs, left, right)$

$$
\begin{array}{lll}
\{? \quad xs \stackrel{?}{=} [\,] & \rightarrow \{\| & left = [\,],\ right = [\,]\}, \\
\quad xs \stackrel{?}{=} [hd \mid tl],\ hd \leq mid & \rightarrow \{\| & left = [hd \mid ls], \\
 & & \quad part(mid, tl, ls, right) \\
 & \}, & \\
\quad xs \stackrel{?}{=} [hd \mid tl],\ hd > mid & \rightarrow \{\| & right = [hd \mid rs], \\
 & & \quad part(mid, tl, left, rs) \\
 & \} &
\end{array}
$$

}

Operation of the Program If xs is the empty list, define $left$ and *right* to be empty lists. If xs is not empty, then let *hd* and *tl* be the head and tail (respectively) of xs. If *hd* is at most *mid*, define *ls* and *right* by $part(mid, tl, ls, right)$, and define $left$ as *hd* followed by *ls*. If *hd* exceeds *mid*, define $left$ and rs by $part(mid, tl, left, rs)$, and define *right* as *hd* followed by rs.

Reasoning About the Program We check that shared mutables are not modified in parallel composition, and reason about the program by induction on the length of xs.

Base Case: If xs is the empty list, then $left$ and *right* are correctly defined as empty lists.

Induction Step: Assume that the program is correct for all *mid*, xs, where the length of xs is at most k, for some $k \geq 0$, and prove that the program is correct for all xs of length $k+1$. If the length of xs is $k+1$, then

xs is nonempty; let hd and tl be the head and tail (respectively) of xs. The length of tl is k. Next, consider the case where hd is at most mid. From the induction assumption, $part(mid, tl, ls, right)$ defines ls and $right$ to be the elements of tl that are at most mid, and that exceed mid, respectively. Since, xs is hd followed by tl, and $hd \leq mid$, it follows that the sequence of elements of xs that are at most mid is hd followed by the sequence of elements of tl that are at most mid, and the sequence of elements of xs that exceed mid is the sequence of elements of tl that exceed mid. Hence, in this case, the definitions of $left$ and $right$ are correct. A similar argument applies for the case where $hd > mid$.

18.8 In Place Quicksort

Program $q1$ Program $q1$ has two input parameters, l, and r, both of which are permanents, and it has one input-output parameter C which is a one-dimensional array of numbers. Let C^{init} be the initial value of C, and let C^{final} be the value of C on termination of the program. Then C^{final} is to be a permuation of C^{init}, where $C^{final}[l, \ldots, r]$ is $C^{init}[l, \ldots, r]$ in increasing order, and the other elements of C are to remain unchanged. (If $l \geq r$ then C^{final} is C^{init}.)

$q1(l, r, C)$
int $C[\,]$;
$\{?\ \ l < r\ \ \rightarrow \{;\ split(l, r, C, mid),$
$\qquad\qquad \{\|\ \ q1(l, mid-1, C),\ q1(mid+1, r, C)\}$
$\qquad\qquad \}$
$\}$

Execution of $split(l, r, C, mid)$ permutes C and assigns a value to mid such that $l \leq mid \leq r$, and such that all elements in $C[l, \ldots, mid-1]$ are at most $C[mid]$, and all elements in $C[mid+1, \ldots, r]$ exceed $C[mid]$.

Operation of the Program If $l \geq r$, then $q1$ takes no action, leaving C unchanged. If $l < r$, then *split* is called, and after *split* terminates execution, $C[l, \ldots, mid-1]$ and $C[mid+1, \ldots, r]$ are sorted in parallel.

Reasoning About the Program First check that shared mutables in parallel composition are not modified. Array C is shared by $q1(l, mid-1, C)$ and $q1(mid+1, r, C)$, but no element of C is shared by both blocks. Hence the restriction on parallel composition is satisfied.

We reason about the program by induction on $r - l$.

Base Case: If $r - l \leq 0$, then C is left unchanged, and this is correct according to our specifications.

Induction Step: Assume that $q1(l, r, C)$ is correct for all l, r, and C for which $r - l \leq k$, for some $k \geq 0$, and prove that the program is correct for $r - l = k + 1$. If $r - l = k + 1$ then $l < r$. In this case *split* is called, and execution of $split(l, r, C, mid)$ permutes C and assigns a value to mid such that $l \leq mid \leq r$, and such that all elements in $C[l, \ldots, mid - 1]$ are at most $C[mid]$, and all elements in $C[mid + 1, \ldots, r]$ exceed $C[mid]$. Hence, $mid - 1 - l \leq k$, and $r - (mid + 1) \leq k$. From the induction assumption, $q1(l, mid - 1, C)$ and $q1(mid + 1, r, C)$ are correct, and sort $C[l, \ldots, mid - 1]$ and $C[mid + 1, \ldots, r]$. Therefore, $C[l, \ldots, r]$ is sorted correctly.

Program *split*

$split(l, r, C, mid)$
int $C[\,]$, $left$, $right$, $temp$;
{? $l \leq r$ $\rightarrow$
  {; $left := l + 1, right := r$, $s = C[l]$,
    $part1(l, r, C, s, left, right)$, $temp := l$,
    $swap(temp, right, C)$, $mid = right$
  }
}

Operation of the Program If $l > r$ then *split* terminates execution without taking any action. If $l \leq r$, then program $split(l, r, C, mid)$ calls $part1(l, r, C, s, left, right)$ after setting $left = l + 1$, $right = r$ and $s = C[l]$; program *part* leaves s unchanged, modifies $left$ and $right$, and permutes elements of $C[l + 1, \ldots, r]$ so that, at termination of $part1$, $left = right + 1$,

and all elements in $C[l+1,\ldots,right]$ are at most s, and all elements in $C[right+1,\ldots,r]$ exceed s.

After termination of *part*1, program *swap* is called to exchange $C[l]$ (which is s) with $C[right]$. After the swap, all elements in $C[l,\ldots,right-1]$ are at most s, and $C[right]=s$, and all elements in $C[right+1,\ldots,r]$ exceed s. The program terminates after *mid* is defined as *right*.

Program *part*1 Program *part*1 moves $left$ rightwards and moves *right* leftwards until they cross (i.e., $left = right+1$), so that the following invariant is maintained:

invariant:
$l+1 \le left \le r+1$ and $l \le right \le r$, and
all elements of $C[l+1,\ldots,left-1]$ are at most s,
and all elements of $C[right+1,\ldots,r]$ exceed s.

From the invariant it follows that $left \le right+1$.

```
part1(l, r, C, s, left, right)
int C[ ], left, right;
{? left ≤ right → {;
                    {|| left_rightwards(r, C, s, left),
                        right_leftwards(l + 1, C, s, right)
                    },
                    left ≤ right → {; swap(left, right, C),
                                      left := left + 1,
                                      right := right − 1
                                   },
                    part1(l, r, C, s, left, right)
                  }
}

left_rightwards(r, C, s, left)
int C[ ], left;
{? left ≤ r, C[left] ≤ s →
```

$\{;\ left := left + 1,\ left\_rightwards(r, C, s, left)\}$
}

$right\_leftwards(l, C, s, right)$
int $C[\,]$, $right$;
$\{?\ right \geq l,\ C[right] > s\ \rightarrow$
$\{;\ right := right - 1,\ right\_leftwards(l, C, s, right)\}$
}

$swap(i, j, C)$
int i, j, $C[\,]$, $temp$;
$\{;\ temp := C[i],\ C[i] := C[j],\ C[j] := temp\}$

Operation of the Program The invariant holds initially because $left = l + 1$ and $right = r$. Programs $left\_rightwards(r, C, s, left)$ and $right\_leftwards(l + 1, C, s, right)$ are executed in parallel. The only mutables that change value in these programs are $left$ and $right$, and these mutables are not shared. Therefore, the restriction on parallel composition is satisfied.

Program $left\_rightwards(r, C, s, left)$ moves $left$ rightwards from $l + 1$, i.e., it increases $left$, until $left = r + 1$ or $C[left] > s$, and it maintains the invariant. Likewise, $right\_leftwards(l + 1, C, s, right)$ moves $right$ leftwards from r, i.e., it decreases $right$, until $right = l$ or $C[right] \leq s$, and it maintains the invariant. If $left = r + 1$ or $right = l$ then $left > right$, and the program terminates execution. Consider the case where $C[right] \leq s < C[left]$, and $left \leq right$ at termination of the parallel composition block. In this case, $C[left]$ and $C[right]$ are exchanged, and then $left$ is incremented and $right$ is decremented, maintaining the invariant. Then $part1$ is calls itself.

Reasoning About the Program We reason about the program by induction on $right + 1 - left$.

Base Case: Consider the case where $right + 1 - left = 0$. In this case the program terminates. If the invariant holds, the program terminates correctly. (The program may not terminate correctly if the invariant does not hold.)

Induction Step: Assume that the program is correct, for all parameters provided $right+1-left \leq k$, for some $k \geq 0$, provided the invariant holds when the program is initiated, and prove the program correct for $right + 1 - left = k + 1$ provided the invariant holds when the program is initiated.

If $right + 1 - left = k + 1$ then $left \leq right$. In this case the parallel composition of *right_leftwards* and *left_rightwards* maintains the invariant and decreases $right + 1 - left$, or it leaves $left$ and $right$ unchanged. In the latter case, $C[right] \leq s < C[left]$, and therefore, $C[left]$ and $C[right]$ are exchanged, and then $left$ is incremented and $right$ is decremented, maintaining the invariant, and reducing $right + 1 - left$. Therefore, when *part*1 is called recursively, the invariant is maintained, and $right + 1 - left \leq k$. From the induction assumption, the recursive call to *part*1 terminates execution with the correct values for C, $left$ and $right$.

18.9 Programs with Programs as Parameters

Let m be a mutable tuple. Let t be a mutable tuple, and let the value of t be the representation for a program call, i.e., the value of t is $f(arg_0, \ldots, arg_l)$, or equivalently, the value of t is $\{"f", arg_0, \ldots, arg_l\}$, where f is the name of a program. We shall develop a program $amt(i, m, t)$ that executes $\{;\ i := m[j],\ t\}$ for each index j of m, in sequence for j going from 0 to $sizeof(m)-1$.

For example, if m is a 3-tuple, the execution of $amt(k, m, close(k))$ causes the following sequence of commands to be executed:
$k := m[0],\ close(k),\ k := m[1],\ close(k),\ k := m[2],\ close(k)$.

Consider another example: If v is a 2-tuple, the execution of $amt(j, v, g(j, b, j))$ causes the following sequence of commands to be executed:
$j := v[0],\ g(j, b, j),\ j := v[1],\ g(j, b, j)$.

The name *amt* stands for *a*ll elements of a *m*utable *t*uple. Programmers find it helpful to develop similar higher-order programs for all elements of lists, trees, and other data structures.

```
amt(i, m, t)
int j, n;
tuple i, m, t;
{;  n := sizeof(m), j := 0, fmt(i, j, n, m, t)}
```

$fmt(i, j, n, m, t)$
int j, n;
tuple i, m, t;
$\{?\ j < n \ \rightarrow\ \{;\ i := m[j],\ t,\ j := j + 1,\ fmt(i, j, n, m, t)\}$
$\}$

Program *amt* has two local integers, n and j. Initially, n is $sizeof(m)$, and j is 0. Program *amt* calls $fmt(i, j, n, m, t)$ that repeatedly executes $\{;\ i := m[j], t\}$, for increasing values of j while $j < n$.

Next, consider an application of *amt*. Let *close* be a program with a single argument i where i is a mutable tuple $\{w\}$, where w is a permanent. Program *close* defines w to be the empty list. Program *close* is:

$close(i)$
tuple i;
$\{?\ i \stackrel{?}{=} \{v\}\ \rightarrow\ v = [\,]\}$

Let m be a tuple, each element of which is a mutable tuple of the form $\{w\}$, where w is a permanent list. The call $amt(i, m, close(i))$ causes *close* to be executed with each element of m as its argument. For example, let m be $\{a, b\}$, where a and b are mutable tuples, and the values of a and b are $\{u\}$ and $\{v\}$, respectively, where u and v are permanent lists. The call $amt(i, m, close(i))$ defines both u and v to be the empty list [ ].

18.10 A Distributor

Program *distributor* has two arguments x and m, where x is a permanent list, and m is a mutable tuple. Permanent x is not modified by *distributor*. Mutable tuple m is an input-output argument of *distributor*. Each element of list x is a two-tuple $\{i, msg\}$, where i and msg are permanents, and i reduces to an integer, where $0 \leq i < sizeof(m)$. Each element $m[i]$ of m is of the form $\{w\}$ where w is a permanent list. The distributor inspects each element $\{i, msg\}$ of its input list x, in order, and places msg on w where $m[i] = \{w\}$. Operationally, the program reads an input stream of messages x, and distributes the incoming messages to one of $sizeof(m)$ output streams of messages; the output stream on which a given message is placed is specified by the index i in the incoming tuple $\{i, msg\}$.

```
distributor(x, m)
tuple m;
{? x ?= [{i, msg} | xs], m[i] ?= {w} →
        {; w = [msg | ws], m[i] := {ws}, distributor(xs, m)},
   x ?= [ ] → amt(i, m, close(i))
}
```

Operation of the Program If x is the empty list, program $amt(i, m, close(i))$ defines the output list w within tuple $m[i]$ to be the empty list, for all i. If x is nonempty, then it is of the form, $\{i, msg\}$ where $m[i]$ is of the form $\{w\}$ and w is a permanent; let the tail of x be xs. In sequence, define w to be $[msg \mid ws]$, where ws is undefined, then the mutable tuple $m[i]$ becomes $\{ws\}$, and then $distributor(xs, m)$ is called.

Reasoning About the Program We reason about the program by induction on the length of x. The reasoning is straightforward, and is left to the reader.

18.11 Mutable Linked Structures

Operations on mutable linked structures in PCN are similar to operations on linked structures in C, except that mutable tuples are used in place of pointers.

We shall develop programs to search, add elements to, and delete elements from, a mutable linear linked list. Each element of the mutable linked list is a 3-tuple $\{x, v, y\}$, where x is the *key* field, v is the *info* field and y is the *next* tuple in the list (if there is a next tuple), or the empty tuple if there are no later tuples. Variables x and v are permanents. (The program can be modified to handle mutable x and v by declaring the variables.) For convenience we use the following macro definitions $key = 0$, $info = 1$, and $next = 2$.

A mutable linked list m with two members, where the first member has key "*steve*" and info field "*anna*", and the second has key "*alain*" and info field "*mariann*" is as follows: the value of m is {"*steve*","*anna*",a}, where the value of a is {"*alain*","*mariann*",b}, where the value of b is {}.

Searching Mutable Linked Structures Program *search* has two input parameters m and x, and one output parameter z. Parameter m is mutable, and x and z are permanents. Parameter m is a mutable linear linked list. Parameter x is a key. The program sets z to be the contents of the info field of the first tuple in m containing key x, if there is such a tuple; if there is no tuple with key x in m the program sets z to $\{\}$.

In our example, the call *search*(m,"*steve*",z) will set z to "*anna*", and the call *search*(m,"*sharon*",z) will set z to $\{\}$.

$search(m, x, z)$
tuple m;
$\{?\ sizeof(m) == 0 \rightarrow z = \{\},$
$\quad sizeof(m) \neq 0, m[key] \neq x \rightarrow search(m[next], x, z),$
$\quad sizeof(m) \neq 0, m[key] == x \rightarrow z = m[info]$
$\}$

Operation of the Program

1. If m is the empty list, then define z to be $\{\}$, as required by the specification. (This is the base case in an induction on the length of m.)

2. If m is nonempty then it is a 3-tuple. If $m[key]$ is different from x then keep searching, and if $m[key] = x$, then define z as $m[info]$ and terminate execution.

We reason about the program by induction on the length of m.

Adding Elements to Mutable Linked Structures Next, we present a program to add an element with *key* field k and *info* field x immediately after tuple l in the list. Here k and x are permanents that are not modified by the program, and l is an input-output mutable tuple.

$addtolist(l, k, x)$
tuple l, w;
$\{?\ sizeof(l) == 0 \rightarrow \{;\ w := \{\}, l := \{k, x, w\}\},$
$\quad sizeof(l) \neq 0 \rightarrow \{;\ w := l[next],\ l[next] := \{k, x, w\}\}$
$\}$

Operation of the Program If l is the empty list when the program is called then l becomes the list $\{k, x, w\}$ where w is the empty list. If l is nonempty, let the tuple that follows l when the program is called, be w. We want to place a tuple with key k and info x after l and before w. Therefore, set $l[next]$ to $\{k, x, w\}$.

Deleting Elements from Mutable Linked Structures Next, we present a program to delete an element from the list. Program *delete* has three arguments: a mutable list m, a key d, and a boolean $found$. Argument m is an input-output mutable linked list, d is an input permanent, and $found$ is an output permanent. If m contains a tuple with a key d then the first such tuple is deleted from list m, and $found$ is set to *true*. If m does not contain a tuple with key d, then *delete* does not modify any variable, and $found$ is set to $false$. Program *delete* calls program *cut* which has four arguments, p, q, d and $found$, where d and $found$ are as before, and q is the tuple that follows p in list m.

The operation of program *cut* is as follows. If q is the empty list then $found$ is set to $false$; if q is nonempty and the key in tuple q is d then $found$ is set to *true* and q is removed from the list. If the key in tuple q is not d then p becomes q and then q becomes $p[next]$, thus p and q move down the list by one tuple. Program *delete* calls *cut* with $p = m$ and $q = p[next]$.

Tuple q is removed from the list by executing the assignment $p[next] := q[next]$.

```
delete(m, d, found)
tuple m, p, q;
{? sizeof(m) == 0      →      found = false,
   sizeof(m) ≠ 0       →      {; p := m,
                                 q := p[next],
                                 cut(p, q, d, found)
                              }
}

cut(p, q, d, found)
tuple p, q;
```

$$
\begin{array}{ll}
\{? \ sizeof(q) \neq 0,\ q[key] \neq d & \rightarrow \{;\ p := q,\ q := p[next],\ cut(p,q,d,found)\}, \\
\quad sizeof(q) \neq 0,\ q[key] == d & \rightarrow \{;\ p[next] := q[next],\ found = true\}, \\
\quad sizeof(q) == 0 & \rightarrow found = false \\
\}
\end{array}
$$

18.12 Distributing to Arbitrarily Many Destinations

In this section we shall describe a program which is a generalization of the distributor given earlier. In program *distributor*, a stream of messages is farmed out into n streams, where n is constant; the messages on the incoming stream are associated with integers which indicate which output stream the messages should be directed towards. Next, we develop a program *dist* in which the incoming stream is farmed out into a variable number of output streams. Each incoming message is associated with a key, and the key indicates the output stream that the incoming message should go out on. The incoming stream can have command messages that add or delete keys. (Anthropomorphically, the distributor is similar to a secretary who puts messages into pigeon-holes depending on the name of the addressee. People can change pigeon-holes, or leave, or new people can get pigeon-holes.)

Program *dist* has an input argument z which is a permanent list, and an input-output argument m which is a mutable linear linked list of tuples as in the previous program. The *info* field of a tuple is a mutable tuple t where the value of t is $\{x\}$, where x is a permanent list. (In anthropomorphic terms, x is the name of the pigeon-hole and the corresponding key is the name of the person who owns that pigeon-hole.) The elements of list z are :

1. tuples of the form $\{k, b\}$ where k is a key, and b is arbitrary, or
2. $add(k, x)$ where k is a key and x is an undefined permanent, or
3. $del(k, found)$ where k is a key and $found$ is an undefined permanent.

Program *dist* inspects the elements of the input list z in sequence. If the next element of z is a tuple of the form $\{k, b\}$, then it searches m for a tuple with key k; if it finds a tuple $\{k, t, nex\}$, for arbitrary nex and where $t = \{x\}$, then it places b on list x; if it does not find such a tuple in linked list m then it takes no action.

If the next element of z is $add(k, x)$ then it adds a tuple with key k and info t, where the value of t is $\{x\}$, to linked list m. If the next element of z is $del(k, found)$, then it searches for a tuple with key k in m, and if it finds one, the tuple is deleted and $found$ is set to *true*; if it does not find such a tuple then $found$ is set to $false$.

```
dist(z, m)
tuple w, m;
{? z ≟ [{k, b} | zs]          → {; search(m, k, w),
                                   w ≟ {x} → {; x = [b | xs], w := {xs}},
                                   dist(zs, m)
                                 },
   z ≟ [add(k, x) | zs]       → {; w := {x},
                                   addtolist(m, k, w),
                                   dist(zs, m)
                                 },
   z ≟ [del(k, found) | zs]   → {; delete(m, k, found), dist(zs, m)},
   z ≟ [ ]                    → aml(i, m, close(i))
}
```

Programs *addtolist* and *delete* given earlier were for tuples in which the info field was a permanent; these programs must be modified for the case where the info field is a mutable tuple (by declaring the tuple appropriately). Program *aml* is similar to program *amt*; the difference is that *aml* operates on a mutable linked list while *amt* operates on a mutable tuple. We leave the development of *aml* to the reader.

19 Instructions About Where to Execute Programs

For purposes of efficiency, programmers may want to specify the processors on which programs are to be executed. For this purpose the annotation '@' is added as a suffix to the program call [6]. Specifying processors does not change the semantics of programs.

The syntax of an annotated program call is:

---

program-call :: *program-call*@*location*
location :: *variable* | *integer* | *relative-location*

The description of *relative-location* is found later in this section.

A PCN program runs on a computer in which processors are numbered $0 \ldots n$, where 0 is the host machine. A program p can be executed on a processor numbered i, for any integer i where $0 < i \leq n$ by executing $p@i$ or $p@v$, where v is a variable with reduced value i.

PCN can execute on a network of processors where the topology of the network can take any form: It can be a local area network connecting workstations, or a hypercube, or a mesh, to name a few examples. Programmers find it convenient to develop their programs for a given topology, and to have their topology mapped to that of the machine on which their programs execute [6]. As in Strand [6], a virtual network of processors can be defined by including the annotation `machine(`*topology*`)` at the head of a file, where, in the current implementation, *topology* is `ring` or `torus`, where a ring is a circular list, and a torus is restricted to two dimensions. At most one virtual network can be defined for a source file.

The processor in the virtual network at which a program is to be executed can be specified by appending @*relative-location* to the program call, where for

a

ring: *relative-location* can be `fwd` for forward, `bwd` for backward, or `random`;

torus: *relative-location* can be `north`, `east`, `south`, `west`, or `random`.

The relative-location specifies a processor in the virtual network by specifying its relationship (forward, backward, north, south, east, west, or random) with respect to the processor executing the parallel composition block. For instance, execution of a parallel composition statement $\{\| \ p(left)@bwd, p(right)@fwd\}$ in a processor q in a virtual ring would cause $p(left)$ and $p(right)$ to be executed on the processor following q and preceding q, respectively, on the virtual ring.

The *relative-location* at which a program $p(\ldots)$ is to be executed can be specified at run-time by executing the statement $p\ldots)@'x$, where x is a permanent. Execution of $p(\ldots)@'x$, is as follows:

---

repeat skip until x is reducible;
execute $p@loc$ where "loc" is the reduced value of x.

For instance, if the reduced value of x is "fwd", then $p(\ldots)@'x$ is executed as $p(\ldots)@fwd$.

If a program call is executed in an address space then the called program will not be executed in another address space if any argument of the called program is a mutable. (See the section called *Architectures, Efficiency and Implementation.*) Also all blocks composed using interleaved composition execute on the same processor, except for blocks that do not reference mutables.

20 Compilation and Modules

Collections of related PCN programs are written in files, for convenience. All PCN programs can be put in a single file, but it is usually more convenient to partition programs in some logical maner, and put each set of related programs in a separate file [6]. A file containing PCN programs is called a **module**, and the name of the file is also the module name. The syntax of a module is:

module:: `-exports(`$\prec$ *program-name* $\succ$`)`
`-foreign(`$\prec$ *library-name* $\succ$`)`
$\prec$ *program* $\succ$

where *program-name* is the name of one of the programs in the module, and *library-name* is the name of a library containing C object code, and all names are quoted.

An example of a module is:

```
-exports("member", "sum")
-foreign("algebra", "diffeqns")
```
.... definition of programs including `member` and `sum` ...

A program p within a module m can be called by programs in other modules if and only if the name p appears in the `-exports(...)` statement in module m. A program in a module references a PCN program in another module by prefixing the name of the referenced program with the name of the module in which the referenced program is located, followed by the symbol ':'. For instance, a program in a module B references a program p in another module D as D:p. A program references another program in the same module without the '*module-name*:'prefix.

PCN programs can also call C (and in later releases, Fortran) programs. Consider a PCN program p that calls a C program f that appears in a library L. The module in which p appears must have library L declared as foreign, by including L in the `-foreign(`$\prec$ *library-name* $\succ$`)` statement that appears in the module.

A PCN source file can begin with macro definitions and file inclusion statements. A macro replaces a name in the the text file by a string of characters,

as in C. The syntax of a macro-definition is:
`#define variable_name replacement_string`
where replacement_string is the sequence of characters (excluding trailing blanks) on the remainder of the line. As in C, a line in a PCN source file of the form `#include "filename"` is replaced by the text of the file called "filename".

21 Acknowledgment

The PCN project is a team effort. Institutions that have participated in the effort include the Center for Research in Parallel Computation, California Institute of Technology, Argonne National Laboratories and Aerospace Corporation. Enhancements to PCN are planned with participation from Rice University and Los Alamos National Laboratories.

Scientists who have contributed to the PCN project include Sharon Brunett, Jan Lindheim, Dave Long, Dong Lin, Berna Massengill and Seth Noble at Caltech, Ian Foster and Steve Tuecke at Argonne, and Joe Bannister, A. Campbell, Ray Chowkwanyun, Mel Cutler, Melody Hancock, Carl Kesselman, Craig Lee at Aerospace Corp. Ian Foster played a major role in the development of the notation. A sizable part of the compiler and run-time system was developed at Argonne by Ian Foster and Steve Tuecke. A programming environment with a variety of support tools is being developed at Aerospace Corp. The development of PCN has been aided by applications written in PCN for weather modeling and DNA sequence matching at Argonne, computing trajectories of space vehicles and tracking objects in space at Aerospace, and Taylor-Couette flows at Caltech. We are grateful to scientists who have given of their time in helping develop PCN applications, particularly to Herb Keller at Caltech.

We are grateful to our sponsors at the National Science Foundation, the Air Force Office of Scientific Research and the Office of Naval Research for their support and advice; in particular, we wish to thank Nat Macon, Harry Hedges, Charles Holland and Andre van Tilborg.

Many ideas in PCN are derived from ideas in UNITY and STRAND.

References

[1] Aho, A.V., J. E. Hopcroft, and J. D. Ullman, Addison-Wesley, *Data Structures and Algorithms*, Reading, Massachusetts.

[2] Campbell, A., R.M. Chowkwanyun, C.F.Kesselman, C. A. Lee, and S. Taylor, 'A Database System to Support the Program Composition Environment', Aerospace Corp., Report No. TOR-0090(5920-05)-1, April 1990.

[3] Chandy, K. M., and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, Massachusetts, 1988.

[4] Chandy,K.M., and S.Taylor, 'The Composition of Concurrent Programs,' in *Proceedings Supercomputing '89*, Reno, Nevada, Nov.13-17, 1989, ACM.

[5] Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[6] Foster,I., and S.Taylor, *Strand, New Concepts in Parallel Programming*, Prentice-Hall, 1989.

[7] Foster,I., and S.Taylor, 'A Portable Run-Time System for PCN', Argonne National Laboratory Report No. ANL/MCS-TM-137, January 1990.

[8] Hoare, C.A.R., *Communication Sequential Processes*, Prentice-Hall International, London, U.K., 1984.

[9] Hoare, C.A.R., 'Quicksort,' Computer J., Vol.5, No.1, pp10-15, 1962.

[10] Seitz, C.L., and J. Seizovic, and W. Su, 'The C Programmer's Abbreviated Guide to Multicomputer Programming', Caltech Computer Science Technical Report Caltech-CS-TR-88-1, 19 January 1988, revised 17 April 1989.

[11] Shapiro,E., 'The Family of Concurrent Logic Programming Languages,' in *ACM Computing Surveys*, Vol.21, No.3, pp412-510, September 1989.