Alef User's Guide

Bob Flandrena
bobf@plan9.att.com

Introduction

The Alef programming language provides a convenient means for implementing concurrent programs. An Alef program looks like a C program: the syntactic structure is similar and the expression syntax is almost identical. But, despite the similarity of the language constructs, most Alef programs are structurally dissimilar and operate differently.

This document describes some of Alef's capabilities, concentrating on concurrency features and language facilities not found in C. Alef provides language support for two process synchronization models: shared variables and message passing. The styles can be freely mixed and the choice often depends on the details of a program design. Programs in the shared variable style rely on locks to synchronize access to shared data. The message passing model benefits from Alef primitives that control, send on, receive from, and multiplex message channels. This document focuses on the latter synchronization model, emphasizing its interaction with abstract data types and language-supplied process management primitives.

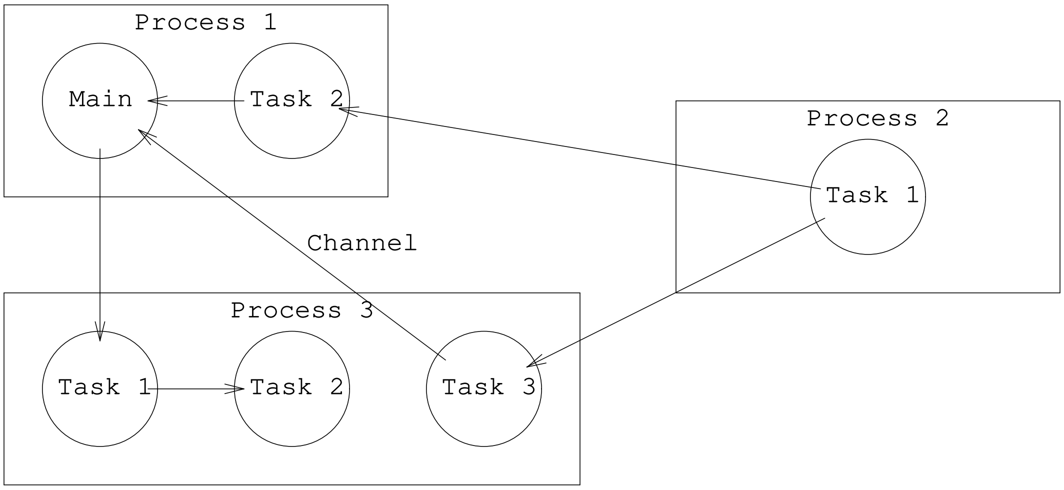
The *Alef Language Reference Manual* by Phil Winterbottom contains the formal definition of the language and is a necessary adjunct to this document. Also, we assume a basic understanding of parallel programming and a familiarity with ANSI C. Alef compilers are available on Plan 9 and on Silicon Graphics systems running IRIX. The implementations are almost identical; this document explicitly identifies the few places where system dependencies intrude. Although some examples are expressed in terms of the Plan 9 system model, it is usually easy to produce an equivalent Unix implementation. Throughout this paper, when reference is made to the C language, it is the ANSI C standard being compared rather than the Plan 9 dialect, which has some extensions.

Overview

Alef consists of four parts: the Alef compiler, run-time support functions, header files, and libraries. The Alef compiler uses the ANSI C preprocessor, so the syntax and semantics of preprocessor directives are identical to those of C. However, because of differences in the declaration syntax, the Alef compiler does not accept C header files. The Alef system header files are stored in the alef subdirectory of the system header file directory. Alef and C programs implement incompatible stack models, so object modules from the two languages cannot be linked with each other and separate system libraries are needed for each compiler. The Alef system libraries are stored in the alef subdirectory of a system library directory. The Alef compiler and loader automatically search the proper directories for the system headers and libraries.

All Alef programs begin by including header file alef.h. This file contains declarations of the complex data types used by the Alef library functions and the prototypes of the functions themselves. Unlike C, the Alef run-time system continually interacts with a program during its execution. It provides functions that manage threads of execution and pass messages between them. Most of these functions implement language operators and are implicitly called by the compiler, but some, such as those that manipulate locks, can be invoked explicitly by the application program.

An Alef program consists of one or more processes each of which contains one or more tasks. *Processes* are threads of execution that are preemptively scheduled by the system and may execute in parallel on a multi-processor. *Tasks* are controlled by the Alef run-time and execute cooperatively. The first task of each program is implicitly created when the program starts; it begins execution at the function named main . A process may create additional processes and tasks each of which begins execution at a specified function. Threads of execution often communicate on *channels* as illustrated in the following diagram:



This section provides an overview of Alef's process creation and communication primitives. Later sections then expand on this groundwork.

Alef provides convenient mechanisms for starting threads of execution. The proc primitive creates a *process* that begins execution at the function supplied as its operand. For example, the statement:

```
proc func(1, 2);
```

starts a process at the function named func and passes it two arguments. The current process continues execution at the statement following the proc statement.

An Alef *channel* is a mechanism for passing data between threads of execution. A channel is similar to a pipe, but unlike a pipe, which is typeless, a channel can only transport the single type specified in its declaration. The statement

```
chan(int) c;
```

declares a channel named c that transports integer values. Unlike variables, channels are not allocated when they are declared; instead, the alloc primitive must be used to allocate and initialize a channel:

```
alloc c;        /* allocate channel named 'c' */
```

The send operator, <-= , transmits the result of the expression supplied as its right operand on the channel specified as its left operand:

```
c <-= 1+1;      /* send a 2 on channel 'c' */
```

The unary receive operator, <- , receives a message on the channel supplied as its operand. The received value may then be manipulated by other operators; for example, it can be assigned to a variable or used as an operand in an expression:

```
	int i;

	i = <-c;		/* assign value received on c to i */
	func(<-c);		/* pass value received on c to func */
```

We can combine processes and channels to implement a simple message passing program:

```
#include	<alef.h>

void
receive(chan(byte*) c)
{
	byte *s;

	s = <-c;
	print("%s\n", s);
	terminate(nil);
}

void
main(void)
{
	chan(byte*) c;

	alloc c;
	proc receive(c);

	c <-= "hello world";
	print("done\n");
	terminate(nil);
}
```

The main process declares and allocates a channel that transports a string address. It then creates a new process with the channel as its argument and sends the address of a string on the channel before printing a message and terminating. The new process begins execution in function receive, where it blocks on the channel until a message arrives and then prints the string before terminating. The order in which the print statements execute is indeterminate, depending on such external factors as system load, the number of processors, and system scheduling policy.

This example illustrates a control structure common to Alef programs: an executive process starts other threads of execution and communicates with them via channels. Subsequent examples elaborate on this basic architecture to illustrate the capabilities of channels and the Alef process control model.

Types

Alef's basic types look similar to those of C but some behave differently. The byte data type is a combination of C's char and unsigned char types with the semantics of the latter. The sizes of some integer types also differ; ints and lints are at least 32 bits and 64 bits long. The single floating point type, float, is always double precision.

A complex type is a collection of basic types or complex types. Alef provides four complex types: union, aggr, adt, and tuple. Unions and aggrs are similar to C unions and structs. Adts and tuples are new and are described in later sections.

An aggr statement is similar to a combination of C struct and typedef statements. The Alef declaration:

```
aggr Point {
        int x;
        int y;
};
```

is equivalent to the C statement

```
typedef struct {
        int x;
        int y;
} Point;
```

An Alef complex type cannot be declared and defined in the same statement; it must be declared first and bound to a variable subsequently. Unlike C, the members of a complex type may be named or unnamed. An unnamed member declaration contains a type specification but no following tag. A complex type may contain many unnamed members, but there can be only one unnamed member of each type. For example, we can declare a Circle

```
aggr Circle {
        int radius;
        Point;
};
Circle c;
```

consisting of an unnamed center point and a radius but we must name the end points of a Line

```
aggr Line {
        Point p1;
        Point p2;
};
Line ln;
```

because there cannot be two unnamed Point members in the aggregate.

Named members of an Alef complex type are referenced using C's syntax. The . operator selects the member specified by its right operand from the complex type on the left; the -> operator selects the member from a pointer to the type. When a member reference does not match any named member, the unnamed members are searched breadth-first for a member with the desired name. In effect, the members of unnamed inner complex types, to arbitrary depth, may be referenced as if they were members of the containing complex type as long as their names are unique within the collection. For example, we can directly refer to the center coordinates of a Circle as c.x and c.y but references to the end coordinates of a line must be fully qualified as ln.p1.x, ln.p1.y, ln.p2.x, and ln.p2.y.

As in C, complex types can be assigned, but Alef provides special promotion rules when the source complex type contains unnamed members. When the operands of an assignment expression evaluate to different complex types, the compiler searches the unnamed members of the right hand complex type breadth-first for a member of the same type as the left hand side and selects that member for assignment. For example,

```
        Point p;
        Circle c;

        p = c;
```

assigns the unnamed Point member of the Circle named c to the Point named p. The promotion rules also apply to the evaluation of function arguments and return values. In the first case arguments are compared to the formal parameters in the function prototype and promoted, if necessary. In the latter case, the return value is evaluated relative to the type of the function. For example, in

```
int
eqpt(Point p1, Point p2)
{
        return p1.x == p2.x && p1.y == p2.y;
}

void
main(void)
{
        Point p;
        Circle c;

        if(eqpt(c, p)) ...
}
```

the first argument to `eqpt` is synthesized by promoting the `Point` member of the `Circle` aggregate to the type of the formal parameter. Similarly, if `eqpt` were changed to accept the addresses of `Points`,

```
        if(eqpt(&c, &p)) ...
```

Alef would promote the address of the `Circle` aggregate to the address of its unnamed `Point` member.

The ability to directly reference members of unnamed inner complex types allows us to avoid the C requirement of attaching dummy names to nested complex types and then having to specify dummy qualifiers on each member reference. Consider the definition of a new complex type that is either a `Line` or a `Circle`:

```
aggr Shape {
        int     type;
        union {
                Circle;
                Line;
        };
};
Shape s;
```

Since the `union`, `Line`, and `Circle` members are unnamed, we avoid unwieldy references like `s.dummy.circ.pt.x`; `s.x` can only refer to the *x* coordinate of the `Circle` member, and `s.p1.x` unambiguously selects the *x* coordinate of the first `Point` of the `Line` member.

An unnamed member can be directly referenced by specifying the type name of the member. For example,

```
        Shape s;

        memset(&s.Circle, 0, sizeof(s.Circle));
```

clears the `Circle` member of the `Shape` aggregate.

Tuples

A *tuple* is a complex type whose members are all unnamed. Most programs process tuples as a unit; if the members are accessed often, the tuple should be an `aggr` with named members. Tuples are often used to bundle several values so they can be transported on a channel, passed as an argument, or returned from a function as a unit.

A tuple declaration consists of the `tuple` keyword followed by a parenthesized list of type specifications for each member:

```
tuple(int, byte*, int) t;
```

defines a tuple named t containing two integers and an address. The tuple keyword can be omitted when the syntax is unambiguous, but it is easier to supply it in all declarations than to remember when it is needed. Tuples can be nested to arbitrary depth:

```
tuple(int, tuple(byte, byte*), int) t;
```

declares a tuple containing an integer, a tuple, and an integer.

Tuples may be used in any context appropriate for a complex type: as assignment operands, as function arguments, as function return values, or as channel messages. The sizeof operator yields the size of a tuple in bytes. As with other complex types, tuples are always passed by value.

A tuple declaration declares a tuple data type and binds it to one or more variables of that type. Alef also provides a tuple operator that constructs an unnamed tuple containing its operands. Syntactically, the tuple operator is specified as a parenthesized list of two or more expressions:

```
(1+2, "abc", var)
```

defines an unnamed tuple containing an int, a byte pointer, and the value of the variable named var. Although a single expression enclosed in parentheses may look like a tuple with one member, it is parsed as a parenthesized scalar.

A tuple may be the source or destination operand of an assignment expression. As with other assignment expressions, the result of a tuple assignment is the value of the right hand side. When the destination operand is a tuple variable, the source must be a named tuple of the same type or an unnamed tuple. In the latter case, the type of each member of the unnamed tuple must exactly match the type of the corresponding member of the destination tuple. The correspondence can be forced by casting the individual members of the unnamed tuple to the proper types or by casting the unnamed tuple to the type of the destination tuple. For example, the code fragment

```
tuple(int, int, byte*) t;
byte i;

t = (0, (int) i, "abc");
t = (tuple (int, int, byte*)) (0, i, "abc");
```

illustrates two equivalent ways of assigning values to the tuple named t. Notice that the variable named i must be explicitly converted from a byte to an int before assignment. Either method is acceptable; the cast is simply applied at different points during the evaluation of the right hand tuple.

When an unnamed tuple appears on the left hand side of an assignment statement all of its members must be variables or the keyword nil. In this case each member of the source tuple is assigned to the variable in the corresponding member of the destination tuple; in effect, the right hand tuple is decomposed into its constituent members. When a member of the destination tuple is nil, the corresponding member of the source tuple is discarded. If the types of corresponding members differ, the source value is promoted to the type of the destination. This differs from assignments to named tuples, where promotion never occurs and explicit casts are necessary. For example,

```
float  a;
byte  *b;
tuple(int,   int,  byte*)  t;

t = (100,  200,  "xyz");
(nil,  a,  b) = t;
```

assigns 200.0 to the variable a and the string "xyz" to b.

Aggregates and unnamed tuples may be assigned to each other. When an aggregate is assigned to a tuple, the members of the aggregate, named and unnamed, are assigned, after promotion, to the corresponding variables in the destination tuple. For example,

```
Circle  c;
Point  p;
int  rad;

(rad,  p) = c;
```

decomposes a Circle into its constituent radius and center point. When an unnamed tuple is assigned to an aggregate, the members of the aggregate are loaded with the corresponding values of the tuple. The types of the source and destination members must match exactly, either by casting each member of the tuple or by casting the constructed tuple to the type of the aggregate. For example,

```
Circle  c;

c = (Circle)(3,   (1.0,1.0));
c = (3,  (Point)(1.0,1.0));
```

illustrates two equivalent ways of loading a Circle aggregate using casts to convert the float coordinates to integers. Notice that the statement

```
c = (3,  (1,1));
```

is acceptable only because the types of the members of the source tuple exactly match the types of the members of the Circle aggregate. An aggregate may not be assigned to a named tuple or vice-versa; only unnamed tuples may be used with aggregates in assignment expressions.

When the value assigned to a member of a tuple affects the evaluation of members of the source tuple, the compiler detects the dependency and performs the evaluation based on the original values of the right hand members. Thus, the code fragment

```
int  a,  b;

(b,a)  =  (a,b);
```

correctly swaps the values of the variables a and b.

The promotion rules governing the assignment of tuples also apply when a tuple is passed as a function argument or is returned by a function. Arguments are evaluated relative to the corresponding formal parameter in the function prototype and return values are matched against the return type of the containing function. Members of unnamed tuples are promoted when the tuple is passed into or returned from a function. However, when a named tuple is supplied as an argument or returned by a function, the types of its members must exactly match the types of the members of the corresponding formal parameter or the function return type.

Unnamed tuples are especially useful for bundling several values so they can be passed into or returned from a function as a unit. This strategy narrows the function interface without requiring dummy variables to hold constant information. In the code fragment

```
Point
midpoint(Point p1, Point p2)
{
        return ((p1.x+p2.x)/2,(p1.y+p2.y)/2);
}

void
main(void)
{
        Point p;

        p = midpoint((1,1), (3,1));
}
```

the arguments to function midpoint and its return value are synthesized from the constituent coordinates.

Unnamed tuples are also ideal for returning several unrelated values from a function. Consider the library function strtoui, which converts a text string to an unsigned binary value. It accepts a pointer to the string, a pointer to a pointer and an integer base. It returns a binary value and updates the second argument to point to the character in the string following the last digit of the extracted number. A tuple containing an error code, the binary value, and the new string pointer simplifies the function interface

```
tuple(int, uint, byte*)
strtoui(byte* str, int base)
{
        int val;

        while(*str != 0 && whitespace(*str))
                str++;
        if(str == nil || *str == 0)
                return(0, 0, str);
        while(*str && !whitespace(*str)) {
                if(!validdigit(*str, base))
                        return (-1, val, str+1);
                /* extract digit into val */
                str++;
        }
        return(1, val, str);
}
```

The code to extract numbers from a string would be:

```
        int ret;
        uint val;
        byte *p, *newp;

        while(*p) {
                (ret, val, newp) = strtoui(p, 10);
                if(ret == 0)
                        break;
                if(ret < 0) {
                        *newp = 0;
                        print("bad number %s\n", p);
                } else
                        print("%d\n", val);
                p = newp;
        }
```

A tuple is displayed by assigning it to an unnamed tuple and then printing the variables of that tuple. For example,

```
tuple(int, byte*, byte*) func();
int a;
byte *b, *c;

(a, b, c) = func();
print("tuple = (%d %s %s)\n", a, b, c);
```

prints the values of the tuple returned by func.

Processes

An Alef process is a preemptively scheduled thread of execution. An Alef program may have as many processes as desired, subject to system restrictions on the number of active processes per user. The proc primitive accepts a list of function invocations as its operand. It scans the list from left to right, starting a new process for each member. The new process begins execution at the specified function with the arguments given in the invocation. The arguments are evaluated in the original process before the new process is created.

On multi-processors, Alef processes may execute in parallel. Since the processes are scheduled by the operating system, the degree of parallel execution and process interleaving is beyond the scope of the language. On a single processor system, Alef processes always interleave execution. The processes comprising a program communicate by exchanging messages on channels or by accessing shared data. The Alef language definition does not specify whether processes share a common address space, but all current implementations use the shared memory model. Programs that communicate via shared data tend to be smaller and faster but may not be portable to an Alef implementation that does not support shared memory. Message passing programs work well for all memory models and are usually easier to debug because channels provide most of the process synchronization. The examples in this document assume a shared memory implementation.

When the proc primitive starts a new process neither process is dominant; Alef processes are more siblings than parent and child. One process may control another with a protocol, but that interaction is determined by the application, not the language.

A program must explicitly terminate all processes to completely stop execution. When a process terminates, other processes in the program are not notified and continue to execute. Several library functions terminate threads of execution. The exits function calls the termination functions previously registered with atexit before terminating the process and all of its tasks. The terminate function terminates a thread of execution; when the last thread in the last process is terminated, terminate calls exits. Alef library functions assume that application-supplied exit functions are only invoked at program termination so an application must ensure that exits is only called by the last process to terminate. In general, only one process in a program should explicitly call exits; others should use terminate. When a process or task returns from the function started by the proc or task statement it calls terminate implicitly.

Alef processes are well suited for decoupling the time-critical processing of asynchronous events from a long-running calculation. Instead of polling for events during a computation, an Alef program uses a slave process to service the events while the master process continues the calculation. The slave process blocks in a system call until an event occurs, handles it, and sends a message on a channel to the master process, which services it when convenient. The event interface in the master process is expressed entirely in terms of channel operations; in effect, the system call is converted to a channel message.

Consider a program that must accept keyboard input while engaged in a long-running calculation. We could implement the keyboard handler as a slave process:

```
void
kbdproc(chan(int) kbdc)
{
	byte c;
	int n, fd, ctlfd;

	fd = open("/dev/cons", OREAD);
	ctlfd = open("/dev/consctl", OWRITE);
	write(ctlfd, "rawon", 5);		/* set raw mode */
	for(;;) {
		n = read(fd, &c, 1);
		if(n <= 0 || c == 0x04) {
			kbdc <-= -1;
			return;
		}
		n = processinput(c);
		if(n)
			kbdc <-= n;
	}
}
```

Kbdproc opens the keyboard, sets it to raw mode, then continually reads bytes, passing each to function *processinput* for low-level processing. The exact form of the processing is not of interest; it could be something like accumulating multiple bytes into a single character or recognizing escape sequences. Valid input values are sent to the executive process on the channel supplied as the argument. When EOF (0x04) is detected or an error occurs, the slave process sends a negative value on the channel and exits. The executive process

```
void
main(void)
{
	int r;
	chan(int) kbd;

	alloc kbd;
	proc kbdproc(kbd);

	for(;;) {
		r = <-kbd;
		if(r < 0)
			terminate(nil);
		/* process the input value */
	}
}
```

allocates a channel, starts the slave process and receives integers from the slave process. The actual processing of each input value is unimportant: for example, input could be accumulated into a command and executed. What is important is that time-critical keystroke processing is confined to the slave process, which executes concurrently with the executive process. Although the processes execute asynchronously, their interaction is synchronous: the executive process blocks until data is received and the slave blocks in the send when the executive is not receiving.

We can extend the program to accept mouse input by adding a slave process for the mouse. It sends an *Mevent* aggregate

```
aggr Mevent {
        Point;
        int     buttons;
};
```

containing the mouse coordinates and the button state on a channel to the executive process each time a mouse event occurs. The executive process must simultaneously receive on the channels from the mouse and keyboard slave processes. The Alef alt statement provides this functionality. Its syntax is similar to that of a switch statement:

```
alt {
case <-c0:          /* receive & discard input on channel c0 */
        break;
case r = <-c1:      /* receive input on channel c1 into r*/
        break;
case c2 <-= 1:      /* send 1 on c2 when there is a receiver*/
        break;
}
```

Each case is associated with a send or receive operation on a different channel. When no channel is ready, the process blocks until a channel operation can succeed. When only one channel is ready, the case statement associated with it is selected and the operation is executed. When more than one channel is ready, a case associated with a ready channel is selected at random. The alt statement does not have a special case corresponding to the default case in a switch statement; all cases must depend on channel send and receive operations.

The mouse slave process is similar to the keyboard slave process:

```
void
mouseproc(chan(Mevent) mc)
{
        int fd, n;
        byte buf[1024];
        Mevent m;

        fd = open("/dev/mouse", OREAD);
        for(;;) {
                n = read(fd, buf, sizeof(buf));
                if(n <= 0)
                        continue;
                m = decodemouse(buf, n);
                mc <-= m;
        }
}
```

After opening the mouse device, the process continually reads the device, invokes decodemouse to decode the buffer into an Mevent aggregate, and sends that to the executive process on the channel supplied as its argument.

The executive process allocates two channels, starts the slave processes, and uses the alt statement to select between keyboard events and mouse events on the channels:

```
void
main(void)
{
	int r;
	Mevent m;
	chan(int) kbd;
	chan(Mevent) mouse;

	alloc kbd, mouse;
	proc kbdproc(kbd), mouseproc(mouse);

	for(;;) {
		alt {
		case r = <-kbd:
			if(r < 0)
				terminate(nil);
			/* process keyboard input */
			break;
		case m = <-mouse:
			/* process mouse event */
			break;
		}
	}
}
```

The introduction of the second input device exposes a deficiency in the termination strategy. On end-of-file or an error, the keyboard slave process sends a negative value to the executive process and terminates. The executive also terminates, leaving the mouse slave process dangling. We remedy this by saving the process numbers of the slave processes in global variables and introduce a new channel used exclusively to communicate a termination condition. The executive adds the new channel to the selection in the alt statement, and kills all slave processes when a value is received on it:

```
int		kbdpid, mousepid;

void
kbdproc(chan(int) kbdc, chan(int) termc)
{
	byte c;
	int n, fd, ctlfd;

	kbdpid = getpid();
	fd = open("/dev/cons", OREAD);
	ctlfd = open("/dev/consctl", OWRITE);
	write(ctlfd, "rawon", 5);	/* set raw mode */

	for(;;) {
		n = read(fd, &c, 1);
		if(n <= 0 || c == EOF) {
			termc <-= -1;
			return;
		}
		n = processinput(c);
		if(n)
			kbdc <-= n;
	}
}
```

```
void
mouseproc(chan(Mevent) mc, chan(int) termc)
{
	int fd, n;
	byte buf[1024];
	Mevent m;

	mousepid = getpid();
	fd = open("/dev/mouse", OREAD);
	for(;;) {
		n = read(fd, buf, sizeof(buf));
		if(n < 0) {
			termc <-= 1;
			return;
		}
		m = decodemouse(buf, n);
		mc <-= m;
	}
}

void
main(void)
{
	int r;
	Mevent m;
	chan(int) kbd, term;
	chan(Mevent) mouse;

	alloc kbd, mouse, term;
	proc kbdproc(kbd, term), mouseproc(mouse, term);

	for(;;) {
		alt {
		case <-term:	/* kill slave processes */
			postnote(PNPROC, mousepid, "kill");
			postnote(PNPROC, kbdpid, "kill");
			exits(nil);
		case r = <-kbd:
			/* process keyboard input */
			break;
		case m = <-mouse:
			/* process mouse event */
			break;
		}
	}
}
```

As in the previous example, low-level input processing is confined to the slave processes and the executive just selects among inputs and processes the data. New input sources can be added by allocating a channel, writing a slave process, and adding a case to the alt statement. In this example, most of the processing takes place in the master process. Using channel messages to represent device events insulates that process from the details of the system interface, frees it from time constraints, and allows it to use Alef's channel operations to demultiplex several event streams. This structure is common to many Alef programs that handle multiple input sources.

Asynchronous Channels

So far the channels in our examples have been synchronous: a sender blocks until another thread of execution attempts to receive on the channel. This mode of operation may be inappropriate when the processing in the master process is lengthy; the keyboard slave process blocks after the first character of type-ahead.

Alef provides asynchronous channels to decouple the sending and receiving processes. An asynchronous channel behaves like a queue with a capacity specified in the channel declaration. Messages are accepted until all slots in the queue are occupied; a subsequent attempt to send blocks until a receiver removes a message. If the channel is of sufficient capacity, senders never block and the sending and receiving processes execute asynchronously. The receive operation is the same for both synchronous and asynchronous of channels; if a message is present, it is received; if not, the receiver blocks. Asynchronous channels not only facilitate parallel execution but also insulate the processing from many races.

An asynchronous channel is declared by appending a capacity specification, which looks like an array dimension, following the channel type. Thus,

```
chan(int)[100]    kbd;
```

defines an asynchronous channel with a capacity of 100 integer messages. The capacity specification is not part of the channel type information; instead, it is an attribute of a channel variable that is applied when the variable is allocated. If the variable is not allocated, for instance, if it is assigned a previously allocated channel, it inherits the type of that channel regardless of the type in its own declaration. This distinction allows a channel variable to reference either synchronous or asynchronous instantiations of a channel. The compiler accepts a capacity specification on any channel declaration, for example, as a formal parameter or the type returned by a function, but the only declaration that matters is the one governing the last allocation of the channel.

We can support type-ahead in our example by using an asynchronous channel between the keyboard and executive processes. We need only change the keyboard channel declaration in the main function; the send and receive operations and the declaration of the keyboard channel parameter of the keyboard slave process remain the same. Supporting type-ahead may be justified but it is not obvious that mouse-ahead is equally desirable.

Tasks

In addition to processes, Alef provides a synchronously scheduled thread of execution called a *task*. Every Alef process consists of one or more tasks; in the trivial case, a process has a single task that is always selected for execution. Alef tasks are co-routines: when one has control, other tasks in the process remain blocked until the selected task gives up control by receiving or sending on a channel, blocking on a QLock (described later), terminating execution, or starting another task. When no task in a process is ready, the Alef run-time blocks until an external event makes one or more of the tasks runnable. When a task executes a system call that blocks or takes a long time, other tasks in the process are prevented from executing. In general, when a thread of execution can block indefinitely in a system call, it should probably be implemented as an Alef process.

An algorithm can often be implemented using either tasks or processes. Both are ideal for encapsulating a computation but when parallel execution and blocking considerations are unimportant, tasks are usually preferable. Tasks are cheap to create and schedule; processes are more appropriate for long-running computations where the startup expense can be amortized. The task scheduling mechanism provides implicit synchronization between threads of execution that cannot be guaranteed with preemptive scheduling. Finally, tasks in the same process always share a common address space; processes may or may not share memory.

The syntax of task creation is identical to that of process creation except the proc keyword is replaced by the task keyword. When a task is started, control always passes to the new task, which runs until it relinquishes control. A task terminates by returning from its entry function or by calling terminate. When one task terminates, another runnable task in the process is selected for execution; if no tasks are ready, the Alef run-time blocks until a task becomes runnable. When the last task in a process calls terminate the process itself terminates. When a task terminates, its stack is reclaimed but other resources held by it, such as dynamic memory, locks, or open files are not reclaimed. If a task calls exits, the process and all of its tasks terminate. Here, all held resources, including the tasks stack space, are not reclaimed and remain unavailable to other processes.

In our example, we can use tasks to partition the processing in the executive process. One task processes keyboard inputs, another handles mouse inputs, and the main thread, the implicit task inherited at program start-up, blocks on the termination channel. The executive process becomes:

```
void
kbdtask(chan(int) kbdc)
{
	int r;

	for(;;) {
		r = <-kbdc;
		/* process keyboard input */
	}
}

void
mousetask(chan(Mevent) mc)
{
	Mevent m;

	for(;;) {
		m = <-mc;
		/* process mouse input */
	}
}

void
main(void)
{
	chan(int)[100] kbd;
	chan(int) term;
	chan(Mevent) mouse;

	alloc kbd, mouse, term;
	proc kbdproc(kbd, term), mouseproc(mouse, term);
	task kbdtask(kbd), mousetask(mouse);

	<-term;		/* main thread blocks here */
	postnote(PNPROC, mousepid, "kill");
	postnote(PNPROC, kbdpid, "kill");
	exits(nil);
}
```

After starting the keyboard and mouse slave processes, the executive passes control to the keyboard input task. That task receives on the keyboard channel; if data is present, it is processed, if not, the task blocks and control returns to the main task. It starts the mouse task, which receives on the channel from the mouse slave process. When that channel is empty, control transfers either to the keyboard task, if a message is present in the keyboard channel, or back to the main task, which immediately blocks on the termination channel. At this point, the three tasks in the process are suspended and remain so until data is available on one or more of the channels. A task associated with a ready channel is selected for execution and it runs until it relinquishes control. In effect, we use the Alef run-time scheduler to simulate the operation of the alt statement. The keyboard and mouse tasks are implicitly terminated when the main task calls exits. The tasks need not be explicitly terminated. They hold no resources except the memory occupied by their stacks and since there are no other active processes, there is no demand for this memory and it will be freed by the system when this process terminates. Notice that the keyboard and mouse slave processes require no modification; the channels insulate them from structural changes in the executive process.

Resource Servers

The interface to many system services does not mesh well with Alef's multi-threaded model of computation. We have already used a process to convert a blocking system call into a channel operation. We can apply this strategy to almost any system resource by implementing a server for the resource in a process that communicates with clients via channels. Not only can we multiplex the resource among several clients but we can also implement the client interface to the resource entirely in terms of channel operations.

Consider the case of alarms in a multi-task process. Since the system provides only one alarm per process, two tasks cannot simultaneously set alarms. Further, when an alarm expires, it is difficult to guarantee that the proper task is selected for execution since that requires the cooperation of the currently executing task. Finally, unlike system calls, which return an error indication when interrupted, most Alef run-time functions, including channel send and receive operations, absorb signals, so the application is unlikely to see the alarm. We can solve these problems by implementing an alarm server to administer alarms.

An alarm server is a process that accepts alarm requests on a common channel and replies to clients on a private channel when the alarm interval expires. Alef channels transport any Alef type and since a channel is a type, we can send the identity of the return channel on an existing channel. A client sends the alarm server a message containing the number of milliseconds until the alarm occurs and an allocated return channel. Since a channel message must be singular, we use a tuple to bundle the values. The alarm server sends an integer value on the return channel when the alarm expires. Notice that the alarm server does not actually generate operating system alarms that interrupt its clients; instead, it uses channel messages to simulate the expiration of an alarm.

The alarm server must simultaneously count down pending alarms and service new requests on its input channel. Since it can't count alarms while blocked in a receive operation, it must only attempt to receive on the input channel when it knows that the operation will not block.

The unary ? operator is a predicate returning the state of the channel supplied as its operand. When the operator precedes the channel, it is called the *can-receive* operator, and evaluates to 1 if a message is available on the channel. The postfix form of the operator, called *can-send*, evaluates to 1 when it is possible to send on the channel without blocking. Because the check of channel status and a subsequent send or receive operation on the channel are not atomic, the can-send and can-receive operators are only reliable when there is no opportunity for transfer of control between the execution of the predicate and the channel operation. For example, although it may look correct, the code segment

```
if(?ch)
        a = <-ch;
```

only works when there is a single receiver on the channel or when the channel connects tasks in the same process; it is inappropriate when more than one process attempts to receive on a channel because execution can be preempted between the test and the receive operation. In our case this isn't a problem because the alarm server is the only process attempting to receive on its input channel. However, we illustrate the solution with a general technique that accommodates multiple receivers. We achieve this by using an alt statement and an asynchronous channel that is always full to ensure that a receive operation will succeed. The code fragment

```
chan(int)[1]  dummy;

alloc dummy;
dummy <-= 1;            /* fill the channel */
while(?ch)
        alt {
        case a = <-ch:
                /* process message */
                break 2;  /* break from while loop */
        case <-dummy:
                dummy <-= 1;    /* refill channel */
                break;
        }
```

the alt statement never blocks; if there is nothing available on the input channel named ch, the case associated with the always-full channel is selected, the channel is refilled, and the loop repeats. Eventually, either the can-receive predicate evaluates to zero or data appears on the channel. In the latter case, the message is received in the first case of the alt statement and the special form of the break statement terminates the while loop by jumping out of two levels of nested control. Notice that the always-full channel must be asynchronous, otherwise the thread blocks when it first sends on the channel and never awakens.

This trick allows the alarm server to poll its input channel for new alarms while counting down existing alarms. The alarm server looks like:

```
void
alarmproc(chan(tuple(int,   chan(int)))  alrmch)
{
        uint t;
        int a, dt;
        chan(int)[1]  dummy;
        chan(int)  reply;

        alarmpid  = getpid();
        alloc dummy;
        dummy <-= 1;
        alist = nil;
        t = msec();
```

```
        for(;;) {
                if(alist == nil) {
                        /* no alarms - get client request */
                        (a, reply) = <-alrmch;
                        addalarm(alist, a, reply);
                        t = msec();
                } else while(?alrmch) {
                        alt {
                        case (a, reply) = <-alrmch:
                                addalarm(alist, a, reply);
                                break 2;
                        case <-dummy:
                                dummy <-= 1;
                                break;
                        }
                }
                sleep(1);               /* sleep 1ms */
                dt = msec()-t;
                t += dt;

                for(each alarm in alist) {
                        if(alarm.msec <= dt) {
                                /* send alarm to client */
                                alarm.ch <-= 1;
                                deletealarm(alarm);
                        } else
                                alarm.msec -= dt;
                }
        }
}
```

The alarm server maintains a list of pending alarms. When the list is empty, the server blocks on the input channel until a request arrives; otherwise, it tests the channel before attempting to receive. When there are pending alarms the server wakes up approximately every millisecond, calls the function msec to get the current time in milliseconds, and decrements the count for each alarm in the list. When a count reaches zero, the server sends an integer on the private channel to the client and removes the alarm from the list. The complete implementation is more complicated, involving clock wrap-around and other details irrelevant here, but the structure of the algorithm is common to servers that multiplex a resource among several clients.

With an alarm server we can modify our program to handle double-click events; for example, a double click of a mouse button. When the mouse task in the executive process receives an input, it sets an alarm for, say, 500 milliseconds, and uses an alt statement to receive a message on either the alarm channel or the normal input channel. If the alarm occurs before an input, the original event is interpreted as a single-click; if the proper mouse event occurs before the alarm expires, the input is processed as a double-click.

The receive loop in the mouse task is modified as follows:

```
void
mousetask(chan(Mevent) mc, chan(tuple(int, chan(int))) alarm)
{
	Mevent m, lastm;
	chan(int) dummy, ch;

	alloc dummy;
	ch = dummy;
	for(;;) {
		alt {
		case m = <-mc:
			if((m.buttons&0x07) == 0)
				break;
			if(ch == dummy) {
				/* no alarm pending */
				alloc ch;
				alarm <-= (500, ch);
				lastm = m;
			} else {
				task consumealarm(ch);
				ch = dummy;
				if(lastm.buttons == m.buttons
				&& eqpt(lastm.Point, m.Point))
					doubleclick(m);
				else
					singleclick(m);
			}
			break;
		case <-ch:	/* alarm expired */
			unalloc ch;
			ch = dummy;
			singleclick(lastm);
			break;
		}
	}
}

void
consumealarm(chan(int) ch)
{
	<-ch;
	unalloc ch;
}
```

We replace the receive operation with an alt statement that receives on the channel from the mouse slave process and another channel referenced by the variable named ch . That variable either refers to a return channel from the alarm server or a dummy channel that is always empty. When an alarm is pending, ch refers to the return channel from the alarm server, both cases of the alt are active, and either a mouse event message or an integer indicating an alarm timeout is received. When no alarm is pending, ch refers to the dummy channel, the second case of the alt is never selected, and only mouse events are received. We begin by waiting for a mouse event. When a button click occurs, we save it, allocate a new channel, send a tuple containing it and the alarm interval to the alarm server, and set ch to refer to the new channel. When the alt repeats, we can receive either a mouse event or an alarm expiration. If an alarm occurs, we deallocate the return channel, process the last mouse event as a single click, and assign the always-empty channel to ch , thereby deactivating the second case of the alt statement. When a button click occurs before the alarm expires, we check the mouse location and the button state to determine if it is a double-click or a single-click and then assign the dummy channel to ch to deactivate the second case of the alt statement. At this point we must decide what to do with the alarm that is pending in the alarm server. We could send a message to the server canceling the alarm, but this strategy is tricky and complicates the alarm server. Instead, we could let the alarm expire and discard the alarm message before deallocating the channel. We choose the second model and use a new task consumealarm to harvest the unneeded alarm. Notice that this strategy is only appropriate when the alarm interval is approximately the same as the event rate. For example, if events occurred thousands of times a second and the alarm interval was on the order of seconds, we would expect to continually generate thousands of consumer tasks during normal operation. We could add alarms for keyboard events by making similar modifications to the keyboard task.

The executive process must allocate the input channel to the alarm server, start and terminate the alarm server process and pass the alarm channel to the mouse task. It looks like:

```
void
main(void)
{
        chan(int)[100]   kbd;
        chan(int)   term;
        chan(Mevent)   mouse;
        chan(tuple(int,   chan(int)))   alarm;

        alloc  kbd,  mouse,  term,  alarm;
        proc  kbdproc(kbd,  term),  mouseproc(mouse,  term),
                alarmproc(alarm);
        task  kbdtask(kbd),  mousetask(mouse,  alarm);

        <-term;          /* main  thread  blocks  here  */
        postnote(PNPROC,   mousepid,  "kill");
        postnote(PNPROC,   kbdpid,  "kill");
        postnote(PNPROC,   alarmpid,  "kill");
        exits(nil);
}
```

This example illustrates several points. First, an always-empty channel allows selective deactivation of a case in an alt statement. Second, an always-full channel allows us to ensure that a channel operation will not block. Third, the ability to send a channel on another channel allows us to implement a server that accepts requests on a common channel and communicates with each client on a private channel. Finally, since tasks are cheap, it is often better to use a task to consume an unwanted message than it is to cancel the message.

Miscellaneous Channel Topics

Channels are not only a mechanism for transferring messages between threads of execution, but also a fundamental data type of the language, and as such, can be manipulated like other data types, subject to certain constraints. The most obvious restriction is that channels cannot be initialized at compile time and must be allocated before use. While they cannot be operands of most arithmetic and shift operators and can only be compared for equality or inequality, channels can be operands for many other operators. Some, like send, receive, can-send, and can-receive, apply only to channels. Others, like assignment and comparison for equality, accept channel operands, as long as both operands evaluate to the same type. Unary operators, such as sizeof, indirection, and address-of also accept a channel operand. Finally, an array can contain channels but each member must be allocated individually. For example,

```
        chan(int)[100]   ch[2];

        alloc  ch[0],  ch[1];
```

declares and allocates two asynchronous channels.

Almost any type of data may be sent or received on a channel. Messages are sent by-value, so it is expensive to send a large data structure; for efficiency, the address of a data structure can be sent, but this introduces a dependency on the shared memory model. Only arrays and functions cannot be sent on channels. Of course, the address of an array or function can be sent and an array can be wrapped in an aggregate for transport.

Abstract Data Types

An abstract data type is a complex type that is similar to a C++ class. It defines storage for its data members and the operations, called methods, that can be performed on them. Data members may be accessed only by member methods unless they are explicitly exported. Methods are known globally unless they are explicitly restricted to internal invocation. Note that the default scope of data members is opposite to that of methods.

An abstract data type is declared using an extended complex type declaration syntax. The adt keyword replaces the aggr or union keyword; it is followed by the name of the type, a list of its members, and prototypes of the functions implementing the type methods. The data members and function prototypes may be specified in any order, but conventionally the prototypes follow the declarations of all data members. An abstract data type may not have a data member and method with the same name. Abstract data types share most of the semantics of complex types. For example, you can declare an array of abstract data types, take the address of one, assign it to a variable of the same type, reference it indirectly, or pass it to a function.

In our examples the details of the mouse and keyboard device interface are isolated in the slave processes. We can implement Mouse and Keyboard abstract data types to formalize and restrict the interface. Usually the functions implementing the methods on an abstract data type are kept in a single source file; for brevity, we bundle the Mouse and Keyboard abstract types with the calling code. The declarations for the Mevent, Mouse and Keyboard abstract data types are:

```
    adt Mevent  {
            Point;
            int       buttons;

            int       fill(*Mevent,   byte*,  int);
    };

    adt Mouse  {
                      Mevent;
            extern    chan(Mevent)       ch;
                      chan(int)          term;
                      int                fd;
                      int                pid;

                      Mouse*             init(byte*,   chan(int));
                      void               close(*Mouse);
            intern    void               mproc(*Mouse);
    };
```

```
adt Keyboard {
        extern  chan(int)[100]   ch;
                chan(int)        term;
                int              kbdfd;
                int              ctlfd;
                int              pid;

                Keyboard*        init(byte*, chan(int));
                void             close(*Keyboard);
                int              ctl(*Keyboard, byte*);
        intern  void             kproc(*Keyboard);
};
```

The Mouse abstract data type declares all data and methods relating to the mouse device. It supplies methods to initialize and close a mouse and to start the mouse slave process, and contains an unnamed Mevent abstract type that provides a method to decode a buffer of raw mouse data. The Keyboard abstract data type provides similar functionality with an additional control method to set the keyboard mode. Methods preceded by the intern keyword can only be invoked by other functions of the type. The extern keyword allows access to data members by code outside the type. We restrict access to the functions that start the mouse and keyboard slave processes and export the channels so the executive process can receive on them.

A method customarily expects the address of an instantiation of the abstract type as the first parameter. Alef provides a shorthand declaration syntax that tells the compiler to provide the address of the type instance automatically. An implicit argument is indicated by preceding the name of the abstract type with an asterisk as the first formal parameter in the method prototype. Notice that this syntax differs from that of a pointer to the abstract type, where the asterisk follows the type name. In our example, all methods except init expect an implicit parameter.

The promotion rules for unnamed members of a complex type also apply to unnamed members of an abstract data type. When an abstract data type is an unnamed member of another abstract data type, the promotion rules apply to the methods as well as the data members of the inner member. When a method cannot be found in an abstract type, the compiler searches its unnamed abstract data type members breadth-first for a matching method. This promotion strategy implements a form of inheritance; if the outer abstract type provides a method of a given name, it takes precedence, if not, it inherits the method from a member. In the latter case the address of the inner member is passed to the method as an implicit parameter, even when the method is invoked relative to the outer abstract type. For example, if the fill method is invoked on an instantiation of a Mouse abstract type, the address of the unnamed Mevent member of that type is supplied as the first argument.

A method is an Alef function with a name of the form *typename* . *method* where *typename* is the name of the abstract data type. For example, the formal definition of the initialization function for the Mouse abstract data type is Mouse.init . The implementation of the Mouse abstract data type becomes:

```
int
Mevent.fill(Mevent *m, byte *buf, int n)
{
        if(n < 10)
                return 0;
        /* decode 'buf' into 'm' */
        return 1;
}
```

```
Mouse*
Mouse.init(byte *device, chan(int) term)
{
        Mouse *m;

        alloc m;
        m->fd = open(device, OREAD);
        if(m->fd < 0) {
                unalloc m;
                return nil;
        }
        alloc m->ch;
        m->term = term;
        proc m->mproc();
        return m;
}

void
Mouse.close(Mouse *m)
{
        if(m->pid)
                postnote(PNPROC, m->pid, "kill");
}

void
Mouse.mproc(Mouse *m)
{
        int n;
        byte buf[1024];

        m->pid = getpid();
        for(;;) {
                n = read(m->fd, buf, sizeof(buf));
                if(n < 0) {
                        m->term <-= -1;
                        continue;
                }
                if(m->fill(buf, n))
                        m->ch <-= m->Mevent;
        }
}
```

We consolidate the channel allocation, slave process initialization, and termination operations in an abstract data type instead of scattering them about the executive process. When the slave process is started with the statement

```
        proc m->mproc();
```

in the initialization method, an implicit pointer to the Mouse abstract data type is passed as the argument. The slave process relies on the promotion rules to invoke the fill method of the Mevent abstract type and to supply the address of the unnamed Mevent member as the first argument to the decoding function.

The implementation of the Keyboard abstract data type is similar:

```
Keyboard*
Keyboard.init(byte     *device,   chan(int)   term)
{
         Keyboard   *k;
         byte  buf[128];

         alloc  k;
         k->kbdfd   = open(device,    OREAD);
         if(k->kbdfd    < 0) {
                  unalloc  k;
                  return  nil;
         }

         sprint(buf,    "%sctl",   device);
         k->ctlfd   = open(buf,   OWRITE);
         if(k->ctlfd    < 0) {
                  unalloc  k;
                  close(k->kbdfd);
                  return  nil;
         }

         alloc  k->ch;
         k->term   = term;
         k->ctl("rawon");
         proc  k->kproc();
         return  k;
}

void
Keyboard.kproc(Keyboard      *k)
{
         byte  c;
         int  n;

         k->pid  = getpid();
         for(;;)   {
                  n = read(k->kbdfd,    &c,  1);
                  if(n  <=  0  ||  c  ==  EOF)  {
                           k->term   <-=  -1;
                           continue;
                  }
                  n = processinput(c);
                  if(n)
                           k->ch  <-=  n;
         }
}

void
Keyboard.close(Keyboard      *k)
{
         if(k->pid)
                  postnote(PNPROC,    k->pid,   "kill");
}
int
Keyboard.ctl(Keyboard      *k,  byte  *msg)
{
         return  write(k->ctlfd,    msg,  strlen(msg));
}
```

A member of an abstract data type is referenced relative to a variable containing an instantiation of the type. The variable may be the type itself or a pointer to the type through any number of levels of indirection. The variable name is qualified with the name of a data member or type method using the same syntax used to select members of other complex types; the . and -> operators work as expected. A method name is followed by a parenthesized list of arguments, for example,

```
Keyboard k, *kp;

k.ctl("rawon");
kp->ctl("rawon");
```

invokes the ctl method of the Keyboard abstract data type in two equivalent ways. The prototype for the ctl method specifies an implicit first parameter, so the Alef compiler supplies the address of the variable as the first argument; in the first case, &k ; in the second, kp .

Sometimes we wish to invoke a method when we lack an instance of the type. This usually occurs when the abstract type provides an initialization method that returns a pointer to the initialized type. Alef provides a mechanism for referencing a method without an instantiation of the type using the syntax . *typename* . *method* . For example, the initialization function of the Mouse data type is invoked by .Mouse.init(*args*). If the method expecting an implicit parameter is invoked in this manner, nil is supplied as the first argument. A method reference of this type is equivalent to invoking the method using a nil base address, for example, ((Mouse*)nil)-> init(*args*).

All state information pertaining to an abstract data type should be stored in data members of the type and not in global variables. The type methods can then be expressed entirely in terms of the type instantiation supplied as an argument, allowing independent processing of multiple instances of the type.

Since the Mouse and Keyboard abstract data types hide the details of initialization, the executive process is simplified:

```
void
main(void)
{
	Mouse *m;
	Keyboard *k;
	chan(tuple(int, chan(int))) alarm;
	chan(int) term;

	alloc term, alarm;

	m = .Mouse.init("/dev/mouse", term);
	if(m == nil)
		exits("mouse");

	k = .Keyboard.init("/dev/cons", term);
	if(k == nil) {
		m->close();
		exits("keyboard");
	}

	proc alarmproc(alarm);
	task kbdtask(k->ch), mousetask(m->ch, alarm);
```

```
            <-term;                 /* main thread blocks here */
            k->close();
            m->close();
            postnote(PNPROC, alarmpid, "kill");
            exits(nil);
    }
```

The alarm server could also be implemented as an abstract data type using a similar approach. We leave that as an exercise for the reader.

Error Handling

Alef provides three mechanisms for detecting and handling errors.

The `check` statement tests an assertion and aborts the process with an error message when the assertion fails. The keyword is followed by a logical expression and an optional string separated from the expression by a comma. If the expression evaluates to zero, a message containing the name of the source file, the line number of the `check` statement, the name of the current function, the optional string, and the last system error message is printed on standard error and the process exits. Other processes are unaffected. When the optional string is not specified, the string "check" is supplied in its place. For example,

```
            n = write(-1, buf, len);
            check n == len, "write error";
```

prints a message of the form

```
    test.l:7:main() write error, errstr(file not open)
```

The error handler is invoked indirectly through the global variable `ALEFcheck`, declared as follows:

```
            void (*ALEFcheck)(byte*, byte*);
```

If the default error handling strategy is inappropriate, an application can set this varia to the address of its own error handler. The first parameter is the string containing the file, line number and function name and the second is the optional string supplied to the `check` statement. An error handler should never return; most code, including many library functions, assumes that sequential execution cannot resume after a failed assertion. `Check` statements can be compiled out of production code by invoking the compiler with the -c command line flag.

The `rescue` statement defines a block of code to be executed by a `raise` statement. A rescue block usually contains error recovery code, but there is no formal enforcement of this use. A rescue block is never entered by sequential execution from the statement preceding the block; only a `raise` or a `goto` statement can cause its execution. When a rescue block is entered, execution flows through the end of a `rescue` block to the following statement. Rescue blocks are well-suited for specifying a recovery action near the point where a resource is claimed instead of replicating the recovery at subsequent points of error.

An optional name may follow the `rescue` or `raise` keyword. When a name is specified in a `raise` statement, the named rescue block is executed; otherwise the rescue block immediately preceding the `raise` statement in the code is executed. Rescue blocks may be chained; one block may contain a `raise` statement that transfers control to another rescue block. Finally, rescue block names are syntactically equivalent to labels. Their scope is the containing function; they share the name space with labels without conflicting with variable names, and they may be the destination of a `goto` statement. A rescue block may only be invoked from within the function where it is defined.

We illustrate the use of the rescue and raise statements with a function that searches a directory for a file beginning with a specified string and returns the name of the file.

```
byte*
findfile(byte *dirname, byte *string)
{
        int n, dirfd, fd;
        byte *buf, buf2[512];
        Dir d;

        n = strlen(string);
        buf = malloc(n);
        rescue {
                unalloc buf;
                return nil;
        }

        dirfd = open(dirname, OREAD);
        if(dirfd < 0)
                raise;
        rescue closedir{
                close(dirfd);
                raise;
        }

        while(dirread(dirfd, &d, sizeof(d)) == DIRLEN) {
                sprint(buf2, "%s/%s", dirname, d.name);
                fd = open(buf2, OREAD);
                if(fd < 0)
                        continue;
                rescue {
                        close(fd);
                        continue;
                }

                if(read(fd, buf, n) <= 0)
                        raise;
                close(fd);
                if(strncmp(buf, string, n) == 0) {
                        close(dirfd);
                        unalloc buf;
                        buf = malloc(strlen(d.name)+1);
                        strcpy(buf, d.name);
                        return buf;
                }
        }
        werrstr("no match");
        raise closedir;
        return nil;             /* needed to fool compiler */
}
```

Notice that each raise statement, except the last, chains back through the rescue blocks, each of which closes a file descriptor or frees memory. The final raise statement invokes the rescue block named closedir by name because the intervening rescue block closes a file descriptor that is no longer open. The return statement at the end of the function is never executed but is required to convince the compiler that the function always returns a value.

Parallel Execution

The par statement executes the statements in its body in parallel and joins the threads of execution at the end. In effect, par forks and executes a process for each statement in its range and then waits for all processes to terminate before proceeding. The par statement is useful when part of a computation can be performed in parallel and the remainder must be single-threaded. Algorithms of this type could be implemented using processes and channels or locks, but the modest synchronization requirements are satisfied by the par primitive.

The par statement:

```
par {
        statement 1;
        statement 2;
            ...
        statement n;
}
```

is a block containing any number of statements. Since braces can be used to group statements, each statement can be arbitrarily complicated. The flow of execution is not sequential; each statement is executed independently and control then transfers to the end of the par block. Further, the order of execution is undefined. The semantics of par ensure that the code following the body of the par block is not executed until all statements within the block have completed execution.

All branches of a par statement share the automatic variables and parameters of the function containing the statement. If a par statement calls a function, the stack expansion is unique to the statement and is inaccessible to other statements. When the par completes, any per-statement stack growth is reclaimed and the single thread of execution continues in the context of the current function.

The par primitive is ideal for implementing a read-ahead algorithm. Consider a program that displays the contents of a file containing blocks of compressed image data. It performs two compute-intensive tasks: decompressing the data on input and displaying a large and potentially complex image. Assuming that the blocks are accessed sequentially, we can decrease the screen update time on a multi-processor by overlapping the two operations. The following Alef code sketches the algorithm:

```
void
main(int, byte **argv)
{
        byte *active, *passive;
        int fd, n;

        fd = open(argv[1], OREAD);
        check fd >= 0, "open error";

        active = malloc(BUFSIZE);
        passive = malloc(BUFSIZE);
```

```
            n = decode(fd, active, BUFSIZE);     /* first block */
            check n > 0, "read error";
            while(active != nil) {
                    par {
                            display(active, BUFSIZE);

                            if(decode(fd, passive, BUFSIZE) <= 0)
                                    passive = nil;
                    }
                    (active, passive) = (passive, active);
                    waitforinput();
            }
    }
```

The function named decode reads, validates and decompresses the input data into the buffer supplied as the second argument. Function display displays a buffer of decompressed data and waitforinput solicits user input. After filling the first buffer with data, calls to display and decode execute in parallel. The buffer switch following the par statement is not executed until both operations complete.

Allocators

The alloc primitive and malloc library function dynamically allocate memory. Alloc accepts a list of pointers as its operand; each pointer is set to the address of an area that is the size of the object referenced by the pointer. Malloc returns a pointer to an area of memory at least as big as the number of bytes specified by its single argument. Both clear the allocated memory to zero and always yield a valid address; if an allocation cannot be satisfied, the process aborts with a check message describing the symptoms.

Channels can only be allocated with the alloc statement; it allocates the necessary space and initializes the channel for subsequent use. Alloc not only allocates channels and complex data types but also basic types:

```
    typedef  byte*   Ptrs[5];

            byte  **p;
            int  *ip;
            Ptrs  *c;

            alloc  p, ip, c;
            alloc  *c[0], *c[1], *c[2], *c[3], *c[4];
```

allocates a byte, an integer, five pointers, and five one-byte fields and assigns the allocated addresses to variables p, ip, c, and the five elements of c, respectively. Alloc does not allocate beyond the first level when pointers have more than one level of indirection; here we must explicitly allocate each member of c after allocating c itself. The operands of alloc are evaluated left to right so the statement:

```
            alloc  c, *c[0], *c[1], *c[2], *c[3], *c[4];
```

is legal because the variable c is guaranteed to be allocated before its members.

Unalloc frees a block of dynamically allocated memory; the block could have been allocated using alloc or malloc. When its argument is a channel, unalloc waits for pending channel operations to complete before freeing the channel and its buffers; all other types of arguments cause unalloc to operate identically to free.

Iterators

The Alef iterator operator :: repeatedly executes the statement containing it using an implicit incrementing counter. On each iteration, the iterator expression is replaced by the value of the counter. The operands of the iterator statement are expressions yielding values defining the range of the iteration. The expressions are evaluated once; the left operand specifies the starting value, the right operand the terminating value. The statement is executed with the counter successively taking on values from the starting value to one less than the terminating value. When the counter equals the terminating value, the iteration ceases. If the terminating value is less than the starting value, the statement is not executed.

Iterators provide a useful shorthand when processing arrays but care should be exercised to limit the range of the iteration. For example, we could specify our previous example as

```
alloc c, *c[0::5];
```

but it would produce unexpected results; since iteration is over the entire statement, the variable c is reallocated as each of its members is allocated. A correct implementation would allocate c in a separate statement. An iterator is accepted wherever an expression is expected. For instance, the program

```
int i, a[10];

a[i=0::10] = i;
print("%d ", a[0::10]);
```

prints the string "0 1 2 3 4 5 6 7 8 9 ". The elegantly obscure

```
typedef float Matrix[4][4];
void mul(Matrix r, Matrix a, Matrix b){
        int i, j, k;

        r[0::4][0::4] = 0;
        r[i=0::4][j=0::4] += a[i][k=0::4]*b[k][j];
}
```

multiplies two 4x4 matrices. When a statement contains more than one iterator, the order of their evaluation is undefined. However, when the result of an iterator is assigned to a variable, the assignment is performed before the statement is executed and the variable may be referenced without regard to order of evaluation. In the matrix example, the assignment of the iterator to the variable k is guaranteed to occur before the operands of the multiply are evaluated.

QLocks and Locks

The QLock abstract data type implements blocking mutual exclusion. QLock.lock claims a free lock or blocks the requesting thread until the lock is freed. QLock.unlock frees a lock. When more than one thread is blocked on a lock, the order of service when the lock frees is undefined; currently a last-in, first-out model is used. QLock.canlock returns zero if a lock is held, or claims it and returns one when it is available. QLocks can be used to synchronize tasks in the same process; when one task blocks contending for a QLock, control transfers to another task that is ready to run or blocks in the Alef run-time until one is ready.

When a QLock is allocated with alloc or alloc, either as itself or a member of a complex type, it is properly initialized and ready for use. A process can only deallocate a QLock when it knows that no other processes will attempt to use the lock. A QLock should never be deallocated when it is held: the list of threads blocked on the QLock is lost and they will never be awakened.

Locks and channels support two models of computation that are complementary and often interchangeable. Neither synchronization model is inherently superior; each has strengths and weaknesses that affect its suitability for a particular implementation. The channel model is slightly slower and bigger but isolates the processing and formalizes the flow of control. The locking model is fast and small but can scatter synchronization details throughout the program. Message passing algorithms work well when messages are small and they are typically easier to debug since the run-time synchronization code has been thoroughly exercised. However, locks can also provide debugging support and restricted semantics, especially if they are implemented as an abstract data type within a larger abstract type. Our example programs use channels and slave processes to multiplex mouse and keyboard input to an executive process. Many programs of this style also manage a display. We could design the display update logic similarly, but the output data, perhaps a bitmap, may be large enough to make message passing impractical. Instead, we could implement the display update code as a Display abstract data type containing a QLock to synchronize accesses to the device. In this case, the concept of locking the display during update is natural and easily restricted and avoids the potentially large message passing overhead.

The QLock abstract type is implemented in terms of the Lock abstract type, which provides spin locks. Locks are seldom needed by most applications; QLocks are sufficient for all but the most specialized circumstances. Lock.lock claims an available lock or loops on a held lock until it is freed. When several processes contend for a lock, the lock is granted to one at random when it frees. A lock is freed with Lock.unlock; the process freeing a lock need not be the process that claimed it. Lock.canlock returns zero if the lock is held or claims the lock and returns one if it is available. Locks cannot be used to synchronize tasks in the same process; since a task does not block while contending for a spin lock, control can never transfer to another task.

Guarded Blocks

The Alef compiler provides a syntactic mechanism to lock a critical section of code. A *guarded block* allows only one thread to execute its statements at a time. When one thread enters the block no other thread may enter until the first thread leaves. At that point, a thread waiting to execute the block is randomly selected to claim the block and begin executing its statements.

A block is guarded when its opening brace is preceded by an exclamation point:

```
<unguarded code>
!{
        <code in guarded block>
}
<unguarded code>
```

When an exclamation point precedes the opening brace of a function, the entire function is guarded.

The Plan 9 library function that reads the system clock:

```
intern int fd = -1;

int
time()
{
        int t;
        byte b[20];

        memset(b, 0, sizeof(b));
        if(fd < 0)
                fd = open("/dev/time", OREAD|OCEXEC);
        if(fd >= 0) {
                seek(fd, 0, 0);
                read(fd, b, sizeof(b));
        }
        t = atoi(b);
        return t;
}
```

opens the clock file once, reads the ASCII string containing the current clock value, and converts it to binary. Since the file is held open, the function must seek to the beginning before each read; this two-step access introduces a race. If a process is preempted between the seek and the read, another process can execute the same code, leaving the file incorrectly positioned. When the original process regains control, the read fails. We can combine the seek and read into an atomic operation by placing them in a guarded block:

```
        if(fd >= 0) !{
                seek(fd, 0, 0);
                read(fd, b, sizeof(b));
        }
```

When one process enters the critical section, other processes block after the if statement until the original process finishes. There is the potential for another race when the clock is opened, but it is less critical; the file could be opened twice, wasting a file descriptor but producing correct results. A race-free implementation of the function guards all file manipulation:

```
        !{
                if(fd < 0)
                        fd = open("/dev/time", OREAD|OCEXEC);
                if(fd >= 0) {
                        seek(fd, 0, 0);
                        read(fd, b, sizeof(b));
                }
        }
```

Guarded blocks and the par statement implicitly synchronize threads of execution. In both cases care must be taken to ensure that the flow of execution reaches the synchronization point at the bottom of the block. The compiler limits control transfer to the scope of a guarded block or par statement but cannot enforce this constraint when a subroutine is called from within a block. For example, it is inadvisable to call terminate from within a guarded block; when another thread in the program attempts to execute the block, it blocks forever.

The ... Formal Parameter

The formal parameter ... indicates that the function expects a variable number of arguments. The syntax accepts ... anywhere in the parameter list, but ... consumes all following parameters, so it only makes sense as the final formal parameter. Within the function ... is the name of a variable of type pointer to void and can be used in any context appropriate for that type. Typically ... is used as an address on the stack from which following arguments can be extracted. The stack architecture is implementation dependent, so although Alef provides the ... syntax as a convenience, code that uses it in this manner must still make assumptions about the stack width and argument passing conventions.

Since ... is a pointer, it can be used in pointer arithmetic or array indexing. For example, a function to compute the average of a variable number of integer arguments

```
int
avg(int n, ...)
{
        int i, tot;

        tot = 0;
        for(i = 0; i < n; i++)
                tot += ((int*)...)[i];
        return tot/n;
}
```

casts ... to an integer pointer and then indexes beyond it to pick up each integer argument.

Most commonly ... is not used directly but is passed as the base of a variable length argument list to the library function doprint for formatting. The Alef function

```
byte    *argv0;

void
fatal(byte *fmt, ...)
{
        byte buf[1024], *p;

        p = doprint(buf, buf+sizeof(buf), fmt, ...);
        *p = 0;
        fprint(2, "%s: %s\n", argv0, buf);
        exits(buf);
}
```

prints a fatal error message prefixed by the program name and exits with an error condition.

The Become Statement

The become statement transfers control in one of two ways. If its operand evaluates to anything other than a function, become behaves as a return statement yielding the result of the evaluation. When the operand is a function returning exactly the same type as the current function, the current function is replaced by the operand function and control is transferred to the beginning of that function. There is a crucial difference between a subroutine call and this form of function invocation: the former creates a new subroutine context on the stack before transferring control; the latter replaces the current function context with the new context and transfers control. When a function invokes itself or when several functions invoke each other via mutual recursion using become, the computation runs in constant space. When used in this form, the become statement is useful for implementing functional style programs.

We can use the become statement to implement a finite state machine that runs in constant space. Suppose we have a state machine with four states (A, B, C, D), three inputs (1, 2, 3) and the following transitions:

state/input	1	2	3
A	A	B	C
B	B	A	B
C	A	B	D
D	D	D	D

The machine starts in state A and continues until reaching the terminal state D. As it passes through each state, the state name is printed. The following Alef program implements this state machine:

```
void	stateA(chan(int));
void	stateB(chan(int));
void	stateC(chan(int));

int	readcpid;

void
readc(chan(int) c)
{
	int fd, fd2;
	byte ch;

	readcpid = getpid();
	fd = open("/dev/cons", OREAD);
	fd2 = open("/dev/consctl", OWRITE);
	check fd >= 0 && fd2 >= 0, "keyboard open";
	write(fd2, "rawon", 5);
	for(;;) {
		check read(fd, &ch, 1) == 1, "read error";
		c <-= ch;
	}
}

void
stateA(chan(int) c)
{
	print("A");
	switch(<-c) {
	default:
	case '1':	become stateA(c);
	case '2':	become stateB(c);
	case '3':	become stateC(c);
	}
}

void
stateB(chan(int) c)
{
	print("B");
	switch(<-c) {
	default:
	case '3':
	case '1':	become stateB(c);
	case '2':	become stateA(c);
	}
}
```

```
void
stateC(chan(int) c)
{
        print("C");
        switch(<-c) {
        default:        become stateC(c);
        case '1':       become stateA(c);
        case '2':       become stateB(c);
        case '3':       print("\n");    /* terminal state */
                        break;
        }
        if(readcpid)
                postnote(PNPROC,readcpid, "kill");
        exits(nil);
}

void
main(void)
{
        chan(int) c;

        alloc c;
        proc readc(c);
        stateA(c);
}
```

Of course, we could implement this simple example with a transition matrix and a recursive become statement, but we choose this form to emphasize the flow of control.

Alef Libraries

The Alef system library libA.a corresponds to the Plan 9 C library libc.a. Most Alef functions in libA.a are functionally identical to their C language counterparts. Occasionally, the names of complex types or their members vary; the Alef header file alef.h is the definitive record of interfaces and system aggregate names. Auxiliary Alef libraries provide regular expression matching, network interfaces, and graphics operations similar to corresponding C libraries. In general, the manual pages for the C library functions accurately describe the functioning of their Alef counterparts.

The Alef buffered I/O library, libbio.a, is the only library that differs substantially from the C library of the same name. In the Alef implementation, the Biobuf structure is implemented as an abstract data type providing buffered I/O operations. Header file <bio.h> declares the abstract data type and its methods. With the exception of the C function getd, which has no Alef counterpart, the Alef Bio methods are functionally similar to the C library functions. The names of the methods differ slightly; the leading B of the C language function is deleted. Most Alef Bio methods expect an implicit parameter, so the first argument of each C function is not needed. Finally, unlike the C version of the library, the Alef package supports parallel execution. See the manual page for details.

Miscellaneous Features

There is no USED compiler pseudo-op in Alef as there is in the Plan 9 C compilers. Unreferenced formal parameters are indicated by discarding the parameter name in the function definition. This action suppresses the declared but not used warning issued when a program is compiled with the '-w' flag. For example, the main function definition

```
void
main(int, byte**)
{
        ...
}
```

can be used for a program that does not process the command line.

Two global variables control the creation of Alef tasks and processes. They contain default values that are appropriate for most Alef programs, but may be overridden for unusual applications. The integer variable `ALEF_stack` contains the number of bytes of stack space allocated for each Alef task or process. The default value is 16,000 bytes. If this figure is too small or too large, set `ALEF_stack` to the desired value before executing the `task` or `proc` statement creating the thread.

The variable `ALEF_rflags` contains a bitmask specifying the resources inherited by an Alef process. Appropriate values for Plan 9 are described in the *fork* (2) manual page. By default `ALEF_rflags` contains the value `RFPROC|RFMEM|RFNOWAIT` specifying that the processes share data segments and that the new process is not a child of the original process. The SGI Unix version of Alef uses the values given in the *sproc* (3) manual page. By default, the `PR_SFDS|PR_SADDR` flags specify that processes share data segments and file descriptors. Processes are always disassociated from each other in the Unix implementation; there is currently no way to change this behavior.

The default resource specification works well for Alef programs that have minimal interaction with the external environment. However, the default behavior may be inappropriate when the Alef process model does not mesh with that of the system. Consider an Alef program that emulates the action of a shell: it reads a command, executes it in a child process, and waits for its termination. If the command is started using the `proc` statement, the default resource specification is inappropriate: the processes shouldnt share memory and they must retain a parent-child relationship. The program should set `ALEF_rflags` to the desired value, execute the `proc` statement, wait for the child to terminate, and then reset `ALEF_rflags` to the default state.