PINE*

ERIC GROSSE† AND W. M. COUGHRAN, JR.‡

Abstract. Pine is a C preprocessor for scientific computation allowing array operations and function overloading. This is a reference manual for the system.

1. Hello, world. For the canonical beginning,

```
mkdir hello; cd hello
cp /usr/num/src/pine/chk/hello/* .
mk
```

The command `pine` invoked by `mk` translates the function `top.g` to equivalent C in `top.c`. There is a default `main` program loaded that initializes floating point and calls user function `top`. There is an optional command flag `-f` for `pine` that asks for short precision (`float`), useful for looking for stability problems. To get a readable version of the function, run `pine -t top.g` and then `psi top.dvi` for screen display, or `lp top.dvi` for paper. Additional hints may be found in `/usr/ehg/mkfile`.

The syntax of source text is described in the companion manual for the *grove* tree editor. Each file describes one function and in that file each variable, function, or other symbol is the root name of a tree. The tree with root `#` is the executable code; other trees constitute the symbol table. On some computer systems, putting one function per `.c` file causes serious inefficiency in header processing and object size; run `pinecat lib/*.c > big.c` to work around this problem.

2. Types. In these trees constituting the symbol table, sons of a root node give the properties of the symbol. For example, `x( Real, Array(n), Nonneg )` where `x` is a node at the root level, declares a variable to be a real array which contains only non-negative entries.

The builtin types include `Logical`, `Int`, `Real`, `Interval`, `Complex`, and `String`. Alternatively, a variable can be declared with the name of a structure type, which is just a symbol declared `Struct(`$field1$`,`$field2$`,`...`)`, where fieldname is itself declared like an ordinary variable. Fields are accessed by expressions of the form $field1(str)$. An integer or real can be automatically read from the corresponding shell environment variable by declaring the variable to have the property `Environment,` with a child supplying the default value.

On top of all this are rectangular arrays, declared `Array` $(l_1\!:\!u_1,\ldots,l_p\!:\!u_p)$. p and l_i are required to be compile-time constants. For non-argument arrays, space is automatically allocated upon function entry and freed upon exit. For argument arrays, u_i must be an integer constant or expression; in the latter case, the variable is assigned the incoming bound. The compiler compensates for differences in l_i between caller and callee. For example, the user can declare a grid as `0:2*k` and call a library function which declares it as `1:n`; at function entry, n is assigned $2k+1$. (This is implemented by collecting equations for such bounds and symbolically solving.) The function `New(`a`,`len_1`,...)` frees the space on the heap formerly taken by the array

---

* Typeset on February 3, 1992. **DRAFT.** *To retain the freedom to make incompatible changes in the language in light of experience, we do not seek outside users at present. Distribution of this manual is not a promise of a future AT&T software product, but rather is a reference manual for our own use and a cheap attempt to convey some of the lessons learned in our experiment.*

† AT&T Bell Laboratories, Murray Hill NJ 07974. `research!ehg` or `ehg@research.att.com`.

‡ AT&T Bell Laboratories, Murray Hill NJ 07974. `research!wmc` or `wmc@research.att.com`.

a and allocates new space as indicated by the new lengths. `Renew` is like `New`, but contents of the old array are copied, up to the minimum of the old and new u_i. When a function returns an array result, the compiler allocates in the caller an array according to the declaration of the result variable in the function declaration and frees the array when done using it. Often the result array is declared as empty by the caller and the callee uses `New` to reallocate the array to some size that it computes.

Arrays are subscripted as usual, `a[i,j]`. The use of square brackets instead of parentheses here is intended to improve human readability; they are regarded as equivalent by pine. The notation `a[l:h]` means slice a subarray from l to h. `Array(x1,x2,...)` constructs a (1-dimensional) array, with type determined by that of `x1`. Scalar operations generalize componentwise to arrays. For example, if `A` and `B` are arrays of the same size, then `C := A * B` is another array of the same size, and $c_{i,j} := a_{i,j} b_{i,j}$. Multiplication in the sense of linear algebra is a different operator, denoted by a period: `y := A.x` means $y_i := \sum_{j=1}^{n} a_{i,j} x_j$. All the arrays used componentwise in an expression should have the same number of dimensions and the same length in each dimension, but may have different origins. (That is, their p's and $u_i - l_i$'s must be equal, but their l_i's may differ.) The result should be independent of the order of componentwise expansion.

Subscripting a `String`, as in `s[i]`, indexes a particular character; adding an `Int`, as in `s+i`, refers to a substring.

3. Functions. Function names are declared with a property `Args` listing the formal argument names, and a property `Result` with a variable describing the value returned. (`Result` may be omitted if the function returns `Void`.) In particular, the function being compiled is so declared, and the value of the function is the value of the `Result` variable upon exit. The name of the function being compiled is derived from the filename by stripping off the `.g` suffix. Variables which may be assigned function names are declared `FunctionVar`; function literals, the usual case, are declared `Function`.

Formal arguments are declared `In, Out,` or both. A formal array argument that might be reallocated should be declared with the property `Movable`. To guard against surprises from an unfamiliar library function, the compiler's type checker requires that the actual argument must also be declared `Movable`. Formal arguments can be explicitly declared `ByRef`, indicating call by reference or value-result. By default, input-only arguments are passed by value, output and `Movable` arguments by reference.

Array arguments are by default assumed not to have been sliced or transposed, and therefore to have contiguous elements; to remove this assumption, add the property `Slice` to the argument declaration.

To inform the compiler (and library user) about allowed aliasing, a property `Overlap` may be added to a function declaration to specify classes of arguments and global variables that may overlap at function call or during its execution. In the example `Overlap[group1(a,b),resid(x,y)]` arguments `a` and `b` may overlap, `x` and `y` may overlap, but `a` and `y` may not overlap. The names `group1` and `resid` are comments.

Functions that (call something that) change the floating point modes should be declared `ChangesFloatModes`.

Overloaded functions are declared by the property `Overload`, followed by the "property" list `f1,f2,...` of specific functions that are candidates for substitution. The first function f_i whose declaration matches is the one selected. Each property of a formal argument must be a property of the corresponding actual argument. If a scalar argument is expected and an array is found, the function will be applied componentwise. All such componentwise arguments must be of the same dimension and lengths. To require that a property `p` not appear in the actual argument, use `~(p)`.

The rule just given would not distinguish between `Array(m)` and `Array(m,n)`. To remedy this, array variables are automatically given the property p`d`, where p is the number of subscripts; thus vectors get the property `1d` and matrices `2d` and hence function declarations can match the correct sorts of objects. The properties `Array(m)` and `Array(n)` are not regarded as different types. An `Array` variable may be declared with an `Args` property when more than simple `Int` type checking is desired.

4. Control structures. There are three basic control structures: `If`, `For`, and `Catch/Throw`. All return `Void`. Use of `Catch/Throw` is discouraged because it tends to make program transformation difficult.

A conditional has the form

```
If
  bool_1
  expr_1
  ...
  bool_n
  expr_n
```

First, $bool_1$ is evaluated. If true, then $expr_1$ is evaluated; otherwise, $bool_2$ is evaluated, and so on. When convenient, write $bool_i$ in the form $intvar == const$; this generates a C switch statement.

Comments are names beginning with `#` and may act as statement brackets delimiting a block of code, as in

```
If
  y>x
  # swap
    t:=x
    x:=y
    y:=t
```

This is like the comma operator of C, but not restricted to simple expressions. To "comment out" code, there is a macro `Ignore` that discards its arguments.

The functions `&` and | evaluate their first argument and immediately return if the resulting value is sufficient to determine the final result; otherwise, they evaluate their second argument, then their third argument, and so on.

The loop construct has the form

```
For
  bool
  expr_1
  expr_2
  ...
```

where *bool* must be either 1) a general `Logical` expression, corresponding to a `while` in other languages, or else 2) a range expression of the form $l \leq i \leq u$, corresponding to a do-loop. The latter implies that the index i is to incremented implicitly and will not be changed inside the loop. If $\geq$ is used instead, the index is decremented. The $expr_q$ expressions constitute the body of the loop.

An exit construct is provided by `Catch` and `Throw`. To evaluate `Catch(`f`,`g`)`, the main expression f begins evaluation as usual. If `Throw` is executed somewhere in f, the "error" expression g is immediately executed; otherwise, g is skipped. If a `Throw` is encountered in g then control is passed to the next higher `Catch` in the surrounding text.

An integer variable `execnt` is automatically provided to count the number of times the function has been called. This allows for special initialization of variables on first execution and for more selective debug printing.

There is a default `main` program arranges for `awk()` (from the `gen` library) to read command line arguments as input files, initializes floating point status and then calls user function `top`. (See the source of `ieee0()` for details of floating point behavior.) The function `Error()` prints its arguments and aborts execution.

5. Program transformation aids. A key reason for pine's implementation as a grove processor rather than as an extension to C is to allow convenient source-to-source transformations.

These rewrite rules can be specified like
`#macros.0(++(++(%?,%ma),+=(%ma,1)),...)`
and
`#Type macros(#splines(Spline,Array(n)),...)`,
which are applied respectively to the code tree rooted at `#` and to the entire file, respectively. Because of the way header files are loaded in grove, the first level children, which are comments, must be unique. Add the node `brkpnt` as the younger brother of the replacement tree to pause after the rule has been globally applied.

Miscellaneous tools: `Apply(f,x,y,...)` means `f(x), f(y), ...`; `String(x)` means `"x"`; `FunctionColon` means `"f:"`, where `f` is the function being compiled; `UniqueInt` means a (compile time constant) integer, guaranteed different from previous `UniqueInt`'s in this function.

`#macros.6` are applied after overloaded functions have been identified and temporaries created for functions that return arrays. At this point subscripting, slicing, function call, and field selection have all been converted to their explicit forms. Also, the first child of each nonterminal is a variable giving the result type; if necessary, this is done by inserting a dummy variable.

From one point of view, pine is just a fancy macro processor. Traditionally, macros like `square(x) => x*x` would fail if x has side effects. (No parenthesization tricks are necessary, because the substitutions are done at a structural level.) Side effects are one of the important disadvantages of macros compared to functions. To overcome this, pine recognizes expressions of the form `Let(t,x)` in the substitution part of macros as meaning that t should be replaced in the rule by either a temporary containing the value x or if possible by x itself. The type of t is inherited from x. Macros have the advantage over functions of being inherently polymorphic, and both the values and the text of the arguments are available. For example, an output rule for interval expressions might be:

```
Putlab(%?,%mx) =>
  Let(x,%mx)
  Put(String(%mx),"=[",left(x),",",right(x),"]").
```

An expression `NewTemp(t,x)` explicitly creates a temporary of the same type as `x`, which is sometimes useful in more elaborate source transformations. Still, macros can be tricky; `swap(i,J(i))` isn't a simple macro application.

As mentioned in section 2, it is customary to gather common declarations in an include file. Thus library functions `lib1` and `lib2` might have argument `x`, all declared in file `lib.g`. This is convenient for the package implementer to read and type, but would lead to name conflicts if general user code that wishes to call `lib1` included `lib.g`. To work around this problem, the command

```
pine_hide lib.g > safe.g
```

adds the prefix `lib_` to any symbol flagged with the property `Hide`. A makefile can keep the internal and external include files automatically in sync. This is a bit ponderous, but in practice works more smoothly than depending on the programmer to keep function declarations consistent in source and header files. As a further effort to avoid name clashes, we are experimenting with extending `Include` to allow explicit listing of which symbols are to be imported, with other symbols being hidden by default.

6. Extras. The assignment operator `Identify(a,b)`, typeset as $a \equiv b$, copies the dope vector, not the contents of the array. The operation `Diag(a,i)` for `i=0` gives the prime diagonal, `i=1` gives superdiagonal, and so on. `Rotate(a)` updates the dope vector for `a` so that subscript 2 becomes subscript 1, ..., 1 becomes p without actually moving all the elements to form the transpose. `Flatten(a)` converts a multiply indexed array to a singly indexed array for the same elements. Flattening is not possible in general on an array that has been sliced or transposed; an error message is printed in such a case. `ColumnMatrix(a)` makes an n-vector into an n-by-1 matrix.

An empty array is one with a constant $u_i < l_i$. Non-empty, compile-time constant size arrays are allocated on the stack and hence may not have `New` applied. If `String` has an integer constant as child, a C char array is allocated rather than a heap pointer.

This rest of this section describes features that have been added because they seem essential to call existing C libraries. They are implemented, but interactions with the rest of the language have not necessarily been thoroughly examined yet.

In addition to the mandatory declaration of base type, a variable may be declared as a `Pointer`. `New`(*pointer*) allocates space for the type of object pointed to, usually a structure. `Deref` is the function name for dereferencing a pointer; `Loc` returns the address of a variable. Applying a field selector to a pointer to a structure is interpreted as an implicit dereferencing. `NullPointer` is the null value, which may be assigned and compared, but not dereferenced. If a pointer argument is expected and an array is found, the address of the first array element is passed. Pointers are listed as an extra, because we are exploring here the use of arrays and subscripts in place of pointers. Compared with the hardware environment when BCPL and C were designed, registers and adds are less precious and aliasing is more of a concern.

The type `Char` and modifiers `Register`, `Static`, `Short`, `Long` and `Unsigned` are also recognized as in C. `Real` is 64-bit floating point. `Extern` means global scope; to identify the unique function in which such a variable is allocated storage, add `Definition`. (Children of `Definition` are emitted as C initializers.) To add other types, update `#TypeModifiers` or `#BasicTypes`. System include files can be referenced via `#include`. C ensures that `Extern` and `Static` variables are initialized to 0.

7. Libraries. Our objective has been to make a core language as small as possible and put most of the functionality in libraries extending the language in various independent directions. Each is implemented by a combination of macro include files and compiled functions.

Since we contemplate thousands of functions, documentation is a serious concern. Paper is still expedient for some of the more mathematical details, but to promote easy searching and programmer convenience we wish to avoid the common spectacle of a shelf filled with library manuals. Instead, we put the libraries in `/netlib/pine` and adopt the scheme of online files per library

`index` one paragraph per file of attribute value pairs;

`changes` a log of modifications;

`dependencies` call graph and other file dependencies;

`include/fun.g` specifies calling sequences;

`intro.tex` gives a general overview of the library, describes the mathematical background of the problem, a discussion of pitfalls, and so on.

This is augmented by example code and regressions tests.