The CPC manual

Juliusz Chroboczek, Gabriel Kerneis
<jch@pps.jussieu.fr>, <kerneis@pps.jussieu.fr>

12 June 2009

Chapter 1

The CPC language

CPC is a programming language designed for any situation where even-driven programming is suitable — most notably, for writing concurrent programs. The semantics of CPC is defined as a source-to-source translation from CPC into plain C using a technique known as *translation into Continuation Passing Style* (CPS) [SW74, Plo75].

The main abstraction provided by CPC is a *continuation*, roughly corresponding to what other concurrent programming systems term a *thread* or *lightweight process*[1].

Structure of a CPC program Just like a plain C program, a CPC program is a set of functions. Functions in a CPC program are partitioned into "cps" functions and "native" functions; a global constraint is that a cps function can only ever be called by another cps function, never by a native function. The precise set of contexts where a cps function can be called is defined in Sec. 1.1.1.

Intuitively, cps code is "interruptible": it is possible to interrupt the flow of a block of cps code in order to pass control to another piece of code or to wait for an event to happen. Native code, on the other hand, is "atomic": if a sequence of native code is executed, it must be completed before anything else is allowed to run.

Technically, native function calls are executed by using the machine's native stack. Cps function calls, on the other hand, are executed by using a lightweight stack-like structure known as a continuation. This arrangement makes CPC context switches extremely fast; the tradeoff is that a cps function call is an order of magnitude slower than a native call. Thus, computationally expensive code should be implemented in native code whenever possible.

Execution of a CPC program starts at a native function called `main`. This function usually starts by registering a number of continuations with the CPC runtime (using `cpc_spawn`, Section 1.1.5), and then passes control to the CPC runtime (by calling `cpc_main_loop`, Section 1.1.3).

[1] From the programmer's point of view, the main difference is that a thread has a long-term identity (a thread or process identifier) which makes it possible to have constructs such as *join* or *kill*. Continuations, on the other hand, are transient: after execution of some code, the former continuation no longer exists, and a new continuation has been created.

Implementatin of CPC CPC is implemented as a source-to-source translation from the CPC language, a conservative extension of C, into event-driven C code. The resulting C code is linked against the *CPC scheduler*, a modest event loop that handles scheduling of CPC continuations.

The CPC scheduler manipulates three data structures: a queue of runnable continuations, a priority queue of sleeping continuations, and a set of queues of continuations blocked on condition variables or waiting for I/O.

1.1 The CPC language

CPC is a conservative extension of the 1999 edition of the C programming language; thus, the syntax of CPC is defined as a set of productions to be added to the grammar defined in the ISO C99 standard [ISO99].

In addition to the reserved words in C99, CPC reserves the words `cps`, `cpc_yield`, `cpc_done`, `cpc_spawn`, `cpc_wait`, `cpc_sleep`, `cpc_io_wait`, `cpc_attach`, `cpc_detach` and `cpc_detached`.

1.1.1 CPS contexts

Any instruction, declaration, or function definition in CPC can be in *cps context* or in *native context*. Cps context is defined as follows:

- the body of a cps function is in cps context (Sec. 1.1.2;
- the body of a `cpc_spawn` statement is in cps context (Sec. 1.1.5);
- the body of a `cpc_detached` or `cpc_attached` statement is in cps context (Sec. 1.1.9).

Any construct that is not in cps context is said to be in native context.

1.1.2 CPS functions

function-specifier ::= `cps`

Functions can be declared as being CPS-converted by adding `cps` to the list of functions specifiers. The effect of such a declaration is to put the body of the function in cps context, thus making it possible to use most of the CPC features.

block-item ::= function-definition

Functions can be defined within other functions, as in Algol-family languages; the inner function can access the variables bound by the outer one. Only cps functions can be inner functions, and they must be within other cps functions.

Free variables of inner functions are copies of the variables of the enclosing function; thus, a change to the value of the free variable is not visible in the enclosing function.

1.1.3 Bootstrapping

```
void cpc_main_loop(void);
```

`cpc_main_loop` Since `main` is a native function, some means is necessary to pass control to cps code. The function `cpc_main_loop` invokes the CPC scheduler; it returns when all continuations have been exhausted (i.e. where there is nothing more to do).

1.1.4 CPC statements

statement ::= cpc-statement

CPC has a number of statements not present in the C language.

1.1.5 Cooperating: yielding, spawning

cpc-statement	::=	`cpc_yield`;
	|	`cpc_done`;
	|	`cpc_spawn` statement

`cpc_yield` The `cpc_yield` statement causes the current continuation to be suspended, and placed at the end of the queue of runnable continuations. Control is passed back to the CPC main loop. This statement is only allowed in cps context.

`cpc_done` The `cpc_done` statement causes the current continuation to be discarded, and control to be passed back to the main CPC loop. This statement is only allowed in cps context.

`cpc_spawn` The `cpc_spawn` statement causes a new continuation that executes the argument to `cpc_spawn` to be created and placed at the end of the queue of runnable continuations. Execution then proceeds after the `cpc_spawn` statement (control is *not* passed back to the main CPC loop). This statement is valid in arbitrary context.

1.1.6 Synchronisation: condition variables

cpc-statement ::= `cpc_wait`(expression);

```
typedef struct cpc_condvar cpc_condvar;
void cpc_signal(cpc_condvar *);
void cpc_signal_all(cpc_condvar *);
```

`cpc_wait` The `cpc_wait` statement places the current continuation on the list of continuations waiting on the condition variable passed as argument to `cpc_wait`. Control is passed back to the CPC loop. This statement is only valid in cps context.

`cpc_signal` The function `cpc_signal` causes the first of the continuations waiting on the condition variable passed as argument to be moved to the tail of the queue of runnable continuations. Execution proceeds at the instruction following the call to `cpc_signal`.

`cpc_signal_all` The function `cpc_signal_all` causes all of the continuations waiting on the condition variabled passed as argument to be moved to the tail of the queue of runnable continuations. This function guarantees that the continuations will be run in the order in which they were suspended.

1.1.7 Sleeping

cpc-statement ::= `cpc_sleep`(expression[, expression[, expression]]);

cpc_sleep The statement `cpc_sleep` takes three arguments: a time in seconds, a time in microseconds, and a condition variable. It causes the current continuation to be suspended until either the specified amount of time has passed, or the condition variable is signalled, whichever happens first.

The third argument can be omitted if no interruption is necessary. The second argument can be omitted if sub-second accuracy is not needed.

This statement is only valid in cps context.

1.1.8 Waiting for I/O

cpc-statement ::= `cpc_io_wait`(expression, expression[, expression]);

cpc_io_wait The statement `cpc_io_wait` takes three arguments: a file descriptor, a direction, and a condition variable. The direction can be one of `CPC_IO_IN`, meaning input, or `CPC_IO_OUT`, meaning output.

This statement causes the current continuation to be suspended until either the given file descriptor is available for I/O in the given direction, or the given condition variable is signalled, whichever happens first.

This statement is only valid in cps context

1.1.9 Interaction with native threads

cpc-statement ::=
- `cpc_detach`;
- `cpc_attach`;
- `cpc_detached` statement
- `cpc_attached` statement

A continuation can be scheduled to be run by a native thread; intuitively, the continuation "becomes" a native thread. When this happens, we say that the continuation has been *detached* from the CPC scheduler. The opposite operation is known as *attaching* a detached continuation to the CPC scheduler.

cpc_detach, cpc_attach The `cpc_detach` statement detaches the current continuation; the following statements are executed in a dedicated native thread. The opposite operation is performed by `cpc_attach`, which causes the current continuation to be scheduled by the CPC scheduler.

The `cpc_detach` and `cpc_attach` statements can only appear in cpc context.

cpc_detached, cpc_attached The body of a `cpc_detached` statement is run detached : an implicit `cpc_detach` is executed upon entering the body, and a `cpc_attach` is executed upon exiting. The `cpc_attached` construct is dual : a `cpc_attach` is executed upon entry, and a `cpc_detach` is executed upon exit.

1.2 Limitations and implementation notes

Not all legal C code is allowable in CPC. Some of the limitations described below are fundamental to the implementation technique of CPC; others are just artefacts of the current implementation, and will be lifted in a future version.

1.2.1 Fundamental limitations

The use of the `longjmp` library function, and its variants, is not allowed in CPC code.

1.2.2 Current limitations

Old-style ("K&R") function definitions are not supported.

1.2.3 Time complexity of CPC operations

The current implementation of CPC implements all the CPS operations in constant time, with the following exceptions:

- at the end of every iteration of the main loop (running all the runnable continuations once), a `select` system call is made; this call runs in time proportional to the number of the highest active file descriptor;
- when a continuation is queued on two structures simultaneously (because of `cpc_sleep` or `cpc_io_wait` with a non-null last argument), invoking it requires dequeueing it from the second queue, which takes linear time in the worst case;
- the `cpc_sleep` instruction runs in worst-case time proportional to the number of currently sleeping continuations;

Chapter 2

The CPC library

The functions in the CPC library are themselves written in CPC, using only the primitives documented in Chapter 1.

All the functions in the CPC core library are declared in the file `cpc-lib.h`.

2.1 Barriers

```
typedef struct cpc_barrier cpc_barrier;

cpc_barrier *cpc_barrier_get(int count);
cps void cpc_barrier_await(cpc_barrier *barrier);
```

A barrier is a synchronisation construct that allows a set of continuations to be woken up at the same time. A barrier is conceptually a queue of continuations and a count of continuations remaining to wait for.

`cpc_barrier_get` The function `cpc_barrier_get` returns a new barrier initialised to wait for `count` continuations.

`cpc_barrier_await` The function `cpc_barrier_await` causes the current continuation to wait on the barrier given in argument. This function first decrements the barrier's count; if the count reaches zero, it wakes up all of the continuations waiting on the barrier. Otherwise, it suspends the current continuation.

The function `cpc_barrier_await` guarantees that the continuations are run in the order in which they were suspended.

2.2 Input/Output

2.2.1 Setting up file descriptors

```
int cpc_setup_descriptor(int fd, int nonagle);
```

The function `cpc_setup_descriptor` sets up the file descriptor `fd` into non-blocking mode, making it suitable for use by the CPC runtime. If `nonagle` is

true (non-zero), the descriptor is assumed to refer to a socket and has the Nagle algorithm disabled (the socket option `TCP_NODELAY` is set).

This function returns 1 in case of success, -1 in case of failure.

2.2.2 Input/Output

```
cps int cpc_write(int fd, void *buf, size_t count);
cps int cpc_write_timeout(int fd, void *buf, size_t count,
                          int secs, int micros);
cps int cpc_read(int fd, void *buf, size_t count);
cps int cpc_read_timeout(int fd, void *buf, size_t count,
                         int secs, int micros);
```

The functions `cpc_write` and `cpc_read` are CPC's versions of the `write` and `read` system calls. They return the number of octets read/written in case of success; in case of failure, they return -1 with `errno` set.

The versions with `timeout` appended return after `secs` seconds and `micros` micreseconds if no I/O has been possible. In this case, they return -1 with `errno` set to `EAGAIN`.

Bibliography

[ISO99] Information technology — programming language C. International standard ISO/IEC 9899:1999, 1999.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975. Also published as Memorandum SAI–RM–6, School of Artificial Intelligence, University of Edinburgh, Edinburgh, 1973.

[SW74] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG–11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974.