A HOPE TUTORIAL

BY ROGER BAILEY

Using one of the new generation of functional languages

Editor's note: In this article we have boldfaced the output of the Hope interpreter to distinguish it from the input. The Hope interpreter is available for downloading from BYTEnet Listings *at* (617) 861-9774. *This version runs under* PC-DOS 2.0; *you will need the files* HOPE.EXE *and* SYS.HOP. *Related articles are the* BYTE U.K. *column on page* 341 *of this issue and* BYTE U.K. *on page* 385 *of the May issue.*

IN A LANGUAGE LIKE PASCAL, a function is a piece of "canned" program for doing standard operations like finding square roots. When we want the square root of a positive number stored in a variable x, we write sqrt(x) at the point in the program where we want the value, such as writeln(1.0 + sqrt(x)). This is called an application of the function. The value represented by x is called the argument or actual parameter. In this context the function sqrt computes the square root of x, 1.0 is added to it, and the result is then printed.

We can also define our own functions specifying how the result is computed using ordinary Pascal statements. Here's a function that returns the greater of its two argument values:

```
function max(x,y:INTEGER):INTEGER;
begin
  if x>y
  then max := x
  else max := y
end;
```

The identifiers x and y are called formal parameters. They're used inside the definition to name the two values that will be supplied as arguments when the function is applied. Here's how we might use max to filter out negative values: writeln(max(z,0));.

A more interesting case is when the actual parameter is a function application itself or involves one. We can use max to find the largest of three numbers by writing max(a,max(b,c)).

Combining functions like this is called composition. The expression is evaluated "inside out" because the outer application of max can't be evaluated until the value of its second argument is known. The inner application of max is therefore evaluated first using the values of b and c; the result is used as the actual parameter in the outer application.

Another way to combine functions is to define more powerful ones by using simpler ones as building blocks. If we often need to find the largest of three numbers, we might define

```
function MaxOf3(x,y,z:INTEGER):INTEGER;
begin
  MaxOf3 := max(x,max(y,z))
end;
```

and apply it by writing MaxOf3(a,b,c).

PROGRAMMING WITH FUNCTIONS

Pascal is called an imperative language because programs written in it are recipes for doing something. If our programs consist only of functions, we can concentrate on the results and ignore how they're computed. Forget that

(continued)

Roger Bailey can be reached at the Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, England.

sqrt is a piece of code and think of sqrt(x) as a way of writing a value in your program, and you'll get the idea. You can think of MaxOf3 in the same way if you ignore the way it works inside. By defining a toolkit of useful functions and combining them, we can build powerful programs that are short and easy to understand.

In Pascal, functions can return only simple data objects such as numbers or characters, but real programs use big data structures and can't easily be written using these functions. In Hope, functions can return any type of value, including data structures equivalent to Pascal's arrays, records, and much more. Programming in Hope has the flavor of simply writing down the answer by writing an expression that defines it. This expression will contain one or more function applications to define smaller parts of the answer. These won't usually be built in like sqrt, so we'll have to define them ourselves, but we'll still think of them as definitions of data objects, not as algorithms for computing them.

A Simple Hope Example—Conditionals

Let's see how we can define max in Hope. Like Pascal, Hope is a strongly typed language; we must tell the compiler about the types of objects in our programs so it can check that they're used consistently. The function definition comes in two parts—the declaration followed by one or more recursion equations. First we declare the argument and result types:

```
dec max : num # num -> num;
```

dec is a reserved word (must be in lowercase) signaling the start of the declaration of the function max, which takes two numbers as arguments and returns a single number as its result. Read the symbol : as "takes a." Names consist of uppercase and lowercase letters (which are distinct) and digits and must start with a letter. The current fashion is to use lowercase. You can separate symbols with any number of blanks and new lines for clarity. A space or new line is needed only when adjacent symbols might be confused as one symbol without it.

The next part of the declaration gives the types of the arguments. Integers are of the predefined type num (in lowercase). Read # as "and a"; alternatively, you can use the reserved word X. Read -> as "yields." The semicolon marks the end of the declaration. max needs only one recursion equation to define it.

```
--- max(x,y) <= if x>y then x else y;
```

Read the symbol --- as "the value of." The expression max(x,y) is called the left-hand side of the equation. It defines x and y as formal parameters or local names for the values that will be supplied when the function is applied. Parameter names are local to the equation, so x and y won't be confused with any other x or y in the program. The symbol <= is read as "is defined as."

The rest of the equation (called the right-hand side) defines the result. It's a conditional expression. The symbols if, then, and else are reserved words. If the value of the subexpression x > y is true, the value of the whole conditional expression is the value of x; otherwise, it's the value of y. The two alternative values can be defined by any Hope expression.

While Pascal's conditional statement causes one of two actions to be performed, Hope's conditional expression specifies one of two values. Hope doesn't specify the order in which the two expressions are evaluated. On a computer that uses parallel processing, such as the Imperial College ALICE machine, it's even possible to evaluate all three expressions in parallel and throw away one of the results when the value of the condition is known.

Using Functions We've Defined

A Hope program consists of a single expression containing one or more function applications. When the expression is evaluated, the result and its type are printed on the screen. Here's a simple program that uses max:

```
max(10,20) + max(1,max(2,3));
23 : num
```

The rules for evaluating the expression are the same as those in Pascal. Function arguments are evaluated first, the functions are applied, and finally other operations are performed in the usual order of priority.

We can also use existing functions to define new ones. Here's the Hope version of MaxOf3:

```
dec MaxOf3 : num # num # num -> num;
--- MaxOf3(x,y,z) <= max(x,max(y,z));
```

A More Interesting Example

Just as Pascal's conditional statement is replaced by Hope's conditional value, so the repetitive statement is replaced by the repetitive value. Here's a Pascal function that multiplies two numbers using repeated addition:

```
function mult(x,y:INTEGER):INTEGER;
  var prod:INTEGER;
  begin
    prod := 0;
    while y>0 do
      begin
        prod := prod + x;
        y := y-1
      end;
    mult := prod
  end;
```

It's hard to be sure this function does enough additions (it took me three tries to get it right), and this seems to be a general problem with loops in programs. A common way of checking imperative programs is to simulate their execution. If we do this for input values of 2 and 3, we'll

(continued)

find that prod starts with the value 0 and gets values of 2, 4, and 6 on successive loop iterations, which suggests that the definition is correct.

Hope doesn't have any loop structures, so we must write all the additions that the Pascal program performed in a single expression. It's much easier to see that this has the right number of additions:

```
dec mult : num # num -> num ;
--- mult(x,y) <= 0 + x + x + . . .
```

or would be if we knew how many times to write +x. The hand simulation suggests we need to write it y times, which is tricky when we don't know the value of y. What we do know is that for a given value of y, the expressions mult(x,y) and mult(x,y – 1) + x would have the same number of +x terms if written out in full. The second one always has two terms, whatever the value of y, so we'll use it as the definition of mult:

```
--- mult(x,y) <= mult(x,y – 1) + x;
```

On the face of it we've written something ridiculous, because it means we must apply mult to find the value of mult. Remember, however, that this is really shorthand for 0 followed by y occurrences of +x. When y is zero, the result of mult is also zero because there are no +x terms. In this case, mult isn't defined in terms of itself, so if we add a special test for it, the definition terminates. A usable definition of mult is

```
--- mult(x,y) <= if y=0 then 0 else mult(x,y – 1) + x;
```

Functions that are defined using themselves like this are called recursive. Every Pascal program using a loop can be expressed as a recursive function in Hope. All recursive definitions need one case (called the base case) where the function isn't defined in terms of itself, just as Pascal loops need a terminating condition.

Another Way of Using Functions

Hope enables us to use a function with two arguments as an infix operator. We must assign it a priority and use it as an infix operator everywhere, including the equations that define it. The definition of mult used as an infix operator looks like this:

```
infix mult : 8;
dec mult : num # num -> num;
--- x mult y <= if y=0 then 0 else x mult(y – 1) + x;
```

A bigger number in the infix declaration means a higher priority. Since a priority of 8 is higher than that of the subtraction operator, mult's second argument, y – 1, in the recursion equation must be in parentheses. Most of Hope's standard functions are supplied as infix operators.

Other Kinds of Data

Hope provides two other primitive data types. A truval is equivalent to Pascal's Boolean data type and has values

(continued)

true or false. We've already seen an example of an expression that defines a truval: x>y. > is a standard function whose type is num # num -> truval. We can use truvals in conditional expressions and combine them with the standard functions and, or, and not.

Single characters are of type char, with values 'a', 'b', and so on. Characters are most useful as components of data structures such as character strings.

Data Structures

Practical programs need data structures, and Hope has two standard kinds already built in. The simplest kind, called a tuple, corresponds to a Pascal record. We can bind a fixed number of objects of any type together into a tuple. For example: (2,3) and ('a',true) are tuples of the type num # num and char # truval, respectively.

We use tuples when we want a function to define more than one value. Here's one that takes the time of day defined in terms of seconds since midnight and converts it to hours, minutes, and seconds:

```
dec time24 : num -> num # num # num;
--- time24(s) <= (s div 3600,
                  s mod 3600 div 60,
                  s mod 3600 mod 60);
```

div is the built-in integer division function, and mod gives the remainder after integer division. If we type an application of time24 at the terminal, the resulting tuple and its type will be printed on the screen in the usual way.

```
time24(45756);
(12,42,36) : (num # num # num)
```

The second standard data type, called a list, corresponds roughly to a one-dimensional array in Pascal. It can contain any number of objects (including none at all), but they must all be the same type. For example, [1,2,3] is an expression of type list(num).

There are two standard functions for defining lists. The infix operator :: (pronounced "cons") defines a list in terms of a single object and a list containing the same type of object:

10 :: [20,30,40] defines the list [10,20,30,40]

Don't think of :: as adding 10 to the front of [20,30,40]. It really defines a new list, [10,20,30,40], in terms of two other objects without changing their meaning, rather in the same way that 1 + 3 defines a new value of 4 without changing the meaning of 1 or 3.

The other standard list function is nil, which defines a list with no elements in it. We can represent every list by an expression consisting of applications of :: and nil. When we write an expression like

['c','a','t']

Hope considers it to be just a shorthand way of writing

(continued)

```
'c' :: ('a' :: ('t' :: nil))
```

Another shorthand way of writing lists of characters is to enclose the character string in double quotes: "cat". When the result of a Hope program is a list of numbers, it's printed out in the concise bracketed notation; if it's a list of characters, it's printed in quotes.

Every data type in Hope is defined by a set of primitive functions like :: and nil. They're called constructor functions and aren't defined by recursion equations. When we defined a tuple, we were actually using a standard constructor called , (pronounced "comma"). Later on we'll see how constructors are defined for other types of data.

Functions That Define Lists

If we wanted to write a Pascal program to print the first n natural numbers in descending order, we'd probably write a loop that printed one value out on each iteration. For example,

```
for i:=n downto 1 do write(i);
```

In Hope we write one expression that defines all the values at once, rather like we did for mult:

```
dec nats : num -> list(num);
--- nats(n) <= if n=0 then nil else n::nats(n-1);
```

nil is useful for writing the base case of a recursive function that defines a list. If we try the function at the terminal by typing

```
nats(10);
[10,9,8,7,6,5,4,3,2,1] : list(num)
```

the numbers are in descending order because that's the way we arranged them in the list, not because they were defined in that order. The values in the expression defining the list are treated as though they were all generated at the same time. On the ALICE machine they actually are generated at the same time. To get the results of a Hope program in the right order, we must put them in the right place in the final data structure.

If we want the list of the natural numbers n through 1 in ascending order, we need to use another built-in operation, <> (pronounced "append"), that concatenates two lists.

```
--- nats(n) <= if n=0 then nil else nats(n-1) <> [n];
```

We put n in brackets to make it into a (single-item) list because <> expects both its arguments to be lists. We could also have written (n::nil) instead of [n].

Data Structures as Parameters

Suppose we have a list of integers and we want to write a function to add up all its elements. The declaration will look like this:

```
dec sumlist : list(num) -> num;
```

We need to refer to the individual elements of the actual parameter in the equations defining sumlist. We do this using an equation whose left-hand side looks like this:

```
--- sumlist(x :: y) . . .
```

This is an expression involving list constructors and corresponds to an actual parameter that is a list. x and y are formal parameters, but they name individual parts of the actual parameter value. In an application of sumlist like

```
sumlist([1,2,3])
```

the actual parameter will be "dismantled" so that x names the value 1 and y names the value [2, 3]. The complete equation will be

```
--- sumlist(x :: y) <= x + sumlist(y);
```

Notice there's no base case test. As we might expect, it's the empty list, but we can't test for it directly in the equation because there's no formal parameter that refers to the whole list. In fact, if we write the application

```
sumlist(nil)
```

we'll get an error message because we can't dismantle nil to find the values of x and y. We must cover this case separately using a second recursion equation:

```
--- sumlist(nil) <= 0;
```

The two equations can be given in either order. When sumlist is applied, the actual parameter is examined to see which constructor function was used to define it. If the actual parameter is a nonempty list, the first equation is used, because nonempty lists are defined using the :: constructor. The first number in the list gets named x and the remaining list y. If the actual parameter is the empty list, the second equation is used because empty lists are defined using the constructor nil.

Pattern Matching

An expression composed of constructors appearing on the left-hand side of a recursion equation is called a pattern. Selecting the right recursion equation and dismantling the actual parameter to name its parts is called pattern matching. When you write a function, you must give a recursion equation for each possible constructor defining the argument type.

Sometimes we don't need to dismantle the actual parameter, and we can use a formal parameter in the pattern that matches the whole object, irrespective of what constructors were used to define it. As an example, let's see how we could define our own version of the append function to concatenate two lists. Let's call it cat.

```
infix cat : 4;
dec cat : list(num) # list(num) -> list(num);
--- (h :: t) cat r <= h :: (t cat r);
--- nil cat r <= r;
```

The first list parameter is matched by the pattern (h::t)

(continued)

so that its first item (the "head") and the remaining list (the "tail") can be referred to separately on the right-hand side. The second recursion equation covers the case when the first list is empty. The second list parameter is matched by the pattern r whether it's empty or not.

In addition to writing enough recursion equations to satisfy all the parameter constructors, we must also be careful not to write sets of equations in which more than one pattern might match the actual parameters, because that would be ambiguous.

We can write patterns to match arguments that are tuples in the same way. When we wrote mult(x,y) you probably thought the parentheses and the comma had something to do with the function application. In fact, we were constructing a tuple, and the parentheses were needed only because the tuple constructor (,) has a low priority. Hope treats all functions as having only one argument. This can be a tuple when you want the effect of several arguments. Without the parentheses

```
mult x, y
```

would be interpreted as

```
( mult ( x ), y )
```

A recursion equation with the left-hand side

```
--- mult ( x, y ) <= ...
```

is just a pattern match on a tuple. The first item in the tuple gets named x and the second one y.

We can even use pattern matching on num parameters. These are defined by two constructors called succ and 0. succ defines a number in terms of the next lower one. 0 has no arguments and defines the value zero. Surely 0 is a value, not a function? Well, we're already used to thinking of function applications as another way of writing values, so it's quite consistent to think of 0 as a function application. Here's a version of mult that uses pattern matching to identify the base case:

```
infix mult : 8;
dec mult : num # num -> num;
--- x mult 0       <= 0;
--- x mult succ(y) <= (x mult y)+x;
```

We can read succ(y) as "the successor of some number that we'll call y." Instead of naming the actual parameter y as we did in the original version of mult, we're naming its predecessor.

Simplifying Expressions

In Pascal programs we can simplify complex expressions by removing common subexpressions and evaluating them separately. Instead of writeln((x+y)*(x+y));, we would probably write z := x+y; writeln(z*z);, which is clearer and more efficient. Hope programs consist only of expressions, and it's even more important to simplify them. We do this by using a qualified expression:

(continued)

```
let z == x+y in z*z;
```

This looks like an assignment, but it isn't. == is read as "is defined as," and z is local to the expression following the in. If we write something like

```
let z == z+1 in z*z;
```

we're actually introducing a new variable, z, to use in the subexpression z*z. It hides the original one in the subexpression z+1.

There's a second form of qualified expression for people who like to use variables first and define their meanings later. It looks like this:

```
z*z where z == x+y;
```

The result of the qualified expression is the same whether we define it using let or where. x+y is evaluated first, and its value is used in the main expression.

The qualifying expression will often be a function application that defines a data structure. If we want to name part of the structure, we can use a pattern on the left-hand side of the == symbol.

```
dec time12 : num -> num # num;
--- time12(s) <= (if h>12 then h-12 else h,m) where
                  (h,m,s) == time24(s);
```

We'll use this construction most often when we write recursive functions that define tuples. Suppose, for example, that we want to form a string of words from a sentence. For simplicity, a word is taken to be any sequence of characters, and words are separated in the sentence by any number of blanks. The sentence and a single word will be list (char) objects and the final sequence of words a list(list(char)).

It's fairly straightforward to obtain the first word. Here's a function that does it:

```
dec firsttry : list(char) -> list(char);
--- firsttry(nil)     <= nil;
--- firsttry(c :: s)  <= if c=' '
                         then nil
                         else c :: firsttry(s);
```

One of the nice features of Hope is that we can type in and print out any kind of value, so it's easy to check out the individual functions of our program separately. If we test firsttry we'll see

```
firsttry ( "You may hunt it with forks and Hope" ) ;
"You" : list ( char )
```

But there's a problem here, because we're going to need the rest of the sentence if we're to find the remaining words. We must arrange for the function to return the remaining list as well as the first word. This is where tuples come in:

```
dec firstword : list(char) -> list(char) # list(char);
--- firstword(nil)     <= (nil, nil);
```

(continued)

```
--- firstword(c :: s)  <= if c = ''
                          then (nil,s)
                          else ((c :: w, r) where
                              (w,r) == firstword(s));
```

The qualified expression is in parentheses so it only applies to the expression after else; otherwise we would evaluate firstword recursively as long as the sentence is nonempty, even if it starts with a blank. This version of the function produces

```
firstword("Hope springs eternal . . .");
("Hope","springs eternal . . .") : (list(char) # list(char))
```

We can use this to define a function to split the sentence into a list of its individual words:

```
dec wordlist : list(char) -> list(list(char));
--- wordlist(nil)      <= nil;
--- wordlist(c :: s)   <= if c = ''
                          then wordlist(s)
                          else (w :: wordlist(r) where
                           (w,r) == firstword (c :: s));
```

which we can test by typing an application at the terminal:

```
wordlist(" While there's life there's Hope ");
["While","there's","life","there's","Hope"] :
list(list(char))
```

So far we've concentrated on features of Hope that have something in common with traditional languages such as Pascal, but without many of their limitations, such as fixed-size data structures. We've also been introduced to the functional style of programming in which programs are no longer recipes for action but definitions of data objects.

Now we'll introduce features of Hope that lift it onto a much higher level of expressive power and enable us to write programs that not only are extremely powerful and concise but that can be checked for correctness at compile time and mechanically transformed into more efficient versions.

More Powerful Functions

The Hope compiler can spot many common kinds of errors by checking the types of all objects in expressions. This is harder than checking at run time, but it is more efficient and saves the embarrassment of discovering an error at run time in a rarely executed branch of the air traffic control system we just wrote.

However, strict type checking can be a nuisance if we want to perform some operation that doesn't depend on the type of the data. Try writing a Pascal procedure to reverse an array of either 10 integers or 10 characters, and you'll see what I mean.

Hope avoids this kind of restriction by allowing a function to operate on more than one type of object. We've already used the standard constructors :: and nil to define a list(num), a list(char), and a list(list(char)). The standard equality function = compares any two objects of the same type. Functions with this property are called polymorphic. Pascal's built-in functions abs and sqr and operators like > and = are polymorphic in a primitive kind of way.

We can define our own polymorphic functions in Hope. The function cat we defined earlier concatenates lists of numbers, but we can use it for lists containing any type of object. We do this by first declaring a kind of "universal type" called a type variable. We use this in the declaration of cat where it stands for any actual type.

```
typevar alpha;
infix cat : 8;
dec cat : list (alpha) # list(alpha) -> list(alpha);
```

This says cat has two parameters that are lists and defines a list, but it doesn't say what kind of object is in the list. However, alpha always stands for the same type throughout a given declaration, so all the lists must contain the same type of object. The expressions [1,2,3] cat [4,5,6] and "123" cat "456" are valid, while the expression [1, 2, 3] cat "456" is not. The interpretation of a type variable is local to a declaration so it can have different interpretations in other declarations without confusion. [*Editor's note: In the version of* Hope *available on* BYTEnet, *the type variables* alpha *and* beta *are predefined.*]

Of course, it only makes sense for a function to be polymorphic as long as the equations defining it don't make any assumptions about types. In the case of cat, it's defined using only :: and nil, which are polymorphic themselves. However, a function like sumlist uses + and can only be used with lists of numbers as parameters.

Defining Your Own Data Types

Tuples and lists are quite powerful, but for more sophisticated applications we'll need to define our own types. User-defined types make programs clearer and help the type checker to help the programmer. We introduce a new data type in a *data declaration*.

```
data vague == yes ++ no ++ maybe;
```

data is a reserved word and vague is the name of the new type. == is pronounced "is defined as" and ++ is pronounced "or." yes, no and maybe are the names for the constructor functions of the new type. We can now write function definitions that use these constructors in pattern matches:

```
dec evade : vague     -> vague ;
--- evade ( yes )     <= maybe ;
--- evade ( maybe )   <= no ;
```

The constructors can be parameterized with any type of object, including the type that's being defined. We can define types like lists, whose objects are of unlimited size using this kind of recursive definition. Here's a user-defined binary tree that can contain numbers as its leaves:

```
data tree == empty ++ tip(num) ++ node(tree # tree);
```

(continued)

There are three constructors: empty has no parameters and defines a tree with nothing in it, tip defines a tree in terms of a single number, and node defines a tree in terms of two other trees. Figure 1 shows a typical tree.

Here's an example of a function that manipulates trees. It returns the sum of all the numbers in the tree:

```
dec sumtree : tree -> num;
--- sumtree (empty)    <= 0;
--- sumtree (tip(n))   <= n;
--- sumtree (node(l,r)) <= sumtree(l) + sumtree(r);
```

(continued)

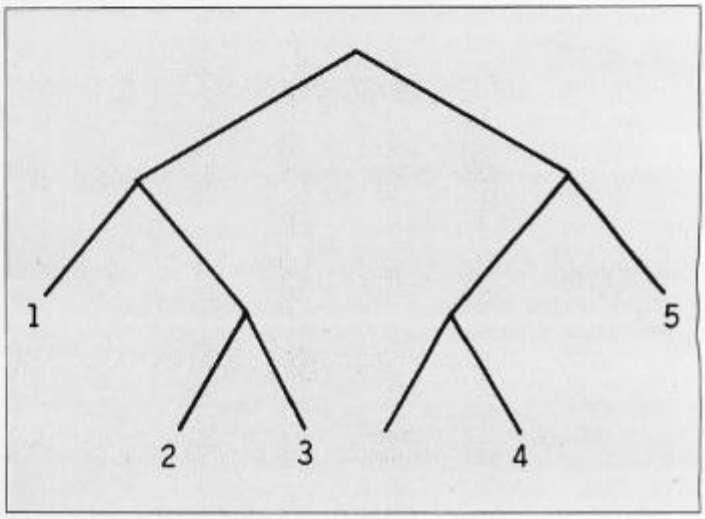

Figure 1: A *typical binary tree.*

Listing 1a: *The polymorphic function* flatten *can operate on trees of any type object.*

```
dec flatten : tree(alpha) -> list(alpha);
--- flatten(empty)      <= nil;
--- flatten(tip(x))     <= x :: nil;
--- flatten(node(x,y))  <= flatten(x) <> flatten(y);
```

Listing 1b: *These examples demonstrate the function* flatten *on various types of trees.*

```
flatten(node(tip(1),node(tip(2),tip(3))));
[ 1, 2, 3 ] : list

flatten(node(tip("one"),
             node(tip("two"),
                  tip("three"))));
["one","two","three"] : list(list(char))

flatten(node(tip(tip('a')),
             node(tip(empty),
                  tip(node(tip('c'),
                           empty)))));
[ tip('a'),empty,node(tip('c'), empty) ] : list(tree(char))
```

Unfortunately, there's no shorthand for writing tree constants like there is for list constants, so we've got to write them out the long way using constructors. If we want to use sumtree to add up all the numbers in the example tree, we must type in the expression

```
sumtree(node(node(tip(1),
                  node(tip(2),
                       tip(3))),
             node(node(empty,
                       tip(4)),
                  tip(5))));
```

This isn't really a drawback because programs that manipulate complex data structures like trees will generally define them using other functions. However, it's very useful to be able to type any kind of constant data structure at the terminal when we're checking out an individual function like sumtree. If we want to test a Pascal program piecemeal, we'll usually have to write elaborate test harnesses or stubs to generate test data.

Making Data More Abstract

The identifier list isn't really a Hope data type. It's called a type constructor and must be parameterized with an actual type before it represents one. We did this every time we declared a list(num) or a list(char). The parameter can be a user-defined type, as with a list(tree), or even a type variable, as in list(alpha), which defines a polymorphic data type. Constructing new data types like this is a compile-time operation, not to be confused with constructing new data values, which is a run-time operation.

You can define your own polymorphic data types. Here's a version of the binary tree we defined earlier that can have any type of value in its leaves:

```
data tree(alpha) == empty ++
                    tip(alpha) ++
                    node(tree(alpha) # tree(alpha));
```

Once again, alpha is taken to be the same type throughout one instance of a tree. If it's a number, then all references to tree(alpha) are taken as references to tree(num).

We can define polymorphic functions that operate on trees of any type of object because our tree constructors are now polymorphic. Listing 1 shows a function to "flatten" a binary tree into a list of the same type of object.

Even More Concise Programs

The importance of polymorphic types and functions is that they let us write shorter, clearer programs. It's rather like the way Pascal subroutines let us use the same code to operate on different data values, but much more powerful. We can write one Hope function to reverse a list of numbers or characters, while we'd need to write two identical Pascal subroutines to reverse an array of integers and an array of characters.

We can use polymorphic functions whenever we're con-

(continued)

cerned only with the "shape" of a data structure and not with the objects in it. Sometimes, however, we'll also want to apply some function to the primitive data items in the structure. Here's a function that defines a list(num) whose elements are the squares of another list(num) using a function called square:

```
dec square : num -> num;
--- square(n) <= n*n;

dec squarelist : list(num) -> list(num);
--- squarelist(nil)     <= nil;
--- squarelist(n :: l)  <= square(n) :: squarelist(l);
```

Every time we write a function to process every element of a list, we'll write something almost identical to squarelist. Here's a function to define a list of factorials:

```
dec fact : num -> num;
--- fact(0)        <= 1;
--- fact(succ(n)) <= succ(n) * fact(n);

dec factlist : list(num) -> list(num);
--- factlist(nil)     <= nil;
--- factlist(n :: l)  <= fact(n) :: factlist(l);
```

factlist has exactly the same "shape" as squarelist; it just applies fact instead of square and then applies itself recursively. Values that differ between applications are usually supplied as actual parameters. Hope treats functions as data objects, so we can do this in a perfectly natural way. A function that can take another function as an actual parameter is called a higher-order function. When we declare it we must give the type of formal parameter standing for the function in the usual way. The declaration of fact tells us it's num -> num. Read this as "a function mapping numbers to numbers."

Now let's see how we can use this idea to write factlist and squarelist as a single higher-order function. The new function needs two parameters—the original list and the function that is applied inside it. Its declaration will be

```
dec alllist : list(num) # (num -> num) -> list(num);
```

The "shape" of alllist is the same as factlist and squarelist, but the function we apply to each element of the list will be the formal parameter.

```
--- alllist(nil, f)    <= nil;
--- alllist(n :: l, f) <= f(n) :: alllist(l,f);
```

We use alllist like this:

```
alllist( [2,4,6], square );
[ 4,16,36 ] : list ( num )

alllist( [ 2,4,6 ], fact );
[ 2,24,720 ] : list ( num )
```

Notice that there is no argument list after the functions square or fact in the application of alllist, so this construction will not be confused with functional composition. fact(3) represents a function application, but fact by itself represents the unevaluated function.

Higher-order functions can also be polymorphic. We can use this idea to write a more powerful version of alllist that will apply an arbitrary function to every element of a list of objects of arbitrary type. This version of the function is usually known as map:

```
typevar alpha, beta ;

dec map : list(alpha) # (alpha -> beta) -> list(beta);
--- map(nil, f)    <= nil;
--- map(n :: l, f) <= f(n) :: map(l,f);
```

The definition now uses two type variables, alpha and beta. Each one represents the same actual type throughout one instance of map, but the two types can be different. This means we can use any function that maps alphas to betas to generate a list of betas from any list of alphas.

The actual types aren't restricted to scalars, which makes map rather more powerful than we might realize at first sight. Suppose we've got a suitably polymorphic function that finds the length of a list:

```
typevar gamma;
dec len : list(gamma) -> num;
--- len(nil)    <= 0;
--- len(n :: l) <= 1 + len(l);

len( [2,4,6,8] ) + len("cat");
7 : num
```

We can use map to apply len to every element of a list of words defined by wordlist:

```
map(wordlist("The form remains, the function never
  dies"), len);
[ 3,4,8,3,8,5,4 ] : list ( num ) ;
```

In this example alpha is taken to be of type list(char) and beta to be a number, so the type of the function must be (list(char) -> num). len fits the bill if gamma is taken to be of type char.

Common Patterns of Recursion

map is powerful because it sums up a pattern of recursion that turns up frequently in Hope programs. We can see another common pattern in the function len used above. Here's another example of the same pattern:

```
dec sum : list(num) -> num;
--- sum(nil)    <= 0;
--- sum(n :: l) <= n + sum(l);
```

The underlying pattern consists of processing each element in the list and accumulating a single value that forms the result. In sum, each element contributes its value to the final result. In len, the contribution is always 1 irrespective of the type or value of the element, but the pattern is identical. Functions that display this pattern are of type (list(alpha) -> beta).

(continued)

In the function definition, the equation for a nonempty list parameter specifies an operation whose result is a beta. This is + in the case of len and sum. One argument of the operation will be a list element, and the other will be defined by a recursive call, so the type of the operation needs to be

```
( alpha # beta -> beta )
```

This operation differs between applications, so it must be a parameter. Finally, we need a parameter of type beta to specify the base case result. The final version of the function is usually known as reduce, and its definition looks like this:

```
dec reduce : list(alpha) #
             (alpha # beta -> beta) #
             beta
            -> beta;

--- reduce(nil,f,b)    <= b;
--- reduce(n :: l,f,b) <= f(n,reduce(l,f,b));
```

To use reduce as a replacement for sum we'll need to supply the standard function + as an actual parameter. The word nonop must precede the function + in the parameter list, so the compiler won't try to use it as an infix operator here.

```
reduce([1,2,3],nonop +, 0);
6 : num
```

If we use reduce as a replacement for len, we're not interested in the first argument of the reduction operation because we always add 1 whatever the list element is. Here's a function that ignores its first argument:

```
dec addone : alpha # num -> num;
--- addone(_ , n ) <= n + 1;
```

We use _ to represent any argument we don't want to refer to.

```
reduce("a map they could all understand", addone, 0);
31 : num
```

Like map, reduce is much more powerful than it first appears because the reduction function needn't define a scalar. Here's a candidate that inserts an object into an ordered list of the same kind of object:

```
dec insert : alpha # list(alpha) -> list(alpha);
--- insert(i,nil)      <= i :: nil;
--- insert(i, h :: t)  <= if i < h
                          then i :: (h :: t)
                          else h :: insert(i,t);
```

Actually, this isn't strictly polymorphic, as its declaration suggests, because it uses the built-in function <, which is only defined over numbers and characters, but it shows the kind of thing we can do. If we use it to reduce a list of characters,

```
reduce ( "All sorts and conditions of men", insert, nil ) ;
"  Aacddefiillmnnnnoooorssstt" : list ( char )
```

we'll see that it actually sorts them. The sorting method (insertion sort) isn't very efficient, but the example shows something of the power of higher-order functions and of reduce in particular. It's even possible to use reduce to get the effect of map, but that's left as an exercise for the reader, as they say.

Of course, map and reduce work only on list(alpha), and we'll need to provide different versions for our own structured data types. This is the preferred style of Hope programming because it makes programs largely independent of the "shape" of the data structures they use. Here's an alternative kind of binary tree and a reduce function for it. The tree holds data at its nodes rather than its tips.

```
data tree(alpha) == empty ++
                    node(tree(alpha) # alpha #
                    tree(alpha));
dec redtree : tree(alpha) #
              (alpha # beta -> beta) #
              beta -> beta;

--- redtree (empty, f, b)       <= b;
--- redtree (node(l, v, r), f, b)<=
             redtree(l, f, f (v, redtree(r,f,b)));
```

Here's the Hope version of tree-sort using the new kind of tree and the two kinds of reduce to construct and flatten them. First, a suitable tree-insertion function:

```
dec instree : alpha # tree (alpha) -> tree(alpha);
--- instree (i,empty)      <= node(empty,i,empty);
--- instree (i,node(l,v,r)) <=
              if i<v
                then node(instree(i,l),v,r)
                else node(l,v,instree(i,r));
```

The tree-sort function is now almost trivial to write:

```
dec sort : list(alpha) -> list(alpha);
--- sort(l) <= redtree(reduce(l,instree,empty),
    nonop ::,nil);

sort("Mad dogs and Englishmen");
"  EMaadddegghilmnnnoss" : list ( char )
```

Anonymous Functions

When we used map and reduce, we had to define extra functions like fact and square to pass in as parameters. This is a nuisance if we don't need them anywhere else in the program and especially if they're trivial, like sum or addone. For on-the-spot use in cases like this, we can use an anonymous function called a lambda-expression. Here's a lambda-expression corresponding to sum:

```
lambda(x,y) => x + y
```

The symbol lambda serves to introduce the function and x and y are its formal parameters. The expression x + y

(continued)

Hope functions possess "full rights" and can be passed as actual parameters like any data object.

is the function body. The definition is just a recursion equation with => instead of <=. Here's another lambda-expression used as the actual parameter of reduce:

```
reduce( [ "toe","tac","tic" ], lambda(a,b) =>
                                  b <> a, nil);
"tictactoe" : list(char)
```

There can be more than one recursion equation in the function definition. They're separated from each other by the symbol |, and pattern matching is used to select the appropriate one. Here's an example that uses pattern matching in a lambda-expression to avoid division by zero when the function it defines is executed:

```
map([1,0,2,0,3],lambda(0) => 0 |
          (succ(n)) => 100 div succ(n));
[ 100,0,50,0,33 ] : list ( num )
```

Functions That Create Functions

As we've seen, Hope functions possess "full rights" and can be passed as actual parameters like any data object. It should be no surprise that we're allowed to return a function as the result of another function. The result can be a named function or an anonymous function defined by a lambda-expression. Here's a simple example:

```
dec makestep : num -> (num -> num);
--- makestep(i) <= lambda x => i + x;

makestep ( 3 ) ;
lambda x => 3 + x : num -> num
```

As we can see from trying makestep, its result is an anonymous function that adds a fixed quantity to its single argument. The size of the increment was specified as an actual parameter to makestep when the new function was created and has become "bound in" to its definition. If we try the new function, we'll see that it really does add 3 to its actual parameter.

```
makestep (3) (10);
13 : num
```

There are actually two applications here. First we apply makestep to 3, then the resulting anonymous function is applied to 10. Finally, here's a function that has functions as both actual parameter and result:

```
dec twice : (alpha -> alpha) -> (alpha -> alpha);
--- twice(f) <= lambda x => f(f(x));
```

Here we're creating a new function that has a single argument and some other function f bound into its definition. The new function has the same type as f. We can see its effect using a simple function like square:

```
twice(square);
lambda x => square(square(x)) : num -> num

twice (square) (3);
81 : num
```

The new function applies the bound-in function to its argument twice. We can even bind in twice itself, generating a new function that behaves like twice except that the function eventually bound in will be applied four times.

```
twice(twice);
lambda x => twice(twice(x)) :
    (alpha -> alpha) -> (alpha -> alpha)

twice(twice) (square) (3);
43046721 : num
```

Conclusion

You've seen how a Hope program is just a series of functions that are regarded as definitions of parts of a data structure—the "results" of the program—and how the powerful idea of higher-order functions allows us to capture many common program patterns in a single function.

Some of these ideas will already be familiar to users of LISP, but they appear in a purer form in Hope because there are no mechanisms for updating data structures like LISP's SETQ and RPLACA or for specifying the order of evaluation like GO and PROG. Unlike LISP programs, Hope programs are free from side effects and possess the mathematical property of referential transparency.

You've seen features that are primitive or lacking in LISP and in most imperative languages. The data declaration lets you define complex data types without worrying about how they're represented, and pattern matching lets you decompose them, so you can use abstract data types directly without writing access procedures and without the hassle of inventing lots of new names. The typing mechanism lets the compiler check that you're using data objects in a correct and consistent way, while the idea of polymorphic types stops the checking from being too restrictive and lets you define common data shapes with a single function.

Higher-order functions and polymorphic types let us write very concise programs. Programmers are more productive and their programs are easier to understand and to reason about. Referential transparency further improves our ability to reason about programs and makes it possible to transform them mechanically into programs that are provably correct but more efficient in their use of space or time. You can find out more about this by reading John Darlington's "Program Transformation" on page 201. Finally, referential transparency frees the meaning of Hope programs from any dependence on the order they're evaluated in, making them ideal for parallel evaluation on suitable machines. You'll be seeing more of Hope and languages like it in the future. ■