

# Continuation Passing for C

## A space-efficient implementation of concurrency

Juliusz Chroboczek

PPS

Université de Paris 7

jch@pps.jussieu.fr

### Abstract

Threads are a convenient abstraction for programming concurrent systems. They are however expensive, which leads many programmers to use coarse-grained concurrency where a fine-grained structure would be preferable, or use more cumbersome implementation techniques.

Cooperative threads can be easily implemented in a language that provides *first-class continuations*. Unfortunately, CPS conversion, the standard technique for adding continuations to a language, is not natural for typical imperative languages. This paper defines a notion of CPS conversion for the C programming language.

Continuation Passing C (CPC) is a concurrent extension of C with very cheap threads. It is implemented as a series of source-to-source transformations, including CPS conversion, that convert a threaded program in direct style into a purely sequential C program. In this paper, we describe CPC and the transformations that are used in its implementation.

### Introduction

*Threads*, or *lightweight processes*, are a convenient abstraction for programming concurrent systems. Unfortunately, typical implementations use a fair amount of memory for every thread, causing the programmer to either use a small number of complex threads where a large number of simple ones would be a better way of structuring his program, or use more cumbersome programming techniques, such as *event loops*.

The functional programming community has been using *first-class continuations* to implement lightweight threads. In some functional programming languages (notably Scheme [18, 11] and SML/NJ [1]), first class continuations are a primitive construct; in others, they may be implemented using a source-to-source transformation called *conversion to Continuation Passing Style* [19, 15] (*CPS conversion* for short).

Many imperative programming languages, however, including C, do not contain enough of the  $\lambda$ -calculus for CPS conversion to be natural, and this sort of techniques are therefore mostly confined to the functional programming community.

This paper introduces *Continuation Passing C* (CPC), a concurrent extension to the C programming language [9] that provides

very cheap threads. CPC is implemented as a series of well-known source-to-source transformations that start with a multi-threaded program written in direct style and, after CPS conversion, end in a pure C program. The major contributions of this work are the definition of a direct notion of CPS conversion for a subset of the C programming language, and the definition of series of transformations that convert an arbitrary C program into an equivalent program in the CPS-convertible subset. Together, these two techniques provide a notion of CPS conversion for the whole C language (see, however, the limitations of this technique in Sec. 6.1).

This paper is structured as follows. In Sec. 1, we outline a few of the known techniques used for implementing concurrency. In Sec. 2, we introduce conversion to Continuation Passing Style (CPS conversion), the fundamental program transformation technique used in CPC. In Sec. 3, we describe the syntax and main features of CPC. Sections 4 and 5 describe the transformations performed by CPC; the former shows CPS conversion itself, while the latter shows the transformations that prepare for CPS. In Sec. 6, we describe the limitations of our algorithms and some future directions; and in Sec. 7, we outline relationship between the techniques used in CPC and event-loop programming, a manual technique for writing highly efficient concurrent programs. Finally, in Sec. 8, we give the results of a few benchmarks that measure the performance of CPC.

## 1. Programming concurrent systems

Programming is sometimes seen as the task of writing *batch* programs. A batch program proceeds by reading a finite amount of input data, performing some computations thereon, then producing a finite amount of result data and terminating.

Many programs, however, are *concurrent* systems — they interact with their environment over multiple channels throughout their execution. That may be because they need to interact with a human user (perhaps through a graphical user interface), or because they need to interact with other programs (perhaps through a network).

A number of techniques have been proposed for writing concurrent programs. The most common one is to divide a program into a number of independent threads of control, usually called *processes* or simply *threads*, that communicate through message passing or shared memory. Such threads can be *cooperatively scheduled*, in which case passing of control from one thread to another is explicit, or *preemptively scheduled*, in which case passing of control is done asynchronously, when some external agent (usually the operating system) decides so.

Another technique is called *event-loop programming*. An event-loop program interacts with its environment by reacting to a set of stimuli called *events*. At any given point in time, to every event is associated a piece of code known as the *handler* for this event. A

[copyright notice will appear here]

global scheduler, known as the *event loop*, repeatedly waits for an event to occur and invokes the associated handler.

Thread programming has the advantage of making the flow of control of every thread explicit, which allows threaded programs to have a relatively transparent structure. This is not the case for event-loop programs, where any given activity that the program must perform is distributed across a potentially large number of event handlers.

Ideally, every thread in a threaded program would consist of a small amount of code having a well-defined role and communicating with just a few other threads. Typical implementations of threads, however, use a distinct hardware stack for every thread, which makes threads rather expensive. This cost causes threaded programs to be written as a few large threads interacting with their peers in complex and wonderful ways.

### 1.1 Cheap cooperative user-space threads

This author believes that a convenient system for programming concurrent systems should have the following properties:

1. be based on cooperatively scheduled contexts;
2. provide very cheap contexts;
3. not require the programmer to perform a complex transformation that hides the flow of control.

We will examine these properties in turn.

**Cooperative scheduling** The difference between programming with preemptively and cooperatively scheduled contexts can be summarised as follows: in a preemptive system, the programmer must make all cases of mutual exclusion (critical sections) explicit; to the contrary, in a cooperative system, it is points of cooperation that have to be made explicit. In most concurrent systems, cooperation happens mainly during I/O and when synchronising between contexts (e.g. when waiting on a condition variable or a message queue); given the right libraries, very few explicit cooperation points need to be inserted. Additionally, we note that a missing critical section can lead to race conditions that are difficult to reproduce; a missing cooperation primitive, on the other hand, results in an obvious deadlock or performance problem, which can usually be reproduced in a deterministic manner and easily diagnosed. (See, however, Sec. 6.2 about cases where native threads cannot be avoided.)

**Cheap contexts** Just like it is essential that function call should be sufficiently cheap for the programmer to use functions as the fundamental tool for structuring his programs and not merely as a way to factor common code, execution contexts should be sufficiently cheap for the programmer to create as many as he needs for every context to play a well-defined role. Coarse-grained concurrency requires complex sharing of data between multiple contexts, leading to complex protocols for modifying shared data structures. With fine-grained contexts, every context manipulates at most two or three pieces of data, which it shares with at most one other thread each, leading to a dramatic simplification of the behaviour of the program.

**No complex transformations** From the above two considerations, it is sometimes concluded that threads should be avoided as the main abstraction for concurrent programming, and an event loop should be used instead [14]. Most of the arguments in favour of event loops, however, are really arguments in favour of cooperative scheduling and of cheap execution contexts [20]; not coincidentally, just the arguments outlined above.

The use of an event loop imposes on the programmer a complex code transformation which hides the control flow of individual execution contexts in a maze of interaction between event handlers

(we discuss the transformations involved in Sec. 7). I believe that such a transformation is better done by a computer program than by hand.

**Cooperatively scheduled threads** From this, it would seem that a good abstraction that a concurrent programming system can offer to the programmer are cooperatively scheduled threads. This should not be taken to imply that these threads should have an actual representation at runtime; and, in fact, CPC doesn't keep such thread information.

## 1.2 Implementation techniques

A number of researchers have been working on ways to provide cheap cooperative threads. This section aims at outlining a few of these techniques.

### 1.2.1 User-space thread libraries

The most common way of providing cooperative threads is with a user-space thread library to be used by a stock compiler. Unfortunately, such implementation do not provide cheap threads : as the compiler is thread-agnostic, every thread needs to be equipped with a hardware stack, leading to the usage of at the very least one page of physical memory per thread, as well as vast swathes of virtual space.

### 1.2.2 Ad hoc interpreters

The simplest way of providing cheap threads is to write an interpreter especially structured for making threads cheap [6]. In practice, this implies encapsulating all of the information specific to an execution context in a single data structure, so that a context switch requires saving and restoring a single pointer.

The trouble with this technique is that it only produces interpreters, and, because of the extra indirection required to access any context-specific data, not very fast ones at that. Thus, a program written in such a system will typically consist of two parts: the concurrent part, which is written in the *ad hoc* language, and the compute-bound part, which is written in a traditional compiled sequential language.

### 1.2.3 Ad hoc compilation

A more recent trend is to compile code in a manner designed especially for concurrent programs. This approach holds the promise to provide cheap threads while avoiding the inherent problems of interpretative techniques.

**Whole program transformation** Concurrency is not parallelism, and a concurrent program can be converted to a purely sequential one by doing a whole program transformation. An interesting example of this technique is the programming language Squeak [5] (not to be confused with the Smalltalk dialect of the same name), the expressivity of which is carefully limited so as to be possible to translate into a finite-state automaton which is then used to generate C code that is processed by a standard compiler.

**Traditional thread libraries and static analysis** At the other extreme lies the idea to use a mostly standard thread library, but minimise the cost of threads by performing static analysis to conservatively estimate the amount of space needed for stacks.

It is not clear to me how well this approach works. Static analysis tends to be complex task and its results fragile, in the sense that small changes to the analysed program may lead to wide variation in the results of a conservative analysis. Still, this is without doubt a promising direction.

**Continuation passing** The standard hardware implementation of call stacks being the problem, it is natural to want to represent the

dynamic chain information in a more compact form. Continuation Passing (Sec. 2) is a technique that encapsulates this information in an abstract data structure, the continuation; and, with a suitably designed language, continuation passing can be implemented as a purely local transformation. Depending on the extent to which their expressivity is restricted, continuations can cost as little as a few bytes.

One example of this kind of technique is the Cilk programming language. The initial version of Cilk [8] used fully general continuations, but required the programmer to perform a complex transformation to make continuations explicit himself. Cilk 5 [7] uses a different technique, based on call-by-name invocation of processes, which allows writing natural code but severely restricts the contexts in which process creation can happen.

CPC is a concurrent programming system based on continuation passing. By using a fully general transformation into continuation-passing style, CPC allows arbitrary concurrent code to be compiled efficiently.

## 2. CPS conversion

*Conversion into Continuation Passing Style* [19, 15], or *CPS conversion* for short, is a program transformation technique that makes the flow of control of a program completely explicit.

Intuitively, the *continuation* of a fragment of code is an abstraction of the action to perform after its execution. CPS conversion consists in replacing every function  $f$  in a program with a function  $f^*$  taking an extra argument, its *continuation*. Where  $f$  would return with value  $v$ ,  $f^*$  invokes or *resumes* its continuation with the argument  $v$ .

A CPS-converted function therefore never returns, but makes a call to a continuation. As all of these calls are in tail position, a converted program doesn't use the native call stack; the information that would normally be in the call stack (the dynamic chain) is encoded within the continuation.

The usual presentation of the CPS conversion [15] is as follows:

$$\begin{aligned} a^* &= \lambda k.ka \\ x^* &= \lambda k.kx \\ (\lambda x.M)^* &= \lambda k.k(\lambda x.M^*) \\ (MN)^* &= \lambda k.M^*(\lambda n.N^*(\lambda n.mnk)) \end{aligned}$$

where  $a$  is a constant,  $x$  a variable, and  $M$  and  $N$  are arbitrary  $\lambda$ -terms.

In a first order language, such as C only constants and variables can appear on the left hand side of an application. With this simplifying hypothesis, the call-by-value CPS conversion becomes (up to  $\beta$ -reduction):

$$\begin{aligned} a^* &= \lambda k.ka \\ x^* &= \lambda k.kx \\ (\lambda x.M)^* &= \lambda k.k(\lambda x.M^*) \\ (fN)^* &= \lambda k.N^*(\lambda n.fnk) \end{aligned}$$

where  $f$  is a constant or a variable.

This translation has three interesting properties, which are part of the continuation-passing folklore.

**CPS conversion need not be global** The above definition would seem to imply that CPS conversion is an “all or nothing” deal, and that the complete program must be converted. This is in fact not the case: there is nothing preventing a converted function from calling a function that has not been converted. On the other hand, a function that has not been converted cannot call a function that has, as it does not have a handle to its own continuation. (This kind of restriction is present in all constructs based on monadic translations, for example

the IO monad of the Haskell programming language or the `cilk` monad of Cilk [7].)

It is therefore possible to perform CPS conversion on just a subset of the functions constituting a program, as long as the above restriction is obeyed. This allows CPS-converted code to call native code, for example system calls or standard library functions. Additionally, at least in the case of CPC, a CPS function call is much slower than a native function call; being able to only convert the functions that need the full flexibility of continuations avoids this overhead most of the time.

**Continuation transformers are linear** Continuations are manipulated linearly [2]: when a CPS-converted function receives a continuation, it will use it exactly once, and never duplicate or discard it.

This property is essential for memory management in CPC: as CPC uses the C allocator (`malloc` and `free`) rather than a garbage collector for managing continuations, it allows reliable reclaiming of continuations without the need for costly devices such as reference counting.

**Continuations are abstract data structures** At first sight, continuations are functions. Careful examination of the CPS conversion process shows, however, that the only operations that are ever performed on continuations are calling a continuation, which we call *resume*, and prepending a function application to the body of a continuation, which we call *push*. Thus, continuations are abstract data structures, of which functions are one particular concrete representation. In that representation, the two operations have the following form:

$$\begin{aligned} \text{resume}(k, x) &= kx \\ \text{push}(m, k) &= \lambda n.mnk \end{aligned}$$

This property is not really surprising: as continuations are merely a representation for the dynamic chain, it is only natural that the operations that are performed on a continuation should roughly correspond to those that can be performed on a stack.

As C doesn't have full first-class functions (closures), CPC uses this property to implement continuations as arrays.

## 3. The CPC language

CPC is a conservative extension of the C programming language [9]. In other words, every C program is a CPC program, and has the same meaning in the two languages; additionally, there is nothing a CPC program can do with a pure C function that cannot be done by a C program.

At the centre of the implementation is the CPC scheduler. The scheduler manipulates three data structures: a queue of runnable continuations, which are resumed in a round-robin fashion; a priority queue of continuations waiting on a timeout, and an array of queues of continuations waiting on I/O, one per active file descriptor.

### 3.1 Basic flow of control

The main addition that CPC makes to C is the “function qualifier” `cps`. Intuitively, a function that was declared `cps` is interruptible, meaning that a context switch can happen in its body, while a C function is uninterruptible (executed “atomically”).

Cooperation between threads is achieved by the `cpc_yield` statement, which causes the current continuation to be suspended, and the next runnable continuation in the scheduler's queue to be resumed.

A new thread is created with the `cpc_spawn` statement, which takes the form

`cpc_spawn s`

where  $s$  is an arbitrary statement. Execution of `cpc_spawn` does not in itself suspend the current continuation; it merely causes the continuation associated to  $s$  to be enqueued in the scheduler's queue of runnable continuations.

Every location in a CPC program is said to be in *native C context* or in *CPS context*. The set of CPS contexts is defined as follows:

- the body of a `cps` function is in CPS context; and
- the argument of a `cpc_spawn` statement is in CPS context.

With the single exception of `cpc_spawn`, which is allowed in any context, CPC statements and calls to `cps` functions are only allowed in CPS context.

A consequence of the above restriction is that a `cps` function can only ever be called by another `cps` function.

### 3.2 Condition variables

CPC introduces a new type, `cpc_condvar`, which holds condition variables. There is one CPC statement for the manipulation of condition variables, `cpc_wait`, which suspends the calling thread on the condition variable given as its argument. The effect of this statement is to enqueue the current continuation on the condition variable  $c$  and pass control to the next runnable continuation.

The functions `cpc_signal` and `cpc_signal_all` can be used for waking up continuations enqueued on a condition variable.

### 3.3 Waiting on an external event

There are two kinds of external events that a CPC program can wait for: a timeout or a file descriptor becoming ready for I/O.

Waiting for a timeout is done with the primitive `cpc_sleep`, which takes the amount of time to wait for. Waiting for I/O is done using `cpc_io_wait`, which takes a file descriptor and a direction.

The waiting primitives in CPC can wait on a single event; in order to wait on one of a set of events, the programmer needs to create multiple threads. When a waiting operation is thus split into multiple threads, and one of the waiters is woken up, it is necessary to wake up the remaining waiters; the waiting primitives therefore take an optional supplementary argument, a condition variable that will cause the wait to be interrupted if it is signalled.

### 3.4 Other features

The current implementation of CPC also includes two other minor features. The statement `cpc_done` discards the current continuation, in effect killing the running thread. Conversely, the statement `cpc_fork` duplicates the current continuation, in effect forking the running thread.

Note that these instructions manipulate continuations non-linearly, which apparently violates one of the properties of Sec. 2. The need for a refined memory allocator is avoided by performing a deep copy of the current continuation in `cpc_fork`.

### 3.5 Bootstrapping

When a CPC program is started, the native function `main` is called by the C runtime. As a native function cannot call a `cps` function, some means is needed to pass control to CPS context.

Calling the function `cpc_main_loop` passes control to the CPC scheduler, which returns when the queues of runnable, sleeping and waiting continuations are all empty. The `main` function of a CPC program typically consists of a number (possibly one) of invocations of `cpc_spawn` followed by a call to `cpc_main_loop`.

### 3.6 Example

The following is a complete program in CPC. It behaves like the Unix command `cat`, copying data from its input to its output, but times out after one second.

Two threads are used: one does the actual input/output, while the other one merely sleeps for a second. The two threads communicate through the condition variable  $c$  and the boolean variable `done`. (This example is in fact very slightly incorrect, as it does not deal correctly with partial writes and interrupted system calls.)

```
#include <unistd.h>

char buf[512];
int done = 0;

int
main()
{
    cpc_condvar *c;
    c = cpc_condvar_get();

    cpc_spawn {
        while(1) {
            int rc;
            cpc_io_wait(0, CPC_IO_IN, c);
            if(done) break;
            rc = read(0, buf, 512);
            if(rc <= 0) break;
            cpc_io_wait(1, CPC_IO_OUT, c);
            if(done) break;
            write(1, buf, rc);
        }
        cpc_signal(c);
    }

    cpc_spawn {
        cpc_sleep(1, 0, c);
        done = 1;
        cpc_signal(c);
    }

    cpc_main_loop();
    cpc_condvar_release(c);
    return 0;
}
```

The program above contains a number of common idioms (notably cooperation on message passing, implemented with a condition variable and a shared boolean). A library for CPC that encapsulates a number of such idioms (cooperation on I/O, cooperation on message passing, barriers, etc.) is currently being implemented.

## 4. CPS conversion for C

Consider a function `cps void f() {...}` that is to be CPS-converted into a function `void f*( $\kappa$ ) {...}`. If the body of  $f$  is just `return`, then the body of  $f^*$  is just an invocation of  $\kappa$ . If the body of  $f$  is just a call to another CPS-converted function  $g$ , then the body of  $f^*$  is just a call to  $g^*(\kappa)$ .

Suppose now that the body of  $f$  is two calls in sequence, `g(); h();`; then the continuation of the call to  $g()$  consists of a call to  $\kappa' = h^*(\kappa)$ , and hence the body of  $f^*$  is a call to  $g^*(\kappa')$ . This obviously generalises to an arbitrary number of function calls.

While a  $\lambda$ -term consists of just function calls, this is not the case of a C program, which contains relatively complex control structures. Thus, in order to CPS-convert an arbitrary C function,

we proceed as follows: we first convert the program to an equivalent program in an “easy” form, which we subsequently CPS-convert.

We first describe the CPS conversion itself. The algorithm for conversion to the “easy” CPS-convertible form is described in Sec. 5.

#### 4.1 CPS-convertible form

Let  $\text{cps } f(\dots)\{A\}$  be a function. We say that such a function is in *CPS-convertible form* when, up to some non-essential details, its body  $A$  consists of a sequence  $A_0$  of C code containing no return statements followed with a straight-line sequence  $A_1$  of cps function calls.

More precisely, we define by simultaneous induction two notions of CPS-convertible form: (void-)CPS-convertible form, which can be converted to a continuation of no arguments, and, for any variable  $v$ ,  $v$ -CPS-convertible form which can be converted to a continuation of one argument bound to the variable  $v$ . The definition is as follows:

- if  $A$  is in  $v$ -CPS-convertible form for some  $v$ , then  $A$  is in CPS-convertible form;
- **return**; is in CPS-convertible form;
- **return**  $v$ ; is in  $v$ -CPS-convertible form;
- if  $A'$  is in CPS-convertible form, then

$$A = f(x_1 \dots x_n); A',$$

where  $f$  is a cps function, is in CPS-convertible form; additionally, if  $n \geq 1$ , then  $A$  is in  $x_n$ -CPS-convertible form;

- if  $A'$  is in  $v$ -CPS-convertible form, then

$$A = v=f(x_1 \dots x_n); A',$$

where  $f$  is a cps function, is in CPS-convertible form; additionally, if  $n \geq 1$ , then  $A$  is in  $x_n$ -CPS-convertible form.

CPS-convertible form can be seen as a version of A-normal form [13] that doesn't rely on lexical scoping to propagate values, but instead passes all values as explicit function parameters.

A function  $\text{cps } f(\dots)\{A\}$  is in CPS-convertible form if it is of the form

```

cps f(⋯)
{
  A0;
  if (e1) { A1; B1 }
  else if (e2) { A2; B2 }
  ...
  else { An; Bn }
}

```

where the  $A_i$  are sequences of pure C statements containing no return statement, and the  $B_i$  are in CPS-convertible form.

#### 4.2 Continuations

A *closure* is a pair  $(f', x_1 \dots x_n)$ , where  $f'$  is a function pointer, and  $x_1 \dots x_n$  is a tuple of values. A (concrete) *continuation* is a sequence of closures.

Intuitively,  $f'$  is the image by CPS conversion of a function  $f$ , and the continuation  $\kappa = ((f', x_1 \dots x_n) \cdot \kappa')$  represents the function

$$\lambda(y_1 \dots y_k).f'(x_1 \dots x_n, y_1 \dots y_k, \kappa')$$

where  $n + k + 1$  is the number of arguments of  $f'$ , or, equivalently,  $n + k$  is the number of arguments of  $f$ ; in particular, if  $f$  takes exactly  $n$  arguments, then  $\kappa$  represents a function of no arguments. This interpretation leads to the following definition of the

operations on continuations:

$$\begin{aligned} \text{push}(f, x_1 \dots x_n, \kappa) &= (f, x_1 \dots x_n) \cdot \kappa \\ \text{resume}((f, x_1 \dots x_n) \cdot \kappa, y_1 \dots y_k) &= \\ &f(x_1 \dots x_n, y_1 \dots y_k, \kappa) \end{aligned}$$

#### 4.3 CPS conversion

We are now ready to define the CPS conversion itself. This is not a direct translation of the one defined in Sec. 2. The most obvious difference is that C functions take multiple arguments, and are not identified with their curried equivalents; thus, the conversion must take into account the number of arguments taken by a function.

Perhaps more subtly, it produces results that are in  $\eta$ -reduced form, or, equivalently, it performs elimination of tail calls on the fly. The implications of this choice are explored further in Sec. 6.1.

We define two notions of CPS conversion, which correspond closely to the two notions of CPS-convertible form. The map  $\cdot^*$  maps a CPS-convertible sequence of statements to a continuation of no arguments; the map  $\cdot^{**}$  maps a  $v$ -CPS-convertible sequence to a continuation of a single argument. (This scheme could in principle be generalised to continuations of multiple arguments, allowing the implementation of functions returning multiple values.)

The definition is as follows:

- if  $A$  is **return**;, then  $A^*$  is  $\text{resume}(\kappa)$ ;
- if  $A$  is **return**  $v$ ;, then  $A^*$  is  $\text{resume}(\kappa, v)$  and  $A^{**}$  is  $\text{resume}(\kappa)$ .
- if  $A$  is  $f(x_1 \dots x_n); B$ , then

$$A^* = \text{push}(\kappa, f, x_1 \dots x_n); B^*,$$

and

$$A^{**} = \text{push}(\kappa, f, x_1 \dots x_{n-1}); B^{**};$$

- if  $A$  is  $y = f(x_1 \dots x_n); B$ , then

$$A^* = \text{push}(\kappa, f, x_1 \dots x_n); B^{**},$$

and

$$A^{**} = \text{push}(\kappa, f, x_1 \dots x_{n-1}); B^{**}.$$

Let now  $\text{cps } f(x_1 \dots x_k)\{A\}$  be a function in CPS-convertible form. By the definition in Sec. 4.1,  $f$  is of the form

```

cps f(x1 ... xk)
{
  A0;
  if (e1) { A1; B1 }
  else if (e2) { A2; B2 }
  ...
  else { An; Bn }
}

```

where the  $A_i$  are plain C code, and the  $B_i$  are in CPS-convertible form. The CPS conversion of  $f$  is a function

```

cps f'(x1 ... xn, κ)
{
  A0;
  if (e1) { A1; B1* }
  else if (e2) { A2; B2* }
  ...
  else { An; Bn* }
}

```

where  $B_i^*$  is the CPS conversion of  $B_i$  with respect to  $\kappa$ .

## 5. Transformation into CPS-convertible form

In the previous section, we have shown how to perform a CPC conversion for the CPS-convertible subset of CPC. In this section, we show how every CPC program can be transformed into a semantically equivalent program that is in that particular form.

We give two algorithms for conversion into CPS-convertible form. The *idealised algorithm*, presented in Sec. 5.1, is fairly easy to explain, but produces code that is very inefficient. The algorithm actually used by CPC is described in Sec. 5.2.

### 5.1 Transformation into CPS-convertible form: idealised algorithm

The idealised algorithm is structured a a sequence of steps. We show the operation of every step on the following CPC function:

```
cps void g() {
    int i;
    i = 0;
    while(1) {
        f(i);
        i++;
    }
}
```

where *f* is a cps function.

#### 5.1.1 Making the flow of control explicit

In this step, we ensure that any internal part of an expression has only trivial flow of control. Given any statement of the form

```
s[e[e']]
```

where *e'* is either a CPC function call or an expression with side effects, we replace it with

```
x = e';
s[e[x]]
```

where *x* is a fresh variable of the right type.

This step has no effect on our example.

#### 5.1.2 Elimination of control structures

In this step, loops (*while*, *do...while* and *for*) and case constructs are replaced with their equivalents in terms of *gotos*. After this step, our example has the following form:

```
cps void g() {
    int i;
    i = 0;
    1: f(i);
    i++;
    if(1) goto 1;
}
```

In order to simplify the rest of the exposition, we “cheat” by simplifying this snippet to the following equivalent code:

```
cps void g() {
    int i;
    i = 0;
    1: f(i);
    i++;
    goto 1;
}
```

#### 5.1.3 Making branch continuations explicit

In this step, we add an explicit *goto* to any conditional branch that doesn't end in either *goto* or *return*. Thus, a conditional such as

```
if(e) { s1; } else { s2; }
```

becomes

```
if(e) { s1; goto 1; } else { s2; goto 1; }
1: ;
```

where *1* is a fresh label.

This step has no effect on our example.

#### 5.1.4 Elimination of *gotos*

It is a well-known fact in the compiler folklore that a tail call is equivalent to a *goto*. It is less well-known that a *goto* is equivalent to a tail call [18].

This step replaces all *gotos* with the equivalent tail calls. This transformation is obviously only valid in systems that implement elimination of tail calls, which, as we noted in Sec. 4.3, is the case in CPC.

After this step, our example looks as follows:

```
cps void g() {
    int i;
    i = 0;
    cps void l() { f(i); i++; l(); return; }
    l();
    return;
}
```

Note that the function *l* introduced in this step has a free variable *i*, which C does not allow (unlike ALGOL-60, Pascal and most functional languages). This “inner function” will be eliminated during  $\lambda$ -lifting (Sec. 5.1.6).

#### 5.1.5 $\beta$ -expansion in tail position

At this point, the function *g* is in CPS-convertible form; *l*, however, is not. We put *l* into CPS-convertible form by encapsulating the final fragment into a new function *h* ( $\beta$ -expanding this final fragment), and iterating this process until all functions are in CPS-convertible form:

```
cps void g() {
    int i;
    i = 0;
    cps void h() { i++; l(); return; }
    cps void l() { f(i); h(); return; }
    l();
    return;
}
```

#### 5.1.6 $\lambda$ -lifting

All that remains to be done at this point is to eliminate inner functions. A standard technique for doing that is  $\lambda$ -lifting [10].  $\lambda$ -lifting consists in making all free variables into explicit arguments, and adding an extra argument to all calls to the lifted function.

$\lambda$ -lifting, however, is only correct in a call-by-name language. In a call-by-value imperative language, it has the effect of creating a copy of the free variable, which makes any mutation performed by the inner function invisible to the outer one. Consider for example the following code:

```
cps void f() {
    int i;
    cps void g() { i++; }
    i = 1;
    g();
    printf("%d\n", i);
    return;
}
```

After  $\lambda$ -lifting, this becomes

```
cps void g(i) { i++; }
cps void f() {
    int i;
    i = 1;
    g(i);
    printf("%d\n", i);
    return;
}
```

which prints 1 rather than 2.

There is one case, however, in which  $\lambda$ -lifting is correct in call-by-value: if a function is only ever called in tail position, the changes it performs to the values of its free variables can never be observed; it may therefore be lifted with no change to the semantics of the program. As all the inner functions that we introduced (during elimination of *gotos* and  $\beta$ -expansion) are only called in tail position,  $\lambda$ -lifting is correct in our case.

After  $\lambda$ -lifting, our example has the following form:

```
cps void h(int i) { i++; l(i); return; }
cps void l(int i) { f(i); h(i); return; }
cps void g() {
    int i;
    i = 0;
    l(i);
    return;
}
```

## 5.2 Conversion into CPS-convertible form: actual algorithm

The algorithm described in Sec. 5.1 performs more transformations than strictly necessary. Indeed, it starts by converting every single loop into a set of *gotos* (Sec. 5.1.4), and then converts all of those *gotos* into expensive, CPS function calls. The overhead of such a coarse transformation is probably non-negligible.

For this reason, the CPC translator uses a more careful algorithm: it only performs those transformations that are necessary in order to arrive to a CPS-convertible form. More precisely, a loop is converted into *gotos* only when it contains CPS function calls or *return* statements. Similarly, a *goto* is converted to a function call when it follows a CPS function call. As this introduces new CPS function calls, which produce new opportunities for the other transformations, all the steps are intermixed in an iterative process.

The CPC translator is thus structured into just three steps:

1. conversion into CPS-convertible form;
2.  $\lambda$ -lifting;
3. CPS conversion.

Step (1) invokes on an as-needed basis all of the transformations in Sec. 5.1. As to step (2), it might encounter not only free variables, as in usual  $\lambda$ -lifting, but also *free* or *escaping gotos*, *gotos* to a label that is in the enclosing function. Thus, elimination of *gotos* can be invoked again by step (2).

A further efficiency improvement is obtained by inlining ( $\beta$ -reducing) functions that either immediately return to their caller or merely call another function after permuting their arguments; such functions are often introduced by the process described in Sec. 5.1.3. This inlining happens after step (2).

## 6. Limitations and further extensions

### 6.1 Limitations

While our aim is to allow any reasonable C construct in CPS context, this is not quite the case in the current implementation of

CPC. A number of limitations of our scheme of translation have been identified.

C's facility for non-local transfer of control, *longjmp* and its variants, cannot be used in a CPC program. While the continuation provided by CPC could be used to implement non-local transfer of control, an implementation would need to be able to capture all of the transfers escaping from a piece of C code to map them to CPC transfers; such "wildcard exception handlers", while available in most programming languages, cannot be implemented within portable C.

A more serious limitation is that it is currently impossible to reliably take the address of a local variable in CPS context. There are two reasons for that: first, due to  $\lambda$ -lifting, local variables may be duplicated; in effect, from the programmer's point of view, they "move around". Second, local variables may vanish due to the elimination of tail calls performed by the CPS conversion step; and, as we noted in Sec. 5.1.4, conversion to CPS-convertible form relies on the fact that tail calls are eliminated.

The simple way of lifting this limitation would be to include an extra layer of indirection, in effect converting local variables to references to heap-allocated storage (perhaps using an *ad hoc* allocator). In order to avoid the related efficiency cost in cases where it is not necessary, we will need a simple pass of static analysis to conservatively determine the set of variables whose addresses may escape; C's ability to take the address of a field within a compound data structure (an array or a structure) makes this somewhat cumbersome.

CPC does not allow communication between threads through shared local variables. Indeed, if a local variable is in scope in the bodies of two distinct *cpc\_spawn* statements,  $\lambda$ -lifting will duplicate it, thus removing all sharing. While this limitation is consistent with the C language (which does not allow shared access to local variables due to the lack of inner functions), it might be convenient to have it lifted; that too could be done by accessing the shared variables through an additional indirection.

While there is no good reason for this restriction, we do not currently allow pointers to *cps* functions.

Finally, we do not allow *cps* functions to take a variable number of arguments. While the CPS conversion could in principle be generalised to such function, I do not currently feel that this minor limitation justifies the resulting increase in complexity.

### 6.2 Interaction with native threads

CPC provides a cooperatively scheduled user-space implementation of threads, and such a system is convenient for I/O bound applications. There are some cases, however, where using the system's native (preemptively scheduled) threads is necessary.

A cooperative threads package requires all interaction with the environment to go through non-blocking or asynchronous interfaces; most operating systems, however, still fail to provide non-blocking alternatives to all of their blocking interfaces. The most egregious case in the Unix environment is the interface to the network resolver (*gethostbyname* and *getaddrinfo*), but even something as simple as disk I/O cannot be done asynchronously in many implementations.

Native threads are also desirable when a program contains compute-bound sections: in a cooperative threads package, CPU-intensive code needs to be peppered with explicit calls to a co-operating primitive (*cpc\_yield*), which is cumbersome at best. In some environments, it may furthermore be possible for native threads to use more than one physical processor. (It may be argued, however, that on a system with relatively cheap processes such as Unix, full-fledged processes communicating through byte streams (pipes, sockets) might be a better solution to this kind of problem.)

One technique for dealing with such situations consists in allowing a cooperative thread to be *detached* from the cooperative scheduler and become a native thread [12]. When the blocking or CPU-intensive activity is over, the thread can be reattached to the cooperative scheduler. The Fair Threads library [17] expands on this scheme by allowing for multiple cooperative schedulers, each in its own native thread, between which threads are free to migrate. It is not yet clear to me to what extent the full generality of the latter scheme is necessary.

In a continuation-passing system such as CPC, such an abstraction can be implemented in a manner that is both simple and efficient. A pool of native threads communicate with the cooperative scheduler through two blocking queues of continuations: one queue of recently detached threads, and one queue of threads waiting to be reattached. In normal circumstances, both queues are empty, and a number of native threads are blocked on the first queue. When a cooperative thread needs to detach from the native scheduler, it enqueues its continuation on the first queue, which causes one of the native threads to be woken up; when a native threads needs to reattach, it enqueues its continuation on the second queue.

## 7. Relation with event loop programming

The code generated by the CPS transformation looks very much like event loop code. There is one major difference: where an event handler takes a single tuple of arguments, a function in CPS style takes a continuation, which is a complex structure consisting of multiple tuples of arguments and pointers to code.

Consider however the case of a CPC program in which all calls to cps functions occur in tail position; at runtime, all continuations passed in such a program consist of a single closure (a function pointer and a tuple of arguments), which corresponds exactly to an event handler. Any program in CPS-convertible form can be converted to such form by first eliminating all recursion [3] and then inlining all cps function calls that are not in tail position. Another way of achieving a similar result would be to apply a defunctionalising transformation [16] to continuations. Note that both schemes require global knowledge of the program being translated.

It is my guess that programmers writing event loop code have a mental model that is expressed in terms of threads, and apply some combination of the two techniques outlined above on the fly.

## 8. Experimental results

All the benchmarks described in this section were performed on a machine with an Intel Pentium 4 processor at 2.6 GHz with 512 MB of memory running Linux 2.6.8 with swap disabled.

I compared CPC with the following thread libraries:

- nptl, the native thread library in GNU libc 2.3 (I used the version from libc 2.3.5);
- LinuxThreads, the native thread library of earlier GNU libc releases (I used the version bundled with libc 2.3.5);
- GNU Pth version 2.0.1;
- State Threads (ST) version 1.5.1.

Nptl and LinuxThreads are kernel thread libraries, while GNU Pth and ST are cooperative user-space thread libraries.

**Micro-benchmarks** I wrote a number of benchmarks that were each aimed at measuring the performance of a single feature of CPC. The most important in my view is the ability of CPC to use massive numbers of threads; on a machine with 512 MB of physical memory and no swap space, CPC can handle up to 7.4 million continuations, implying an average memory usage of roughly 69 bytes per continuation. This figure compares very favourably to

both kernel and user-space thread libraries (see Fig. 1), which my tests have shown to be limited on the same system to anywhere from 250 to 30 000 threads in their default configuration, and to 99 000 threads at most after some tuning.

Time measurements, as shown in Fig. 2, yielded somewhat more mixed results. A loop with a single function call is roughly 100 times slower when the call is CPS-converted than when it is native. This rather disappointing result can be explained by two factors: first, due to the translation techniques used, after CPS conversion the loop consists of four CPS and four native function calls. The remaining factor of 20 can be attributed to the inefficiency on modern hardware of making an indirect function call, which prevents accurate branch prediction. The fact that CPS-converted functions are mostly opaque to the native compiler might also play a role.

The situation is much better when measuring the speed of the concurrency primitives. Both spawning a new continuation and switching to an existing one were measured as being five times faster than in the fastest thread library available to me.

nptl	250
LinuxThreads	1 533
Pth	70 000 (est.)
ST	29 000
ST (4 kB stacks)	99 300
CPC	7 390 000

All thread libraries were used in their default configuration, except where noted. Pth never completed, the value 70 000 is an educated guess.

**Figure 1.** Number of threads possible in various thread libraries

	call	cps-call	switch	cond	spawn
nptl	$4.3 \cdot 10^{-3}$		2.7	10.1	6.3
LinuxThr.	$4.3 \cdot 10^{-3}$		2.2	7.3	42
Pth	$4.3 \cdot 10^{-3}$		23.2		59
ST	$4.3 \cdot 10^{-3}$		2.7		3.6
CPC	$4.3 \cdot 10^{-3}$	0.33	0.57	0.67	0.78

All times are in microseconds. The columns are as follows:

call: native function call;

cps-call: call of a CPS-converted function;

switch: context switch;

cond: context switch on a condition variable;

spawn: thread creation.

**Figure 2.** Speed of various thread libraries

**Full-size benchmark** In the absence of reliable information on the relative dynamic frequency of CPS function calls and thread synchronisation primitives in CPC programs, it is difficult to estimate the performance of CPC programs from the above data. In order to experimentally determine this information, Boucher [4] has written a small web server in CPC. The server consists of 1700 lines of CPC code, more than half of which is devoted to the HTTP parser; all cooperation between contexts happens on input/output operations or on condition variables, with the exception of two explicit `cpc_yield` operations. In the presence of 1000 simultaneous connection attempts, his server appears to be five times faster than `thttpd`, a standard web server implemented in hand-written event-loop style (15 ms mean connection time for Boucher's web server, against 60 ms for `thttpd` when running on a Pentium-M processor downclocked to 600 MHz in order to avoid being limited by the client's performance). Boucher's web server has not yet been tested

with larger numbers of simultaneous clients due to a limitation of the current implementation of the CPC runtime.

## Conclusion

CPC is a concurrent extension of the C programming language that provides extremely cheap cooperatively scheduled threads. By combining in a novel way a dozen well-known source-to-source transformations, the CPC implementation provides a relatively clean semantics and very modest memory requirements, comparable to those of hand-written event-loop driven code.

CPC is not finished yet. In particular, the current implementation has a number of annoying limitations that need to be lifted. And while the little experience that we have had with programming in CPC seems to show that it is a pleasant language to program in, we need to write more complex programs in order to find out what its limitations might be.

The threads provided by CPC are purely cooperative user-space entities. In some situations, however, native threads are a necessary evil. CPC should be easy to extend to allow cooperative threads to become native ones.

**Software availability** The CPC translator is available from <http://www.pps.jussieu.fr/~jch/software/cpc/>.

## References

- [1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press. 1992.
- [2] J. Berdine, P. W. O'Hearn, U. S. Reddy, H. Thielecke. Linear Continuation Passing. *Higher-Order and Symbolic Computation*. 2002.
- [3] R. S. Bird. Notes on Recursion Elimination. *Communications of the ACM*, June 1977, **20**:6, pp. 434-439. 1977.
- [4] Édouard Boucher. *TuS, un serveur web en CPC*. Rapport de stage, École Nationale Supérieure des Télécommunications, Paris, France. 2005
- [5] L. Cardelli and R. Pike. Squeak: A language for communicating with mice. In B. A. Barsky, editor, *Computer Graphics (Proc. of SIGGRAPH'85)* **19**, pp. 199-204, July 1985.
- [6] James Gosling, David S. H. Rosenthal, Michele J. Arden. *The NeWS Book: an introduction to the Network/Extensible Window System*. Springer-Verlag. 1989.
- [7] Matteo Frigo, Charles E. Leiserson and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, Montreal, Canada. 1998.
- [8] Michael Halbherr, Yuli Zhou and Chris Joerg. MIMD-Style Parallel Programming Based on Continuation-Passing Threads. *Proceedings of the 2nd International Workshop on Massive Parallelism*, Capri, Oct. 1994.
- [9] ISO/IEC 9899:1999 Information technology — Programming Language C. 1999.
- [10] T. Johnsson. Efficient Compilation of Lazy Evaluation. *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction*. Montreal, 1984.
- [11] R. Kelsey, W. Clinger, J. Rees (eds.). Revised<sup>5</sup> Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, **11**:1, August 1998 and *ACM SIGPLAN Notices*, **33**:9, September 1998. 1998.
- [12] Simon Marlow, Simon Peyton Jones and Wolfgang Thaller. Extending the Haskell Foreign Function Interface with Concurrency. *Proceedings of the Haskell Workshop*, Snowbird, Sept 2004.
- [13] Atsushi Ohori. A Curry-Howard isomorphism for compilation and program execution. *Proceedings of TLCA'99*, Springer LNCS 1581, 258–179. 1999.
- [14] John Ousterhout. Why Threads Are A Bad Idea (for most purposes). Invited talk at the 1996 *USENIX Technical Conference*. 1996.
- [15] Gordon D. Plotkin. Call-by-name, Call-by-value and the  $\lambda$ -calculus. *Theoretical Computer Science* **1**. 1975.
- [16] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM National Conference*, pp. 717-740, August 1972.
- [17] M. Serrano, F. Boussinot and B. Serpette. Scheme Fair Threads. *Proceeding of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP)*. Verona, Italy, 2004.
- [18] Guy Lewis Steele, Jr. and Gerald Jay Sussman. Lambda: The Ultimate Imperative. AI Lab Memo AIM-353. MIT AI Lab. March 1976.
- [19] 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.
- [20] J. Robert von Behren, Jeremy Condit and Eric A. Brewer. Why Events Are a Bad Idea (for High-Concurrency Servers). In *Proceedings of HotOS'03: 9th Workshop on Hot Topics in Operating Systems*, May 2003, Lihue (Kauai), Hawaii, USA. 2003.