

Continuation-Passing C

Compiling threads to events through continuations

Gabriel Kerneis · Juliusz Chroboczek

Received: date / Accepted: date

Abstract In this paper, we introduce Continuation Passing C (CPC), a programming language for concurrent systems in which native and cooperative threads are unified and presented to the programmer as a single abstraction. The CPC compiler uses a compilation technique, based on the CPS transform, that yields efficient code and an extremely lightweight representation for contexts. We provide a complete proof of the correctness of our compilation scheme. We show in particular that lambda-lifting, a common compilation technique for functional languages, is also correct in an imperative language like C, under some conditions enforced by the CPC compiler. The current CPC compiler is mature enough to write substantial programs such as Hekate, a highly concurrent BitTorrent seeder. Our benchmark results show that CPC is as efficient, while significantly cheaper, as the most efficient thread libraries available.

Keywords Concurrent programming · Lambda-lifting · Continuation-Passing Style

Contents

1	Introduction	2
2	Related work	4
3	The CPC language	6
4	The CPC compilation technique	10
5	CPS conversion	12
6	Rewriting CPC terms into CPS-convertible form	20
7	Lambda-lifting	22
8	Optimised reduction rules	27
9	Correctness of lambda-lifting	38
10	Guaranteeing non-interference	47
11	Implementation	52
12	Experimental results	54
13	Hekate, a BitTorrent seeder written in CPC	56
14	Conclusions and further work	58

G. Kerneis
Laboratoire PPS, Université Paris Diderot, Case 7014, 75205 Paris Cedex 13, France
E-mail: kerneis@pps.jussieu.fr

J. Chroboczek
Laboratoire PPS, Université Paris Diderot, Case 7014, 75205 Paris Cedex 13, France

1 Introduction

Most computer programs written today are *concurrent* programs, which need to perform multiple tasks at a given time. For example, a network server needs to serve multiple clients at a time; a program with a graphic interface needs to expect keystrokes and mouse clicks at multiple places; and a network program with a graphic interface (e.g. a web browser) needs to do both.

Threads The dominant abstraction for concurrency is provided by *threads*, or *lightweight processes*. When programming with threads, a process is structured as a dynamically changing number of independently executing threads that share a single heap (and possibly other data, such as global variables).

Ideally, the threads provided by a programming language should be lightweight enough to allow the programmer to encapsulate a single activity in each thread. In most cases, however, the thread abstraction provided by the programming language tends to be identical to the one provided by the operating system; such threads are designed to be efficient to implement, but not necessarily to be lightweight or convenient to program with. In effect, the programmer is forced to multiplex multiple activities in a single thread. In the rare languages where threads are sufficiently lightweight, they tend to be slow and unable to exploit multiple processors or processor cores.

Events Another technique for concurrency 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 global scheduler, known as the *event loop*, repeatedly waits for an event to occur and invokes the associated handler. Event-driven programs are believed to be faster and lighter than their threaded counterparts; however, like most lightweight threads, they do not scale easily to multiple processors.

Event-driven programming is also very difficult: in an event-driven program, the flow of control is no longer explicit, and what is logically a sequence of statements is often split into many tiny event-handlers. The event-driven programmer must maintain at all times a mental model of the current computation, and manually link these tiny event-handlers into long chains of callbacks. Debugging such code is even more difficult than writing it, since the stack is not visible in a debugger.

Continuation Passing C CPC is a programming language designed for large scale concurrency. The threads provided by CPC do not correspond to any operating system entity; instead, they are implemented in different manners in different places of the program. Using multiple implementation techniques in various places yields threads that are lightweight, efficient, and able to exploit multiple processors.

1.1 User and kernel threads

There are two main categories of threads that can be provided to the programmer. User-space threads, sometimes known as *green* threads, are implemented in the programming language runtime. When implemented correctly, user-space threads are fast and lightweight; additionally, since the user-space scheduler resides entirely in user-space, it is easily customised on

a per-language or even per-application basis to provide the chosen semantics (for example determinacy of scheduling, cooperativity, etc.).

There are, however, a number of issues with the exclusive use of user-space threads. First, with user-space threads, all blocking system interfaces (system calls and standard library interfaces) must be wrapped in non-blocking wrappers; this is difficult most of the time, and sometimes impossible (e.g. in the case of Unix's `gethostbyname` library function). Second, in the case of cooperative threads, a single user-space thread that fails to yield control of the processor in a timely manner may block the whole program. And finally, user-space threads cannot make use of multiple processors, a significant flaw in these times of multi-core and multithreaded processors.

The alternative are system-provided, preemptive *kernel* threads. Kernel threads have none of the flaws of user-space threads: invoking a blocking system call or failing to yield in a kernel thread doesn't prevent other threads from running, and, when implemented correctly, kernel threads can use multiple processors or processor cores.

There are a few reasons, however, that can make programming with kernel threads inconvenient. First, kernel threads are preemptive: this implies that all data shared between multiple threads must be protected using mutual exclusion and synchronisation primitives, a difficult and error-prone task that, if not done correctly, leads to unreproducible and difficult to debug issues. Second, kernel threads are nondeterministic, which means that determinacy, when desired, can only be achieved by using synchronisation primitives. And, finally, kernel threads cannot be implemented in quite as lightweight a manner as user-space threads, as each kernel thread requires at least two pages of physical memory (one for a kernel and one for a user-space stack) and the associated virtual memory structures.

1.2 CPC threads

CPC provides the programmer with a thread abstraction that unifies user and kernel threads. A CPC thread is implemented, at different times during its lifetime, as either a cooperative user-space thread or a native thread attached to a thread pool. In the first case, we say that the CPC thread is *attached* to the user-space scheduler; in the second case, that it is attached to the thread pool, or *detached*. Transitions between these states are managed explicitly by the programmer (a fair amount of syntactic sugar is provided to make this task easier).

1.3 CPC compilation strategies

In order for CPC threads to be an efficient abstraction and to avoid burdening the programmer with multiplexing multiple activities in a single thread, we want context switches and transitions between the attached and detached states to be as cheap as possible. For this to be the case, we want a thread's dynamic state to be as small as possible, so that it can be easily saved and restored. Ideally, a thread's dynamic state should be a single pointer to a heap-allocated data structure, in which case a context switch is just a pointer swap.

In an interpreter, such an implementation of threads is easily achieved by encapsulating all of the interpreter's thread-local state in a heap allocated structure and systematically accessing this state through a single pointer [36, 19]. In a compiler, a similar effect is achieved by a technique known as *conversion to continuation-passing style* (CPS), which consists in making manipulations of the dynamic chain ("the stack") explicit. CPS conversion is described in more detail in Section 5.

Unfortunately, CPS-converted code is somewhat slower than direct-style (“normal”) code. Hence, CPC implements two compilation strategies. Most code is compiled into direct style, where function activation records and local variables reside on the native processor stack. Some code, however, is compiled using a continuation-passing transform; we say that such code is in *cps context*. Most “interesting” operations are only allowed in *cps context*: this includes cooperating when attached to the user-space scheduler, or transitioning between different attached states. As with the implementation of threads, choosing the compilation strategy is the responsibility of the programmer (and, again, this is made simpler by the use of syntactic sugar).

The CPS transform can be shown to be correct in purely functional languages. In a call-by-value language with imperative variables such as C, it would seem to require “boxing” of every variable (adding an extra layer of indirection by wrapping it within a heap-allocated structure), which would have a prohibitive cost (Section 7). However, as we show in Section 10.1, it turns out that boxing most imperative variables is not necessary.

1.4 Contributions

This paper makes the following main contributions. First, we introduce CPC, a programming language in which native and cooperative threads are unified and presented to the programmer as a single abstraction. Second, we introduce a compilation technique, based on the CPS transform, that yields efficient code and an extremely lightweight representation for contexts, needing no boxing of mutable variables in the typical case. Finally, we provide a complete proof of the correctness of our compilation scheme, including the use of lambda-lifting in the context of an imperative language.

2 Related work

2.1 Thread and event libraries

Thread libraries Most thread libraries in C offer either preemptive or cooperative threads.

The simplest implementation of preemptive threads is a 1:1 mapping to the underlying kernel threads, as done for example by both Linux thread libraries (LinuxThreads [28] and the Native POSIX Thread Library [14]). The mapping from programmer-visible threads to kernel threads may be more complex than that: DCE threads, unbound threads in Solaris 8 [47] and scheduler activations [2] (formerly implemented in the NetBSD kernel [54]) are three examples of the “N:M model” where a number of programmer-visible threads are scheduled on a smaller number of kernel threads. Pure user-space implementations of preemptive threads also exist, such as MIT Pthreads [38].

Cooperative threads, on the other hand, are provided by many user-space libraries with different aims: Pth [17] focuses on portability, State Threads [43] on efficiency for network servers, shared-stack threads [20] on memory consumption for embedded systems, FairThreads [8] on a clear semantics, Capriccio [5] on automatic cooperation before blocking system calls, etc. Although most cooperative threads libraries are implemented in user-space, Windows fibers [32] are an example of cooperative kernel threads.

It is also possible to provide both models in a single library. Boussinot’s FairThreads [8] introduced the idea of unified threads, either “linked” to a cooperative scheduler or “unlinked” as preemptive threads. Since Boussinot is mostly concerned with the semantics of

threads, the current FairThreads implementation maps every thread to a native kernel thread; CPC threads, which can be either attached or detached, are very close to FairThreads and are implemented in a much cheaper manner.

Event libraries Since most event-driven programs are structured around a similar event loop, waiting for the same set of events, libraries such as libevent [39] and libev [27] are largely used to replace the boilerplate code with concisely defined event-handlers. They help reduce code size and, most importantly, avoid duplicating the efforts of building highly-efficient event loops.

Hybrid architectures, mixing events and native threads, aim at scaling event-driven systems to multiple processors. For instance, the asymmetric multi-process event-driven architecture (AMPED), used in the Flash web server [35], delegates disk operations to helper threads, and the staged event-driven architecture (SEDA) used in the Haboob server [52] decomposes the program into a collection of event loops executed by multiple thread pools. CPC programs are currently compiled to a hybrid architecture similar to AMPED, with a main event loop scheduling attached threads and user-defined thread pools executing detached threads.

None of these frameworks, however, solve the fundamental issue which makes event-driven programming so difficult: the flow of control is split into many tiny handlers, a phenomenon called “stack ripping” by Adya et al. [1]. There have been a few attempts to mitigate this intrinsic complexity. Tame [26] is an extension of the C++ language providing composable events in a threaded style. It uses a variant of Duff’s device [15] to allow “rendez-vous” (cooperation points) inside “tamed” (cooperative) functions. Eel [12] follows the opposite approach, staying within the C language and a classical API, but offering tools to easily visualise and debug complex, event-driven programs. Unfortunately, such tools need to be rewritten for every event loop library, because each has its own callback convention. CPC, for instance, lacks a continuation-aware debugger which would reconstruct and display the stack in a transparent way.

2.2 Continuations and concurrency

Continuations offer a natural framework to implement concurrency systems in functional languages: they capture “the rest of the computation” much like the state of an imperative program is captured by the call stack. Thread-like primitives may be built with either first-class continuations, or encapsulated within a continuation monad.

The former approach is best illustrated by Concurrent ML constructs [40], implemented on top of SML/NJ’s first-class continuations, or by the way coroutines and engines are typically implemented in Scheme using the `call/cc` operator [21, 16]. Stackless Python [49] uses first-class continuations to implement generators, which are in turn used to implement concurrency primitives.

Explicit translation into continuation-passing style, often encapsulated within a monad, is used in languages lacking first-class continuations. In Haskell, the original idea of a concurrency monad is due to Scholz [42], and extended by Claessen [10] to a monad transformer yielding a concurrent version of existing monads. Li et. al [30] use a continuation monad to lazily translate the thread abstraction exposed to the programmer into events scheduled by an event loop. In Objective Caml, Vouillon’s “lwt” [50] provide a lightweight alternative to native threads, with the ability to execute blocking tasks in a separate thread pool.

2.3 Compilation techniques

The two main techniques used by CPC, CPS conversion [46,37] and lambda-lifting [24], are fairly standard techniques for compiling functional languages. CPS conversion has been used, for example, for Scheme (Rabbit [44]) and ML (SML/NJ [3]). Danvy and Schultz provide many examples of the uses of lambda-lifting [13].

There are fewer occurrences of similar techniques applied to imperative languages, and they do not aim at compiling concurrent systems. Van Wijngaarden [53] proposes a preprocessor for ALGOL 60, surprisingly close to the transformations performed by CPC, considered by Reynolds [41] as the earliest example of continuation-passing style. He uses the fact that ALGOL allows labels as first-class values to implement continuations, then converting `goto` statements into calls to inner procedures. Since ALGOL allows inner procedures, lambda-lifting is not needed. Thielecke [48] translates van Wijngaarden's ideas to C, as a pedagogical introduction to continuations for C programmers; he makes some simplifying assumptions that allow him to avoid the pitfalls of lambda-lifting in an imperative language. To the best of our knowledge, neither of these works has yielded effective implementations.

3 The CPC language

CPC is a modest extension to the C programming language [23]. Every C program is a correct CPC program, and has exactly the same semantics.

Just like a plain C program, a CPC program is a set of functions. Code in a CPC program is divided into “cps” and “native” code, with the global constraint that a cps function can only ever be called by another cps function, never by a native function. (A native function may spawn a new thread that executes a cps function, of course.)

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

The `cps` keyword is used for marking a function to be CPS-converted. Syntactically, `cps` is a function specifier, and can appear before any function.

3.1 CPC primitives

CPC provides a set of primitive cps functions, which allow the programmer to schedule threads and wait for some events. These primitive functions could not have been defined in user code: they must have access to the internals of the scheduler to operate. Since they are cps functions, they can only be called in cps context.

In addition to these functions, CPC extends the C language with one statement, `cps_spawn`, and a small amount of syntactic sugar (`cpc_attached`, `cpc_detached`, etc.)

Creating threads

```
typedef cpc_sched;  
cps cpc_sched *cpc_attach(cpc_sched *sched);
```

The function `cpc_attach` attaches the current thread to the scheduler or thread pool passed as an argument. It returns the previous scheduler, which makes it easy to restore the current threads state to its previous value.

The statements `cpc_attached` and `cpc_detached` execute their body, respectively, attached to the default CPC scheduler and detached (i.e. attached to the default thread pool). They are pure syntactic sugar: for example,

```
cpc_detached { body; }
```

is equivalent to

```
cpc_sched *previous = cpc_attach(cpc_default_threadpool);
body;
cpc_attach(previous);
```

Notice how this implementation guarantees that the body is executed detached and the scheduler is properly restored, whether the function was invoked detached or not. This property is useful for library functions, which must behave identically when invoked from either detached or attached code.

Cooperating and scheduling

```
cps void cpc_yield();
cpc_spawn {
    ...
}
```

The function `cpc_yield` is used for cooperating: when attached to a scheduler, it causes the current thread to be queued at the end of the scheduler's runnable threads. It has no effect in a detached thread.

The compound instruction `cpc_spawn` creates a new thread that executes the instruction's body and schedules it for execution by CPC's default scheduler. Unlike all other CPC constructs, it can be executed in non-cps context; since execution of a CPC program starts at the native function `main`, this is essential for bootstrapping.

Condition variables

```
typedef struct cpc_condvar cpc_condvar;

cps int cpc_wait(cpc_condvar *cond);
void cpc_signal(cpc_condvar *cond);
void cpc_signal_all(cpc_condvar *cond);
```

In order to provide synchronisation between threads, CPC provides traditional condition variables [22]. The function `cpc_wait` causes the calling thread to be suspended and placed at the end of the list of threads waiting on the condition variable `cond`. The function `cpc_signal` wakes up the first thread waiting on `cond`, while `cpc_signal_all` wakes up all the threads waiting on `cond`.

Since CPC condition variables are not associated with a lock, they are potentially subject to race conditions. For this reason, the condition variable functions are only legal in a thread attached to a scheduler.

CPC condition variables are deterministic: threads will be woken up in the order in which they were suspended. We have found this property to be useful in order to enforce rough fairness between clients of a network server [4].

Input/Output CPC does not provide any input/output primitives; it expects the programmer to either use native, non-blocking system calls, or to use blocking system calls in a detached thread.

```
cps int cpc_io_wait(int fd, int direction, cpc_condvar *cond);
```

In order to make non-blocking I/O useful, CPC provides a primitive that can block on a file descriptor. A call to `cpc_io_wait` blocks until either the file descriptor `fd` is usable in the direction specified by `direction`, or the condition variable `cond` is signaled. In an attached thread, this simply stores the current continuation in the sundry scheduler's queue, and schedules a different thread; in a detached one, it behaves as a wrapper around `select` (and `cond` must be null in that case).

3.2 The CPC library

Programming directly with the CPC primitives is somewhat awkward (see the first example below), so CPC comes with a standard library. The standard library is written in CPC, with no knowledge of CPC internals.

All functions in the CPC library are designed to work identically when called from an attached or a detached thread. For example, the `cpc_write` function detects whether it is called from a detached thread; if it is, then it simply invokes the `write` system call; if it is not, then it invokes a non-blocking wrapper around `write`.

A different approach is taken in the `cpc_getaddrinfo` function, which just calls the `getaddrinfo` function within the body of `cpc_detached`. As noted above, `cpc_detached` has the expected effect even when the caller was already detached.

3.3 Examples

Timed cat The following is a complete program in raw CPC, with no calls to the CPC library. 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 slightly incorrect, as it does not deal correctly with partial writes and interrupted system calls.)

```
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 same code using the CPC library reads as follows:

```

char buf[512];
int done = 0;

int
main()
{
    cpc_spawn {
        cpc_timeout *timeout = cpc_timeout_get(1, 0);
        cpc_condvar *cond = cpc_timeout_condvar(timeout);
        while(!cpc_timeout_expired(timeout)) {
            int rc;
            rc = cpc_read_c(0, buf, 512, cond);
            if(rc <= 0) break;
            cpc_full_write_c(1, buf, rc, cond);
        }
        cpc_timeout_destroy(timeout);
    }

    cpc_main_loop();
    return 0;
}

```

The functions starting with `cpc_` are taken from CPC's standard library. The function `cpc_read` has the same semantics as the `read` system call (at least one byte of data is read), while `cpc_full_write` always attempts to write the number of bytes passed as parameter. The functions `cpc_read_c` and `cpc_full_write_c` are similar to `cpc_read` and `cpc_full_write`, but can be interrupted by a condition variable.

A `cpc_timeout` is a small structure containing two condition variables and a thread; the condition variable returned by `cpc_timeout_condvar` is signaled when the timeout expires.

Non-blocking disk reads The following example (taken from Hekate, see Section 13, with slight modifications), sends a chunk of data from a memory-mapped disk file over a network socket. The function `prefetch` triggers an asynchronous read of an area of virtual memory into physical memory; the function `incore` determines whether a region of virtual memory is backed by physical memory (these are thin wrappers around the `posix_madvise` and `mincore` system calls, respectively).

```

prefetch(source, length);
cpc_yield();
if(!incore(source, length)) {
    cpc_yield();
    if(!incore(source, length)) {
        cpc_detached {
            rc = cpc_write(fd, source, length);
            if(rc < 0)
                return -1;
        }
        goto done;
    }
}
rc = cpc_write(fd, source, length);
if(rc < 0)
    return -1;
done:
;

```

In this code, we first trigger an asynchronous read of the on-disk data, and immediately yield to threads servicing other clients in order to give the kernel a chance to perform the read. When we are scheduled again, we check whether the read has completed; if it has, we perform a non-blocking write; if it hasn't, we yield one more time and, if that fails again, eventually delegate the work to a native thread (which can block).

Note that this code contains a race condition: the prefetched block of data could have been swapped out before the `cpc_write` completes. This possibility is too rare, however, to be concerned about.

Note further that the call to `cpc_write` in the `cpc_detached` block could be replaced by a call to `write`: we are in a native thread here, so the non-blocking wrapper is not needed. However, the CPC runtime is smart enough to detect this case, and `cpc_write` simply behaves as `write` when invoked from a native thread; for simplicity, we choose to use the CPC wrappers throughout our code.

4 The CPC compilation technique

Outline The CPC translator turns a CPC program into an equivalent C program written in continuation-passing style. Because the C language features a number of constructs, such as loops and gotos, not easily handled by a direct CPS conversion, the translator is structured as

a series of source-to-source transformations which simplify the program into *CPS-convertible form*, a subset of C for which the CPS is easy to define. This sequence of transformations consists of the following passes:

- boxing a small number of variables that would otherwise make the following passes incorrect;
- making the flow of control explicit, by introducing `goto` statements;
- converting `goto` statements to tail calls, which introduces inner functions;
- lambda-lifting, to get rid of inner functions.

CPS-convertible form is a limited enough subset of C for the translator to perform a correct partial CPS conversion in one pass. The converted program is then compiled by a standard C compiler and linked to the CPC scheduler to produce the final executable.

Correctness and efficiency concerns All of these passes are well-known compilation techniques for functional programming languages, but lambda-lifting and CPS conversion are not correct in general for an imperative call-by-value language such as C. A standard workaround consists in boxing every mutated variable; in an imperative language, however, this would mean that almost every variable is boxed, which would impose an unacceptable cost due to the extra memory accesses.

Our compilation technique keeps boxing at a reasonable level (10% in Hekate). We prove that, even in an imperative language, it is correct to apply lambda-lifting to functions called in tail position (Section 7). Since functions introduced by the `goto` elimination pass are necessarily called in tail position, the translator needs only box these local variables whose address has been retained with the “&” operator (“extruded” variables, see Section 10.1).

Boxing is not enough to guarantee the correctness of CPS conversion in the presence of “shared” variables, i.e. variables that might be modified by several functions (such as global or extruded variables). Outdated copies of those shared parameters, evaluated too early because of CPS conversion, might be stored in continuations. We work around this pitfall by inserting statements to reevaluate potentially obsolete parameters; this happens seldom enough to induce no noticeable overhead (see Section 10.3).

Proof of the compilation scheme In Sections 5 to 10, we prove the correctness of the CPS conversion, lambda-lifting and boxing passes. We do not write these proofs in CPC, because the semantics of C is too broad and complex for our purposes: the CPC translator leaves most parts of converted programs intact, transforming only control structures and function calls. Instead, we define a simple language with restricted values, expressions and terms.

Definition 1 (Values, expression and terms)

Values are either boolean and integer constants or **1**, a special value for functions returning void.

$$v ::= \mathbf{1} \mid \mathbf{true} \mid \mathbf{false} \mid n \in \mathbf{N}$$

Expressions are either values or variables. We deliberately omit arithmetic and boolean operators, with the sole concern of avoiding boring cases in the proofs.

$$expr ::= v \mid x \mid \dots$$

Terms are made of assignments, conditionals, sequences, recursive functions definitions and calls.

$$T ::= \text{expr} \mid x := T \mid \text{if } T \text{ then } T \text{ else } T \mid T ; T \\ \mid \text{letrec } f(x_1 \dots x_n) = T \text{ in } T \mid f(T, \dots, T)$$

Our language focuses on the essential details affected by the transformations: recursive functions, conditionals and memory accesses. Loops, for instance, are ignored because they can be expressed in terms of recursive calls and conditional jumps. Since lambda-lifting and CPS-conversion happen after the goto elimination pass, our language includes inner functions (although they are not part of the C language) and excludes gotos. We show in Section 5.3 how introducing goto statements and converting them into tail calls allows us to put any CPC program in CPS-convertible form.

We constrain the language even further for the proof of CPS conversion in order to reflect the particular structure of “CPS-convertible” terms, which contain no more inner functions, free variables or non-tail calls (see Definition 2 in Section 5.1).

The reduction rules for the full language and the CPS-convertible one are detailed when we prove the correctness of lambda-lifting (Section 5.1) and CPS-conversion (Section 7.3). Both use a simplified memory model, without pointers, and enforce that local variables are not accessed outside their scope as ensured by our boxing pass. The limitations of this model and workarounds are discussed in Section 10.

5 CPS conversion

Conversion into Continuation Passing Style [46,37], or *CPS conversion* for short, is a program transformation technique that makes the flow of control of a program explicit and provides first-class abstractions for it.

Intuitively, the *continuation* of a fragment of code is an abstraction of the action to perform after its execution. For example, consider the following computation:

$$f(g(5) + 4);$$

The continuation of $g(5)$ is $f(\cdot + 4)$ because the return value of g will be added to 4 and then passed to f .

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 its continuation with the argument v .

A CPS-converted function therefore never returns, but makes a call to its continuation. Since 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.

This translation has three well-known interesting properties, on which we rely in the implementation of CPC:

CPS conversion need not be global The CPS conversion is not an “all or nothing” deal, in which the complete program must be converted: 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.

It is therefore possible to perform CPS conversion on just a subset of the functions constituting a program (in the case of CPC, such functions are annotated with the `cps` keyword). This allows `cps` code to call code in direct style, 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 direct function call; being able to only convert the functions that need the full flexibility of continuations avoids this overhead as much as possible.

Continuations are abstract data structures The classical definition of CPS conversion implements continuations as functions. However, continuations are abstract data structures, of which functions are only one particular concrete representation.

The CPS conversion process performs only two operations on continuations: calling a continuation, which we call *invoke*, and prepending a function application to the body of a continuation, which we call *push*. This property is not really surprising: as continuations are merely a representation for the dynamic chain, it is natural that there should be a correspondence between the operations available on a continuation and a stack.

Since C doesn't have full first-class functions (closures), CPC uses this property to implement continuations as arrays (see Section 11).

Continuation transformers are linear CPS conversion introduces linear (or "one-shot") continuations [7,9]: 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 reliably reclaiming continuations without the need for costly devices such as reference counting.

5.1 CPS-convertible form

CPS conversion is not defined for all terms of our C-like language; instead, we restrict ourselves to a subset of terms, which we call the *CPS-convertible* subset. As we shall see in Section 6, every term of our full language can be converted to an equivalent CPS-convertible term.

Definition 2 (CPS-convertible terms)

$v ::= \mathbf{1} \mid \mathbf{true} \mid \mathbf{false} \mid n \in \mathbf{N}$	(values)
$expr ::= v \mid x \mid \dots$	(expressions)
$F ::= f(expr, \dots, expr) \mid f(expr, \dots, expr, F)$	(nested function calls)
$Q ::= \varepsilon \mid Q ; F$	(tail)
$T ::= expr \mid x := expr ; T \mid \mathbf{if } e \mathbf{ then } T \mathbf{ else } T \mid Q$	(head)

A CPS-convertible term has two parts: the head and the tail. The head is a (possibly empty) sequence of assignments, possibly embedded within conditional statements. The tail is a (possibly empty) sequence of function calls in a highly restricted form: their parameters are (side-effect free) expressions, except possibly for the last one, which can be another function call of the same form. Values and expressions are left unchanged.

A program in CPS-convertible form consists of a set of mutually-recursive functions with no free variables, the body of each of which is a CPS-convertible term.

The essential property of CPS-convertible terms, which makes their CPS conversion immediate to perform, is the guarantee that there is no cps call outside of the tails. It makes continuations easy to represent as a series of function calls (tails) and separates them clearly from imperative blocks (heads), which are not modified by the CPC translator.

Stack machine We define the semantics of CPS-convertible terms through a set of small-step reduction rules. We distinguish three kinds of reductions: \rightarrow_T to reduce the head of terms, \rightarrow_Q to reduce the tail, and \rightarrow_e to evaluate expressions.

These rules describe a stack machine with a store σ to keep the value of variables. Since free and shared variables have been eliminated in earlier passes (see Sections 7 and 10), there is a direct correspondence at any point in the program between variable names and locations, with no need to dynamically maintain an extra environment.

We use contexts as a compact representation for stacks. The head rules \rightarrow_T reduce triples made of a term, a context and a store: $\langle T, C[], \sigma \rangle$. The tail rules \rightarrow_Q , which merely unfold tails with no need of a store, reduce couples of a tail and a context: $\langle Q, C[] \rangle$. The expression rules do not need context to reduce, thus operating on couples made of an expression and a store: $\langle e, \sigma \rangle$.

Contexts Contexts are sequences of function calls. In those sequences, function parameters shall be already evaluated: constant expressions are allowed, but not variables. As a special case, the last parameter might be a “hole” instead, written \ominus , to be filled with the return value of the next, nested function.

Definition 3 (Contexts) Contexts are defined inductively:

$$C ::= [] \mid C[] ; f(v_1, \dots, v_n) \mid C[] ; f(v_1, \dots, v_n, \ominus)$$

Definition 4 (CPS-convertible reduction rules)

$$\langle x := expr ; T, C[], \sigma \rangle \rightarrow_T \langle T, C[], \sigma[x \mapsto v] \rangle \quad \text{when } \langle expr, \sigma \rangle \rightarrow_e^* v \quad (1)$$

$$\langle \text{if } expr \text{ then } T_1 \text{ else } T_2, C[], \sigma \rangle \rightarrow_T \langle T_1, C[], \sigma \rangle \quad \text{when } \langle expr, \sigma \rangle \rightarrow_e^* \text{true} \quad (2)$$

$$\langle \text{if } expr \text{ then } T_1 \text{ else } T_2, C[], \sigma \rangle \rightarrow_T \langle T_2, C[], \sigma \rangle \quad \text{when } \langle expr, \sigma \rangle \rightarrow_e^* \text{false} \quad (3)$$

$$\langle expr, C[] ; f(v_1, \dots, v_n), \sigma \rangle \rightarrow_T \langle \varepsilon, C[] ; f(v_1, \dots, v_n) \rangle \quad (4)$$

$$\langle expr, C[] ; f(v_1, \dots, v_n, \ominus), \sigma \rangle \rightarrow_T \langle \varepsilon, C[] ; f(v_1, \dots, v_n, v) \rangle \quad \text{when } \langle expr, \sigma \rangle \rightarrow_e^* v \quad (5)$$

$$\langle expr, [], \sigma \rangle \rightarrow_T v \quad \text{when } \langle expr, \sigma \rangle \rightarrow_e^* v \quad (6)$$

$$\langle Q, C[], \sigma \rangle \rightarrow_T \langle Q[x_i \setminus \sigma x_i], C[] \rangle \quad \text{for every } x_i \text{ in } \text{dom}(\sigma)$$

$$\langle Q ; f(v_1, \dots, v_n), C[] \rangle \rightarrow_Q \langle Q, C[] ; f(v_1, \dots, v_n) \rangle \quad (7)$$

$$\langle Q ; f(v_1, \dots, v_n, F), C[] \rangle \rightarrow_Q \langle Q ; F, C[] ; f(v_1, \dots, v_n, \ominus) \rangle \quad (8)$$

$$\langle \varepsilon, C[] ; f(v_1, \dots, v_n) \rangle \rightarrow_Q \langle T, C[], \sigma \rangle \quad \text{when } f(x_1, \dots, x_n) = T \text{ and } \sigma = \{x_i \mapsto v_i\} \quad (9)$$

We do not detail the rules for \rightarrow_e , which simply looks for variables in σ and evaluates arithmetical and boolean operators.

Explicit non-interference Rule 6 deserves some specific explanation. This essential rule translates the fact that the evaluation order of function parameters in tails does not matter.

The rationale for this rule comes from the following remark: since there are no more shared or free variables in CPS-convertible terms, there can be no “interference” issues, i.e. functions modifying variables outside of their own scope. In particular, a function cannot modify the variables of its caller. It follows that any variable can be replaced by its value when the evaluation of a tail begins. We prove that this assumption holds for CPS-convertible terms in Theorem 5 (Section 10.2).

In this section, we deliberately restrict the language to make this “non-interference” result obvious. This avoids introducing sophisticated lemmas to prove that function parameters remain constant throughout the evaluation of tails.

5.2 CPS terms

Unlike classical CPS conversion techniques [37], our CPS terms are not continuations, but a procedure which builds and executes the continuation of a term. Construction is performed by **push**, which adds a function to the current continuation, and execution by **invoke**, which calls the first function of the continuation, optionally passing it the return value of the current function.

Definition 5 (CPS terms)

$$\begin{aligned}
 v &::= \mathbf{1} \mid \mathbf{true} \mid \mathbf{false} \mid n \in \mathbf{N} && \text{(values)} \\
 \text{expr} &::= v \mid x \mid \dots && \text{(expressions)} \\
 Q &::= \mathbf{invoke} \mid \mathbf{push} f(\text{expr}, \dots, \text{expr}) ; Q \mid \mathbf{push} f(\text{expr}, \dots, \text{expr}, \square) ; Q && \text{(tail)} \\
 T &::= \mathbf{invoke} \text{ expr} \mid x := \text{expr} ; T \mid \mathbf{if} e \mathbf{then} T \mathbf{else} T \mid Q && \text{(head)}
 \end{aligned}$$

Continuations and reduction rules A continuation is a sequence of function calls to be performed, with already evaluated parameters. We write \cdot for appending a function to a continuation, and \square for a “hole”, i.e. an unknown parameter.

Definition 6 (Continuations)

$$\mathcal{C} ::= \varepsilon \mid f(v, \dots, v) \cdot \mathcal{C} \mid f(v, \dots, v, \square) \cdot \mathcal{C}$$

The reduction rules for CPS terms are very similar to the rules for CPS-convertible terms, except that they use continuations instead of contexts.

Definition 7 (CPS reduction rules)

$$\langle x := expr ; T, \mathcal{C}, \sigma \rangle \rightarrow_T \langle T, \mathcal{C}, \sigma[x \mapsto v] \rangle \quad (10)$$

when $\langle expr, \sigma \rangle \rightarrow_e^* v$

$$\langle \text{if } expr \text{ then } T_1 \text{ else } T_2, \mathcal{C}, \sigma \rangle \rightarrow_T \langle T_1, \mathcal{C}, \sigma \rangle \quad (11)$$

if $\langle expr, \sigma \rangle \rightarrow_e^* \text{true}$

$$\langle \text{if } expr \text{ then } T_1 \text{ else } T_2, \mathcal{C}, \sigma \rangle \rightarrow_T \langle T_2, \mathcal{C}, \sigma \rangle \quad (12)$$

if $\langle expr, \sigma \rangle \rightarrow_e^* \text{false}$

$$\langle \text{invoke } expr, f(v_1, \dots, v_n) \cdot \mathcal{C}, \sigma \rangle \rightarrow_T \langle \text{invoke}, f(v_1, \dots, v_n) \cdot \mathcal{C} \rangle \quad (13)$$

$$\langle \text{invoke } expr, f(v_1, \dots, v_n, \square) \cdot \mathcal{C}, \sigma \rangle \rightarrow_T \langle \text{invoke}, f(v_1, \dots, v_n, v) \cdot \mathcal{C} \rangle \quad (14)$$

when $\langle expr, \sigma \rangle \rightarrow_e^* v$

$$\langle \text{invoke } expr, \varepsilon, \sigma \rangle \rightarrow_T v \quad \text{when } \langle expr, \sigma \rangle \rightarrow_e^* v$$

$$\langle Q, \mathcal{C}, \sigma \rangle \rightarrow_T \langle Q[x_i \setminus \sigma x_i], \mathcal{C} \rangle \quad (15)$$

for every x_i in $\text{dom}(\sigma)$

$$\langle \text{push } f(v_1, \dots, v_n) ; Q, \mathcal{C} \rangle \rightarrow_Q \langle Q, f(v_1, \dots, v_n) \cdot \mathcal{C} \rangle \quad (16)$$

$$\langle \text{push } f(v_1, \dots, v_n, \square) ; Q, \mathcal{C} \rangle \rightarrow_Q \langle Q, f(v_1, \dots, v_n, \square) \cdot \mathcal{C} \rangle \quad (17)$$

$$\langle \text{invoke}, f(v_1, \dots, v_n) \cdot \mathcal{C} \rangle \rightarrow_Q \langle T, \mathcal{C}, \sigma \rangle \quad (18)$$

when $f(x_1, \dots, x_n) = T$ and $\sigma = \{x_i \mapsto v_i\}$

Well-formed terms Not all CPS term will lead to a correct reduction. If we **push** a function expecting the result of another function and **invoke** it immediately, the reduction blocks:

$$\langle \text{push } f(v_1, \dots, v_n, \square) ; \text{invoke}, \mathcal{C}, \sigma \rangle \rightarrow \langle \text{invoke}, f(v_1, \dots, v_n, \square) \cdot \mathcal{C}, \sigma \rangle \not\rightarrow$$

Well-formed terms avoid this behaviour.

Definition 8 (Well-formed term) A continuation queue is *well-formed* if it does not end with:

$$\text{push } f(expr, \dots, expr, \square) ; \text{invoke}.$$

A term is *well-formed* if every continuation queue in this term is well-formed.

5.3 From CPS-convertible terms to CPS terms

We define the CPS conversion as a mapping from CPS-convertible terms to CPS terms.

Definition 9 (CPS conversion)

$$(Q ; f(expr, \dots, expr))^{\mathbf{A}} = \text{push } f(expr, \dots, expr) ; Q^{\mathbf{A}}$$

$$(Q ; f(expr, \dots, expr, F))^{\mathbf{A}} = \text{push } f(expr, \dots, expr, \square) ; (Q ; F)^{\mathbf{A}}$$

$$\varepsilon^{\mathbf{A}} = \text{invoke}$$

$$(x := expr ; T)^{\mathbf{A}} = x := expr ; T^{\mathbf{A}}$$

$$(\text{if } expr \text{ then } T_1 \text{ else } T_2)^{\mathbf{A}} = \text{if } expr \text{ then } T_1^{\mathbf{A}} \text{ else } T_2^{\mathbf{A}}$$

$$expr^{\mathbf{A}} = \text{invoke } expr$$

In the rest of this section, we prove that this mapping yields an isomorphism between the reduction rules of CPS-convertible terms and well-formed CPS terms, whence the correctness of our CPS conversion (Theorem 1).

We first prove two lemmas to show that \blacktriangle yields only well-formed CPS terms. This leads to a third lemma to show that \blacktriangle is a bijection between CPS-convertible terms and well-formed CPS terms.

CPS-convertible terms have been carefully designed to make CPS conversion as simple as possible. Accordingly, the following three proofs, while long and tedious, are fairly trivial.

Lemma 1 *Let Q be a continuation queue. Then Q^\blacktriangle is well-formed.*

Proof By induction on the structure of a tail.

$$\varepsilon^\blacktriangle = \mathbf{invoke}$$

and

$$(\varepsilon ; f(\text{expr}, \dots, \text{expr}))^\blacktriangle = \mathbf{push} f(\text{expr}, \dots, \text{expr}) ; \mathbf{invoke}$$

are well-formed by definition.

$$((Q ; F) ; f(\text{expr}, \dots, \text{expr}))^\blacktriangle = \mathbf{push} f(\text{expr}, \dots, \text{expr}) ; (Q ; F)^\blacktriangle$$

and

$$(Q ; f(\text{expr}, \dots, \text{expr}, F))^\blacktriangle = \mathbf{push} f(\text{expr}, \dots, \text{expr}, \square) ; (Q ; F)^\blacktriangle$$

are well-formed by induction. \square

Lemma 2 *Let T be a CPS-convertible term. Then T^\blacktriangle is well-formed.*

Proof Induction on the structure of T , using the above lemma. \square

Lemma 3 *The \blacktriangle relation is a bijection between CPS-convertible terms and well-formed CPS terms.*

Proof Consider the following mapping from well-formed CPS terms to CPS-convertible terms:

$$\begin{aligned} (\mathbf{push} f(\text{expr}, \dots, \text{expr}) ; Q)^\blacktriangledown &= Q^\blacktriangledown ; f(\text{expr}, \dots, \text{expr}) \\ (\mathbf{push} f(\text{expr}, \dots, \text{expr}, \square) ; Q)^\blacktriangledown &= Q' ; f(\text{expr}, \dots, \text{expr}, F) \\ &\text{with } Q^\blacktriangledown = Q' ; F \end{aligned} \quad (*)$$

$$\begin{aligned} \mathbf{invoke}^\blacktriangledown &= \varepsilon \\ (x := \text{expr} ; T)^\blacktriangledown &= x := \text{expr} ; T^\blacktriangledown \\ \mathbf{if} \text{ expr} \mathbf{ then } T_1 \mathbf{ else } T_2^\blacktriangledown &= \mathbf{if} \text{ expr} \mathbf{ then } T_1^\blacktriangledown \mathbf{ else } T_2^\blacktriangledown \\ (\mathbf{invoke} \text{ expr})^\blacktriangledown &= \text{expr} \end{aligned}$$

(*) The existence of Q' is guaranteed by well-formedness:

- $\forall T, T^\blacktriangledown = \varepsilon \Rightarrow T = \mathbf{invoke}$ (by disjunction on the definition of \blacktriangledown),
- here, $Q \neq \mathbf{invoke}$ because $(\mathbf{push} f(\text{expr}, \dots, \text{expr}, \square) ; Q)$ is well-formed,
- hence $Q^\blacktriangledown \neq \varepsilon$.

One checks easily that $(T^\blacktriangledown)^\blacktriangle = T$ and $(T^\blacktriangle)^\blacktriangledown = T$. \square

To conclude the proof of isomorphism, we also need an (obviously bijective) mapping from contexts to continuations:

Definition 10 (Conversion of contexts)

$$\begin{aligned} ([])^{\Delta} &= \varepsilon \\ (C[] ; f(v_1, \dots, v_n))^{\Delta} &= f(v_1, \dots, v_n) \cdot \mathcal{C} \\ &\quad \text{with } (C[])^{\Delta} = \mathcal{C} \\ (C[] ; f(v_1, \dots, v_n, \ominus))^{\Delta} &= f(v_1, \dots, v_n, \Box) \cdot \mathcal{C} \\ &\quad \text{with } (C[])^{\Delta} = \mathcal{C} \end{aligned}$$

The correctness theorem follows:

Theorem 1 (Correctness of CPS conversion) *The \blacktriangle and \triangle mappings are two bijections, the inverses of which are written \blacktriangledown and \triangledown . They yield an isomorphism between reduction rules of CPS-convertible terms and CPS terms.*

Proof Lemma 3 ensures that \blacktriangle is a bijection between CPS-convertible terms and well-formed CPS terms. Moreover, \triangle is an obvious bijection between contexts and continuations.

To complete the proof, we only need to apply \blacktriangle , \triangle , \blacktriangledown and \triangledown to CPS-convertible terms, contexts, well-formed CPS terms and continuations (respectively) in every reduction rule and check that we get a valid rule in the dual reduction system. The result is summarized in Figure 1. \square

$$\begin{array}{lcl}
\langle x := \text{expr} ; T, C[\], \sigma \rangle \rightarrow_T \langle T, C[\], \sigma[x \mapsto v] \rangle & \Leftrightarrow & \langle x := \text{expr} ; T, \mathcal{C}, \sigma \rangle \rightarrow_T \langle T, \mathcal{C}, \sigma[x \mapsto v] \rangle \\
\langle \text{if } \text{expr} \text{ then } T_1 \text{ else } T_2, C[\], \sigma \rangle \rightarrow_T \langle T_1, C[\], \sigma \rangle & \Leftrightarrow & \langle \text{if } \text{expr} \text{ then } T_1 \text{ else } T_2, \mathcal{C}, \sigma \rangle \rightarrow_T \langle T_1, \mathcal{C}, \sigma \rangle \\
\langle \text{if } \text{expr} \text{ then } T_1 \text{ else } T_2, C[\], \sigma \rangle \rightarrow_T \langle T_2, C[\], \sigma \rangle & \Leftrightarrow & \langle \text{if } \text{expr} \text{ then } T_1 \text{ else } T_2, \mathcal{C}, \sigma \rangle \rightarrow_T \langle T_2, \mathcal{C}, \sigma \rangle \\
\langle \text{expr}, C[\] ; f(v_1, \dots, v_n), \sigma \rangle \rightarrow_T \langle \varepsilon, C[\] ; f(v_1, \dots, v_n) \rangle & \Leftrightarrow & \langle \text{invoke } \text{expr}, f(v_1, \dots, v_n) \cdot \mathcal{C}, \sigma \rangle \rightarrow_T \langle \text{invoke}, f(v_1, \dots, v_n) \cdot \mathcal{C} \rangle \\
\langle \text{expr}, C[\] ; f(v_1, \dots, v_n, \ominus), \sigma \rangle \rightarrow_T \langle \varepsilon, C[\] ; f(v_1, \dots, v_n, v) \rangle & \Leftrightarrow & \langle \text{invoke } \text{expr}, f(v_1, \dots, v_n, \square) \cdot \mathcal{C}, \sigma \rangle \rightarrow_T \langle \text{invoke}, f(v_1, \dots, v_n, v) \cdot \mathcal{C} \rangle \\
\langle \text{expr}, [\], \sigma \rangle \rightarrow_T v & \Leftrightarrow & \langle \text{invoke } \text{expr}, \varepsilon, \sigma \rangle \rightarrow_T v \\
\langle Q, C[\], \sigma \rangle \rightarrow_T \langle Q[x_i \setminus \sigma x_i], C[\] \rangle & \Leftrightarrow & \langle Q, \mathcal{C}, \sigma \rangle \rightarrow_T \langle Q[x_i \setminus \sigma x_i], \mathcal{C} \rangle \\
\langle Q ; f(v_1, \dots, v_n), C[\] \rangle \rightarrow_Q \langle Q, C[\] ; f(v_1, \dots, v_n) \rangle & \Leftrightarrow & \langle \text{push } f(v_1, \dots, v_n) ; Q, \mathcal{C} \rangle \rightarrow_Q \langle Q, f(v_1, \dots, v_n) \cdot \mathcal{C} \rangle \\
\langle Q ; f(v_1, \dots, v_n, F), C[\] \rangle \rightarrow_Q \langle Q ; F, C[\] ; f(v_1, \dots, v_n, \ominus) \rangle & \Leftrightarrow & \langle \text{push } f(v_1, \dots, v_n, \square) ; Q', \mathcal{C} \rangle \rightarrow_Q \langle Q', f(v_1, \dots, v_n, \square) \cdot \mathcal{C} \rangle \\
\langle \varepsilon, C[\] ; f(v_1, \dots, v_n) \rangle \rightarrow_Q \langle T, C[\], \sigma \rangle & \Leftrightarrow & \langle \text{invoke}, f(v_1, \dots, v_n) \cdot \mathcal{C} \rangle \rightarrow_Q \langle T, \mathcal{C}, \sigma \rangle \\
& & \text{when } f(x_1, \dots, x_n) = T \text{ and } \sigma = \{x_i \mapsto v_i\}
\end{array}$$

Fig. 1 Isomorphism between reduction rules

6 Rewriting CPC terms into CPS-convertible form

A highly motivated programmer could, in principle, write his code directly in CPS-convertible form. This would actually yield code analogous to event-driven code: each cooperation point would be followed by a call to a cps function encapsulating the rest of the computation, i.e. an event handler.

One aim of the CPC translator is to relieve the programmer from the burden of writing CPS-convertible code directly. It therefore creates automatically those encapsulating functions, taking care to preserve the flow of control between imperative blocks through mutually recursive function calls.

To transform a CPC program into CPS-convertible form, the CPC translator performs two successive passes. During the first pass, it makes continuations explicit with `goto` statements (Section 6.1). In the second pass, it eliminates `gotos`, replacing them with tail calls to inner cps functions (Section 6.2). Since inner functions are not allowed in C, a third transformation is actually necessary: lambda-lifting, that we cover in Section 7.

6.1 Making continuations explicit

When a cps call is not immediately followed by another cps call or a `return`, the CPC translator adds a `goto` statement after it to make the flow of control explicit.

The simplest case is trivial:

```
cpc_yield(); rc = 0;
```

becomes

```
cpc_yield(); goto l;
l: rc = 0;
```

Loops and conditional jumps require more care:

```
while(!timeout) {
    int rc = cpc_read();
    if(rc <= 0) break;
    cpc_write();
}
reset_timeout();
```

is converted to:

```
while_label:
    if(!timeout) {
        int rc = cpc_read(); goto l;
        l:
            if(rc <= 0) goto break_label;
            cpc_write(); goto while_label;
    }
break_label:
    reset_timeout();
```

Since labelled blocks will be encapsulated in separate functions by the next pass, we need to keep track of their continuations, in order to preserve the flow of control from one block to another. To this end, we also add `gotos` at the exit points of labelled blocks. For instance, suppose our first example were included in an `if` statement:

```

if(rc < -1) {
    cpc_yield(); rc = 0;
}
return rc;

```

It would be translated to:

```

if(rc < 0) {
    cpc_yield(); goto l;
    l: { rc = 0; goto exit; }
}
exit: return rc;

```

After this step, every sequence of cps calls (and every labelled block) end with either a `return` or a `goto` statement.

6.2 Eliminating gotos

It is a well-known fact in the compiler folklore that a tail call is equivalent to a `goto`. It is less known that a `goto` is equivalent to a tail call [45,53]: the block of any destination label `l` is encapsulated in an inner function `l()`, and each `goto l`; is replaced by a tail call to `l`.

Coming back to our first example of Section 6.1, the label `l` yields an `l()` function:

```

f(); l(); return;
cps void l() { x = 0; }

```

We see that the first line has become a sequence of function calls: this fragment is in CPS-convertible form.

Loops yield mutually recursive functions, like `l` and `while_label` in our second example:

```

while_label();
cps void while_label() {
    if(!timeout) {
        int rc = cpc_read(); l(); return;
        cps void l() {
            if(rc <= 0) break_label(); return;
            cpc_write(); while_label(); return;
        }
    }
}
cps void break_label() {
    reset_timeout();
}

```

When a label, like `while_label` above, is reachable not only through `gotos`, but also directly through the linear flow of the program, it is necessary to call its associated function at the point of definition; this is what we do on the first line of this example. Also notice how inner functions might include free variables, e.g. `rc` in `l`. This code is no longer valid C.

7 Lambda-lifting

After goto elimination, there are no cps calls left in non-tail position, as gotos have been converted into tail calls to mutually recursive inner functions. But inner functions, which may contain free variables, are neither part of the C standard nor present in our definition of CPS-convertible terms. They must therefore be eliminated with an additional pass, called lambda-lifting.

Lambda-lifting [24] is a standard technique to eliminate free variables in functional languages. It proceeds in two phases [13]. In the first pass (“parameter lifting”), free variables are replaced by local, bound variables. In the second pass (“block floating”), the resulting closed functions are floated to toplevel.

In the following example, the variable `x` is free in `add`. Lifting variable `x` leads to:

```
int f(x) {
  int add(y) { return x + y; }
  return add(4);
}
f(3);
```

Parameter lifting leads to:

```
int f(x) {
  int add(y, x) { return x + y; }
  return add(4, x);
}
f(3);
```

which, after block floating (and alpha-conversion), becomes:

```
int add(y, z) { return z + y; }
int f(x) { return add(4, x); }
f(3);
```

Both the original and the lambda-lifted version of `g` return 7: in this example, lambda-lifting preserves the result of the computation.

7.1 Lambda-lifting in imperative languages

Lambda-lifting is in general not correct in a call-by-value languages with mutable variables such as C.

Here is an example of what may go wrong with mutable variables. The following code returns 1:

```
int f(x) {
  void set() { x = 1; }
  set(); return x;
}
f(0);
```

After lambda-lifting, the transformed code returns 0:

```

int f(x) {
    void set(x) { x = 1; }
    set(x); return x;
}
f(0);

```

The `set` function no longer shares the mutable `x` variable with `f`. Instead, it modifies the lifted copy of `x`: the changes are not reflected on the original variable.

Boxing mutable variables A common workaround consists in boxing every mutable variable that needs to be lifted, either transparently in the translator or explicitly in the source. This is a perfectly reasonable technique in functional languages, where the number of mutable variables is usually small. In an imperative language such as CPC, however, where almost all variables are mutated, the cost would be prohibitive: it would put a huge amount of pressure on the memory allocator [33], and increase the number of memory accesses by systematically adding a level of indirection.

Lambda-lifting functions in tail position It turns out that it is not necessary to box every lambda-lifted variable. In particular, in the absence of “extruded” variables (the addresses of which have been retained with the `&` operator), lambda-lifting functions that are called in “hereditary tail position” turns out to be correct.

Coming back to our example above, notice that the `set` function is not called in tail position: this allows us to observe the incorrect value of (the original) `x` after `set` has returned. If it were called in tail position instead, the fact that it operates on a copy of `x` would remain unnoticed.

Since the functions introduced in the previous steps of the CPC translator are only ever called in tail position, boxing “extruded” variables is enough to ensure correctness (see Section 10.1) while introducing much less overhead than boxing every variable.

Outline of the proof We show that, assuming the absence of extruded variables, lambda-lifting functions called in hereditary tail position is correct in a call-by-name imperative language.

We first give experimental results to show how important it is to avoid unnecessary boxing in a real CPC program (Section 7.2).

In order to reason about lambda-lifting, we need to define reduction rules (Section 7.3) and the lambda-lifting transformation itself (Section 7.4) for our small imperative language. With those preliminary definitions, we are then able to characterise *liftable parameters* (Definition 17) and state our main correctness theorem (Theorem 2, Section 7.5).

It turns out that the “naive” reduction rules defined in Section 7.3 do not provide strong enough invariants to prove the correctness theorem by induction. In Section 8, we therefore define an equivalent, “optimised” set of reduction rules. The actual proof of correctness is carried in Section 9 using those optimised rules.

7.2 Experimental results

In order to verify our assumption about boxing cost, we used a modified version of CPC which blindly boxes every lifted parameter and measured the amount of boxing that it induced in Hekate, the most substantial program written with CPC so far.

Hekate contains 260 local variables and function parameters, spread across 28 cps functions¹. Among them, 125 variables are lifted. A naive lambda-lifting pass would therefore need to box almost 50 % of the variables.

On the other hand, boxing extruded variables only carries a much smaller overhead: in Hekate, the current CPC compiler boxes only 5 % of the variables in cps functions. In other words, 90 % of the lifted variables in Hekate are safely left unboxed (see Section 11 for more detail).

7.3 Naive reduction rules

In this section, we define naive reduction rules for our small subset of C. Let us recall the structure of a term in this language (Definition 1, p. 11):

$$\begin{aligned}
 v &::= \mathbf{1} \mid \mathbf{true} \mid \mathbf{false} \mid n \in \mathbf{N} && \text{(values)} \\
 \text{expr} &::= v \mid x \mid \dots && \text{(expressions)} \\
 T &::= \text{expr} \mid x := T \mid \mathbf{if } T \mathbf{ then } T \mathbf{ else } T \mid T ; T \\
 &\quad \mid \mathbf{letrec } f(x_1 \dots x_n) = T \mathbf{ in } T \mid f(T, \dots, T) && \text{(terms)}
 \end{aligned}$$

This language is somewhat less constrained than CPS-convertible terms, since it has inner functions and function calls in any position.

Environments and stores Handling inner functions requires explicit closures in the reduction rules, and means that the direct correspondence between variable names and locations no longer holds. We need environments, written ρ , to bind variables to locations; the store, now written s , consequently binds locations to values.

Environments and *stores* are partial functions, defined with a single operator which extends and modifies a partial function: $\cdot + \{ \cdot \mapsto \cdot \}$.

Definition 11 The modification (or extension) f' of a partial function f , written $f' = f + \{x \mapsto y\}$, is defined as follows:

$$\begin{aligned}
 f'(t) &= \begin{cases} y & \text{when } t = x \\ f(t) & \text{otherwise} \end{cases} \\
 \text{dom}(f') &= \text{dom}(f) \cup \{x\}
 \end{aligned}$$

Definition 12 (Environments of variables and functions) Environment of variables are defined inductively by

$$\rho ::= \varepsilon \mid (x, l) \cdot \rho,$$

i.e. the empty domain function and $\rho + \{x \mapsto l\}$ (respectively).

Environments of functions, on the other hand, associate function names to closures:

$$\mathcal{F} : \{f, g, h, \dots\} \rightarrow \{[\lambda x_1 \dots x_n. T, \rho, \mathcal{F}]\}.$$

Reduction 1 “Naive” reduction rules

$$\begin{array}{c}
\text{(VAL)} \frac{}{v^s \xrightarrow[\mathcal{F}]{\rho} v^s} \quad \text{(VAR)} \frac{\rho \ x = l \in \text{dom } s}{x^s \xrightarrow[\mathcal{F}]{\rho} s \ l^s} \\
\text{(ASSIGN)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho} v^{s'} \quad \rho \ x = l \in \text{dom } s'}{x := a^s \xrightarrow[\mathcal{F}]{\rho} \mathbf{1}^{s'+\{l \mapsto v\}}} \quad \text{(SEQ)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho} v^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho} v^{s''}}{a ; b^s \xrightarrow[\mathcal{F}]{\rho} v^{s''}} \\
\text{(IF-T.)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho} \mathbf{true}^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho} v^{s''}}{\mathbf{if } a \text{ then } b \text{ else } c^s \xrightarrow[\mathcal{F}]{\rho} v^{s''}} \quad \text{(IF-F.)} \frac{a^s \xrightarrow[\mathcal{F}]{\rho} \mathbf{false}^{s'} \quad c^{s'} \xrightarrow[\mathcal{F}]{\rho} v^{s''}}{\mathbf{if } a \text{ then } b \text{ else } c^s \xrightarrow[\mathcal{F}]{\rho} v^{s''}} \\
\text{(LETREC)} \frac{b^s \xrightarrow[\mathcal{F}']{\rho} v^{s'} \quad \mathcal{F}' = \mathcal{F} + \{f \mapsto [\lambda x_1 \dots x_n. a, \rho, \mathcal{F}']\}}{\mathbf{letrec } f(x_1 \dots x_n) = a \text{ in } b^s \xrightarrow[\mathcal{F}]{\rho} v^{s'}} \\
\text{(CALL)} \frac{\mathcal{F} f = [\lambda x_1 \dots x_n. b, \rho', \mathcal{F}'] \quad \rho'' = (x_1, l_1) \dots (x_n, l_n) \quad l_i \text{ fresh and distinct} \\ \forall i, a_i^{s_i} \xrightarrow[\mathcal{F}]{\rho} v_i^{s_i+1} \quad b^{s_{n+1}+\{l_i \mapsto v_i\}} \xrightarrow[\mathcal{F}'+\{f \mapsto \mathcal{F} f\}]{\rho'' \cdot \rho'} v^{s'}}{f(a_1 \dots a_n)^{s_1} \xrightarrow[\mathcal{F}]{\rho} v^{s'}}
\end{array}$$

Reduction rules We use classical big-step reduction rules (see Reduction 1, p. 25) for our language, with a small trick in the (letrec) and (call) rules to allow recursive calls without using a fix-point operator.

We define two functions, Env and Loc, to extract the environments and locations contained in the closures of a given environment of functions \mathcal{F} :

Definition 13 (Set of environments, set of locations)

$$\text{Env}(\mathcal{F}) = \bigcup \{ \rho, \rho' \mid [\lambda x_1 \dots x_n. M, \rho, \mathcal{F}'] \in \text{Im}(\mathcal{F}), \rho' \in \text{Env}(\mathcal{F}') \}$$

$$\text{Loc}(\mathcal{F}) = \bigcup \{ \text{Im}(\rho) \mid \rho \in \text{Env}(\mathcal{F}) \}$$

A location l is said to *appear* in \mathcal{F} iff $l \in \text{Loc}(\mathcal{F})$.

Those functions allow us to define fresh locations:

Definition 14 (Fresh location) In the (call) rule, a location is *fresh* when:

- $l \notin \text{dom}(s_{n+1})$, i.e. l is not already used in the store before the body of f is evaluated, and
- l doesn't appear in $\mathcal{F}' + \{f \mapsto \mathcal{F} f\}$, i.e. l will not interfere with locations captured in the environment of functions.

The second condition implies in particular that l does not appear in either \mathcal{F} or ρ' .

¹ These numbers leave out direct-style functions, and a cps function with 221 variables due to pathological macro-expansion in the *curl* library.

7.4 Lambda-lifting

We mentioned before that lambda-lifting can be split into two parts: parameters lifting and block floating. We will focus only on the first part here, since the second one is trivial. Rather than considering the lifting of a given function (“lambda”), we define the lifting of a given parameter x in a term M .

Usually, smart lambda-lifting algorithms strive to minimize the number of lifted variables. Such is not our concern in this proof: parameters are lifted in every function where they might potentially be free. Of course, the CPC compiler actually uses a smarter approach to avoid lifting too many parameters (see Section 11).

Definition 15 (Parameter lifting in a term) Assume that x is defined as a parameter of a given function, g , and that every inner function in g is called h_i (for some $i \in \mathbf{N}$). Also assume that function parameters are unique before lambda-lifting.

Then, the *lifted form* $(M)_*$ of the term M with respect to x is defined inductively as follows:

$$\begin{aligned}
 (\mathbf{1})_* &= \mathbf{1} & (n)_* &= n \\
 (\text{true})_* &= \text{true} & (\text{false})_* &= \text{false} \\
 (y)_* &= y & \text{and } (y := a)_* &= y := (a)_* \quad (\text{even if } y = x) \\
 (a ; b)_* &= (a)_* ; (b)_* \\
 (\text{if } a \text{ then } b \text{ else } c)_* &= \text{if } (a)_* \text{ then } (b)_* \text{ else } (c)_* \\
 (\text{letrec } f(x_1 \dots x_n) = a \text{ in } b)_* &= \begin{cases} \text{letrec } f(x_1 \dots x_n x) = (a)_* \text{ in } (b)_* & \text{if } f = h_i \\ \text{letrec } f(x_1 \dots x_n) = (a)_* \text{ in } (b)_* & \text{otherwise} \end{cases} \\
 (f(a_1 \dots a_n))_* &= \begin{cases} f((a_1)_*, \dots, (a_n)_*, x) & \text{if } f = h_i \text{ for some } i \\ f((a_1)_*, \dots, (a_n)_*) & \text{otherwise} \end{cases}
 \end{aligned}$$

7.5 Correctness condition

A notable fact about CPS-convertible functions is that the inner functions resulting from the elimination of gotos are called exclusively in tail position. We want to show that this structural property ensures the correctness of lambda-lifting in CPC.

We claim that parameter lifting is correct for variables defined in functions whose inner functions are called exclusively in *tail position*. We call those variables *liftable parameters*.

We define tail positions as usual [11]. We also define tail position *with respect to some parameter* x as the set of all tail positions in the function where x is defined.

Definition 16 (Tail position) *Tail positions* are defined inductively as follows:

1. M and N are in tail position in **if** P **then** M **else** N .
2. N is in tail position in $N, M ; N$ and **letrec** $f(x_1 \dots x_n) = M$ **in** N .

In a term M , we define tail positions *with respect to a parameter* x as the set of tail positions in N , when **letrec** $f(\dots, x, \dots) = N$ **in** P is a sub-term of M .

Definition 17 (Liftable parameter) A parameter x is *liftable* in M when:

- x is defined as the parameter of a function g ,

- inner functions in g , named h_i , are called exclusively in tail position in g or in one of the h_i .

In other words, a parameter x defined in a function g is liftable if the inner functions in g are called exclusively in tail position with respect to x .

This leads us to our main theorem:

Theorem 2 (Correctness of lambda-lifting) *If x is a liftable parameter in M , then*

$$M \xrightarrow[\varepsilon]{\varepsilon} v^\varepsilon \text{ implies } \exists t, (M)_* \xrightarrow[\varepsilon]{\varepsilon} v^t.$$

Section 9 is devoted to the proof of this theorem. To maintain invariants during the proof, we need to use an equivalent, “optimised” set of reduction rules; it is introduced in the next section.

8 Optimised reduction rules

The naive reduction rules (Section 7.3) are not well-suited to prove the correctness of lambda-lifting. In this Section, we introduce two optimisations, minimal stores (Section 8.1) and compact closures (Section 8.2), which lead to the definition of an optimised set of reduction rules (Reduction 2, Section 8.3). We then prove that optimised and naive reduction rules are equivalent (Section 8.4).

8.1 Minimal stores

In the naive reduction rules, the store grows faster when reducing lifted terms, because each function call adds to the store as many locations as it has function parameters. This yields stores of different sizes when reducing the original and the lifted term, and that difference cannot be accounted locally, at the rule level.

Consider for instance the first example of Section 7:

```
int f(x) {
  int add(y) { return x + y; }
  return add(4);
}
f(3);
```

At the end of the reduction, the store is $\{l_x \mapsto 3; l_y \mapsto 4\}$; once lifted, the functions become

```
int add(y, z) { return z + y; }
int f(x) { return add(4, x); }
f(3);
```

which yields a different store: $\{l_x \mapsto 3; l_y \mapsto 4; l_z \mapsto 3\}$.

To keep the store under control, we need to get rid of useless variables as soon as possible during the reduction. It is safe to remove a variable from the store once we are certain that it will never be used again, i.e. as soon as the last term in tail position with respect to this variable has been evaluated. This mechanism is similar to the deallocation of a stack frame when a function returns.

We introduce *split environments*, which keep track of which variables are in tail position with respect to the current environment. They are written $\rho_T | \rho$, where the *tail environment* ρ_T contains the variables with respect to which we are in tail position. The *cleaning* operator, $\cdot \setminus \cdot$, removes those variables from the stores:

Definition 18 (Cleaning of a store) The store s cleaned with respect to the variables in ρ , written $s \setminus \rho$, is defined as:

$$s \setminus \rho = s|_{\text{dom}(s) \setminus \text{Im}(\rho)}.$$

8.2 Compact closures

Another source of overhead of naive reduction rules is the inclusion of useless variables in closures. It is safe to remove from the environments contained in closures the variables that are also parameters of the function: when the function is called, and the environment restored, those variables will be hidden by the freshly instantiated parameters.

This is typically what happens to lifted parameters: they are free variables, captured in the closure when the function is defined, but those captured values will never be used since calling the function adds a fresh parameters with the same names. We introduce *compact closures* in the optimised reduction rules to avoid dealing with this hiding mechanism manually.

Definition 19 (Compact closure and environment) A closure $[\lambda x_1 \dots x_n. M, \rho, \mathcal{F}]$ is *compact* if $\forall i, x_i \notin \text{dom}(\rho)$ and \mathcal{F} is compact. An environment is *compact* if it contains only compact closures.

We define a canonical mapping from any environment \mathcal{F} to a compact environment \mathcal{F}_* , restricting the domains of every closure in \mathcal{F} .

Definition 20 (Canonical compact environment) The *canonical compact environment* \mathcal{F}_* is the unique environment with the same domain as \mathcal{F} such that

$$\begin{aligned} \forall f \in \text{dom}(\mathcal{F}), \mathcal{F} f &= [\lambda x_1 \dots x_n. M, \rho, \mathcal{F}'] \\ \text{implies } \mathcal{F}_* f &= [\lambda x_1 \dots x_n. M, \rho|_{\text{dom}(\rho) \setminus \{x_1 \dots x_n\}}, \mathcal{F}'_*]. \end{aligned}$$

8.3 Optimised reduction rules

Combining both optimisations yields the *optimised* reduction rules (Reduction 2, p. 29), used Section 9 for the proof of lambda-lifting.

Theorem 3 (Equivalence between naive and optimised reduction rules) *Optimised and naive reduction rules are equivalent: every reduction in one set of rules yields the same result in the other. It is necessary, however, to take care of locations left in the store by the naive reduction:*

$$M^\varepsilon \xrightarrow[\varepsilon]{\varepsilon|\varepsilon} v^\varepsilon \quad \text{iff} \quad \exists s, M^\varepsilon \xrightarrow[\varepsilon]{\varepsilon} v^s$$

We prove this theorem in the next section.

Reduction 2 Optimised reduction rules

$$\begin{array}{c}
 \text{(VAL)} \frac{}{v^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^s \setminus \rho_T} \quad \text{(VAR)} \frac{\rho_T \cdot \rho \quad x = l \in \text{dom } s}{x^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} s \setminus \rho_T} \\
 \\
 \text{(ASSIGN)} \frac{a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} v^{s'} \quad \rho_T \cdot \rho \quad x = l \in \text{dom } s'}{x := a^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} \mathbf{1}^{s' + \{l \mapsto v\}} \setminus \rho_T} \quad \text{(SEQ)} \frac{a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} v^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s''}}{a ; b^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s''}} \\
 \\
 \text{(IF-T.)} \frac{a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} \mathbf{true}^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s''}}{\mathbf{if } a \text{ then } b \text{ else } c^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s''}} \quad \text{(IF-F.)} \frac{a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} \mathbf{false}^{s'} \quad c^{s'} \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s''}}{\mathbf{if } a \text{ then } b \text{ else } c^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s''}} \\
 \\
 \text{(LETREC)} \frac{b^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s'} \quad \rho' = \rho_T \cdot \rho |_{\text{dom}(\rho_T \cdot \rho) \setminus \{x_1 \dots x_n\}} \quad \mathcal{F}' = \mathcal{F} + \{f \mapsto [\lambda x_1 \dots x_n. a, \rho', \mathcal{F}]\}}{\mathbf{letrec } f(x_1 \dots x_n) = a \text{ in } b^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s'}} \\
 \\
 \text{(CALL)} \frac{\mathcal{F} f = [\lambda x_1 \dots x_n. b, \rho', \mathcal{F}'] \quad \rho'' = (x_1, l_1) \dots (x_n, l_n) \quad l_i \text{ fresh and distincts} \quad \forall i, a_i^{s_i} \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} v_i^{s_i+1} \quad b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{F}' + \{f \mapsto \mathcal{F} f\}]{\rho'' | \rho'} v^{s'}}{f(a_1 \dots a_n)^{s_1} \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s'} \setminus \rho_T}
 \end{array}$$

8.4 Equivalence of optimised and naive reduction rules

This section is devoted to the proof of equivalence between the optimised naive reduction rules (Theorem 3).

To clarify the proof, we introduce intermediate reduction rules (Reduction 3, p. 30), with only one of the two optimisations: minimal stores, but not compact closures.

The proof then consists in proving that optimised and intermediate rules are equivalent (Lemma 5 and Lemma 6, Section 8.4.1), then that naive and intermediate rules are equivalent (Lemma 9 and Lemma 10, Section 8.4.2).

$$\text{Naive rules} \xrightleftharpoons[\text{Lemma 9}]{\text{Lemma 10}} \text{Intermediate rules} \xrightleftharpoons[\text{Lemma 6}]{\text{Lemma 5}} \text{Optimised rules}$$

8.4.1 Optimised and intermediate reduction rules equivalence

In this section, we show that optimised and intermediate reduction rules are equivalent:

$$\text{Intermediate rules} \xrightleftharpoons[\text{Lemma 6}]{\text{Lemma 5}} \text{Optimised rules}$$

We must therefore show that it is correct to use compact closures in the optimised reduction rules.

Compact closures carry the implicit idea that some variables can be safely discarded from the environments when we know for sure that they will be hidden. The following lemma formalises this intuition.

Reduction 3 Intermediate reduction rules

$$\begin{array}{c}
\text{(VAL)} \frac{}{v^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^s \setminus \rho_T} \quad \text{(VAR)} \frac{\rho_T \cdot \rho \quad x = l \in \text{dom } s}{x^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} s \setminus l^s \setminus \rho_T} \\
\text{(ASSIGN)} \frac{a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} v^{s'} \quad \rho_T \cdot \rho \quad x = l \in \text{dom } s'}{x := a^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} \mathbf{1}^{s' + \{l \mapsto v\}} \setminus \rho_T} \quad \text{(SEQ)} \frac{a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} v^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s''}}{a ; b^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s''}} \\
\text{(IF-T.)} \frac{a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} \mathbf{true}^{s'} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s''}}{\mathbf{if } a \text{ then } b \text{ else } c^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s''}} \quad \text{(IF-F.)} \frac{a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} \mathbf{false}^{s'} \quad c^{s'} \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s''}}{\mathbf{if } a \text{ then } b \text{ else } c^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s''}} \\
\text{(LETREC)} \frac{b^s \xrightarrow[\mathcal{F}']{\rho_T | \rho} v^{s'} \quad \rho' = \rho_T \cdot \rho \quad \mathcal{F}' = \mathcal{F} + \{f \mapsto [\lambda x_1 \dots x_n. a, \rho, \mathcal{F}']\}}{\mathbf{letrec } f(x_1 \dots x_n) = a \mathbf{ in } b^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s'}} \\
\text{(CALL)} \frac{\mathcal{F} f = [\lambda x_1 \dots x_n. b, \rho', \mathcal{F}'] \quad \rho'' = (x_1, l_1) \cdot \dots \cdot (x_n, l_n) \quad l_i \text{ fresh and distinct} \\ \forall i, a_i^{s_i} \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} v_i^{s_i+1} \quad b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{F}']{\rho'' | \rho'} v^{s'}}{f(a_1 \dots a_n)^{s_1} \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s' \setminus \rho_T}}
\end{array}$$

Lemma 4 (Hidden variables elimination)

$$\begin{array}{c}
\forall l, l', M^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | \rho} v^{s'} \quad \text{iff} \quad M^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | (x, l') \cdot \rho} v^{s'} \\
\forall l, l', M^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | \rho} v^{s'} \quad \text{iff} \quad M^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | (x, l') \cdot \rho} v^{s'}
\end{array}$$

Moreover, both derivations have the same height.

Proof The exact same proof holds for both intermediate and optimised reduction rules.

By induction on the structure of the derivation. The proof relies solely on the fact that $\rho_T \cdot (x, l) \cdot \rho = \rho_T \cdot (x, l) \cdot (x, l') \cdot \rho$. We show only the (seq) case, the other cases are similar.

(seq) $\rho_T \cdot (x, l) \cdot \rho = \rho_T \cdot (x, l) \cdot (x, l') \cdot \rho$. So,

$$a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot (x, l) \cdot (x, l') \cdot \rho} v^{s'} \quad \text{iff} \quad a^s \xrightarrow[\mathcal{F}]{|\rho_T \cdot (x, l) \cdot \rho} v^{s'}$$

Moreover, by the induction hypotheses,

$$b^{s'} \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | (x, l') \cdot \rho} v^{s''} \quad \text{iff} \quad b^{s'} \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | \rho} v^{s''}$$

Hence,

$$a ; b^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | (x, l') \cdot \rho} v^{s''} \quad \text{iff} \quad a ; b^s \xrightarrow[\mathcal{F}]{\rho_T \cdot (x, l) | \rho} v^{s''}$$

□

Now we can show the required lemmas and prove the equivalence between the intermediate and optimised reduction rules.

Lemma 5 (Intermediate implies optimised)

$$\text{If } M^s \xrightarrow[\mathcal{F}]{\rho_T|\rho} v^{s'} \text{ then } M^s \xrightarrow[\mathcal{F}_*]{\rho_T|\rho} v^{s'}.$$

Proof By induction on the structure of the derivation. We show only the interesting cases, namely (letrec) and (call), where compact environments are respectively built and used.

(letrec) By the induction hypotheses,

$$b^s \xrightarrow[\mathcal{F}'_*]{\rho_T|\rho} v^{s'}$$

Since we defined canonical compact environments so as to match exactly the way compact environments are built in the optimised reduction rules, the constraints of the (letrec) rule are fulfilled:

$$\mathcal{F}'_* = \mathcal{F}_* + \{f \mapsto [\lambda x_1 \dots x_n. a, \rho', \mathcal{F}_*]\},$$

hence:

$$\text{letrec } f(x_1 \dots x_n) = a \text{ in } b^s \xrightarrow[\mathcal{F}_*]{\rho_T|\rho} v^{s'}$$

(call) By the induction hypotheses,

$$\forall i, a_i^{s_i} \xrightarrow[\mathcal{F}_*]{|\rho_T-\rho} v_i^{s_{i+1}}$$

and

$$b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{F}' + \{f \mapsto \mathcal{F} f\}_*]{\rho''|\rho'} v^{s'}$$

Lemma 4 allows to remove hidden variables, which leads to

$$b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{F}' + \{f \mapsto \mathcal{F} f\}_*]{\rho''|\rho'_{|\text{dom}(\rho') \setminus \{x_1 \dots x_n\}}} v^{s'}$$

Besides,

$$\mathcal{F}_* f = [\lambda x_1 \dots x_n. b, \rho'_{|\text{dom}(\rho') \setminus \{x_1 \dots x_n\}}, \mathcal{F}'_*]$$

and

$$(\mathcal{F}' + \{f \mapsto \mathcal{F} f\})_* = \mathcal{F}'_* + \{f \mapsto \mathcal{F}_* f\}$$

Hence

$$f(a_1 \dots a_n)^{s_1} \xrightarrow[\mathcal{F}_*]{\rho_T|\rho} v^{s' \setminus \rho_T}.$$

□

Lemma 6 (Optimised implies intermediate)

$$\text{If } M^s \xrightarrow[\mathcal{F}]{\rho_T|\rho} v^{s'} \text{ then } \forall \mathcal{G} \text{ such that } \mathcal{G}_* = \mathcal{F}, M^s \xrightarrow[\mathcal{G}]{\rho_T|\rho} v^{s'}.$$

Proof First note that, since $\mathcal{G}_* = \mathcal{F}$, \mathcal{F} is necessarily compact.

By induction on the structure of the derivation. We show only the interesting cases, namely (letrec) and (call), where non-compact environments are respectively built and used.

(*letrec*) Let \mathcal{G} such as $\mathcal{G}_* = \mathcal{F}$. Remember that $\rho' = \rho_T \cdot \rho |_{\text{dom}(\rho_T \cdot \rho) \setminus \{x_1 \dots x_n\}}$. Let

$$\mathcal{G}' = \mathcal{G} + \{f \mapsto [\lambda x_1 \dots x_n. a, \rho_T \cdot \rho, \mathcal{F}]\}$$

which leads, since \mathcal{F} is compact ($\mathcal{F}_* = \mathcal{F}$), to

$$\begin{aligned} \mathcal{G}'_* &= \mathcal{F} + \{f \mapsto [\lambda x_1 \dots x_n. a, \rho', \mathcal{F}]\} \\ &= \mathcal{F}' \end{aligned}$$

By the induction hypotheses,

$$b^s \xrightarrow[\mathcal{G}']{\rho_T \cdot \rho} v^{s'}$$

Hence,

$$\mathbf{letrec} \ f(x_1 \dots x_n) = a \ \mathbf{in} \ b^s \xrightarrow[\mathcal{G}]{\rho_T \cdot \rho} v^{s'}$$

(*call*) Let \mathcal{G} such as $\mathcal{G}_* = \mathcal{F}$. By the induction hypotheses,

$$\forall i, a_i^{s_i} \xrightarrow[\mathcal{G}]{|\rho_T \cdot \rho|} v_i^{s_{i+1}}$$

Moreover, since $\mathcal{G}_* f = \mathcal{F} f$,

$$\mathcal{G} f = [\lambda x_1 \dots x_n. b, (x_i, l_i) \cdot \dots \cdot (x_j, l_j) \rho', \mathcal{G}']$$

where $\mathcal{G}'_* = \mathcal{F}'$, and the l_i are some locations stripped out when compacting \mathcal{G} to get \mathcal{F} .

By the induction hypotheses,

$$b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{G}' + \{f \mapsto \mathcal{G} f\}]{\rho'' | \rho'} v^{s'}$$

Lemma 4 leads to

$$b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{G}' + \{f \mapsto \mathcal{G} f\}]{\rho'' | (x_i, l_i) \dots (x_j, l_j) \rho'} v^{s'}$$

Hence,

$$f(a_1 \dots a_n)^{s_1} \xrightarrow[\mathcal{G}]{\rho_T \cdot \rho} v^{s' \setminus \rho_T}.$$

□

8.4.2 Intermediate and naive reduction rules equivalence

In this section, we show that the naive and intermediate reduction rules are equivalent:

$$\text{Naive rules} \xrightleftharpoons[\text{Lemma 9}]{\text{Lemma 10}} \text{Intermediate rules}$$

We must therefore show that it is correct to use minimal stores in the intermediate reduction rules. We first define a partial order on stores:

Definition 21 (Store extension)

$$s \sqsubseteq s' \quad \text{iff} \quad s' |_{\text{dom}(s)} = s$$

Property 1 Store extension (\sqsubseteq) is a partial order over stores. The following operations preserve this order: $\cdot \setminus \rho$ and $\cdot + \{l \mapsto v\}$, for some given ρ , l and v .

Proof Immediate when considering the stores as function graphs: \sqsubseteq is the inclusion, $\cdot \setminus \rho$ a relative complement, and $\cdot + \{l \mapsto v\}$ a disjoint union (preceded by $\cdot \setminus (l, v')$ when l is already bound to some v'). \square

Before we prove that using minimal stores is equivalent to using full stores, we need an alpha-conversion lemma, which allows us to rename locations in the store, provided the new location does not already appear in the store or the environments. It is used when choosing a fresh location for the (call) rule in proofs by induction.

Lemma 7 (Alpha-conversion) *If $M^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s'}$ then, for all l , for all l' appearing neither in s nor in \mathcal{F} nor in $\rho \cdot \rho_T$,*

$$M^{s[l'/l]} \xrightarrow[\mathcal{F}[l'/l]]{\rho_T[l'/l] | \rho[l'/l]} v^{s'[l'/l]}.$$

Moreover, both derivations have the same height.

Proof A lengthy and tedious exhaustive verification. The only non-trivial case is that of the (call) rule, where one must ensure that the fresh locations l_i do not clash with l' . In case they do, we conclude by applying the induction hypotheses twice: first to rename the clashing l_i into a fresh l'_i , then to rename l into l' . \square

To prove that using minimal stores is correct, we need to extend them so as to recover the full stores of naive reduction. The following lemma shows that extending a store before an (intermediate) reduction extends the resulting store too:

Lemma 8 (Extending a store in a derivation)

$$\text{Given the reduction } M^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s'}, \text{ then } \forall t \sqsupseteq s, \exists t' \sqsupseteq s', M^t \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{t'}.$$

Moreover, both derivations have the same height.

Proof By induction on the height of the derivation. We focus on the most interesting case, namely (call), which requires alpha-converting a location (hence the induction on the height rather than the structure of the derivation).

(var), (val) and (assign) are straightforward by the induction hypotheses and Property 1.

(seq), (if-true), (if-false) and (letrec) are straightforward by the induction hypotheses.

(call) Let $t_1 \sqsupseteq s_1$. By the induction hypotheses,

$$\begin{aligned} \exists t_2 \sqsupseteq s_2, a_1^{t_1} &\xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho|} v_1^{t_2} \\ \exists t_{i+1} \sqsupseteq s_{i+1}, a_i^{t_i} &\xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho|} v_i^{t_{i+1}} \\ \exists t_{n+1} \sqsupseteq s_{n+1}, a_n^{t_n} &\xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho|} v_n^{t_{n+1}} \end{aligned}$$

The locations l_i might belong to $\text{dom}(t_{n+1})$ and thus not be fresh. By alpha-conversion (Lemma 7), we chose fresh l'_i (not in $\text{Im}(\rho')$ and $\text{dom}(s')$) such that

$$b^{s_{n+1} + \{l'_i \mapsto v_i\}} \xrightarrow[\mathcal{F}' + \{f \mapsto \mathcal{F} f\}]{(l'_i, v_i) \rho'} v^{s'}$$

By Property 1, $t_{n+1} + \{l'_i \mapsto v_i\} \sqsupseteq s_{n+1} + \{l'_i \mapsto v_i\}$. By the induction hypotheses,

$$\exists t' \sqsupseteq s', b^{t_{n+1} + \{l'_i \mapsto v_i\}} \xrightarrow[\mathcal{F}' + \{f \mapsto \mathcal{F} f\}]{(l'_i, v_i) \rho'} v^{t'}$$

Moreover, $t' \setminus \rho_T \sqsupseteq s' \setminus \rho_T$. Hence,

$$f(a_1 \dots a_n)^{t_1} \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{t' \setminus \rho_T}.$$

□

Now we can show the required lemmas and prove the equivalence between the intermediate and naive reduction rules.

Lemma 9 (Intermediate implies naive)

$$\text{If } M^s \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s'} \text{ then } \exists t' \sqsupseteq s', M^s \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{t'}.$$

Proof By induction on the height of the derivation, because some stores are modified during the proof. We show only the interesting cases, namely (seq) and (call), where Lemma 8 is used to extend intermediary stores. Other cases are straightforward by Property 1 and the induction hypotheses.

(seq) By the induction hypotheses,

$$\exists t' \sqsupseteq s', a^s \xrightarrow[\mathcal{F}]{\rho} v^{t'}.$$

Moreover,

$$b^{s'} \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{s''}.$$

Since $t' \sqsupseteq s'$, Lemma 8 leads to:

$$\exists t \sqsupseteq s'', b^{t'} \xrightarrow[\mathcal{F}]{\rho_T \rho} v^{t'}$$

and the height of the derivation is preserved. By the induction hypotheses,

$$\exists t'' \sqsupseteq t, b^{t'} \xrightarrow[\mathcal{F}]{\rho} v^{t''}$$

Hence, since \sqsupseteq is transitive (Property 1),

$$\exists t'' \sqsupseteq s'', a; b^s \xrightarrow[\mathcal{F}]{\rho} v^{t''}.$$

(call) Similarly to the (seq) case, we apply the induction hypotheses and Lemma 8:

$$\begin{aligned}
\exists t_2 \sqsupseteq s_2, a_1^{s_1} &\xrightarrow[\mathcal{F}]{\rho} v_1^{t_2} && \text{(Induction)} \\
\exists t'_{i+1} \sqsupseteq s_{i+1}, a_i^{t_i} &\xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} v_i^{t'_{i+1}} && \text{(Lemma 8)} \\
\exists t_{i+1} \sqsupseteq t'_{i+1} \sqsupseteq s_{i+1}, a_i^{t_i} &\xrightarrow[\mathcal{F}]{\rho} v_i^{t_{i+1}} && \text{(Induction)} \\
\exists t'_{n+1} \sqsupseteq s_{n+1}, a_n^{t_n} &\xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} v_n^{t'_{n+1}} && \text{(Lemma 8)} \\
\exists t_{n+1} \sqsupseteq t'_{n+1} \sqsupseteq s_{n+1}, a_n^{t_n} &\xrightarrow[\mathcal{F}]{\rho} v_n^{t_{n+1}} && \text{(Induction)}
\end{aligned}$$

The locations l_i might belong to $\text{dom}(t_{n+1})$ and thus not be fresh. By alpha-conversion (Lemma 7), we choose a set of fresh l'_i (not in $\text{Im}(\rho')$ and $\text{dom}(s')$) such that

$$b^{s_{n+1} + \{l'_i \mapsto v_i\}} \xrightarrow[\mathcal{F}' + \{f \mapsto \mathcal{F} f\}]{(l'_i, v_i) \cdot \rho'} v^{s'}.$$

By Property 1, $t_{n+1} + \{l'_i \mapsto v_i\} \sqsupseteq s_{n+1} + \{l'_i \mapsto v_i\}$. Lemma 8 leads to,

$$\exists t \sqsupseteq s', b^{t_{n+1} + \{l'_i \mapsto v_i\}} \xrightarrow[\mathcal{F}' + \{f \mapsto \mathcal{F} f\}]{(l'_i, v_i) \cdot \rho'} v^t.$$

By the induction hypotheses,

$$\exists t' \sqsupseteq t \sqsupseteq s', b^{t_{n+1} + \{l'_i \mapsto v_i\}} \xrightarrow[\mathcal{F}' + \{f \mapsto \mathcal{F} f\}]{(l'_i, v_i) \cdot \rho'} v^{t'}.$$

Moreover, $t' \setminus \rho_T \sqsupseteq s' \setminus \rho_T$. Hence,

$$f(a_1 \dots a_n)^{s_1} \xrightarrow[\mathcal{F}]{\rho} v^{t' \setminus \rho_T}.$$

□

The proof of the converse property — i.e. if a term reduces in the naive reduction rules, it reduces in the intermediate reduction rules too — is more complex because the naive reduction rules provide very weak invariants about stores and environments. For that reason, we add an hypothesis to ensure that every location appearing in the environments ρ , ρ_T and \mathcal{F} also appears in the store s :

$$\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(s).$$

Moreover, since stores are often larger in the naive reduction rules than in the intermediate ones, we need to generalise the induction hypothesis.

Lemma 10 (Naive implies intermediate)

Assume $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(s)$. Then, $M^s \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{s'}$ implies

$$\forall t \sqsupseteq s \text{ such that } \text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(t), \quad M^t \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} v^{s' \upharpoonright_{\text{dom}(t) \setminus \text{Im}(\rho_T)}}.$$

Proof By induction on the structure of the derivation.

(val) Let $t \sqsubseteq s$. Then

$$\begin{aligned} t \setminus \rho_T &= s|_{\text{dom}(t) \setminus \text{Im}(\rho_T)} && \text{because } s|_{\text{dom}(t)} = t \\ &= s'|_{\text{dom}(t) \setminus \text{Im}(\rho_T)} && \text{because } s' = s \end{aligned}$$

Hence,

$$v^t \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^t \setminus \rho_T.$$

(var) Let $t \sqsubseteq s$ such that $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(t)$. Note that $l \in \text{Im}(\rho_T \cdot \rho) \subset \text{dom}(t)$ implies $t \ l = s \ l$. Then,

$$\begin{aligned} t \setminus \rho_T &= s|_{\text{dom}(t) \setminus \text{Im}(\rho_T)} && \text{because } s|_{\text{dom}(t)} = t \\ &= s'|_{\text{dom}(t) \setminus \text{Im}(\rho_T)} && \text{because } s' = s \end{aligned}$$

Hence,

$$x^t \xrightarrow[\mathcal{F}]{\rho_T | \rho} t \ l^t \setminus \rho_T.$$

(assign) Let $t \sqsubseteq s$ such that $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(t)$. By the induction hypotheses, since $\text{Im}(\varepsilon) = \emptyset$,

$$a^t \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} v^{s'|_{\text{dom}(t)}}$$

Note that $l \in \text{Im}(\rho_T \cdot \rho) \subset \text{dom}(t)$ implies $l \in \text{dom}(s'|_{\text{dom}(t)})$. Then

$$\begin{aligned} (s'|_{\text{dom}(t)} + \{l \mapsto v\}) \setminus \rho_T &= (s' + \{l \mapsto v\})|_{\text{dom}(t)} \setminus \rho_T && \text{because } l \in \text{dom}(s'|_{\text{dom}(t)}) \\ &= (s' + \{l \mapsto v\})|_{\text{dom}(t) \setminus \text{Im}(\rho_T)} \end{aligned}$$

Hence,

$$x := a^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} \mathbf{1}^{(s'|_{\text{dom}(t)} + \{l \mapsto v\}) \setminus \rho_T}.$$

(seq) Let $t \sqsubseteq s$ such that $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(t)$. By the induction hypotheses, since $\text{Im}(\varepsilon) = \emptyset$,

$$a^t \xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho} v^{s'|_{\text{dom}(t)}}$$

Moreover, $s'|_{\text{dom}(t)} \sqsubseteq s'$ and $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(s'|_{\text{dom}(t)}) = \text{dom}(t)$. By the induction hypotheses, this leads to:

$$b^{s'|_{\text{dom}(t)}} \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{t^{s''}|_{\text{dom}(s'|_{\text{dom}(t)}) \setminus \text{Im}(\rho_T)}}.$$

Hence, with $\text{dom}(s'|_{\text{dom}(t)}) = \text{dom}(t)$,

$$a ; b^t \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{t^{s''}|_{\text{dom}(t) \setminus \text{Im}(\rho_T)}}.$$

(if-true) and (if-false) are proved similarly to (seq).

(*letrec*) Let $t \sqsubseteq s$ such that $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(t)$.

$\text{Loc}(\mathcal{F}') = \text{Loc}(\mathcal{F}) \cup \text{Im}(\rho_T \cdot \rho)$ implies $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}') \subset \text{dom}(t)$.

Then, by the induction hypotheses,

$$b^t \xrightarrow[\mathcal{F}']{\rho_T | \rho} v^{s' |_{\text{dom}(t) \setminus \text{Im}(\rho_T)}}.$$

Hence,

$$\mathbf{letrec} \ f(x_1 \dots x_n) = a \ \mathbf{in} \ b^t \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s' |_{\text{dom}(t) \setminus \text{Im}(\rho_T)}}.$$

(*call*) Let $t \sqsubseteq s_1$ such that $\text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) \subset \text{dom}(t)$. Note the following equalities:

$$\begin{aligned} s_1 |_{\text{dom}(t)} &= t \\ s_2 |_{\text{dom}(t)} &\sqsubseteq s_2 \\ \text{Im}(\rho_T \cdot \rho) \cup \text{Loc}(\mathcal{F}) &\subset \text{dom}(s_2 |_{\text{dom}(t)}) = \text{dom}(t) \\ s_3 |_{\text{dom}(s_2 |_{\text{dom}(t)})} &= s_3 |_{\text{dom}(t)} \end{aligned}$$

By the induction hypotheses, they yield:

$$\begin{aligned} a'_1 &\xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho|} v_1^{s_2 |_{\text{dom}(t)}} \\ a_2^{s_2 |_{\text{dom}(t)}} &\xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho|} v_1^{s_3 |_{\text{dom}(t)}} \\ \forall i, a_i^{s_i |_{\text{dom}(t)}} &\xrightarrow[\mathcal{F}]{|\rho_T \cdot \rho|} v_i^{s_{i+1} |_{\text{dom}(t)}} \end{aligned}$$

Moreover, $s_{n+1} |_{\text{dom}(t)} \sqsubseteq s_{n+1}$ implies $s_{n+1} |_{\text{dom}(t)} + \{l_i \mapsto v_i\} \sqsubseteq s_{n+1} + \{l_i \mapsto v_i\}$ (Property 1) and:

$$\begin{aligned} \text{Im}(\rho'' \cdot \rho') \cup \text{Loc}(\mathcal{F}' + \{f \mapsto \mathcal{F} f\}) &= \text{Im}(\rho'') \cup (\text{Im}(\rho') \cup \text{Loc}(\mathcal{F}')) \\ &\subset \{l_i\} \cup \text{Loc}(\mathcal{F}) \\ &\subset \{l_i\} \cup \text{dom}(t) \\ &\subset \text{dom}(s_{n+1} |_{\text{dom}(t)} + \{l_i \mapsto v_i\}) \end{aligned}$$

Then, by the induction hypotheses,

$$b^{s_{n+1} |_{\text{dom}(t)} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{F}' + \{f \mapsto \mathcal{F} f\}]{\rho'' | \rho'} v^{s' |_{\text{dom}(s_{n+1} |_{\text{dom}(t)} + \{l_i \mapsto v_i\}) \setminus \text{Im}(\rho'')}}.$$

Finally,

$$\begin{aligned} s' |_{\text{dom}(s_{n+1} |_{\text{dom}(t)} + \{l_i \mapsto v_i\}) \setminus \text{Im}(\rho'')} \setminus \rho_T &= s' |_{\text{dom}(t) \cup \{l_i\} \setminus \{l_i\}} \setminus \rho_T = s' |_{\text{dom}(t)} \setminus \rho_T \\ &= (s' \setminus \rho_T) |_{\text{dom}(t) \setminus \text{Im}(\rho_T)} \quad (\text{by definition of } \cdot \setminus \cdot) \end{aligned}$$

Hence,

$$f(a_1 \dots a_n)^t \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{(s' \setminus \rho_T) |_{\text{dom}(t) \setminus \text{Im}(\rho_T)}}.$$

□

9 Correctness of lambda-lifting

In this section, we prove the correctness of lambda-lifting (Theorem 2, p. 27) by induction on the height of the optimised reduction. Using the height is mandatory because we rewrite some reduction trees during the proof, which prevents us from applying structural induction directly.

We need strong induction hypotheses to ensure that key invariants about stores and environments hold at every step. For that purpose, we define *aliasing free environments*, in which locations may not be referenced by more than one variable, and *local positions*, a generalisation of tail positions. They yield a strengthened version of liftable parameters (Definition 24). We then define lifted environments (Definition 25), to mirror the effect of lambda-lifting in lifted terms captured in closures, and finally reformulate the correctness of lambda-lifting in Theorem 4 with hypotheses strong enough to be provable directly by induction.

Definition 22 (Aliasing) A set of environments \mathcal{E} is *aliasing free* when:

$$\forall \rho, \rho' \in \mathcal{E}, \forall x \in \text{dom}(\rho), \forall y \in \text{dom}(\rho'), \rho x = \rho' y \Rightarrow x = y.$$

By extension, an environment of functions \mathcal{F} is aliasing free when $\text{Env}(\mathcal{F})$ is aliasing free.

The C standard defines aliasing under the name of “linkage”, a concept used to bind global variables or functions defined in separate files to a single memory location. It states explicitly that function parameters have no linkage (Section 6.2.2, alinea 6 in ISO/IEC 9899 [23]). Therefore, guaranteeing aliasing free environments is not an artifact but translates a fundamental property of the C semantics.

Definition 23 (Local position) *Local positions* are defined inductively as follows:

1. M is in local position in $M, x := M, M ; M, \text{if } M \text{ then } M \text{ else } M$ and $f(M, \dots, M)$.
2. N is in local position in **letrec** $f(x_1 \dots x_n) = M$ **in** N .

We extend the notion of liftable parameter (Definition 17, p. 26) to enforce invariants on stores and environments:

Definition 24 (Extended liftability) The parameter x is *liftable* in $(M, \mathcal{F}, \rho_T, \rho)$ when:

1. x is defined as the parameter of a function g , either in M or in \mathcal{F} ,
2. in both M and \mathcal{F} , inner functions in g , named h_i , are defined and called exclusively:
 - (a) in tail position in g , or
 - (b) in tail position in some h_j (with possibly $i = j$), or
 - (c) in tail position in M ,
3. for all f defined in local position in M , $x \in \text{dom}(\rho_T \cdot \rho) \Leftrightarrow \exists i, f = h_i$,
4. moreover, if h_i is called in tail position in M , then $x \in \text{dom}(\rho_T)$,
5. in \mathcal{F} , x appears necessarily and exclusively in the environments of the h_i 's closures,
6. \mathcal{F} contains only compact closures and $\text{Env}(\mathcal{F}) \cup \{\rho, \rho_T\}$ is aliasing free.

We need to extend the definition of lambda-lifting (Definition 15, p. 26) to environments, to reflect changes in lambda-lifted parameters captured in closures:

Definition 25 (Lifted form of an environment)

$$\begin{aligned} &\text{If } \mathcal{F} f = [\lambda x_1 \dots x_n. b, \rho', \mathcal{F}'] \quad \text{then} \\ (\mathcal{F})_* f = &\begin{cases} [\lambda x_1 \dots x_n x. (b)_*, \rho' |_{\text{dom}(\rho') \setminus \{x\}}, (\mathcal{F}')_*] & \text{when } f = h_i \text{ for some } i \\ [\lambda x_1 \dots x_n. (b)_*, \rho', (\mathcal{F}')_*] & \text{otherwise} \end{cases} \end{aligned}$$

Of course, a liftable parameter does not appear in the lifted form of an environment. This will be useful during the proof of correctness:

Lemma 11 *If x is a liftable parameter in $(M, \mathcal{F}, \rho_T, \rho)$, then x does not appear in $(\mathcal{F})_*$.*

Proof Since x is liftable in $(M, \mathcal{F}, \rho_T, \rho)$, it appears exclusively in the environments of h_i . By definition, it is removed when building $(\mathcal{F})_*$. \square

Those invariants and definitions lead to an enhanced correctness theorem:

Theorem 4 (Correctness of lambda-lifting) *If x is a liftable parameter in $(M, \mathcal{F}, \rho_T, \rho)$, then*

$$M^s \xrightarrow[\mathcal{F}]{\rho_T|\rho} v^{s'} \text{ implies } (M)_*^s \xrightarrow[(\mathcal{F})_*]{\rho_T|\rho} v^{s'}$$

Since naive and optimised reductions rules are equivalent (Theorem 3, p. 28), the proof of Theorem 2 (p. 27) is a direct corollary of this theorem:

Corollary 1 *If x is a liftable parameter in M , then*

$$M^\varepsilon \xrightarrow[\varepsilon]{\varepsilon} v^\varepsilon \text{ implies } \exists t, (M)_*^\varepsilon \xrightarrow[\varepsilon]{\varepsilon} v^t.$$

Overview of the proof The proof of the theorem (Section 9.3) is by induction on the height of the derivation. It is based on two key points:

- we must check that x is liftable in every subterm, to enforce the invariants needed during the induction. Since this is tedious and repetitive, we prove the details only for the statements that do not result directly from the definitions. It turns out that aliasing is often the unobvious point: to keep the proof as compact as possible, we first prove some lemmas about aliasing and liftability in Section 9.1.
- the essence of the proof lies in the (call) rule, more precisely when the called function f is one of the lifted functions h_i . In that case, one must rewrite some reductions to put them in the expected form. This requires another set of lemmas, that we introduce in Section 9.2, to be able to switch the location of the original x and its lifted counterpart.

Limitations Theorem 2 shows that lambda-lifting is correct in the case of CPC: since every inner function comes from the elimination of a goto, it is only ever called in tail position. As noted earlier, this proof deliberately overlooks interference issues related to C pointers, addressed in Section 10.

Another limitation is that Theorems 2 and 4 are implications, not equivalences: we do not prove that if a term does not reduce, it will not reduce once lifted. For instance, our proof does not ensure that lambda-lifting does not break infinite loops.

9.1 Aliasing lemmas

We need three lemmas to show that environments remain aliasing free during the induction. The first lemma states that concatenating two environments in an aliasing free set yields an aliasing free set. The other two prove that the aliasing invariant (Invariant 6, Definition 24) holds in the context of the (call) and (letrec) rules, respectively.

Lemma 12 *If $\mathcal{E} \cup \{\rho, \rho'\}$ is aliasing free then $\mathcal{E} \cup \{\rho \cdot \rho'\}$ is aliasing free.*

Proof By exhaustive check of cases. We want to prove

$$\forall \rho_1, \rho_2 \in \mathcal{E} \cup \{\rho \cdot \rho'\}, \forall x \in \text{dom}(\rho_1), \forall y \in \text{dom}(\rho_2), \rho_1 x = \rho_2 y \Rightarrow x = y.$$

given that

$$\forall \rho_1, \rho_2 \in \mathcal{E} \cup \{\rho, \rho'\}, \forall x \in \text{dom}(\rho_1), \forall y \in \text{dom}(\rho_2), \rho_1 x = \rho_2 y \Rightarrow x = y.$$

If $\rho_1 \in \mathcal{E}$ and $\rho_2 \in \mathcal{E}$, immediate. If $\rho_1 \in \{\rho \cdot \rho'\}$, $\rho_1 x = \rho x$ or $\rho' x$. This is the same for ρ_2 . Then $\rho_1 x = \rho_2 y$ is equivalent to $\rho x = \rho' y$ (or some other combination, depending on x, y, ρ_1 and ρ_2) which leads to the expected result. \square

Lemma 13 *Assume that, in a (call) rule,*

- $\mathcal{F} f = [\lambda x_1 \dots x_n. b, \rho', \mathcal{F}']$,
- $\text{Env}(\mathcal{F})$ is aliasing free, and
- $\rho'' = (x_1, l_1) \cdot \dots \cdot (x_n, l_n)$, with fresh and distinct locations l_i .

Then $\text{Env}(\mathcal{F}' + \{f \mapsto \mathcal{F} f\}) \cup \{\rho', \rho''\}$ is aliasing free too.

Proof Let $\mathcal{E} = \text{Env}(\mathcal{F}' + \{f \mapsto \mathcal{F} f\}) \cup \{\rho'\}$. $\mathcal{E} \subset \text{Env}(\mathcal{F})$ so \mathcal{E} is aliasing free. Adding fresh and distinct locations (ρ'') preserves aliasing freedom: as in the proof of Lemma 12, the only new cases are $\rho_1 = \rho_2 = \rho''$ and $\rho_1 = \rho'' \wedge \rho_2 \in \mathcal{E}$ (without loss of generality). In the former, $\rho'' x = \rho'' y \Rightarrow x = y$ holds since the locations of ρ'' are distinct. In the latter, $\rho_1 x = \rho_2 y \Rightarrow x = y$ holds since $\rho_1 x \neq \rho_2 y$ (by freshness hypothesis). \square

Lemma 14 *If $\text{Env}(\mathcal{F}) \cup \{\rho, \rho_T\}$ is aliasing free, then, for all x_i ,*

$$\text{Env}(\mathcal{F}) \cup \{\rho_T \cdot \rho \mid_{\text{dom}(\rho_T \cdot \rho) \setminus \{x_1 \dots x_n\}}, \rho_T, \rho\}$$

is aliasing free.

Proof Let $\mathcal{E} = \text{Env}(\mathcal{F}) \cup \{\rho, \rho_T\}$ and $\rho'' = \rho_T \cdot \rho \mid_{\text{dom}(\rho_T \cdot \rho) \setminus \{x_1 \dots x_n\}}$. Adding ρ'' , a restricted concatenation of ρ_T and ρ , to \mathcal{E} preserves aliasing freedom, as in the proof of Lemma 12. If $\rho_1 \in \mathcal{E}$ and $\rho_2 \in \mathcal{E}$, immediate. If $\rho_1 \in \{\rho''\}$, $\rho_1 x = \rho x$ or $\rho' x$. This is the same for ρ_2 . Then $\rho_1 x = \rho_2 y$ is equivalent to $\rho x = \rho' y$ (or some other combination, depending on x, y, ρ_1 and ρ_2) which leads to the expected result. \square

9.2 Rewriting lemmas

Calling a lifted function has an impact on the resulting store: new locations are introduced for the lifted parameters and the earlier locations, which are not modified anymore, are hidden. Because of those changes, the induction hypotheses do not apply directly in the case of the (call) rule. We use the following three lemmas to obtain, through several rewriting steps, a reduction of lifted terms meeting the induction hypotheses.

Lemma 7 handles alpha-conversion in stores and is used when choosing a fresh location for the (call) rule. Lemma 16 shows that moving a variable from the non-tail environment ρ to the tail environment ρ_T does not change the result, but restricts the domain of the store. Lemma 17 finally adds into the store a fresh location, bound to an arbitrary value.

Lemma 15 (Alpha-conversion) If $M^s \xrightarrow[\mathcal{F}]{\rho_T|\rho} v^{s'}$ then, for all l , for all l' appearing neither in s nor in \mathcal{F} nor in $\rho \cdot \rho_T$,

$$M^{s[l'/l]} \xrightarrow[\mathcal{F}[l'/l]]{\rho_T[l'/l]|\rho[l'/l]} v^{s'[l'/l]}$$

Moreover, both derivations have the same height.

Proof This is the counterpart of Lemma 7 (p. 33) for optimised reduction rules. The proof is exactly the same.

Lemma 16 If $M^s \xrightarrow[\mathcal{F}]{\rho_T|(x,l)\cdot\rho} v^{s'}$ and $x \notin \text{dom}(\rho_T)$ then $M^s \xrightarrow[\mathcal{F}]{\rho_T\cdot(x,l)|\rho} v^{s'|\text{dom}(s')\setminus\{l\}}$. Moreover, both derivations have the same height.

Proof By induction on the structure of the derivation. For the (val), (var), (assign) and (call) cases, we use the fact that that $s \setminus \rho_T \cdot (x, l) = s'|\text{dom}(s')\setminus\{l\}$ when $s' = s \setminus \rho_T$. \square

Lemma 17 If $M^s \xrightarrow[\mathcal{F}]{\rho_T|\rho} v^{s'}$ and k does not appear in either s , \mathcal{F} or $\rho_T \cdot \rho$, then, for all value u , $M^{s+\{k \mapsto u\}} \xrightarrow[\mathcal{F}]{\rho_T|\rho} v^{s'+\{k \mapsto u\}}$. Moreover, both derivations have the same height.

Proof The key idea is to add (k, u) to every store in the derivation tree. A collision might occur in the (call) rule, if there is some j such as $l_j = k$. In that case, l_j is renamed to $l'_j \neq k$ (same technique as the proof of Lemma 7).

By induction on the height of the derivation. We only show the (call) rule; other cases are straightforward by the induction hypotheses.

(call) By the induction hypotheses,

$$\forall i, a_i^{s_i+\{k \mapsto u\}} \xrightarrow[\mathcal{F}]{|\rho_T\cdot\rho} v_i^{s_{i+1}+\{k \mapsto u\}}$$

Because k does not appear in \mathcal{F} ,

$$k \notin \text{Loc}(\mathcal{F}' + \{f \mapsto \mathcal{F} f\}) \subset \text{Loc}(\mathcal{F})$$

For the same reason, it does not appear in ρ' . On the other hand, there might be a j such that $l_j = k$, so k might appear in ρ'' . In that case, we rename l_j in some fresh $l'_j \neq k$, appearing in neither s_{n+1} , nor \mathcal{F}' or $\rho'' \cdot \rho'$ (Lemma 15). After this alpha-conversion, k does not appear in either $\rho'' \cdot \rho'$, $\mathcal{F}' + \{f \mapsto \mathcal{F} f\}$, or $s_{n+1} + \{l_i \mapsto v_i\}$. By the induction hypotheses,

$$b^{s_{n+1}+\{l_i \mapsto v_i\}+\{k \mapsto u\}} \xrightarrow[\mathcal{F}'+\{f \mapsto \mathcal{F} f\}]{\rho''|\rho'} v^{s'+\{k \mapsto u\}}$$

Moreover, $s' + \{k \mapsto u\} \setminus \rho_T = s' \setminus \rho_T + \{k \mapsto u\}$ (since k does not appear in ρ_T). Hence

$$f(a_1 \dots a_n)^{s_1+\{k \mapsto u\}} \xrightarrow[\mathcal{F}]{\rho_T|\rho} v^{s'+\{k \mapsto u\} \setminus \rho_T}.$$

\square

9.3 Proof of correctness

We finally prove Theorem 4. Assume that x is a liftable parameter in $(M, \mathcal{F}, \rho_T, \rho)$. The proof is by induction on the height of the reduction of

$$M^s \xrightarrow[\mathcal{F}]{\rho_T | \rho} v^{s'}.$$

In the liftability subproofs, we detail only the non-trivial cases.

(call) — *first case* First, we consider the (most interesting) case where $\exists i, f = h_i$. x is a liftable parameter in $(h_i(a_1 \dots a_n), \mathcal{F}, \rho_T, \rho)$ hence in $(a_i, \mathcal{F}, \varepsilon, \rho_T \cdot \rho)$ too.

Proof (Liftability)

(3) by definition of a local position, every f defined in local position in a_i is in local position in $h_i(a_1 \dots a_n)$, hence the expected property by the induction hypotheses.

(4) immediate since the premise does not hold : since the a_i are not in tail position in $h_i(a_1 \dots a_n)$, they cannot feature calls to h_i (by (2)).

(6) Lemma 12, p. 39. \square

By the induction hypotheses, we get

$$(a_i)_*^{s_i} \xrightarrow[(\mathcal{F})_*]{|\rho_T \cdot \rho} v_i^{s_{i+1}}.$$

By definition of lifting, $(h_i(a_1 \dots a_n))_* = h_i((a_1)_*, \dots, (a_n)_*, x)$. But x is not a liftable parameter in $(b, \mathcal{F}', \rho'', \rho')$ since the Invariant (4) might be broken: $x \notin \text{dom}(\rho'')$ (x is not a parameter of h_i) but h_j might appear in tail position in b .

On the other hand, we have $x \in \text{dom}(\rho')$: since, by hypothesis, x is a liftable parameter in $(h_i(a_1 \dots a_n), \mathcal{F}, \rho_T, \rho)$, it appears necessarily in the environments of the closures of the h_i , such as ρ' . This allows us to split ρ' into two parts: $\rho' = (x, l) \cdot \rho'''$. Lemma 16 yields:

$$b^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[\mathcal{F}' + \{f \mapsto \mathcal{F} f\}]{\rho''(x, l) | \rho'''} v^{s' |_{\text{dom}(s') \setminus \{l\}}}$$

This rewriting ensures that x is a liftable parameter in $(b, \mathcal{F}' + \{f \mapsto \mathcal{F} f\}, \rho'' \cdot (x, l), \rho''')$.

Proof (Liftability)

(3) every function defined in local position in b is an inner function in h_i so, by Invariant (2), it is one of the h_i and $x \in \text{dom}(\rho'' \cdot (x, l) \cdot \rho''')$.

(4) Immediate since $x \in \text{dom}(\rho'' \cdot (x, l) \cdot \rho''')$.

(5) Immediate since \mathcal{F}' is included in \mathcal{F} .

(6) Immediate for the compact closures. Aliasing freedom is guaranteed by Lemma 13 (p. 40). \square

By the induction hypotheses,

$$(b)_*^{s_{n+1} + \{l_i \mapsto v_i\}} \xrightarrow[(\mathcal{F}' + \{f \mapsto \mathcal{F} f\})_*]{\rho''(x, l) | \rho'''} v^{s' |_{\text{dom}(s') \setminus \{l\}}}$$

The l location is not fresh: it must be rewritten into a fresh location, since x is now a parameter of h_i . Let l' be a location appearing in neither $(\mathcal{F}' + \{f \mapsto \mathcal{F} f\})_*$, nor $s_{n+1} + \{l_i \mapsto v_i\}$ or $\rho'' \cdot \rho_T'$. Then l' is a fresh location, which is to act as l in the reduction of b .

We will show that, after the reduction, l' is not in the store (just like l before the lambda-lifting). In the meantime, the value associated to l does not change (since l' is modified instead of l).

Lemma 11 implies that x does not appear in the environments of $(\mathcal{F})_*$, so it does not appear in the environments of $(\mathcal{F}' + \{f \mapsto \mathcal{F} f\})_* \subset (\mathcal{F})_*$ either. Aliasing freedom also implies that l does not appear in $(\mathcal{F}' + \{f \mapsto \mathcal{F} f\})_*$, so

$$(\mathcal{F}' + \{f \mapsto \mathcal{F} f\})_*[l'/l] = (\mathcal{F}' + \{f \mapsto \mathcal{F} f\})_*.$$

Moreover, l does not appear in $s' \upharpoonright_{\text{dom}(s') \setminus \{l\}}$. The conditions of Lemma 15 are met to rename l to l' :

$$(b)_*^{s_{n+1}[l'/l] + \{l_i \mapsto v_i\}} \xrightarrow[\text{(\mathcal{F}' + \{f \mapsto \mathcal{F} f\})_*}]{\rho''(x, l') | \rho'''} v^{s' \upharpoonright_{\text{dom}(s') \setminus \{l\}}}.$$

We want now to reintroduce l . Let $v_x = s_{n+1} l$. The location l does not appear in $s_{n+1}[l'/l] + \{l_i \mapsto v_i\}$, $(\mathcal{F}' + \{f \mapsto \mathcal{F} f\})_*$, or $\rho''(x, l') \cdot \rho'''$. Then, by Lemma 17,

$$(b)_*^{s_{n+1}[l'/l] + \{l_i \mapsto v_i\} + \{l \mapsto v_x\}} \xrightarrow[\text{(\mathcal{F}' + \{f \mapsto \mathcal{F} f\})_*}]{\rho''(x, l') | \rho'''} v^{s' \upharpoonright_{\text{dom}(s') \setminus \{l\}} + \{l \mapsto v_x\}}.$$

Since

$$\begin{aligned} s_{n+1}[l'/l] + \{l_i \mapsto v_i\} + \{l \mapsto v_x\} &= s_{n+1}[l'/l] + \{l \mapsto v_x\} + \{l_i \mapsto v_i\} && \text{because } \forall i, l \neq l_i \\ &= s_{n+1} + \{l' \mapsto v_x\} + \{l_i \mapsto v_i\} && \text{because } v_x = s_{n+1} l \\ &= s_{n+1} + \{l_i \mapsto v_i\} + \{l' \mapsto v_x\} && \text{because } \forall i, l' \neq l_i \end{aligned}$$

and $s' \upharpoonright_{\text{dom}(s') \setminus \{l\}} + \{l \mapsto v_x\} = s' + \{l \mapsto v_x\}$, we finalize the rewriting by Lemma 4,

$$(b)_*^{s_{n+1} + \{l_i \mapsto v_i\} + \{l' \mapsto v_x\}} \xrightarrow[\text{(\mathcal{F}' + \{f \mapsto \mathcal{F} f\})_*}]{\rho''(x, l') | (x, l) \cdot \rho'''} v^{s' + \{l \mapsto v_x\}}.$$

Hence the result:

$$\begin{aligned} & (\mathcal{F})_* h_i = [\lambda x_1 \dots x_n x. (b)_* \rho', (\mathcal{F}')_*] \\ \rho'' &= (x_1, l_1) \cdot \dots \cdot (x_n, l_n)(x, \rho_T x) \quad l' \text{ and } l_i \text{ fresh and distinct} \\ & \forall i, (a_i)_*^{s_i} \xrightarrow[\text{(\mathcal{F})_*}]{|\rho_T \cdot \rho|} v_i^{s_{i+1}} \\ (x)_*^{s_{n+1}} & \xrightarrow[\text{(\mathcal{F})_*}]{|\rho_T \cdot \rho|} v_X^{s_{n+1}} \quad (b)_*^{s_{n+1} + \{l_i \mapsto v_i\} + \{l' \mapsto v_x\}} \xrightarrow[\text{(\mathcal{F}' + \{f \mapsto \mathcal{F} f\})_*}]{\rho''(x, l') | \rho'} v^{s' + \{l \mapsto v_x\}} \\ \text{(CALL)} & \xrightarrow[\text{(\mathcal{F})_*}]{(h_i(a_1 \dots a_n))_*^{s_1} \xrightarrow{|\rho_T \cdot \rho|} v^{s' + \{l \mapsto v_x\}} \setminus \rho_T} \end{aligned}$$

Since $l \in \text{dom}(\rho_T)$ (because x is a liftable parameter in $(h_i(a_1 \dots a_n), \mathcal{F}, \rho_T, \rho)$), the extra-aneous location is reclaimed as expected: $s' + \{l \mapsto v_x\} \setminus \rho_T = s' \setminus \rho_T$.

(*letrec*) The parameter x is a liftable in (**letrec** $f(x_1 \dots x_n) = a$ **in** $b, \mathcal{F}, \rho_T, \rho$) so x is a liftable parameter in $(b, \mathcal{F}', \rho_T, \rho)$ too.

Proof (Liftability)

(3) and (4) Immediate by the induction hypotheses and definition of tail and local positions.

(5) By the induction hypotheses, Invariant (3) (x is to appear in the new closure if and only if $f = h_i$).

(6) Lemma 14 (p. 40). \square

By the induction hypotheses, we get

$$(b)_*^s \xrightarrow[\mathcal{F}'_*]{\rho_T | \rho} v^{s'}$$

If $f \neq h_i$,

$$(\mathbf{letrec} \ f(x_1 \dots x_n) = a \ \mathbf{in} \ b)_* = \mathbf{letrec} \ f(x_1 \dots x_n) = (a)_* \ \mathbf{in} \ (b)_*$$

hence, by definition of $(\mathcal{F}')_*$,

$$\begin{array}{c} (b)_*^s \xrightarrow[\mathcal{F}'_*]{\rho_T | \rho} v^{s'} \\ \text{(LETREC)} \frac{\rho' = \rho_T \cdot \rho |_{\text{dom}(\rho_T \cdot \rho) \setminus \{x_1 \dots x_n\}} \quad (\mathcal{F}')_* = (\mathcal{F})_* + \{f \mapsto [\lambda x_1 \dots x_n. (a)_*, \rho', F]\}}{(\mathbf{letrec} \ f(x_1 \dots x_n) = a \ \mathbf{in} \ b)_*^s \xrightarrow[\mathcal{F}'_*]{\rho_T | \rho} v^{s'}} \end{array}$$

On the other hand, if $f = h_i$,

$$(\mathbf{letrec} \ f(x_1 \dots x_n) = a \ \mathbf{in} \ b)_* = \mathbf{letrec} \ f(x_1 \dots x_n x) = (a)_* \ \mathbf{in} \ (b)_*$$

hence, by definition of $(\mathcal{F}')_*$,

$$\begin{array}{c} (b)_*^s \xrightarrow[\mathcal{F}'_*]{\rho_T | \rho} v^{s'} \\ \text{(LETREC)} \frac{\rho' = \rho_T \cdot \rho |_{\text{dom}(\rho_T \cdot \rho) \setminus \{x_1 \dots x_n x\}} \quad (\mathcal{F}')_* = (\mathcal{F})_* + \{h_i \mapsto [\lambda x_1 \dots x_n x. (a)_*, \rho', F]\}}{(\mathbf{letrec} \ h_i(x_1 \dots x_n) = a \ \mathbf{in} \ b)_*^s \xrightarrow[\mathcal{F}'_*]{\rho_T | \rho} v^{s'}} \end{array}$$

(*val*) $(v)_* = v$ so

$$\text{(VAL)} \frac{}{(v)_*^s \xrightarrow[\mathcal{F}'_*]{\rho_T | \rho} v^{s \setminus \rho_T}}$$

(*var*) $(y)_* = y$ so

$$\text{(VAR)} \frac{\rho_T \cdot \rho \ y = l \in \text{dom} \ s}{(y)_*^s \xrightarrow[\mathcal{F}'_*]{\rho_T | \rho} s \ l^{s \setminus \rho_T}}$$

(assign) The parameter x is liftable in $(y := a, \mathcal{F}, \rho_T, \rho)$ so in $(a, \mathcal{F}, \varepsilon, \rho_T \cdot \rho)$ too.

Proof (Liftability) (6) Lemma 12, p. 39. \square

By the induction hypotheses, we get

$$(a)_*^s \xrightarrow[(\mathcal{F})_*]{|\rho_T \cdot \rho|} v^{s'}$$

Moreover

$$(y := a)_* = y := (a)_*,$$

so :

$$\text{(ASSIGN)} \frac{(a)_*^s \xrightarrow[(\mathcal{F})_*]{|\rho_T \cdot \rho|} v^{s'} \quad \rho_T \cdot \rho \ y = l \in \text{dom } s'}{(y := a)_*^s \xrightarrow[(\mathcal{F})_*]{\rho_T | \rho} \mathbf{1}^{s' + \{l \rightarrow v\}} \setminus \rho_T}$$

(seq) The parameter x is liftable in $(a ; b, \mathcal{F}, \rho_T, \rho)$. If x is not defined in a or \mathcal{F} , then $(\)_*$ is the identity function and can trivially be applied to the reduction of a . Otherwise, x is a liftable parameter in $(a, \mathcal{F}, \varepsilon, \rho_T \cdot \rho)$.

Proof (Liftability) (6) Lemma 12, p. 39. \square

If x is not defined in b or \mathcal{F} , then $(\)_*$ is the identity function and can trivially be applied to the reduction of b . Otherwise, x is a liftable parameter in $(b, \mathcal{F}, \rho_T, \rho)$.

Proof (Liftability) Trivial. \square

By the induction hypotheses, we get $(a)_*^s \xrightarrow[(\mathcal{F})_*]{|\rho_T \cdot \rho|} v^{s'}$ and $(b)_*^{s'} \xrightarrow[(\mathcal{F})_*]{\rho_T | \rho} v^{s''}$.

Moreover,

$$(a ; b)_* = (a)_* ; (b)_*,$$

hence:

$$\text{(SEQ)} \frac{(a)_*^s \xrightarrow[(\mathcal{F})_*]{|\rho_T \cdot \rho|} v^{s'} \quad (b)_*^{s'} \xrightarrow[(\mathcal{F})_*]{\rho_T | \rho} v^{s''}}{(a ; b)_*^s \xrightarrow[(\mathcal{F})_*]{\rho_T | \rho} v^{s''}}$$

(if-true) and (if-false) are proved similarly to (seq). \square

10 Guaranteeing non-interference

The C language offers two main opportunities for interference that are problematic in CPC: “extruded” and “shared” variables.

Extruded variables are local variables (or function parameters) the address of which has been retained using the “address of” operator (&). There is one exception: `static` variables, which have a fixed address, are converted into global variables by the CPC translator and therefore no longer appear as extruded.

Shared variables are, more generally, variables that might be modified by more than one function. This includes global and `static` variables, as well as every memory area reachable through a pointer: extruded variables and those obtained with `malloc`.

We guarantee non-interference with two techniques, boxing and reevaluation. Both techniques are applied during the first pass of the CPC translator.

Boxing is a common, straightforward technique to preserve mutable variables during lambda-lifting, but it causes an expensive indirection to access boxed variables. We apply it to extruded variables only, in order to keep this cost to an acceptably low level (Section 10.1).

Reevaluation, on the other hand, is a technique specific to the way CPC builds continuations: the parameters of the latest functions in the tail of cps calls are evaluated when the continuation is built, before the earliest functions have been called. We show that, most of the time, changing the evaluation order in such a way is correct (Section 10.2) but, in some uncommon patterns, shared parameters must be reevaluated before their function is called (Section 10.3).

10.1 Boxing of extruded variables

Cost analysis The key point of the efficiency of CPC is that we need not box every mutated variable for lambda-lifting to be correct. As shown in Section 7, even though C is an imperative language, the fact that we only lift functions called in tail position allows us to copy parameters instead of maintaining a single instance of each variable through boxing.

There is one case, however, where boxing cannot be avoided: extruded variables. Performance-wise, we expect boxing only extruded variables to be far less expensive than boxing every lifted variable. In a typical C program, indeed, few local variables have their address retained compared to the total number of variables.

Experimental data confirm this intuition: in Hekate, the CPC translator boxes 13 variables out of 125 lifted parameters, with optimisations enabled (i.e. parameters that “obviously” do not need lifting or boxing are skipped, see Section 11 for further detail). Without optimisations, 29 variables are boxed out of 323 lifted parameters. In both cases, we box about 10% of the lifted variables, which seems an acceptable overhead.

Necessity and correctness of boxing Extruded variables have to be boxed because, after CPS conversion, local variables no longer exist as a single memory area with a fixed address. They are split over several functions, duplicated during lambda-lifting, and copied around with continuations. This is incompatible with the programmer’s expectation of a unique variable, reliably reachable through its address.

Preserving variables across cooperation points is a common issue in implementations of stackless threads. Some frameworks, like Tame [26], require the programmer to box local variables himself or at least to annotate them to let the compiler know they should be boxed. This increases the opportunities for bugs and is more painful when, for example, you need to

change a direct-style function into a cooperative function. The CPC translator, on the other hand, does this automatically: since extruded variables are detected by the compiler, the programmer need not understand the internals of CPC and decide by himself which variable should be boxed and which should not.

The boxing pass yields a program without the “address of” operator (&). Extruded variables are no more allocated on the stack, but on the heap with the `malloc` function (or a more efficient alternative). Calls to the `free` function are inserted before every `return` to release the allocated memory. The heap acts as an opaque storage abstraction, that is left unaffected by the CPC translator, and the addresses of boxed variables remain valid across the inner functions added in the goto elimination pass.

Interaction with other passes One may wonder whether it is correct to perform boxing before every other transformation. It turns out that boxing does not interfere with the other passes, because they do not introduce any additional “address of” operators. The program therefore remains free of extruded variables. Moreover, it is preferable to box early, before introducing inner functions, since it makes it easier to identify the entry and exit points of the original function, where variables are allocated and freed.

Extruded variables and tail recursive calls Although we keep the cost of boxing low, with about 10% of boxed variables, boxing has another, hidden cost: it breaks tail recursive cps calls. Since the boxed variables might, in principle, be used during the recursive calls, one cannot free them beforehand. Therefore, functions featuring extruded variables do not benefit from the automatic elimination of tail recursive calls induced by the CPS conversion. While this prevents CPC from optimising tail recursive calls “for free”, it is not a real limitation: the C standard does not guarantee the elimination of tail recursive calls anyway, for the stack frame should similarly be preserved in case of extruded variables, and C programmers are used not to rely on it.

10.2 Early evaluation of function parameters

Continuation building and evaluation order Another potential source of interference occurs when building continuation from a tail of at least two functions. Consider the following sequence:

```
f(); g(x); return;
```

Since continuations carry function parameters in an evaluated form, function `g` will have its parameter `x` evaluated, and stored in the continuation, before function `f` is called. This violates the evaluation order prescribed by the C language, and leads to wrong results when the functions in the tail interfere — more precisely, when an earlier function modifies some parameters of a later one.

In Section 5.1, we assumed non-interference in cps convertible terms, in order to retain a simple proof of correctness for the CPS conversion. This is exposed in the small-step semantics by Rule 6 (p. 14), which evaluates all function parameters at once: $\langle Q, C[], \sigma \rangle \rightarrow_{\tau} \langle Q[x_i \setminus \sigma x_i], C[] \rangle$. This rule reflects what happens in CPC programs: a continuation for the whole tail is built at once, with every parameter computed before any function call is performed.

Correctness of early evaluation It turns out that, most of the time, there is no need for specific shielding mechanisms because lifted terms do not exhibit interference issues and can therefore be evaluated in any order. In the rest of this section, we prove that such an “early evaluation” process is correct for lambda-lifted terms as long as no shared variable is involved. Section 10.3 will show how to avoid interference when shared variables are used as function parameters.

We first simplify our language to reflect the fact that lambda-lifting removes local functions and free variables. The **letrec** $f = \dots$ **in** \dots construct and the associated (letrec) rule disappear. Instead, we use a constant environment \mathcal{F} holding every function used in the reduced term M . To account for the absence of free variables, the closures in \mathcal{F} need not carry an environment. As a result, in the (call) rule, $\rho' = \varepsilon$ and $\mathcal{F}' = \mathcal{F}$.

The reason why interference does not occur is that a lifted term can never modify the variables that are not in its environment, since it cannot access them anymore through closures:

Lemma 18 *Let M be a lambda-lifted term. Then,*

$$M^s \xrightarrow[\mathcal{F}]{\rho} v^{s'}$$

implies

$$s|_{\text{dom}(s) \setminus \text{Im}(\rho)} = s'|_{\text{dom}(s) \setminus \text{Im}(\rho)}.$$

Proof By induction on the structure of the reduction. The key points are the use of $\rho' = \varepsilon$ in the (call) case, and the absence of (letrec) rules.

(val) and (var) Trivial ($s = s'$).

(assign) By the induction hypotheses,

$$s|_{\text{dom}(s) \setminus \text{Im}(\rho)} = s'|_{\text{dom}(s) \setminus \text{Im}(\rho)} \text{ and } l \in \text{Im}(\rho),$$

hence

$$s|_{\text{dom}(s) \setminus \text{Im}(\rho)} = (s' + \{l \mapsto v\})|_{\text{dom}(s) \setminus \text{Im}(\rho)}.$$

(seq) By the induction hypotheses,

$$s|_{\text{dom}(s) \setminus \text{Im}(\rho)} = s'|_{\text{dom}(s) \setminus \text{Im}(\rho)} \text{ and } s'|_{\text{dom}(s') \setminus \text{Im}(\rho)} = s''|_{\text{dom}(s') \setminus \text{Im}(\rho)}.$$

Since, $\text{dom}(s) \subset \text{dom}(s')$, the second equality can be restricted to

$$s'|_{\text{dom}(s) \setminus \text{Im}(\rho)} = s''|_{\text{dom}(s) \setminus \text{Im}(\rho)}.$$

Hence,

$$s|_{\text{dom}(s) \setminus \text{Im}(\rho)} = s''|_{\text{dom}(s) \setminus \text{Im}(\rho)}.$$

(if-true) and (if-false) are proved similarly to (seq).

(letrec) doesn't occur since M is lambda-lifted.

(call) By the induction hypotheses,

$$(s_{n+1} + \{l_i \mapsto v_i\})|_{\text{dom}(s_{n+1} + \{l_i \mapsto v_i\}) \setminus \text{Im}(\rho'' \cdot \rho')} = s'|_{\text{dom}(s_{n+1} + \{l_i \mapsto v_i\}) \setminus \text{Im}(\rho'' \cdot \rho')}$$

Since $\rho' = \varepsilon$, $\text{Im}(\rho'') = \{l_i\}$ and $\text{dom}(s_{n+1}) \cap \{l_i\} = \emptyset$ (by freshness),

$$(s_{n+1} + \{l_i \mapsto v_i\})|_{\text{dom}(s_{n+1})} = s'|_{\text{dom}(s_{n+1})}$$

so $s_{n+1} = s'|_{\text{dom}(s_{n+1})}$.

Since $\text{dom}(s) \setminus \text{Im}(\rho) \subset \text{dom}(s) \subset \text{dom}(s_{n+1})$,

$$s_{n+1}|_{\text{dom}(s) \setminus \text{Im}(\rho)} = s'|_{\text{dom}(s) \setminus \text{Im}(\rho)}.$$

Finally, we can prove similarly to the (seq) case that

$$s|_{\text{dom}(s) \setminus \text{Im}(\rho)} = s_{n+1}|_{\text{dom}(s) \setminus \text{Im}(\rho)}.$$

Hence,

$$s|_{\text{dom}(s) \setminus \text{Im}(\rho)} = s'|_{\text{dom}(s) \setminus \text{Im}(\rho)}.$$

□

As a consequence, a tail of function calls cannot modify the current store, only extend it with the parameters of the called functions:

Corollary 2 For every tail Q ,

$$Q^s \xrightarrow[\mathcal{F}]{\rho} v^{s'} \text{ implies } s \sqsubseteq s'.$$

Proof We prove the corollary by induction on the structure of a tail. First remember that *store extension* (written \sqsubseteq) is a partial order over stores (Property 1), defined in Section 8.4.2 as follows: $s \sqsubseteq s'$ iff $s'|_{\text{dom}(s)} = s$.

The case ε is trivial. The case $Q ; F$ is immediate by induction ((seq) rule), since \sqsubseteq is transitive. Similarly, it is pretty clear that $f(\text{expr}, \dots, \text{expr}, F)$ follows by induction and transitivity from $f(\text{expr}, \dots, \text{expr})$ ((call) rule). We focus of this last case.

Lemma 18 implies:

$$(s_{n+1} + \{l_i \mapsto v_i\})|_{\text{dom}(s_{n+1} + \{l_i \mapsto v_i\}) \setminus \text{Im}(\rho'' \cdot \rho')} = s'|_{\text{dom}(s_{n+1} + \{l_i \mapsto v_i\}) \setminus \text{Im}(\rho'' \cdot \rho')}.$$

Since $\rho' = \varepsilon$, $\text{Im}(\rho'') = \{l_i\}$ and $\text{dom}(s_{n+1}) \cap \{l_i\} = \emptyset$ (by freshness),

$$(s_{n+1} + \{l_i \mapsto v_i\})|_{\text{dom}(s_{n+1})} = s'|_{\text{dom}(s_{n+1})}$$

so $s_{n+1} = s'|_{\text{dom}(s_{n+1})}$.

The evaluation of *expr* parameters do not change the store: $s_{n+1} = s$. The expected result follows: $s = s'|_{\text{dom}(s)}$, hence $s \sqsubseteq s'$. □

This leads to the expected correctness result:

Theorem 5 (Non-interference in tail evaluation) For every tail Q , $Q^s \xrightarrow[\mathcal{F}]{\rho} v^{s'}$ implies

$$Q[x \setminus s(\rho x)]^s \xrightarrow[\mathcal{F}]{\rho} v^{s'} \text{ (provided } x \in \text{dom}(\rho) \text{ and } \rho x \in \text{dom}(s)).$$

Proof Immediate induction on the structure of tails and expressions: Corollary 2 implies that $s \sqsubseteq s''$ and $\rho x \in \text{dom}(s)$ ensures that $s(\rho x) = s''(\rho x)$ in the relevant cases (namely $Q ; F$ and $f(\text{expr}, \dots, \text{expr}, F)$). □

This result justifies the assumption of non-interference in the reduction rules of CPS-convertible terms (Rule 6, p. 14), in the absence of shared variables.

10.3 Reevaluation of shared variables

Shared variables break the assumption that a closed function can only access its own parameters. This does not affect lambda-lifting: shared variables are either global variables, left unchanged by lambda-lifting, or extruded variables, that are protected with boxing. It does, however, affect CPS conversion: changing the evaluation order of parameters and function calls is incorrect when the former might be modified by the latter.

Consider, for instance, the following tail where `f` might modify the value pointed to by `p`:

```
f(); g(*p); return;
```

CPS conversion would cause `*p` to be evaluated prematurely — before `f` is called — to store it in the continuation.

To work around this issue, the CPC translator forces the reevaluation before function calls featuring global variables and pointer dereference in their parameters. Our example becomes:

```
/* y is a fresh variable */
f(); y = *p; g(y); return;
```

This transformation guarantees that parameters of cps calls are never shared, because shared parameters are replaced by fresh, local variables.

Of course, this breaks the sequence of tail calls into two parts. Transformation into CPS-convertible form will then lead to something like:

```
h() { y = *p; g(y); return; }
f(); h(); return;
```

And lambda-lifting yields:

```
h(p) { y = *p; g(y); return; }
f(); h(p); return;
```

Interaction with other passes We perform reevaluation right after the boxing pass. This ensures that the program is free of “address of” operators and extruded variables, which are replaced, respectively, by pointers and the dereference operator (`*`). Shared variables are therefore restricted to global (and `static`) variables as well as pointer dereferences, which includes both extruded and manually allocated variables.

It is safe to force reevaluation before the goto elimination and lambda-lifting passes: although they add cps calls at the end of existing tails, their parameters are never shared. Indeed, these parameters are introduced by the lambda-lifting pass, which means that they are local, not global or `static`, variables. Moreover, lambda-lifting does not introduce any dereference operator, and the boxing pass guarantees that they are not extruded.

Cost analysis As shown in the above example, reevaluation causes an extra cps call, more if several functions are called with shared parameters. Since cps calls are expensive, one might expect a significant overhead, but it turns out that reevaluation is seldom needed.

It is rare, indeed, that the programmer performs two cps calls in a row. With the additional constraint that the second call must contain shared parameters, not a single reevaluation is needed to compile Hekate for instance. We can therefore safely consider that reevaluation is exceptional in practice, yielding negligible overhead.

11 Implementation

The current implementation of CPC is structured into three parts: the CPC to C translator, implemented in Objective Caml [29] on top of the CIL framework [34], the runtime, implemented in C, and the standard library, implemented in CPC.

The three parts are as independent as possible, and interact only through a small set of well-defined interfaces. This makes it easier to experiment with different approaches. For example, an earlier prototype of the translator was written in Common Lisp; we have experimented with a number of alternative implementations of the runtime, using external event loops and thread pools; and the standard library is being grown as we gain experience with Hekate.

11.1 The CPC translator

The CPC translator is structured as a series of passes, very close to the theoretical description given in the previous sections. It is built on top of CIL [34], an Objective Caml framework to parse, analyse and transform C programs.

Incremental transformations

The major difference between the transformations described earlier and the actual implementation is that most passes are implemented on an as-needed basis: a given subtree of the AST is only transformed if it contains CPC primitives that cannot be implemented in direct style. Our main concern here is to transform the code as little as possible, on the assumption that the gcc compiler is optimised for human-written code.

The first pass is kept straightforward: the CPC translator ensures non-interference, as described in Section 10, boxing extruded variables and forcing reevaluation of shared variables before cps calls. It iterates then over the AST to check whether the cps functions are in CPS-convertible form. To that end, the translator looks for sequences of cps calls and analyses the statement following it:

- in the case of a return, the subterm is already CPS-convertible and the translator goes on;
- in the case of a goto, it is converted to a tail call, with the corresponding label turned into an inner function, and the translator starts another pass;
- for any other statement, a goto is added to make the flow of control explicit, converting enclosing loops too if necessary. The translator then starts another pass and will eventually convert the introduced goto into a tail call.

Once every cps function is in CPS-convertible form, lambda-lifting is performed similarly: rather than lifting every parameter, the translator looks for free variables to be lifted until it reaches a fixed point.

Implementation of continuations

The translator is ultimately responsible for implementing cps function calls. To ensure proper interfacing with hand-written primitive cps functions, CPC uses a well-defined calling convention for cps functions.

Function parameters are pushed on continuations in left-to-right order. They are not aligned on word boundaries, which leads to smaller continuations and easier store and load operations. Although word-aligned reads and writes are more efficient in general, our tests showed little or no impact in the CPC programs we experimented with, on x86 and x86_64 architectures: the worst case has been a 10 % slowdown in a microbenchmark with deliberately misaligned parameters. This tradeoff might need to be reconsidered when we port CPC to an architecture with no hardware support for unaligned accesses, e.g. MIPS or ARM.

Cps functions are always called in tail position, and the correctness of goto-elimination requires that function calls be properly tail-recursive. This has two consequences on the implementation of function calls. First, continuations don't store return addresses, and the frame at the top of the continuation contains a pointer to the next function to execute (similar to the sequences of subroutines' addresses used in "threaded code" [6]). Second, every function is responsible for popping the parameters it was called with before invoking the next function: the caller has no opportunity to perform the popping, since it will never be returned to.

In the case in which it passes a value to its continuation, the callee stores it in a location reserved for that purpose in the next frame by the function that built the continuation (Section 5). The address of this "hole" is computed by subtracting the size of the returned value from the top of the next continuation frame; this works because the return value is passed as the right-most parameter, which is on the top the continuation.

11.2 The CPC runtime

The CPC runtime is in charge of three tasks: managing continuations, providing primitive cps functions and scheduling attached and detached continuations.

Continuations management

The runtime provides the functions to allocate, expand and free continuations. The abstraction exposed to the translator is the allocation and deallocation of activation records on top of the current continuation. The current implementation grows continuations by a multiplying factor and never shrinks them; while this might in principle waste memory in the case of many long-lived continuations with an occasional deep call stack, we believe this case is rare enough not to bother about it.

The runtime also provides facilities to invoke continuations. In a language with proper tail calls, each function would simply invoke the next one directly; in C, which is not properly tail-recursive, doing that leads to unbounded growth of the native call stack. We work around this issue by using a "trampolining" technique [18]: we iteratively extract the top function from the current continuation, pass it the rest of the continuation, and receive the next continuation as a result of its invocation.

Primitive cps functions

The runtime implements the set of primitive cps functions detailed in Section 3.1. This set is easily extensible since the translator treats those primitives like any cps function: the burden of compatibility is left to the runtime, which must manually build and decode continuations using the same conventions as the translator.

Unlike ordinary cps functions, the primitives do not typically return the current continuation when invoked. Instead, they directly manipulate the internal data structures of the scheduler, and return `NULL` to yield back to the main event loop.

The primitives take care to behave properly when the current thread is detached. For example, in attached mode, `cpc_sleep` will set the continuation state to *sleeping*, insert it in the priority queue of sleeping continuations and invoke the next continuation available. In detached mode, it will simply call the blocking native `sleep` function,

12 Experimental results

The CPC language provides no more than half a dozen, very low-level primitives. The standard library and CPC programs are implemented in terms of this small set of operations, and are therefore directly dependent on their performance.

Benchmarking individual CPC primitives however is somewhat misleading. The CPS transform performed by the translator induces a cost that is spread across a CPC program, and that is mostly due to splitting every cps function written by the programmer into a number of smaller functions. Since calls to cps functions are significantly more expensive than direct-style function calls, this cost can be significant.

In the following sections, we present the results of our benchmarks comparing CPC with other thread libraries. In Section 12.1, we show the results of benchmarking the CPC primitives. In Section 12.2, we present the results of a more realistic benchmark, that takes into account the overhead of the CPS conversion.

All the benchmarks described in this section were performed on a machine with an Intel Core 2 Duo processor at 3.6 GHz, downclocked at 2.0 GHz, with 4 GB of memory and running Linux 2.6.35 with swap disabled.

We compared CPC with the following thread libraries:

- nptl, the native thread library in GNU libc 2.11.2 [14];
- GNU Pth version 2.0.7 [17];
- State Threads (ST) version 1.9 [43].

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

12.1 Speed of CPC primitives

We wrote a number of benchmarks that were each aimed at measuring the performance of a single CPC primitive. The most important is the ability of CPC to use massive numbers of threads: on a machine with 4 GB of physical memory and no swap space, CPC can handle up to 50.1 million continuations, implying an average memory usage of roughly 82 bytes per continuation. This figure compares very favourably to both kernel and user-space thread libraries (see Fig. 2), which our tests have shown to be limited on the same system to anywhere from 32 000 to 934 600 threads in their default configuration, and to 961 400 threads at most after some tuning.

Time measurements, as shown in Fig. 3, yielded somewhat more mixed results. Two imbricated loops with a single function call is roughly 10 times slower when the call is CPS-converted than when it is in direct-style. This rather disappointing result can be explained by two factors: first, due to the translation techniques used, after goto elimination

and lambda-lifting, the loop consists of four CPS function calls. Compiling that CPS-convertible code, simple enough to allow tail call optimisation, directly with `gcc` yields a factor of 4.7 compared to CPS-converted calls.

That remaining factor can be attributed to the cost of allocating continuations on the heap [33], and the fact that CPS-converted functions are mostly opaque to the native compiler. The inefficiency on modern hardware of making an indirect function call, which prevents accurate branch prediction, might also play a role although we observed a limited impact even on the most contrived microbenchmarks (less than 10% of overhead).

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 ten times faster than in the fastest thread library available to us. For reasons that we do not understand, waking up on a condition variable turns out to be surprisingly fast in ST.

nptl	32 330
Pth	700 000 (est.)
ST	934 600
ST (4 kB stacks)	961 400
CPC	50 190 000

All thread libraries were used in their default configuration, except where noted. Pth never completed, the value 700 000 is an educated guess.

Fig. 2 Number of threads possible in various thread libraries

	call	cps-call	switch	cond	spawn
nptl (1 core)	$3.0 \cdot 10^{-3}$		1.5	28	15
nptl (2 cores)	$3.0 \cdot 10^{-3}$		0.45	52	13
Pth	$3.0 \cdot 10^{-3}$		6.8	84	11
ST	$3.0 \cdot 10^{-3}$		0.56	0.8	0.67
CPC	$3.0 \cdot 10^{-3}$	$32 \cdot 10^{-3}$	0.06	1.0	0.07

All times are in microseconds (smaller is better). The columns are as follows:

call: direct-style function call;

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

switch: context switch;

cond: context switch on a condition variable;

spawn: thread creation. For nptl on one core, the `sched_compat_yield` kernel variable needs to be set, to fix the utterly broken default behavior of `sched_yield` under Linux.

Fig. 3 Speed of various thread libraries

12.2 Macro-benchmarks

As we have seen above, most CPC operations are faster than their equivalents in common thread libraries; however, a CPC program incurs the overhead of the CPS translation; in the absence of accurate data on the number of cps calls introduced, it is difficult to predict the behaviour of CPC programs.

In order to determine the speed of a realistic set of programs, we have written a set of very simple web servers (less than 200 lines each) that share the exact same structure: a single thread or process waits for incoming connections, and spawns a new thread or process as soon as a client is accepted (there is no support for persistent connections). The servers were benchmarked by repeatedly downloading a tiny file with different numbers of simultaneous clients and measuring the average response time. Because of the simple structure of the servers, and the naive nature of the benchmark, this test yields a fully understood, repeatable benchmark that measures the underlying implementation of concurrency rather than the implementation of the web server.

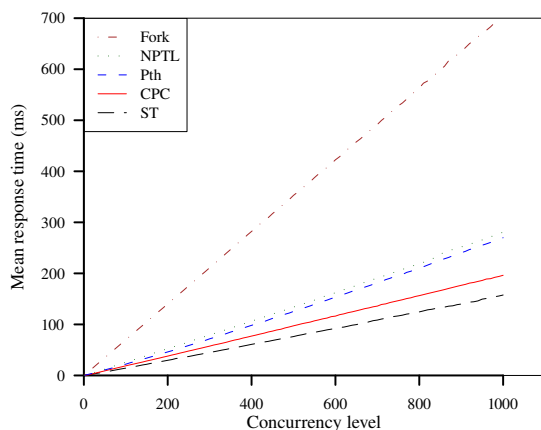


Fig. 4 Web servers comparison. Featuring servers based, in increasing order of efficiency, on the `fork` system call, the *Linux Native Posix Threads Library*, the *Gnu Pth* library, CPC and the *State Threads* library.

Figure 4 presents the results of our experiment. It plots the average serving time per request against the number of concurrent requests; a smaller slope indicates a faster server. It shows that CPC code is as efficient as the fastest thread libraries we tested. A more in-depth analysis of this benchmark is available in a technical report [25].

13 Hekate, a BitTorrent seeder written in CPC

CPC has a very different feel to the other programming languages familiar to us. Since CPC threads are so cheap to create, a single action is easily split into two or three threads; hence, the programming idioms that are useful in CPC are likely to be new.

Hekate is a BitTorrent seeder (an upload-only implementation of the peer-to-peer BitTorrent file-transfer protocol) written entirely in CPC, and meant to help us develop a programming style and a standard library for CPC. Hekate was designed and implemented by two B.Sc. students, Pejman Attar and Yoann Canal, over the course of a few months [4].

In the rest of this section, we give a rough description of Hekate and describe a few of its more interesting features.

General structure Hekate consists in 3,000 lines of CPC code (not counting the DHT code, which is encapsulated in a separate library), roughly one third of which is plain, sequential C code and the rest is threaded code.

Hekate is structured to use at least two threads for each peer: one for reading requests, the other one to push the chunks of files back to the client. Given the cheapness of CPC threads, this model scales to hundreds of clients without any problem.

Scheduling clients When too many clients are connected, some of them are temporarily suspended (the BitTorrent protocol calls this state *choked*). In Hekate, the associated threads are suspended on a condition variable, and woken up in a round-robin manner. To our surprise, it turns out that no explicit queues are necessary: the implicit queues contained in CPC condition variables are sufficient.

Timeouts Like most networking protocols, BitTorrent requires application-layer timeouts: nothing prevents a malicious, buggy or simply slow peer from maintaining an idle connection indefinitely, and unless measures are taken to drop such idle connections, they end up clogging all of Hekate's connection slots.

Implementing timeouts in Hekate is done in less than 30 lines of code, using the functions demonstrated in the second code snippet of Section 3.3. Before every read from a network client, we create a fresh one-minute timeout, which we pass to `cpc_full_read_c` and destroy when the read has completed.

It is interesting to compare this scheme with a typical event-driven program: this kind of constant timeout would be allocated only once per client and rearmed on every read. The timeout callback would then need to synchronise somehow with the read callback to interrupt it. While this is probably more efficient resource wise, the CPC approach composes better and is easier to reason about. Allocating a new timeout (and hence spawning a CPC thread) is cheap enough to make the overhead negligible.

Interacting with external libraries The BitTorrent protocol provides two ways to advertise available content to other peers: over HTTP to dedicated servers (*trackers*), and over a Kademlia distributed hash table (DHT). Hekate advertises itself using both techniques, and in both cases by using external C (not CPC) libraries.

HTTP support is provided by *libcurl* [31]. Interactions with trackers are scheduled by a set of CPC threads, one per tracker, that spend most of their time in the attached state, waiting for a timeout to expire. Since *libcurl* provides a blocking interface, the actual HTTP transactions are encapsulated within calls to `cpc_detached`.

DHT support is provided by the second author's *libdht*², which, unlike *libcurl*, provides a non-blocking interface; because of that, there is no need to detach, and calls to *libdht* are simply made from normal CPC code. A complication arises, however, from the fact that *libdht* expects the caller to provide it with a set of callbacks which, of course, need to be written in C (not CPC). Hence, the callbacks are stub functions that spawn a CPC thread that performs the real work (recall that the `cpc_spawn` statement is legal in non-cps context).

Performance Working on Hekate showed us that CPC allows a pleasant style of concurrent programming which scales reasonably well. Using Hekate to seed the updates to the *World of Warcraft game* (distributed over BitTorrent), we served hundred of clients for days, reaching peaks of 5 MB of data per second with an average of 2.5 MB/s; Hekate's CPU load never rose above 10%.

² Also used in the *Transmission* implementation of BitTorrent.

14 Conclusions and further work

In this paper, we have described CPC, a programming language that provides threads which are implemented, in different parts of the program, either as extremely lightweight heap-allocated data structures, or as native operating system threads. The compilation technique used by CPC is somewhat unusual, since it involves a continuation-passing style (CPS) transform for the C programming language; we have shown the correctness of that CPS transform, as well as the correctness of CPC's compilation scheme.

We believe that CPC is highly adapted to writing high-performance network servers. To convince ourselves of this fact, we have written Hekate, a large scale BitTorrent seeder in CPC. Hekate has turned out to be a maintainable, fast and reliable piece of software.

We enjoyed writing Hekate very much. Due to the lightweight threads that it provides, and due to the determinacy of scheduling of attached threads, CPC threads have a very different feel than threads in other programming languages; discovering the right idioms and the right abstractions for CPC has been (and remains) one of the more enjoyable parts of our work.

For CPC to become a useful production language it must come equipped with a consistent and powerful standard library that encapsulates useful programming idioms in a generally useable form. The current CPC library was written in an on-demand basis, mainly to meet the needs of Hekate; the choice of functions that it provides is therefore somewhat random. Filling in the holes of the library should be a fairly straightforward job. For CPC to scale easily on multiple cores, this extended standard library might also offer the ability to run several event loops, scheduled on different cores, and migrate threads between them.

We have no doubt that CPC can be useful for applications other than high-performance network servers. One could for example envision a GUI system where every button is implemented using three CPC threads: one that waits for mouse clicks, one that draws the button, and one that coordinates with the rest of the system. To be useful in practice, such a system should be implemented using a standard widget library; the fact that CPC integrates well with external event loops imply that this should be possible.

Finally, the ideas used in CPC might be applicable to other programming languages than C. For example, a continuation passing transform might be a way of having threads in Javascript without deploying a new Javascript runtime to hundreds of millions of web browsers.

Software availability

The full CPC compiler, including sources and benchmarking code, is available online at <http://www.pps.jussieu.fr/~kerneis/software/cpc>. The sources of Hekate are available online at <http://www.pps.jussieu.fr/~kerneis/software/hekate>.

References

1. Adya, A., Howell, J., Theimer, M., Bolosky, W.J., Douceur, J.R.: Cooperative task management without manual stack management. In: Proceedings of the 2002 USENIX Annual Technical Conference, pp. 289–302. USENIX Association, Berkeley, CA, USA (2002)
2. Anderson, T.E., Bershad, B.N., Lazowska, E.D., Levy, H.M.: Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.* **10**(1), 53–79 (1992)
3. Appel, A.W.: *Compiling with continuations*. Cambridge University Press (1992)

4. Attar, P., Canal, Y.: Réalisation d'un seeder bittorrent en CPC (2009). URL <http://www.pps.jussieu.fr/~jch/software/hekate/hekate-attar-canal.pdf>
5. von Behren, R., Condit, J., Zhou, F., Necula, G.C., Brewer, E.: Capriccio: scalable threads for internet services. *SIGOPS Oper. Syst. Rev.* **37**(5), 268–281 (2003)
6. Bell, J.R.: Threaded code. *Commun. ACM* **16**(6), 370–372 (1973)
7. Berdine, J., O'Hearn, P., Reddy, U., Thielecke, H.: Linear Continuation-Passing. *Higher-Order and Symbolic Computation* **15**, 181–208 (2002)
8. Boussinot, F.: FairThreads: mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice and Experience* **18**(5), 445–469 (2006)
9. Bruggeman, C., Waddell, O., Dybvig, R.K.: Representing control in the presence of one-shot continuations. In: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation, PLDI '96, pp. 99–107. ACM, New York, NY, USA (1996)
10. Claessen, K.: A poor man's concurrency monad. *J. Funct. Program.* **9**(3), 313–323 (1999)
11. Clinger, W.D.: Proper tail recursion and space efficiency. In: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, PLDI '98, pp. 174–185. ACM, New York, NY, USA (1998)
12. Cunningham, R., Kohler, E.: Making events less slippery with eel. In: Proceedings of the 10th conference on Hot Topics in Operating Systems. USENIX Association, Berkeley, CA, USA (2005)
13. Danvy, O., Schultz, U.: Lambda-lifting in quadratic time. In: Functional and Logic Programming, *Lecture Notes in Computer Science*, vol. 2441, pp. 134–151. Springer-Verlag, Berlin, Germany (2002)
14. Drepper, U., Molnar, I.: The Native POSIX Thread Library for Linux (2005). URL <http://people.redhat.com/drepper/nptl-design.pdf>
15. Duff, T.: Duff's device (1983). URL <http://www.lysator.liu.se/c/duffs-device.html>
16. Dybvig, R.K., Hieb, R.: Engines from continuations. *Comput. Lang.* **14**, 109–123 (1989)
17. Engelschall, R.S.: Portable multithreading: the signal stack trick for user-space thread creation. In: Proceedings of the 2000 USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, USA (2000)
18. Ganz, S.E., Friedman, D.P., Wand, M.: Trampolined style. In: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming, ICFP '99, pp. 18–27. ACM, New York, NY, USA (1999)
19. Gosling, J., Rosenthal, D.S.H., Arden, M.J.: The NeWS book: an introduction to the network/extensible window system (SUN Technical Reference Library). Springer-Verlag, New York, NY, USA (1989)
20. Gu, B., Kim, Y., Heo, J., Cho, Y.: Shared-stack cooperative threads. In: Proceedings of the 2007 ACM symposium on Applied computing, SAC '07, pp. 1181–1186. ACM, New York, NY, USA (2007)
21. Haynes, C.T., Friedman, D.P., Wand, M.: Continuations and coroutines. In: Proceedings of the 1984 ACM Symposium on LISP and functional programming, LFP '84, pp. 293–298. ACM, New York, NY, USA (1984)
22. Hoare, C.A.R.: Monitors: an operating system structuring concept. *Commun. ACM* **17**(10), 549–557 (1974)
23. International Organization for Standardization: ISO/IEC 9899:1999 “Programming Languages – C” (1999)
24. Johnsson, T.: Lambda lifting: Transforming programs to recursive equations. In: Functional Programming Languages and Computer Architecture, *Lecture Notes in Computer Science*, vol. 201, pp. 190–203. Springer-Verlag, Berlin, Germany (1985)
25. Kerneis, G., Chroboczek, J.: Are events fast? Tech. rep., PPS, Université Paris 7 (2009). URL <http://hal.archives-ouvertes.fr/hal-00434374/en/>
26. Krohn, M., Kohler, E., Kaashoek, M.F.: Events can make sense. In: Proceedings of the 2007 USENIX Annual Technical Conference, pp. 1–14. USENIX Association, Berkeley, CA, USA (2007)
27. Lehmann, M.: The libev manual (2010). URL <http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod>
28. Leroy, X.: The LinuxThreads library. URL <http://pauillac.inria.fr/~xleroy/linuxthreads>
29. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The Objective-Caml system (2010). URL <http://caml.inria.fr/>
30. Li, P., Zdancewic, S.: Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, PLDI '07, pp. 189–199. ACM, New York, NY, USA (2007)
31. Libcurl, the multiprotocol file transfer library (2010). URL <http://curl.haxx.se/libcurl/>
32. Microsoft: Windows fibers. URL <http://msdn.microsoft.com/ms682661>
33. Miller, J.S., Rozas, G.J.: Garbage collection is fast, but a stack is faster. AI Memo 1462, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA (1994)

34. Necula, G., McPeak, S., Rahul, S., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Compiler Construction, *Lecture Notes in Computer Science*, vol. 2304, pp. 209–265. Springer-Verlag, Berlin, Germany (2002)
35. Pai, V.S., Druschel, P., Zwaenepoel, W.: Flash: an efficient and portable web server. In: Proceedings of the 1999 USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, USA (1999)
36. Pike, R.: The implementation of Newsqueak. *Software: Practice and Experience* **20**(7), 649–659 (1990)
37. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science* **1**(2), 125–159 (1975)
38. Provenzano, C.: MIT pthreads (1998). URL <http://www.humanfactor.com/pthreads/mit-pthreads.html>
39. Provos, N., Mathewson, N.: Libevent (2010). URL <http://www.monkey.org/~provos/libevent/>
40. Reppy, J.: Concurrent ML: Design, application and semantics. In: Functional Programming, Concurrency, Simulation and Automated Reasoning, *Lecture Notes in Computer Science*, vol. 693, pp. 165–198. Springer-Verlag, Berlin, Germany (1993)
41. Reynolds, J.C.: The discoveries of continuations. *LISP and Symbolic Computation* **6**(3), 233–247 (1993)
42. Scholz, E.: A concurrency monad based on constructor primitives, or, being first-class is not enough. Tech. Rep. B 95-01, Fachbereich Mathematik und Informatik, Freie Universität Berlin, Berlin, Germany (1995)
43. Shekhtman, G., Abbott, M.: State Threads for internet applications (2009). URL <http://state-threads.sourceforge.net/docs/st.html>
44. Steele Jr., G.L.: Rabbit: A compiler for Scheme. Master’s thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA (1978). Technical report AI-TR-474
45. Steele Jr., G.L., Sussman, G.J.: Lambda, the ultimate imperative. AI Memo 353, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA (1976)
46. Strachey, C., Wadsworth, C.P.: Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England (1974). Reprinted in *Higher-Order and Symbolic Computation* **13**(1/2):135–152, 2000, with a foreword [51]
47. Sun Microsystems: Multithreading in the Solaris operating environment. URL <http://www.sun.com/software/whitepapers/solaris9/multithread.pdf>
48. Thielecke, H.: Continuations, functions and jumps. *SIGACT News* **30**(2), 33–42 (1999)
49. Tismer, C.: Continuations and stackless Python. In: Proceedings of the 8th International Python Conference (2000)
50. Vouillon, J.: Lwt: a cooperative thread library. In: Proceedings of the 2008 ACM SIGPLAN workshop on ML, ML ’08, pp. 3–12. ACM, New York, NY, USA (2008)
51. Wadsworth, C.P.: Continuations revisited. *Higher-Order and Symbolic Computation* **13**(1/2), 131–133 (2000)
52. Welsh, M., Culler, D., Brewer, E.: SEDA: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.* **35**(5), 230–243 (2001)
53. van Wijngaarden, A.: Recursive definition of syntax and semantics. In: Formal Language Description Languages for Computer Programming, pp. 13–24. North-Holland Publishing Company, Amsterdam, Netherlands (1966)
54. Williams, N.J.: An implementation of scheduler activations on the NetBSD operating system. In: Proceedings of the 2002 USENIX Annual Technical Conference, FREENIX Track, pp. 99–108. USENIX Association, Berkeley, CA, USA (2002)