The Plan 9 Front Concurrent C Extensions

Benjamin Purcell (spew)
benjapurcell@gmail.com

ABSTRACT

The Plan 9 Front compilers extend the C programming language to provide builtin CSP style concurrency operations. This paper describes the usage and implementation of the extension.

1. Introduction and Motivation

Programming with CSP style concurrency is an essential part of the Plan 9 operating system. Concurrent programs were originally written in Alef which had builtin concurrency operations. When Alef was retired, a library was written to allow access to CSP operations from programs written in C (see *thread*(2)). However, there are a number of deficiencies with *thread*(2): thread creation is inflexible, receiving or sending of multiple channels requires an awkward definition of an array of structures, and the send/receive operations are not type safe. This extension aims to address those concerns to make threaded programs easier and safer to write without the need to maintain a separate compiler infrastructure such as Alef. This document assumes familiarity with *thread*(2).

2. The Extensions

The compiler extension provides for launching new threads and processes, declaring and allocating storage for typed channels, and type safe sending and receiving from channels. It also provides a new control structure for type safe sending or receiving of multiple channels.

2.1. Thread and Process Creation

Threads and processes are created using the keywords `coproc` and `cothread` which have the syntax of a function that takes two arguments. The first argument a function application, and the second is an unsigned int that specifies the stack size for the process or thread. The calls `coproc` and `cothread` return the resultant thread id.

```
int tid, pid
void fn(int arg1, double arg2, char *arg3);
...
tid = cothread(fn(a, b, c), 8192);
pid = coproc(fn(a, b, c), 8192);
```

The function passed to `coproc` and `cothread` can have any signature, though its return value will not be used. Instead of applying the function to its arguments, the calls of `cothread` and `coproc` tell the compiler to check the arguments to the function and then compile a call into *thread*(2) to start the function in a new thread or process with a memory allocated stack (see *malloc*(2)). Due to the type-checking, if `a`, `b`, and `c`, are of an incompatible type to `int`, `double`, and `char*` respectively, then the example above will not compile.

2.2. Channel Declarations

The extension reserves the character @ for declarations of typed channels. A typed channel has a type—referred to as the sending type—associated with it; only values of that type may be sent or received from the channel. The @ symbol has the same precedence as the pointer dereference * and serves a similar purpose. Thus

```
int @c;
```

declares `c` to be a channel for sending/receiving an int;

```
char *@c;
```

declares `c` to be a channel for sending/receiving a pointer to a char; and

```
int *(*@c[3])(int);
```

declares `c` to be an array of three channels for sending/receiving pointers to functions that take an int and return a pointer to an int.

2.3. Channel Allocation

Once a channel is declared, it must be configured for use by applying the compiler extension `chanset` to it. The usage is

```
int @c;
chanset(c, nelem);
```

Where `nelem` is an int that sets the number of values the channel can hold and whether the channel is buffered or unbuffered. See `chancreate` in *thread*(2).

2.4. Channel Operations

The compiler extension allows for sending into and receiving typed values from channels. The syntax for receiving a channel mimics that of channel declarations. That is, for a channel for sending ints and an int as follows:

```
int @c, i;
```

the statement

```
i = @c;
```

receives an int from the channel `c` and assigns the value to `i`. More specifically, the expression `@c` evaluates to the value retrieved from the channel via a call into the *thread*(2) library.

A new binary operator @= is used to send into a channel. The left-hand side of the expression must be a channel and the right-hand side's type must match the type of the sending value of the channel. Thus given

```
char *@c;
```

the statement

```
c @= "hello, world";
```

sends the string into the channel. The expression `c @= val` evaluates to an int: 1 on success and -1 if the send was interrupted.

2.5. Sending/Receiving of Multiple Channels

Channel sending/receiving may be multiplexed on a single statement using a new control flow statement called the alt-switch. It is similar to a switch with the expression value replaced by an @ character and the `case` keywords replaced by a new extension keyword `alt`. Instead of constant expressions, each `alt` is labeled by potential channel sends or receives. An optional default label handles the case where the underlying `doalt` operation (see *thread*(2)) is interrupted.

```
int @ichan, @req, i;
char *@schan, *s;

s = "hello";
switch @{
alt i = @ichan:
     print("%d\n", i);
     ...
     break;
alt @req:
     ...
     break;
alt schan @= s:
     print("Sent hello\n");
     ...
     break;
default:
     print("Interrupted!\n");
}
```

In the example above three potential channel operations are multiplexed on one alt–switch statement. Either an int is received from `@ichan` and assigned to `i`, an int is received from `@req` and its value thrown away, or the string `hello` is sent into `schan`. The operations are multiplexed in the sense that if at least one of those channel operations can proceed, one is chosen at random to be executed and control flow proceeds after the corresponding `alt` label. Otherwise the alt–switch statement blocks until one of the operations can proceed.

A non–blocking alt–switch statement is specified by using two @ symbols:

```
switch @@{
     ...
default:
     print("No channel operations can proceed.\n");
}
```

In this case, the statement does not block if no channel operations can proceed, but immediately continues execution at the default label. If a non–blocking alt–switch is interrupted while in the middle of executing a valid channel operation, then the alt–switch will continue execution at a `case` labeled by `−1`.

The channel send operation in an `alt` label is more restricted than an ordinary channel send in the sense that the right hand side of the @= binary operator must be addressable. Thus

```
alt ichan @= 5:
```

will not compile.

3. Summary of the Extension

In total the extension reserves the following new keywords or symbols

```
@ alt   chanset   cothread   coproc
```

and defines the following new expressions:

Usage Summary	
Channel Operations	
`chanset(chan, nelem)`	Allocates and readies a channel
`chan @= val`	Channel Send
`@chan`	Channel dereference (receive)
Alt–Switch	
`switch @{...}`	Blocking alt–switch
`switch @@{...}`	Non–blocking alt–switch
`alt val = @chan:`	Alt label (receive)
`alt @chan:`	Alt label (receive, value thrown away)
`alt chan @= val:`	Alt label (send)
Thread Creation	
`coproc(fn(...), stksize)`	Starts a process in its own stack
`cothread(fn(...), stksize)`	Starts a thread in its own stack

Figure 1. Summary of compiler extensions and usage. `chan` denotes a typed channel and `val` is of the channel's sending type. `nelem` is an int, `fn` is a function of any signature, and `stksize` is an unsigned int.

4. Implementation Details

The extension does two things: checks types and provides syntactic sugar for calls to `thread`(2).

Threads and processes are created by walking the argument list to find the number of arguments to the function and then compiling a call to `rtthreadcreate` in *thread*(2). For example, given a call

```
cothread(fn(a, b, c), 1024);
```

the compiler first checks the types of the arguments to the function `fn` and then rewrites the above as

```
rtthreadcreate(1024, 3, fn, a, b, c);
```

The library *thread*(2) takes things from there. A call of `coproc` is rewritten as a call to `rtproccreate` in the exact same way.

Each channel declaration declares a new structure that holds both the channel and locations for sending and receiving values of the sending type of the channel. The channel itself is declared as a pointer to that struct. Thus, a channel declaration such as

```
int @c;
```

is rewritten by the compiler as

```
struct {
     Channel;
     int @in;
     int @out;
} *chan;
```

In this rewrite the symbols `@in` and `@out` are not channels but the actual identifier used internally by the compiler to access those members of the structure. They are only accessible within the compiler since @ is reserved.

A more complicated type such as

```
int (*@*chan[3])(int, double);
```

(an array of three pointers to a channel that returns pointers to function pointers), becomes

```
struct {
     Channel;
     int (*@in)(int, double);
     int (*@out)(int, double);
} **chan[3];
```

In other words, if you read the declaration from the outer type inwards toward the symbol, then everything before the @ symbol is associated with the sending type of the channel and everything after is associated with the declaration of the the symbol `chan` itself.

The call

```
chanset(chan, nelem);
```

allocates memory for the channel and does further setup as needed for *thread*(2). It is syntactic sugar for the call

```
chan = rtchancreate(sizeof(*chan), sizeof(chan->@in), nelem);
```

where `chan` has already been defined by the compiler to have a structure type like the examples above.

A channel receive operation `val = @chan` is rewritten as

```
recv(c, c->@out);
val = c->@out;
```

and in the case when there is no left hand side, then as

```
recv(c, c->@out);
```

alone. A channel send expression

```
c @= val;
```

is first type checked so that val is assignable to `c->@in` and then rewritten as

```
send(c, &val);
```

In the case where `val` is not addressable, such as

```
c @= val1 + val2;
```

then the compiler rewrites this as

```
c->@in = val1 + val2;
recv(c, &c->@in);
```

The alt-switch statement is compiled by constructing an `Alt` structure (see *thread*(2)) and then rewriting the alt-switch as a normal switch with case labels corresponding to the value returned by a `doalt` call (see *thread(2))* with the `Alt` structure as an argument. The example

```
int @ichan, @req, i;
char *@schan, *s;

s = "hello";
switch @{
alt i = @ichan:
     print("%d\n", i);
     ...
     break;
alt @req:
     ...
     break;
alt schan @= s:
     print("Sent hello\n");
     ...
     break;
default:
     print("Interrupted!\n");
}
```

is rewritten as

```
int @ichan, @req, i;
char *@schan, *s;

s = "hello";

struct Alt alts[] = {
     {ichan, &i, CHANRCV},
     {req, nil, CHANRCV},
     {schan, &s, CHANSND},
     {nil, nil, CHANEND}
};
switch(doalt(alts)) {
case 0:
     print("%d\n", i);
     ...
     break;
case 1:
     ...
     break;
case 2:
     print("Sent hello\n");
     ...
     break;
default:
     print("Interrupted!\n");
}
```

The appropriate type checking takes place on the values being sent or received in order to preserve type safety.