Continuation-Passing C: Program Transformations for Compiling Concurrency in an Imperative Language

Gabriel Kerneis

Université Paris Diderot – Paris 7

École Doctorale Sciences Mathématiques de Paris Centre

Continuation-Passing C :

Transformations de programmes

pour

compiler la concurrence

dans

un langage impératif

Gabriel Kerneis

Thèse de doctorat
Spécialité informatique

Soutenue le vendredi 9 novembre 2012
devant le jury composé de :

M. Juliusz Chroboczek	*Directeur*
M. Olivier Danvy	*Rapporteur*
M. Thomas Ehrhard	*Président*
M. Jean Goubault-Larrecq	
M. Alan Mycroft	*Rapporteur*

Université Paris Diderot – Paris 7

Continuation-Passing C:

Program Transformations

for

Compiling Concurrency

in

an Imperative Language

Gabriel Kerneis

Abstract

Most computer programs are concurrent programs, which need to perform several tasks at the same time. Among the many different techniques to implement concurrent programs, the most common are threads and events. Event-driven programs are more lightweight and often faster than their threaded counterparts, but also more difficult to understand and error-prone. Additionally, event-driven programming alone is often not powerful enough; it is then necessary to write hybrid code, that uses both preemptively-scheduled threads and cooperatively-scheduled event handlers, which is even more difficult.

This dissertation shows that concurrent programs written in a unified, threaded style can be translated automatically into efficient, equivalent event-driven programs through a series of proven source-to-source transformations.

Our first contribution is a complete implementation of *Continuation-Passing C* (CPC), an extension of the C programming language for writing concurrent systems. The CPC programmer manipulates very lightweight threads, choosing whether they should be cooperatively or preemptively scheduled at any given point. The CPC program is then processed by the CPC translator, which produces highly efficient sequentialized event-loop code, and uses native threads to execute the preemptive parts. This approach retains the best of both worlds: the relative convenience of programming with threads, and the low memory usage of event-loop code.

We then prove the correctness of the transformations performed by the CPC translator. We demonstrate in particular that lambda lifting is correct for functions called in tail position in an imperative call-by-value language without extruded variables. We also show that CPS conversion is correct for a subset of C programs, and that every CPC program can be translated into such a CPS-convertible form.

Finally, we validate the design and implementation of CPC by exhibiting our *Hekate* BitTorrent seeder, and showing through a number of benchmarks that CPC is as fast as the fastest thread librairies available to us. We also justify the choice of lambda lifting by implementing eCPC, a variant of CPC using environments, and comparing its performance to CPC.

Résumé

Concurrence

La plupart des programmes informatiques sont des programmes *concurrents*, qui doivent effectuer plusieurs tâches simultanément. Par exemple, un serveur réseau doit répondre à de multiples clients en même temps ; un jeu vidéo doit gérer les entrées du clavier et les clics de souris tout en simulant l'univers du jeu et en affichant les éléments de décor ; et un jeu en réseau doit effectuer toutes ces tâches en même temps.

Il existe de nombreuses techniques différentes pour implémenter des programmes concurrents. Une abstraction couramment utilisée est le concept de *thread* (ou processus léger) : chaque *thread* encapsule un calcul précis pour l'effectuer isolément. Ainsi, dans un programme à *threads*, les tâches concurrentes sont exécutées par autant de *threads* indépendants, qui communiquent par l'intermédiaire d'une mémoire partagée. L'état de chaque *thread* est stocké dans une structure de pile, qui n'est en revanche pas partagée.

Une alternative aux *threads* est la programmation en style à événements. Un programme à événements interagit avec son environnement en réagissant à un ensemble de stimuli, appelés *événements* ; par exemple, dans un jeu vidéo, les touches frappées au clavier ou les clics de souris. À tout moment est associé à chaque événement un morceau de code appelé le *gestionnaire* de cet événement ; un ordonnanceur global, appelé la *boucle à événements*, attend répétitivement qu'un événement se produise et invoque le gestionnaire correspondant. Un calcul donné n'est pas nécessairement encapsulé dans un unique gestionnaire d'événements : exécuter une tâche complexe, qui requerrait par exemple à la fois des entrées venant du clavier et de la souris, exige de coordonner plusieurs gestionnaires en échangeant les événements appropriés.

Contrairement aux *threads*, les gestionnaires d'événements ne disposent pas d'une pile propre ; les programmes à événements sont ainsi plus légers, et souvent plus rapides, que leurs homologues à base de *threads*. Néanmoins, parce qu'elle nécessite de diviser le flot de contrôle en de multiples petits gestionnaires d'événements, la programmation à événements est difficile et sujette à erreur. De plus, elle n'est souvent pas suffisante en tant que telle, en particulier pour accéder à des interfaces bloquantes ou pour exploiter des processeurs à cœurs multiples.

Il est alors nécessaires d'écrire du code *hybride*, qui utilise à la fois des *threads* ordonnancés *préemptivement* et des gestionnaires d'événements ordonnancés *coopérativement*, ce qui est encore plus difficile.

Puisque la programmation par événements est plus difficile mais plus efficace que la programmation avec des *threads*, il est naturel de vouloir l'automatiser au moins partiellement. D'une part, de nombreuses architectures et techniques de transformations ad-hoc ont été proposées pour mêler *threads* et événements, essentiellement pour des langages impératifs tels que C, Java et Javascript. D'autre part, un certain nombre d'abstractions et de techniques, développées pour implémenter des langages fonctionnels, ont été étudiées en détail et appliquées à la construction de programmes fonctionnels concurrents : par exemple les monades, le style par passage à la continuation (CPS, pour *continuation-passing style*) et la conversion CPS, ou encore la programmation fonctionnelle réactive ont été utilisés dans des langages comme Haskell, OCaml et Concurrent ML. Cette thèse cherche à réconcilier ces deux courants de recherche, en adaptant des techniques de transformation classiques de la programmation fonctionnelle afin de construire un traducteur de *threads* en événements, correct et efficace, pour un langage impératif.

Dans le chapitre 1, nous détaillons comment écrire des programmes concurrents à l'aide de *threads*, d'événements et de continuations. Nous présentons notamment plusieurs styles de programmes à événements, ainsi que les transformations que nous utiliserons par la suite : la *conversion en style par passage à la continuation* (ou conversion CPS) et le *lambda lifting*.

Continuation-Passing C

Continuation-Passing C (CPC) est une extension du langage C pour l'écriture de systèmes concurrents. Le programmeur CPC manipule des *threads* très légers et peut choisir s'ils doivent être ordonnancés préemptivement ou coopérativement en tout point du programme. Le programme CPC est ensuite traité par le *compilateur CPC*, qui produit un code à événement séquentialisé très efficace et utilise les *threads* natifs pour exécuter les parties préemptives. Cette approche offre le meilleur des deux mondes : le confort relatif de la programmation à base de *threads* et la faible occupation mémoire du code à événements.

Dans le chapitre 2, nous commençons par donner un aperçu du langage CPC à travers l'exemple de *Hekate*, un serveur réseau qui est le programme CPC le plus étendu que nous ayons réalisé. Nous montrons en particulier l'utilité d'un concept de *thread* unifié, distinct des *threads* natifs proposés par le système d'exploitation : la possibilité d'alterner sans effort entre ordonnancement coopératif et préemptif offre les avantages du code à événements sans la complexité de gérer manuellement des *pools* de *threads* pour effectuer les opérations bloquantes. Nous donnons une description détaillée du langage CPC. Le cœur du langage est aussi simple que possible, les structures de données plus complexes et les mécanismes de synchronisation étant construit sur la base d'une demi-douzaine de primitives élémentaires. Nous concluons le chapitre en donnant quelques exemples de telles structures de données, incluses dans la bibliothèques standard CPC.

Technique de compilation prouvée

Le compilateur CPC traduit des programmes CPC en des programmes C équivalents grâce à une série de transformations source-source. Chacune de ces étapes emploie des techniques couramment utilisées pour la compilation de langages fonctionnels : conversion CPS, lambda lifting, environnements et traduction de sauts en appels terminaux. Néanmoins, nous les utilisons dans le contexte du C, un langage notoirement hostile à la formalisation. Le simple fait qu'il s'agisse d'un langage impératif rend la moitié de ces techniques indéfinies dans le cas général. De plus, parce que le C permet d'obtenir l'adresse de variables allouées sur la pile, un soin tout particulier doit être accordé pour garantir la correction d'une traduction des *threads*, qui ont leur propre pile, vers les événements, qui n'en ont pas.

Dans le chapitre 3, nous présentons les transformations effectuées par le compilateur CPC. Nous justifions en particulier pourquoi l'étape d'encapsulation (*boxing*) est nécessaire pour garantir la correction de la traduction quand l'adresse de variables de pile est capturée. Puisque le lambda lifting et la conversion CPS ne sont pas corrects en général dans un langage impératif, nous fournissons des preuves de correction pour ces étapes (Chapitres 4 et 5). Dans le chapitre 4, nous prouvons la correction du lambda lifting pour les fonctions appelées en position terminale dans un langage impératif en appel par valeur, en l'absence de variables *extrudées*. Dans le chapitre 5, nous prouvons que la conversion CPS est correcte dans un langage impératif sans variables extrudées, statiques ou globales, bien qu'elle implique l'évaluation anticipée (*early evaluation*) de certains paramètres de fonctions.

Implémentation et évaluation

Une partie essentielle de notre travail est l'implémentation du compilateur CPC. Disposer d'un compilateur fonctionnel est extrêmement utile pour expérimenter, développer des intuitions, vérifier certaines suppositions et effectuer des tests de performance.

Il est tentant, lorsque l'on travaille sur un langage de bas niveau tel que le C, de se concentrer sur l'optimisation d'un petit nombre de détails d'implémentation dans l'espoir d'améliorer les performances. Néanmoins, il est également une maxime bien connue des programmeurs qui met en garde contre ce penchant[1] tant il est vrai qu'optimiser sans avoir au préalable mesuré mène fréquemment à un code plus obscur sans pour autant apporter le moindre gain de performances. Notre implémentation du compilateur CPC a été guidée par la conviction que le lambda lifting et la conversion CPS sont assez efficaces pour ne pas nécessiter de trop nombreuses micro-optimisations. Notre code reste simple et aussi proche des transformations théoriques que possible, mais parvient malgré tout à des performances comparables aux bibliothèques de *threads* les plus efficaces que nous connaissions.

Dans le chapitre 6, nous comparons l'efficacité de programmes CPC à celle d'autres implémentations de la concurrence, et montrons que CPC est aussi rapide que les plus rapides d'entre elles tout en permettant offrant un gain en occupation mémoire d'au moins un ordre

[1] « *Premature optimisation is the root of all evil.* » [Knu74]

de grandeur. Nous effectuons plusieurs séries de mesures. Tout d'abord, nous mesurons individuellement l'efficacité des primitives de concurrence. Nous comparons ensuite le débit de serveurs web, un exemple de concurrence typique, pour évaluer l'impact des transformations effectuées par CPC sur les performances d'un programme complet. Enfin, nous mesurons les performances de Hekate, à la fois sur de matériel embarqué aux ressources limitées et sur un ordinateur courant équipé de plusieurs cœurs soumis à une charge réaliste.

Quoique l'efficacité soit un paramètre essentiel, il est également important d'évaluer l'utilisabilité du langage. Notre but étant de développer un langage de programmation agréable à partir de techniques d'implémentation efficaces et prouvées, nous avons cherché à obtenir un retour de la part d'utilisateurs pour évaluer l'expressivité et le confort apportés par CPC pour l'écriture de programmes concurrents. Grâce au support complet du langage C par le compilateur CPC, il est possible d'écrire de larges programmes utilisant des bibliothèques C existantes. L'écriture d'Hekate, avec deux étudiants de master qui n'avaient jamais utilisé CPC auparavant, a été l'occasion de découvrir des idiomes de programmation associés à la légèreté et au déterminisme des *threads* CPC. Nous utilisons Hekate tout au long de cette thèse pour fournir des exemples de code CPC, mais aussi comme d'une référence pour évaluer l'impact des transformations effectuées par le compilateur CPC sur la taille et la structure du code généré.

Comprendre le code à événements

L'étude de la transformation automatique des *threads* en événements est une occasion de comprendre plus en détail la structure et le fonctionnement des programmes à événements. Comment certains programmeurs parviennent à écrire des programmes si grands et complexes sans devenir fous restera vraisemblablement un mystère pour toujours, mais cette thèse tente d'éclairer un peu la question.

Le code généré par CPC diffère de la plupart des programmes à événements écrits à la main sur deux points : il contient plus de gestionnaires d'événements, et l'utilisation de lambda lifting implique la copie des variables locales d'un gestionnaire au suivant au lieu d'être allouées une fois pour toutes sur le tas. Il est alors intéressant de modifier le compilateur CPC pour générer du code qui se rapproche plus du code écrit par un être humain et d'identifier si les modifications nécessaires correspondent à des transformations de programme connues. Cela permet également de comparer ces différents styles et donne une idée des opérations que le programmeur effectue dans sa tête quand il écrit du code à événements.

Le chapitre 7 est le résultat d'une collaboration avec Matthieu Boutier [Bou11]. Nous y étudions comment les divers styles à événements présentés dans le chapitre 1 peuvent être générés à partir d'une description commune en style à *threads*, à l'aide des transformations classiques que sont la *défonctionalisation* et les *environnements*. Nous implémentons en particulier l'une de ces variantes, sous la forme d'*eCPC*, qui utilise des environnements au lieu du lambda lifting pour stocker les variables locales. Nous mesurons les performances de *CPC* et *eCPC* et montrons que le lambda lifting est plus efficace mais moins facile à déboguer que les environnements dans la plupart des cas.

Contributions

Nous montrons dans cette thèse que des programmes concurrents écrits dans un style à *threads* sont traduisibles automatiquement, par une suite de transformations source-source prouvées, en programmes à événements équivalents et efficaces.

Nos contributions principales sont :

- une implémentation complète du langage CPC (Chapitre 2) ;
- une technique de compilation basée sur des transformations de programmes prouvées (Chapitre 3), en particulier :
  - une preuve de correction du lambda lifting pour les fonctions appelées en position terminale, dans un langage impératif en appel par valeur sans variables extrudées (Chapitre 4),
  - une preuve de correction de la conversion CPS pour les programmes en forme CPS-convertible dans un langage impératif sans variables extrudées, statiques et globales (Chapitre 5) ;
- des résultats expérimentaux évaluant l'utilisabilité et l'efficacité de CPC, notamment :
  - Hekate, un serveur réseau BitTorrent écrit en CPC (Chapitre 2),
  - des mesures expérimentales montrant que CPC est aussi rapide que les bibliothèques de *threads* les plus rapides dont nous disposons, tout en permettant la création d'au moins un ordre de grandeur de *threads* supplémentaires (Chapitre 6) ;
- une implémentation alternative, eCPC, utilisant des environnements au lieu du lambda lifting et permettant d'évaluer le gain apporté par ce dernier (Chapitre 7).

Remerciements

> You are in a maze of twisty little passages, all alike.
>
> *Colossal Cave Adventure*
> Will Crowther

> People are capable of learning like rats in mazes. But the process is slow and primitive. We can learn more, and more quickly, by taking conscious control of the learning process.
>
> *Mindstorms*
> Seymour Papert

Je voudrais remercier tous ceux qui m'ont aidé à sortir du labyrinthe sans me transformer en rat.

Pour sa confiance tout au long du chemin, son soutien sans faille, ses conseils précieux, ses anecdotes innombrables sur les sujets les plus variés et son amitié, merci à Juliusz. Pour l'incroyable minutie de ses corrections jusque dans les moindres détails et son rapport qui résume si bien ma thèse, et pour avoir compris CPC en moins de temps qu'il ne m'en a fallu pour l'expliquer autour d'une tasse de café, merci à Alan Mycroft. Pour ses explications patientes, tant sur les subtilités des continuations que sur la technique de rédaction d'un article scientifique, et pour ses suggestions sur ce manuscrit, merci à Olivier Danvy. Pour son écoute et ses efforts de tous les instants pour faire de PPS un laboratoire où il fait bon travailler, et pour assumer la lourde tâche de présider mon jury, merci à Thomas Ehrhard. Merci enfin à Jean Goubault-Larrecq de témoigner son intérêt pour CPC en acceptant de siéger pour la deuxième fois en deux mois à son sujet. Votre présence à tous m'honore.

Merci à ceux qui ont contribué à cette thèse, parfois sans le savoir, untel par un conseil, un autre par une idée. Parmi ces pollinisateurs, je retiens notamment : Vincent Padovani, pour la mise en page du lemme 5.4.2 et ses longues explorations du *ticket entailment* ; Frédéric Boussinot, pour son invitation à Sophia-Antipolis ; Allan McInnes pour avoir cité mon article

sur *Lambda the Ultimate*[2] et Tom Duff pour avoir été le premier à le commenter ; Andy Key, pour ses passionnantes explications sur *Weave* ; Marco Trudel, pour sa bibliographie sur l'élimination des gotos ; Michael Scott pour son combat acharné contre les *threads* coopératifs ; Boris Yakobowski pour ses rapports de bugs ; et tous les collègues de PPS qui ont supporté mes divagations à un moment ou un autre durant ces quatre années.

Merci aux professeurs qui m'ont formé et guidé jusqu'aux portes du doctorat. Pour l'informatique, je retiens Bruno Petazzoni, Olivier Hudry, Irène Charon, Samuel Tardieu, Alexis Polti et Paul Gastin. Et s'il ne fallait en ajouter qu'une seule, pour les langues bien vivantes, ce serait évidemment Marie-Christine Mopinot — elle appréciera à sa juste valeur, j'en suis sûr, l'effort que constitua la rédaction de ce manuscrit.

Un paragraphe à lui tout seul, il le mérite vraiment, pour Michel Parigot. Il nous rappelle par son action exemplaire qu'*on ne se bat pas dans l'espoir du succès ! Non ! non, c'est bien plus beau lorsque c'est inutile*. Plus qu'utile au contraire, essentiels, irremplaçables combats de celui qui, ne sachant pas que c'était impossible, l'a fait.

Une thèse serait bien insipide sans co-thésards. Les miens furent du meilleur cru : Mehdi pour son affection indéfectible, Matthieu pour assurer la relève sans l'ombre d'une hésitation, Stéphane « le coursier diligent » pour son soutien logistique, Thibaut pour la guerre des grenouilles, Antoine qui a rédigé si vite, Jonas qui finira bientôt, Mathias et Marie-Aude qui ont fini trop tôt, Grégoire pour son soutien dès les premiers instants, Christine car que ferions-nous sans elle ?, Fabien pour les coqs, Florian, Julien et Claire même s'ils sont du LIAFA, Pierre, Flavien, Guillaume, Shahin et Jakub. Sans oublier Pejman, l'auteur incontestable du meilleur programme CPC au monde (Hekate), et de l'un des pires programmes Java (BabelDraw). Un dernier merci pour finir cette longue liste (et j'en oublie forcément) à Odile : si les thésards sont les forces vives du laboratoire, elle en est incontestablement un pilier, indispensable.

Des amis, je ne dirai rien, que les noms. Ils savent chacun à quel point ils comptent pour moi, et combien j'ai pu compter sur eux : *cela est de toute évidence et de toute éternité*. Merci donc à Chloé, Fabien, Sophie, Julien, Christine et Joris. Merci aussi à Grégoire, Julie, Raphaëlle, Antoine et Gambetta.

Dans le guignon toujours présente, je pense enfin à ma famille : à mes parents, à la force de leur confiance, à Nicole et Henriette à qui je dédie la soutenance, à Pierre-Emmanuel et Gildas, et à Georges ; à ma belle famille, aussi, d'un bout à l'autre des Boulay-Claverie-Rakotonirina ; et aux autres plus lointains, mais fidèles, oncle, tante et cousins. Merci à Ermione et Fidji. Merci à Camille, sans qui je n'aurais pas fini. Merci d'avoir supporté mes retards (ce soir encore !), mes absences et mes longs week-ends. Merci d'avoir dit « oui ».

Cambridge, le 25 octobre 2012, à 20 heures.

[2] `http://lambda-the-ultimate.org/node/4157`.

Contents

Introduction

> "And what is the use of a book," thought Alice
> "without pictures or conversation?"
>
> *Alice's Adventures in Wonderland*
> Lewis Carroll

We show in this dissertation that imperative concurrent programs written in threaded style can be translated automatically into efficient, equivalent event-driven programs through a series of proven source-to-source transformations.

In this introduction, we first present the notion of concurrency, and two of its implementations: *threads* and *events*. We also introduce *Continuation-Passing C* (CPC), an extension of the C language for writing concurrent systems, which provides very lightweight threads and compiles them into event-driven code. We then explain how to use another abstraction, *continuations*, and the associated transformation, *conversion into continuation-passing style*, to implement such a translation from threads into events. Finally, we show why it is difficult to perform this translation in an imperative language, and give a brief overview of how the CPC translator manages to keep it correct and efficient nonetheless. We conclude with an outline our main contributions.

The wild land of concurrency

Task management

This section owes much to the enlightening article *Cooperative Task Management Without Manual Stack Management* [Ady+02].

Writing large programs often involves dividing them in a number of conceptually distinct *tasks*: each task encapsulates control flow and often a local state, and all tasks access some shared state and communication channels to coordinate with other tasks. In a *sequential* program, there is a single task or, more generally, tasks are fully ordered and each of them waits for the previous one to complete before executing. Sequential task management makes it easy to coordinate tasks: there is no risk of conflicting accesses to the shared state, and

the programmer can rely on the fact that no two tasks are ever executing simultaneously to reason about the behaviour of the program. However, sequential task management is often too limited.

Most computer programs are *concurrent*: they perform several tasks at the same time. They might repeat a single task a large number of times. For instance, a web server needs to serve hundreds of clients simultaneously. Or they might use a lot of different tasks, cooperating to build a larger system. For example, a video game needs to handle the graphics and the simulation of the game while reacting to keystrokes and mouse moves from the user. Some complex programs might even need to do both. For instance, a network game server needs to simulate a virtual word while sending and receiving updates from hundreds of players over the Internet.

In a concurrent program, the order in which tasks are executed is determined by a scheduler. One distinguishes two kinds of scheduling: *preemptive* task management and *cooperative* task management.

Preemptive task management Preemptively scheduled tasks can be suspended and resumed by the scheduler at any time. They can be interleaved on a single processor or executed simultaneously on several processors or processor cores. This makes them well suited for highly-parallel workloads and high-performance programs.

One must be very careful when accessing shared state with preemptive tasks: since any other task might be modifying the same piece of data at the same time, concurrent accesses to shared resources need to be protected with synchronisation primitives such as locks, semaphores, or monitors. Should these primitives be forgotten, or misused, *race conditions* arise: conflicts when accessing shared data that are often hard to debug because they might depend on a specific, non-deterministic scheduling that is difficult to reproduce. Preemptive scheduling makes programs harder to reason about, because global guarantees about shared state must be manually enforced by the programmer with appropriate locking and synchronisation.

Cooperative task management A cooperatively scheduled task only yields to another one at some explicit points, called *cooperation points*. Between two cooperation points, the programmer enjoys the ease of reasoning associated with sequential task management: a single task accessing shared state at a given time, with no need to add locks or care about race conditions. It is then only necessary to ensure that invariants about shared state hold when cooperating, rather than in any possible interleaving as is the case with preemptive scheduling. Cooperative schedulers can also be deterministic schedulers, providing guarantees on the order of execution which further helps in synchronising tasks, reproducing bugs and controlling fairness between tasks.

However, because of the exclusive nature of cooperative tasks, they cannot use the power of multiple processors or processor cores. Moreover, since tasks cannot be preempted by the scheduler, a single task performing a long computation or stuck in a blocking operation would prevent every other tasks from executing: it is impossible to ensure fairness if a task

does not cooperate. This means that the programmer must make his program yield regularly. Even in network programs, where each I/O is an opportunity to cooperate, this requirement of cooperation sometimes happens to be too limiting.

Hybrid programming There are at least three cases where cooperative task management is not enough. When a program needs to perform blocking system calls, use libraries with blocking interfaces, or perform long-running computations, preemptive scheduling cannot be avoided. Many cooperative programs are therefore in fact *hybrid* programs: they use cooperative scheduling most of time, and fall-back to using preemptive tasks for potentially blocking sections of code. This combines the advantages of both approaches, but is often tedious because concurrent systems rarely provide a straightforward way to switch from preemptive to cooperative task management: the programmer usually has to craft his own solution using two different frameworks, one for each kind of scheduling.

Stack management

As we have seen, concurrent programs can often be divided in distinct tasks encapsulating a control flow and a local state. However, these conceptual tasks do not always map directly to the abstractions provided to the programmer by concurrency frameworks. In some models, each task has its own call stack, to store its control flow and local variables; we then speak of *automatic stack management*. In other models the call stack is shared among all tasks; we speak of *manual stack management*. In the former case, stacks are independent and they are automatically saved and restored upon context switches by the compiler, the library or the operating system. In the latter, the programmer is responsible for multiplexing several conceptual tasks on a single stack, handling the control flow and the local state of each task by himself. Automatic stack management is of course easier for the programmer, but going manual sometimes cannot be avoided, for efficiency reasons or because the target system does not provide concurrency abstractions with automatic stack management.

Threads and *event-driven programming* are the two most common examples of automatic and manual stack management, respectively.

Threads Threads are a widely used abstraction for concurrency. Each thread executes a function, with its own call stack, and all threads share heap-allocated memory. Thanks to automatic stack management, threads are convenient for the programmer: each concurrent task that needs to be implemented is written as a distinct function, and executed in a separate thread. Interaction between threads uses the shared heap, and synchronisation primitives such as locks and condition variables.

Threads are provided either by the operating system, by a user-space library, or directly by the programming language. Generally, threads provided by the operating system are preemptive, and user-space threads cooperative. In both cases, the scheduler has very few hints about the actual stack usage and behaviour of the program. Therefore, it needs to reserve a large, fixed chunk of memory for the call stack upon creation of each thread, and saves and

restores it fully at each context switch. This conservative approach potentially wastes a lot of memory, for instance for idle threads with a shallow stack. Moreover, thread creation and context switching are usually two to three orders of magnitude slower than a function call. In implementations exhibiting these limitations, creating large numbers of short-lived threads is not always possible, and in practice the programmer sometimes needs to multiplex several concurrent tasks on a single thread.

However, threads are not necessarily slower than event-driven code, and careful implementations can yield large performance gains. One solution is to use user-space, cooperative libraries which tend to be faster and more lightweight than native, preemptive threads [Beh+03]. A user-space library does not incur the cost of going through supervisor mode, allowing faster context switches. It can also reduce memory usage by using smaller, or even dynamically-resized, call stacks [Gu+07].

Another approach is to make concurrency constructs part of the programming language, using information from the compiler for optimisations. For example, the compiler can perform static analysis to reduce the amount of memory used, saving only relevant pieces of information on each context switch. This is most effective in the context of cooperative threads because the compiler can also determine the point of cooperation in advance and optimise atomic blocks of code accordingly. It is also possible to add heuristics to choose between different implementations at compile time.

Events Event-driven programs are built around an *event loop* which repeatedly gathers external stimuli, or *events*, and invokes a small atomic function, an *event handler*, in reaction to each of them. Task management is manual in event-driven style: there is no abstraction to represent a concurrent task with its own control flow and local state, and no automatic transfer of control flow from one handler to the next. If performing a task requires interacting with several events, for instance in a server exchanging a number of network messages with a client, the programmer is responsible for writing each event handler and registering them with the event loop in the correct order as the execution of the task flows. Similarly, synchronisation is achieved manually by registering handlers and firing the appropriate events. Large persistent pieces of data are generally shared between event handlers, while short-lived values used in a handful of handlers are kept local and copied from one handler to the next; again, this replicates manually how local variables and heap-allocated data are used in threads.

Events allow concurrency to be implemented in languages that do not provide threads, and cannot be avoided on systems exposing only an asynchronous, callback-based API; even when threads are available, they provide a lightweight alternative, well-suited to highly-concurrent and resource-constrained programs. But events are also an extreme example of the programmer implementing scheduling and context switching entirely by hand, sequentializing and optimising manually concurrency in his program, then relying on the compiler to compile the resulting sequential code efficiently. This is a very tedious task, and it yields programs that are hard to debug because they lack a call stack: information about control flow and local state must be extracted from custom, heap-allocated data structures. As we shall see, this is also not the most efficient approach (Chapter 7).

Event-driven programs are inherently cooperative, since event handlers are guaranteed to run atomically and control is passed around explicitly by the programmer. As explained above, this enables deterministic scheduling and makes reasoning about a particular piece of code easier. The price to pay for this simpler scheduling is that, just like cooperative threads, event-driven programs do not benefit from parallel architectures with multiple cores or processors, and get frozen if a handler performs a blocking operation. In practice, most event-driven programs are therefore hybrid programs, delegating blocking and long-running computations to native threads, or distributing events across several event loops running in independent threads. This hybrid style makes them even harder to debug.

CPC threads

Since event-driven programming is more difficult but more efficient than threaded programming, it is natural to want to at least partially automate it. On the one hand, many architectures for mixing threads and events, and ad-hoc translation schemes have been proposed, mostly for wide-spread imperative programming languages such as C, Java or Javascript. On the other hand, a number of abstractions and techniques, developed to implement functional languages, have been studied extensively and applied to build concurrent functional programs: for instance monads, continuation-passing style (CPS) and CPS conversion, or functional reactive programming, in languages such as Haskell, OCaml or Concurrent ML.

This dissertation seeks to bridge the gap between these two streams of research, adapting well-known transformation techniques from functional programming to build an efficient and correct translator from threads to events in an imperative language.

We propose *Continuation-Passing C* (CPC), an extension of the C language for concurrency designed and implemented with Juliusz Chroboczek. The CPC language offers a unified abstraction, called *CPC threads*, that are neither native threads nor user-space, library-based threads. Most of the time, CPC threads are scheduled cooperatively but the programmer has the ability to switch a thread between cooperative and preemptive mode at any time. To the programmer, CPC threads look like extremely lightweight, user-space threads.

When a CPC thread is created, it is *attached* to the main CPC scheduler, which is cooperative and deterministic. CPC provides a number of primitives that interact with the scheduler, to yield to another thread, sleep, wait for I/O or synchronise on condition variables; the programmer then builds more cooperative functions on top of these primitives. There is a special primitive, `cpc_link`, that allows the programmer to *detach* the current CPC thread from the cooperative scheduler, and execute it in a native, preemptive thread instead. Conversely, `cpc_link` also allows the user to *attach* a detached thread back to the cooperative scheduler.

These unified threads provide the advantages of hybrid programming, without the hassle of combining manually distinct concurrency models. This is not a silver bullet: the programmer must still take care not to block in attached mode, and to use locks properly in detached mode. However, CPC threads eliminate all the boilerplate needed to switch back and forth between

both modes, making it straightforward to call blocking functions asynchronously, without writing callbacks manually.

To compile CPC threads efficiently, we translate CPC programs into event-driven C code, which is then handed over to a regular C compiler. This approach retains the best of both worlds: unified threads for the programmer, easier to reason about and with a simple way to switch scheduling mode, and tiny event handlers at runtime with fast context switches which allow tens of thousands of threads to be created, even on platforms with constrained resources.

The power of continuations

The *CPC translator* translates CPC programs in threaded style into equivalent plain C programs in event-driven style. It automates the manual work usually performed by the event-loop programmer: splitting long-running tasks into small atomic event handlers around cooperation points, creating small data structures to pass local data from one handler to the next, and linking handlers correctly to implement the control flow of the original program. The resulting event handlers are scheduled cooperatively by an event loop, or executed by individual native threads when the programmer detaches a CPC thread to preemptive mode.

In manually written event-driven code, the programmer rolls his own data structures to register event handlers and save the pieces of local state that need to be passed to the next handler. Since we want to perform an automatic source-to-source translation, we seek a systematic way to generate event handlers from a threaded control flow. More precisely, we need some data structure to capture the state of a thread, that is to say its current point of execution and local variables, when it reaches a cooperation point.

Continuations are an abstraction that is widely used to represent the control flow, and in particular to implement concurrency, in functional programming languages. They can be implemented in a number of ways, including as data structures that fulfill our needs to capture the state of CPC threads in an event-driven style. Because the C language does not offer first-class continuations that we could use directly, the CPC translator performs a *conversion into continuation-passing style*, a transformation that introduces continuations in a program written in direct-style.

Continuations and concurrency

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, where `add` represents the arithmetic operator `+`:

Figure 1: Small continuation

```
g(add(f(5), x));
```

The continuation of `f(5)` is `g(add(□, x))` because the return value of `f`, represented by `□` in the continuation, will be added to the value of the variable `x` and then passed to `f`.

A continuation captures the context at some given point in the program: it implicitly records the current instruction, "□" in the previous example, and the local state, for instance the local variable `x` to be added to the return value of `f`. Continuations are therefore perfectly suited to implementing concurrency. If the call to `f` were a cooperation point in a threaded program for instance, saving its continuation would be enough to resume execution after a context switch.

Continuations are most often used in functional programming languages. Some of them, like Scheme [Abe+98] or Scala [Ode+04], provide first-class continuations with control operators, such as `call/cc` or `shift` and `reset` respectively, that allow a program to capture and resume its own continuations. Cooperative threads and other concurrency constructs are then built on top of these operators [HFW86; DH89; RMO09].

In functional languages that do not provide first-class continuations, continuations are encoded using other features such as first-class functions or monads. These constructs can then be used to implement concurrency libraries: concurrency monads in Haskell [Cla99], or lightweight *lwt* threads in OCaml [Vou08]. To some extent, these concurrent programs based on continuations are similar to event-driven programs: the programmer writes many small atomic functions, which make the continuations explicit, and composes them using synchronisation functions provided by the library. However, abstractions provided by functional languages make writing such programs usually less tedious than event-driven code: anonymous lambda-abstractions alleviate the burden of naming every intermediary event handler, and variables can be shared between inner functions and need not be passed explicitly from one handler to the next. With some syntactic sugar to write monads concisely, this yields pleasant and idiomatic code that is in fact more similar to threads than to hard-to-follow, event-driven code written in an imperative language.

Implementing continuations

There are several ways to implement continuations. One approach is to think of continuations as functions, and implement them using closures. The continuation `g(add(□, x))` from Fig. 1 is similar to the function `λr.g(add(r, x))` which waits for the return value of `f`, sums it with `x` and passes the result to `g`. Note that the value of `x` is captured in the closure: local variables are preserved automatically through environments. As explained above, this approach is mainly used in functional languages that do not provide first-class continuations. This technique does not work in the case of CPC because the C language does not feature first-class functions.

Continuations can also be implemented with stacks: capturing a continuation is done by copying the call stack, and resuming it by discarding the current stack and using the saved one instead. This approach is very similar to the way threads are saved and restored in concurrent programming, and it has indeed been shown that threads can implement continuations [KBD98]. It is obviously not usable in the case of CPC since our goal is to compile threads into lightweight event handlers to minimise the amount of memory used per thread.

Finally, continuations can be implemented as a stack of function calls to be performed. Contrary to the native call stack, this stack does not represent the current state of the program, but the rest of the computation as an implicit composition of functions. Consider again Fig. 1:

```
g(add(f(5), x));
```

The continuation of `f(5)` is `add(x)` · `g`. It is a stack of two function calls: first pass the result of `f(5)` to `add(□, x)`, then pass the result to `g(□)`. This is the approach used in CPC. It is similar to event-driven code: we store a list of callbacks to invoke later, along with the values of useful variables, and the return value of each callback is passed to the next. It is straightforward to implement in C, using function pointers for callbacks and structures to store function parameters, and retains only the data relevant to resume the computation.

Conversion into Continuation-Passing Style

Since the C language does not offer first-class continuations, the CPC translator needs to transform CPC programs to introduce continuations. *Conversion into Continuation-Passing Style* [SW74; Plo75], or *CPS conversion* for short, is a program transformation technique that makes the flow of control of a program explicit and provides continuations for it.

CPS conversion consists in replacing every function f in a program with a function $f^\star$ taking an extra argument, its *continuation*. Where f would return with value v, $f^\star$ invokes its continuation with the argument v. Remember the computation that we considered in Fig. 1:

```
g(add(f(5), x));
```

After CPS conversion, the function `f` becomes `f`* which receives its continuation as an additional parameter, λ`r.g(add(r, x))`.

Figure 2: Partial CPS conversion

```
f*(5, λr.g(add(r, x)));
```

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 does not use the native call stack: the information that would normally be in the call stack (the dynamic chain) is encoded within the continuation.

In the context of concurrent programming, having a handle on the continuation makes it easy to implement cooperative threads. Consider for instance the case where `f` has to yield to some other thread before returning its value. One simply needs to store the continuation λ`r.g(add(r, x))`, which has been received as a parameter, and invoke it later, after having run other threads, to resume the computations.

Figure 2 is an example of *partial* CPS conversion: only the function `f` is CPS-converted, `g` and `add` are left in direct (non-CPS) style. It is also possible to perform a full CPS conversion, translating every function.

Figure 3: Full CPS conversion

```
f*(5, λr. add*(x, r, λs. g*(s, k)));
```

Note that in the case of a full CPS conversion, the last called function, `g`*, expects a continuation too: we need to introduce a variable `k` to represent the top-level continuation, the context in which this fragment of code is executed.

We can finally rewrite Fig. 3 without lambda-terms, using the more compact implementation of continuations that we introduced in the previous section.

```
f*(5, add*(x)·g*·k);
```

This last example is fairly close to the CPS conversion actually performed by the CPC translator. The main difference is that CPC performs a partial translation: because a call to a CPS-converted function (or *CPS call*, see Section 2.2.1) is slower than a native call, we only translate these functions, called *CPS functions*, that are annotated as cooperative by the programmer with the `cps` keyword. Hence, in our example, the function `add` would probably be kept in direct-style, while `f` and `g` would be annotated with `cps`.

The hazards of imperative languages

CPS conversion, CPS-convertible form and splitting

The CPC translator is structured as a series of source-to-source passes, transforming CPC programs into plain C event-driven code. Although it might be possible to directly define a CPS conversion for the whole of the C language, we found it too difficult in practice. In particular, in the presence of loops and `goto` statements, the continuations are not as obvious as in the example shown above. In Fig. 1, the control flow merely consists in nested function calls, which makes it easy to express the continuation of `g` in terms of "functions to be called later". Therefore, the CPC translator performs several preliminary passes to bring the code into *CPS-convertible* form, a form suitable for CPS conversion, before the actual CPS-conversion step.

CPS-convertible form is similar to the example shown previously: each call to a function that must be CPS-converted is guaranteed to be followed by a tail call to another CPS-converted function.

```
f(); return g(a1,...,an);
```

Figure 4: CPS calls in CPS-convertible form

If the top-level continuation is `k`, the continuation of `f` is $\texttt{g}(\texttt{a}_1, \ldots, \texttt{a}_n) \cdot \texttt{k}$ and the CPS conversion is straightforward.

To translate a CPC program into CPS-convertible form, the CPS translator replaces the direct-style chunk of code following each CPS call by a call to a CPS function which encapsulates this chunk. We call this pass *splitting* because it splits each original CPS function into many, mutually recursive CPS functions in CPS-convertible form. These functions are similar to event handlers: they are small atomic chunks of code that end with a cooperative action, the continuation (callback) of which is explicit.

These functions encapsulating chunks of code, introduced during the splitting pass, are inner functions, defined within the original, split CPS function. They do not have their own local variables, sharing them instead with the enclosing function: this yields *free variables*, defined in the enclosing function but unbound in the inner ones. For example, in the following code, the local variable `i`, defined in `f`, is a free variable in the inner functions f_1 and f_2.

Figure 5: Split CPS function

```
cps void f(int i) {
    cps int f1() {
        i++;
        cpc_sleep(1);
        f2();
    }
    cps int f2() {
        if(i > 10) return;
        f1();
    }
    f1();
}
```

However, inner functions and free variables are not allowed in the C language; they only exist as intermediary steps in the transformations performed by the CPC translator. Another pass is needed after splitting, to eliminate these free variables and retrieve plain C in CPS-convertible form.

Lambda lifting and environments

There are two common solutions used in functional languages to eliminate free variables: lambda lifting and environments (or boxing). *Lambda lifting* binds free variables in inner functions by adding them as parameters of these functions; the body of functions is not modified, except for adding free variables as parameters at every call of an inner function. As a result, each inner function gets its own local copy of free variables, and passes it the next function. For example, here is Fig. 5 after lambda lifting, with the lifted variable `i` added as a parameter to f_1 and f_2.

```
cps int f1(int i) {
    i++;
    cpc_sleep(1);
    f2(i);
}
cps int f2(int i) {
    if(i > 10) return;
    f1(i);
}
cps void f(int i) {
    f1(i);
}
```

A copy of the variable `i` is made on every call to f_1 and f_2.

On the other hand, with *environments*, free variables are boxed in a chunk of heap-allocated memory and shared between inner functions; the body of the functions is modified so that every access to a shared variable goes through the indirection of the environment. Consider once again Fig. 5, using an environment `e` containing a boxed version of `i`.

```
typedef struct env { int i; } env;
cps int f1(env *e) {
    (e->i)++;
    cpc_sleep(1);
    f2(e);
}
cps int f2(env *e) {
    if(e->i > 10) { free e; return; }
    f1(e);
}
cps void f(int i) {
    env *e = malloc(sizeof(env));
    e->i = i;
    f1(e);
}
```

The environment `e` is allocated and initialised in `f`, freed before returning in f_2, and every access to `i` is replaced by `e->i`.

The CPC translator uses lambda lifting. We have chosen this technique because a compilation strategy based on environments would most certainly have resulted in a significant overhead. We mentioned earlier that boxing is commonly used to compile functional programs; but the overhead of allocating and freeing memory for environments, as well doing indirect memory accesses, is reasonable in a language such as Scheme because most variables are never mutated, and can therefore be kept unboxed. On the other hand in C, where mutated variables are the rule and `const` variables the exception, almost every variable would have to be boxed, hindering compiler optimisations by the use of heap-allocated variables instead of local variables. We have confirmed this intuition indeed, with benchmarks showing that, even with a careful implementation of boxing, lambda lifting is faster than environments in most cases (Chapter 7).

Duplicating mutable and extruded variables

Mutable variables There is a correctness issue when using lambda lifting in an imperative language with mutable variables. Because lambda lifting copies variables, it can lead to incorrect results if the original variable is used after the copy has been modified. For example, the following program cannot be lambda-lifted correctly.

```
cps void f(int rc) {
  cps void set() { rc = 0; return; }
  cps void done() {
    printf("rc = %d\n", rc);
    return;
  }
  set(); done(); return;
}
```

This function sets the variable `rc` to 0, regardless of its initial value, then prints it and returns. The variable `rc` is free in the functions `set` and `done`. We lambda-lift it, and rename it in each function for more clarity.

```
cps void f(int rc₁) {
  set(rc₁); done(rc₁); return;
}
cps void set(int rc₂) {
  rc₂ = 0;
  return;
}
cps void done(int rc₃) {
  printf("rc = %d\n", rc₃);
  return;
}
```

The lambda-lifted version modifies the copied variable $\mathtt{r}_2$, but then copies again $\mathtt{rc}_1$ into $\mathtt{r}_3$ and prints this copy. Therefore, it actually prints the initial value of $\mathtt{rc}_1$, which might be different from 0.

In order to ensure the correctness of lambda lifting, one needs either to replicate the modifications to every copy of the variable, or to enforce that a variable is never modified after it has been copied by lambda lifting. The former idea is hardly usable in practice because it requires to track every lifted variable: this would probably involve indirections, and would be at least as costly as using environments.

As we shall see in Chapter 4, we use the latter solution: even in the presence of mutated variables, the CPC compilation technique ensures the correctness of lambda lifting by enforcing the fact that unboxed lifted variables are never modified after having been copied by lambda lifting. As we explained before, keeping a straightforward lambda-lifting pass without boxing is essential for the efficiency of CPC programs.

Extruded variables The situation gets even worse in the presence of *extruded* variables, variables whose address has been retained in a pointer through the "address of" operator &. These variables can be modified from any function that has access to the pointer, and cannot be copied since that would make their address stored in the pointer invalid.

This is an issue for lambda lifting, which copies variables, but also for CPS conversion. Consider the following example, where the function `f` can modify the extruded variable `x` via the pointer `p`.

```
cps int set(int *p) {
    *p = 1;
    return 2;
}
cps int f() {
    int x = 0, r;
    r = set(&x); return add(r, x);
}
```

The function `f` sets `x` to 1, then adds it to 2, returning 3. If `k` is the current continuation, the continuation of the call to `set(&x)` is $\mathtt{add(x)} \cdot \mathtt{k}$, which yields the following code after CPS conversion.

```
cps void set*(int *p, cont k) {
    *p = 1;
    k(2);
}
cps void f*(cont k) {
    int x = 0;
    set*(&x, add*(x)·k);
}
```

The variable `x`, whose original value is 0, is copied in the continuation `add(x)·k` when creating it. The variable `x` is later modified by `set`, but the copy in the continuation is not updated and the code returns 2 (= 0 + 2) instead of 3.

Encapsulating extruded variables in environments solves the problem: instead of variables, a pointer to the environment is copied in the continuation. Then, the function `set` modifies the boxed variable, and the function `add` accesses the updated version, through the environment. Boxing also solves the problem of pointers to extruded variables becoming invalid when the variable is copied: since variables are boxed only once, their addresses do not change and can be used reliably by the programmer.

The magic of CPC

To preserve its correctness in the hostile context of the C language, surrounded by traps of mutable and extruded variables, the CPC translator could cowardly use systematic boxing to shield all local variables in environments. In practice, however, such a conservative approach would not be acceptable: C programs rely heavily on mutating local stack variables, and allocating all of them on the heap would add a significant overhead. We follow a bolder, less obvious approach.

The CPC translator keeps most variables unboxed, but uses lambda lifting and CPS conversion nonetheless. In fact, only extruded variables are boxed, which amounts in practice to less than 5 % of lifted variables in Hekate, our largest CPC program. Since copying variables changes their address, which in turn breaks pointers to them, we cannot avoid boxing extruded variables anyway. With this lower bound on the amount of boxing, the CPC translator manages to use lambda lifting and CPS conversion with as little boxing as possible, performing correct transformations with a limited overhead.

This result is made possible by the fact that the CPC translator does not operate on arbitrary programs. We have shown that

in an imperative call-by-value language
without extruded, static, and global variables,
CPS conversion and lambda lifting are correct
for programs in CPS-convertible form
obtained by splitting.

More precisely, we will show the following two results:

- Lambda lifting is correct when lifted functions are called in tail position (Theorem 4.1.9). Intuitively, when lifted functions are called in tail position, they never return. Hence, modifying copies of variables is not a problem since the original, out-of-date variables are not reachable anymore. As it turns out, lifted functions in CPC are the inner functions introduced by the splitting pass, which are always called in tail position.
- CPS conversion is correct for programs in CPS-convertible form (Theorem 5.4.1). Intuitively, storing copies of variables in continuations is an issue when another function later in the call chain modifies the original variable. But only local variables are copied into continuations, and a function cannot modify the local variables of other functions—except for extruded variables, which are boxed to avoid this problem.

Contributions

The main contributions of this dissertation are:

- a complete implementation of the CPC language (Chapter 2);
- a compilation scheme based on proven program transformations (Chapter 3), in particular:
  - a proof of correctness of lambda lifting for functions called in tail position in an imperative call-by-value language without extruded variables (Chapter 4),
  - a proof of correctness of the CPS conversion for programs in CPS-convertible form in an imperative language without extruded, static and global variables (Chapter 5);
- experimental results evaluating the usability and efficiency of CPC, including:
  - Hekate, a BitTorrent network server written with CPC (Chapter 2),
  - benchmarks showing that CPC is as fast as the fastest thread libraries available to us while allowing an order of magnitude more threads (Chapter 6);
- an alternative implementation, eCPC, using environments instead of lambda lifting in order to compare the overhead of indirect memory accesses and larger allocations, versus repeated copies of local variables (Chapter 7).

Chapter 1

Background

We have seen in the introduction (page 15) that threads and events are two common techniques to implement concurrent programs. We review them in more details in Section 1.1. We study in particular how to implement a simple program in both styles, what it implies in terms of code readability and memory footprint, and when each style might be more suitable.

As it turns out, events are actually a generic term to describe a wide range of manual techniques for concurrency. In Section 1.2, we compare several event-driven styles from real-world programs, and analyse how the programmer encodes the flow of control and the data flow manually in each of them.

Because threads are more convenient to write concurrent programs but sometimes not available or efficient enough, an idea to keep the best of both worlds is to translate threads into events automatically. In Section 1.3, we review existing techniques to perform this transformation, and previous work on bridging the gap between threads and events.

1.1 Threaded and event-driven styles

1.1.1 Threads

An example of threaded style

Consider the following OCaml program that counts one sheep per second.

Figure 1.1: Sequential count

```
let rec count n animal =
  print_int n; print_endline animal;
  sleep 1.;
  count (n+1) animal

(* start counting sheep *)
count 1 "␣sheep"
```

The function `count` is an infinite counter that displays the number of animals reached so far, sleeps for one second,[1] then calls itself recursively to count the next animal. This is a purely

[1] Hence this program sleeps to count sheep, instead of counting sheep to fall asleep.

sequential function which no longer works if we want to count several animals concurrently: since it never returns, it is impossible to start another counter after calling `count 0 " sheep"`.

To introduce concurrency in this program, one straightforward solution is to use *threads*. In OCaml, we only need to prefix the call to `count` with `Thread.create`. For instance, the following code counts fish and sheep in two separate threads of execution, starting the second thread half a second after the first one.

Figure 1.2: Threaded count

```
(* start counting fish and sheep concurrently *)
Thread.create count 1 "␣fish"
sleep 0.5
Thread.create count 1 "␣sheep"
```

The output looks as follows:

```
1 fish
1 sheep
2 fish
2 sheep
...
```

where one line is issued every half second.

In threaded programs, each task is executed by a separate thread with its own control flow and local variables, independent of other threads. A *scheduler* is responsible for executing each thread in turn, saving the state of the current thread and switching to the next one at points called *context switches*. In Fig. 1.2, the scheduler executes the thread counting fish, then after some time it saves the current point of execution and the value of the local variables `n` and `animal`, switches to the thread counting sheep, restores its local variables, and continues its execution where it left on the previous context switch. Provided the scheduler performs these context switches fairly and often enough, the output looks like a witness of two tasks executing simultaneously.

Implementing threads

Threads in a given program share every resource—memory space, file descriptors, etc.—except their call stack and the CPU registers. Each thread has its own call stack that contains the activation records of the functions that it is currently executing; each activation record holds the information related to a single function execution and they are stacked one above the other as functions call each other [Aho+88, p. 398]. The call stack captures most of the state of a thread's computation: the activation record of a function call stores, for instance, the local variables and function parameters for this call. Some of the state of the computation is also contained in the CPU registers. Most importantly, the stack pointer and program counter capture respectively the current thread and its current instruction. They need to be saved too so that the thread can be resumed in the exact same state later on.

When the scheduler switches from one thread to another, it saves the registers and scheduling information in a data structure called a *process control block*[2] (PCB) [Dei90, p. 57]; the

[2] The terminology is not uniform across operating systems. In Linux, PCB is called *process descriptor* [BC00, Chapter 3] and is stored in a structure called `task_struct`.

call stack need not be saved in most cases since it is already located in memory. The scheduler then decides which thread to run next, grabs its PCB, and copies its content back into the registers. Since the registers contain in particular a pointer to its call stack, it is then ready to resume its execution until the next context switch.

Threads are implemented and scheduled either by the operating system (OS), or by a user-space library. For *user-space threads* in interpreted languages or languages using a virtual machine, performing a context switch is sometimes as simple and efficient as swapping two pointers. *Native threads*, scheduled by the OS, are usually more heavyweight because each context switch is associated with a switch to supervisor mode.

When threads are not suitable

There are a number of reasons why a programmer might not want, or be able, to use threads to write a concurrent program: memory overhead, efficiency, lack of language or platform support.

The most common reason for avoiding threads is because one cannot afford the memory overhead that they entail. Since each thread reserves space for its own stack, it uses a fixed amount of memory. Because of this historical implementation choice, threads are not the ideal abstraction in the case of systems with many idle threads, which do not perform any useful work but keep wasting memory. In embedded systems with limited resources, this quickly becomes an unacceptable overhead as the number of threads increases. This is also an issue for highly concurrent programs that wish to use many threads: for instance, a web server that accepts ten thousands clients and uses three threads per client would need around 4 GB of memory, for stack space alone, using the NPTL Posix threads library on Linux. Previous authors have proposed implementations techniques to reduce this memory overhead, such as InterLISP's spaghetti stacks [BW73], GCC's split stacks [Tay11], or linked-stack [Beh+03] and shared-stack threads [Gu+07]. However, in most real-world cases, programmers use events instead of threads.

Threads are also sometimes avoided for efficiency reasons, because they might interact badly with CPU caches. Because threads share a common memory space, two threads writing in distinct but close variables will hit the same cache line, an issue known as *false sharing*: there is no real conflict between the writes performed by the two threads, but when the variables happen to be in the same cache line, the CPU considers them as a single block. On computers with multiples processors or processor cores, this generates cache coherency traffic between the processors, and this traffic has a significant impact on performance. To avoid false sharing, the programmer might use processes instead of threads, sharing and synchronising explicitly the necessary memory chunks; dthreads is an implementation of threads upon processes that automates this idea [LCB11]. Another alternative is to use a distributed event-driven architecture such as AMPED [PDZ99] or SEDA [WCB01]: the program is split into several processes, each of them executes an independent event loop, and communication and synchronisation are performed by the means of events exchanged through queues between the various processes.

Beyond space and time efficiency, there are other, more fundamental, reasons why a programmer might not be able to use threads. Some programming languages do not provide threads as a concurrency primitive. This is the case for instance of Javascript[Ecm09]: it would be impossible to write the "counting sheep" example (Fig. 1.2) in Javascript because there is no such thing as a `Thread.create` function for this language. (In fact, even the sequential example shown in Fig. 1.1 could not be written because there is no `sleep` function in Javascript.) To help ensure the reactivity of programs, the designers of the language have decided to exclude *synchronous* functions, which might block, and to provide only *asynchronous* alternatives, to be used in event-driven style. We shall see in the next section how to use such functions to write an event-driven equivalent of Fig. 1.2.

Asynchronous interfaces can also be imposed by the underlying OS, for example as a way to perform non-blocking Input/Output operations (I/O). Sometimes, the OS does not provide threads at all, and the sole API for I/O is asynchronous. This is common in particular in embedded systems, where the limited resources preclude the use of threads. For example, some adapters for RAID hard drives developed by IBM around the year 2000 used an architecture called Independent Packet Network (IPN). Each IPN node ran a small firmware, the IPN kernel, whose system calls were all asynchronous, from allocating and freeing memory to reading and writing network packets. This forced the device drivers developed above the IPN kernel to be implemented in purely event-driven style [Key10]. It also happens that asynchronous I/O is an optional complement to threads, designed to use multiple processors and improve performance; using them then yields a hybrid style mixing threads and events. This is the case on Windows, where I/O Completion Ports (IOCP [RN11, Chapter 10]) are the recommended way to perform efficient I/O, in combination with threads.

1.1.2 Events

A common alternative to threads for writing concurrent programs is the use of *events*. In threaded style, each task is contained in a single execution unit, a thread, that is suspended and resumed by a scheduler. In event-driven programming, each task is split in several small functions, called *event handlers* or *callbacks*, that are scheduled by an event loop. The execution of these event handlers is triggered by certain events, like the expiration of a timeout, the availability of data for I/O, or a client connecting to a network server. The event loop repeatedly collects new events, compares them to a set of registered *event listeners* and dispatches them to the relevant event handlers. The event handlers then execute atomically: the event loop starts new event handlers but, contrary to preemptive thread schedulers, it can never suspend or interrupt them. Each event handler is responsible for registering its own event listeners with the event loop to carry on its task.

An example of event-driven style

Consider the example of counting fish and sheep, re-written in event-driven style (Fig. 1.3). The first change with respect to the threaded version is the call to a function `startEventLoop` to launch the program (1): contrary to threads which are part of the OCaml language and have

```
let startEventLoop : unit -> unit = fun ()  -> (* ... *)
let runAfter : float -> (unit -> unit) -> unit = fun t f -> (* ... *)

let rec count n animal =
  print_int n; print_endline animal;
  runAfter 1. (fun () -> count (n+1) animal)          (* 2 *)

runAfter 0.  (fun () -> count 1 "␣fish")              (* 3 *)
runAfter 0.5 (fun () -> count 1 "␣sheep")             (* 4 *)
startEventLoop()                                      (* 1 *)
```

Figure 1.3: Event-driven count

an implicit scheduler embedded in the OCaml runtime, there is no event loop to schedule event handlers provided by the language. It needs to be written by the programmer (a sample implementation is detailed in Fig. 1.4) and invoked explicitly. Note that this is not always the case and depends in fact on the concurrency model offered by the language: Javascript, for instance, provides no thread but offers a `runAfter` function and has an implicit event loop associated with every program.

The second change, more fundamentally tied to the event-driven model, is the introduction of a function `runAfter`. The purpose of `runAfter` is to register an event handler function `f` with the event loop to handle a timeout event: calling `runAfter t f` schedules the execution of the function `f` after `t` seconds. Hence, instead of sleeping for one second then calling itself recursively as it did in threaded style, the function `count` registers a callback with the main event loop to execute its next step one second later (2). Similarly, the tasks counting fish and sheep are scheduled to start with a half-second interval (3 and 4).

Implementing an event loop

Figure 1.4 shows a naive implementation of the functions `runAfter` and `startEventLoop` used in Fig. 1.3.

We need to keep track of the current time (1) and of the list of timeouts (2). The latter is represented as a list of pairs: the first value is the expiration time of the timeout, in seconds since the Epoch, and the second one is the handler to invoke when the timeout triggers. The function `runAfter` computes the expiration time (3) and adds the pair to the list of timeouts (4). The function `startEventLoop` finds the next timeout (5), sleeps for the time remaining until it expires (6), then looks for expired timeouts (7), removes them from the list (8) and executes them (9). It loops until the list of timeouts is empty (10).

Note that this implementation is not efficient for a high number of timeouts because it does not even keep the list of timeouts sorted, hence needing to traverse it every time it looks for the minimal timeout `tmin`. More efficient data structures, like double-ended queues or heaps, are used to implement timeout queues in realistic event-driven programs. It is not complete either: a full-fledged event loop would offer functions to listen to more kinds of events, to stop and to restart event listeners, and it would sometimes use a local rather than global variable `state` to enable several event loops to run independently.

Figure 1.4: Implementation of an event loop

```
type loop_state = {
  mutable current_time : float                                    (* 1 *)
  mutable timeouts : (float * (unit -> unit)) list;               (* 2 *)
}
let state = { timeouts = []; current_time = Unix.gettimeofday () }

let runAfter : float -> (unit -> unit) -> unit = fun t f ->
  let t' = state.current_time +. t in                             (* 3 *)
  state.timeouts <- (t',f) :: state.timeouts                      (* 4 *)

let rec startEventLoop : unit -> unit = fun ()  ->
  match state.timeouts with
  | [] -> ()
  | l ->
     let tmin =                                                   (* 5 *)
        List.fold_left (fun m (t,_) -> min m t) 0. l in
     let timeout = max (tmin -. Unix.gettimeofday()) 0. in
     ignore(Unix.select [] [] [] timeout);                        (* 6 *)
     let current = Unix.gettimeofday () in
     let (now, later) = List.partition
         (fun (t,_) -> t <= current) state.timeouts in            (* 7 *)
     state.current_time <- current;
     state.timeouts <- later;                                     (* 8 *)
     List.iter (fun (_, f) -> f()) now;                           (* 9 *)
     startEventLoop ()                                           (* 10 *)
```

Memory footprint We mentioned that threads must sometimes be avoided because of the space wasted by their fixed-size stacks (page 31). To support this point, we compare the memory footprint of the thread and events implementations. We start an increasing number of tasks counting animals and measure the maximum resident size of the process, on a system with a x86-64 processor and 4 GB of RAM. Figure 1.5 shows the advantage of events over threads when tens of thousands of tasks are required, or for memory constrained devices.

Each call to `runAfter` allocates around 330 bytes to store the closure `fun () -> count (n+1) animal` and the list of timeouts; in comparison, each thread allocates a stack of 8 MB. However, most of these allocations is virtual memory and, in practice, each thread uses 34 kB of physical memory. As a result, starting 10 000 timeouts uses 3 MB of physical memory, whereas creating the same number of threads uses 100 times as much memory—and eats up 82 GB of virtual memory. Scheduling 100 000 timeouts uses only 36 MB of memory; creating that many threads is not even possible on our test system because it exceeds system limits.

The downsides of events

One major downside of event-driven style is that it makes the control flow much harder to follow. Because event handlers are executed atomically, they must never block: if an event handler calls a blocking function—for instance, if it sleeps during one second—it will also block the whole program. Instead, event handlers that need to wait for an operation to

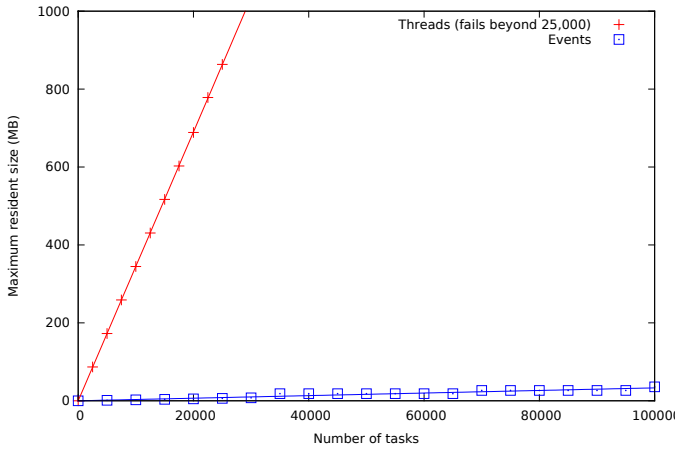


Figure 1.5: Memory footprint of threads and events

complete must use an asynchronous equivalent, with a callback function invoked when the event signaling the completion of the operation triggers. For that reason, in an event-driven program, the control flow of each task is split into many atomic handlers that are linked together by callbacks around each blocking point.

This property is not obvious in Fig. 1.3 because it consists of a single tail recursive function `count`. There is therefore no need to split it: `count` registers itself as a handler for the timeout event. Consider the following function, with a linear control flow but several calls to the blocking function `sleep`:

```
let manySleeps () =
    sleep 1.;
    print_endline "Hello␣";
    sleep 2.;
    print_endline "world...␣";
    sleep 3.;
    print_endline "and␣goodbye!"
```

Figure 1.6: Sequential blocking calls

In event-driven style, it becomes:

```
let manySleeps () =
    runAfter(1., fun () ->
        print_endline "Hello␣";
        runAfter(2., fun () ->
            print_endline "world...␣";
            runAfter(3., fun () ->
                print_endline "and␣goodbye!")))
```

Figure 1.7: Nested callbacks

And in a language without first-class functions, like C, we would need to name every intermediate callback:

```
let manySleeps () =
    let rec manySleeps1 () =
        print_endline "Hello␣";
        runAfter(2., manySleeps2)
```

Figure 1.8: Nested named callbacks

```
and manySleeps2 () =
    print_endline "world... ";
    runAfter(3., manySleeps3)
and manySleeps3 () =
    print_endline "and goodbye!"
in runAfter(1., manySleeps1)
```

Even in this simple, sequential case the control flow is harder to understand in the event-driven versions than in the threaded one. As we shall see in the next section, for larger and more complex functions featuring nested loops, conditional jumps and several blocking points, the event-driven equivalent is even more obscure.

Another weak point of events is the difficulty of debugging. Debugging event-driven code is painful because of the lack of a call stack. Since event handlers return to the event loop whenever they are done, the call stack is reduced to the event loop calling the latest event handler. There is no hint as to which previous event handler registered the current one, which one in turn registered the previous one, and so on. The values of local variables in these past handlers are lost as well, although they could sometimes be very valuable to track a bug of which the effects are visible only after a few turns of the event loop. This phenomenon, sometimes known as "stack ripping" [Ady+02], differs from threaded style where one can inspect the dynamic stack to determine the nested function calls that led to the current point of execution as well as the value of local variables in every intermediary function—except for optimised tail calls, which produce a flat stack similar to event-driven style.[3]

1.2 Control flow and data flow in event-driven code

Since event-driven programs do not use the native call stack to store return addresses and local variables, they must encode the control flow and data flow in data structures, the bookkeeping of which is the programmer's responsibility. This yields a diversity of styles among event-driven programs, depending on the programmer's taste, creativity, and his perception of efficiency. In this section, we analyse how control flow and data flow are encoded in several examples of real-world event-driven programs, and compare them to equivalent threaded-style programs.

1.2.1 Control flow

Two main techniques are used to represent the control flow in event-driven programming: callbacks and state machines.

Callbacks Most of the time, control flow is implemented with *callbacks*. Instead of performing a blocking function call, the programmer calls a non-blocking equivalent that cooperates

[3] Some implementations manage to preserve debugging information in spite of tail call optimisation. For instance, MIT/GNU Scheme's interpreter keeps a copy of stack frames in a ring-buffer, used only when a backtrace is needed. [Mas11, Section 5.2 (The Command-Line Debugger)]

Figure 1.9: Accept loop in threaded style

```
cps int
cpc_accept(int fd)
{
    cpc_io_wait(fd, CPC_IO_IN);
    return accept(fd, NULL, NULL);
}

cps int
accept_loop(int fd)
{
    int client_fd;

    while(1) {
        client_fd = cpc_accept(fd);

        cpc_spawn httpTimeout(client_fd, clientTimeout);
        cpc_spawn httpClientHandler(client_fd);
    }
}
```

with the event loop, providing a function pointer to be called back once the non-blocking call is done. This callback function is actually the continuation of the blocking operation.

Developing large programs raises the issue of composing event handlers. Whereas threaded code has return addresses stored on the stack and a standard calling sequence to coordinate the caller and the callee [JR81], event-driven code needs to define its own strategy to layer callbacks, storing the callback to the next layer in some data structure associated with the event handler. The "continuation stack" of callbacks is often split in various places of the code, each callback encoding its chunk of the stack in an ad-hoc manner.

Consider for instance the accept loop of an HTTP server that accepts clients and starts two tasks for each of them: a client handler, and a timeout to disconnect idle clients. With cooperative threads, this would be implemented as a mere infinite loop with a cooperation point. Figure 1.9 shows such an accept loop written with CPC. The programmer would call `cpc_spawn accept_loop(fd)` to create a new thread that runs the accept loop; the function `accept_loop` then waits for incoming connections with the primitive `cpc_io_wait`, and creates two new threads for each client (`httpTimeout` and `httpClientHandler`), which kill each other upon completion.

Figure 1.10 shows the (simplified) code of the accepting loop in Polipo, a caching web-proxy written by Chroboczek [Chr08]. This code is equivalent to the threaded version in Fig. 1.9, and uses several levels of callbacks.

In Polipo, the accept loop is started by a call to:

```
schedule_accept(fd, httpAccept, NULL);
```

This function stores the pointer to the (second-level) callback `httpAccept` in the `handler` field of the `request` data structure (1), and registers a (first-level) callback to `do_scheduled_`

Figure 1.10: Accept loop callbacks in Polipo

```
FdEventHandlerPtr
schedule_accept(int fd,
                int (*handler)(int, FdEventHandlerPtr, AcceptRequestPtr),
                void *data)
{
    FdEventHandlerPtr event;
    AcceptRequestRec request;
    int done;

    request.fd = fd;
    request.handler = handler;                                  /* 1 */
    request.data = data;
    event = registerFdEvent(fd, POLLOUT|POLLIN,                 /* 2 */
                            do_scheduled_accept,
                            sizeof(request), &request);
    return event;
}

int
do_scheduled_accept(int status, FdEventHandlerPtr event)
{
    AcceptRequestPtr request = (AcceptRequestPtr)&event->data;
    int rc, done;

    rc = accept(request->fd, NULL, NULL);                       /* 3 */
    done = request->handler(rc, event, request);                /* 4 */
    return done;
}

int
httpAccept(int fd, FdEventHandlerPtr event, AcceptRequestPtr request)
{
    HTTPConnectionPtr connection;
    TimeEventHandlerPtr timeout;

    connection = httpMakeConnection();
    timeout = scheduleTimeEvent(clientTimeout,                  /* 5 */
                                httpTimeoutHandler,
                                sizeof(connection), &connection);
    connection->fd = fd;
    connection->timeout = timeout;
    connection->flags = CONN_READER;
    do_stream_buf(IO_READ | IO_NOTNOW, connection->fd, 0,       /* 6 */
                  &connection->reqbuf, CHUNK_SIZE,
                  httpClientHandler, connection);
    return 0;
}
```

For readability reasons, the code has been greatly simplified (in particular, error handling has been omitted completely).

`accept`, through `registerFdEvent`. Each time the file descriptor `fd` becomes ready (not shown), the event loop calls the (first-level) callback `do_scheduled_accept`, which performs the actual `accept` system call (3) and finally invokes the (second-level) callback `httpAccept` stored in `request->handler` (4).

This callback schedules two new event handlers, `httpTimeout` and `httpClientHandler`. The former is a timeout handler, registered by `scheduleTimeEvent` (5); the latter reacts I/O events to read requests from the client, and is registered by `do_stream_buf` (6). Note that those helper functions that register callbacks with the event loop use other intermediary callbacks themselves, just like `schedule_accept` uses `do_schedule_accept`.

In the original Polipo code, things are even more complex since `schedule_accept` is called from `httpAcceptAgain`, yet another callback that is registered by `httpAccept` itself in some error cases. The control flow becomes very hard to follow, in particular when errors are triggered: each callback must be prepared to cope with error codes, or to follow-up the unexpected value to the next layer. In some parts of the code, this style looks a lot like an error monad manually interleaved with a continuation monad. Without a strict discipline and well-defined conventions about composition, the flexibility of callbacks easily traps the programmer in a control-flow and storage-allocation maze.

State machines When the multiplication of callbacks becomes unbearable, the event-loop programmer might refactor his code to use a state machine. Instead of splitting a computation into as many callbacks as it has atomic steps, the programmer registers a single callback that will be called over and over until the computation is done. This callback implements a state machine: it stores the current state of the computation into an ad-hoc data structure, just like threaded code would store the program counter, and uses it upon resuming to jump to the appropriate location.

Figure 1.11 shows how the initial handshake of a BitTorrent connection is handled in *Transmission*[4], a popular and efficient BitTorrent client written in (mostly) event-driven style. Until the handshake is over, all data arriving from a peer is handed over by the event loop to the `canRead` callback. This function implements a state machine, whose state is stored in the `state` field of a `handshake` data structure. This field is initialised to `AWAITING_HANDSHAKE` when the connection is established (not shown) and updated by the functions responsible for each step of the handshake.

The first part of the handshake is dispatched by `canRead` to the `readHandshake` function (1). It receives the buffer `inbuf` containing the bytes received so far; if not enough data has yet been received to carry on the handshake, it returns `READ_LATER` to `canRead` (2), which forwards it to the event loop to be called back when more data is available (3). Otherwise, it checks the BitTorrent header (4), parses the first part of the handshake, registers a callback to send a reply handshake (not shown), and finally updates the state (5) and returns `READ_NOW` to indicate that the rest of the handshake should be processed immediately (6).

Note what happens when the BitTorrent header is wrong (4): the function `tr_handshakeDone` is called with `false` as its second parameter, indicating that some error occurred.

[4] `http://www.transmissionbt.com/`.

Figure 1.11: Handshake state machine in Transmission

```
static ReadState
canRead(struct evbuffer * inbuf, tr_handshake * handshake) {
    ReadState ret = READ_NOW;

    while(ret == READ_NOW) {
        switch(handshake->state) {
            case AWAITING_HANDSHAKE:                              /* 1 */
                ret = readHandshake    (handshake, inbuf);
                break;
            case AWAITING_PEER_ID:                                /* 7 */
                ret = readPeerId       (handshake, inbuf);
                break;
            /* ... cases dealing with encryption omitted */
        }
    }
    return ret;                                                   /* 3 */
}

static int
readHandshake(tr_handshake * handshake, struct evbuffer * inbuf) {
    uint8_t pstr[20], reserved[HANDSHAKE_FLAGS_LEN],
            hash[SHA_DIGEST_LENGTH];

    if(evbuffer_get_length(inbuf) < INCOMING_HANDSHAKE_LEN)
        return READ_LATER;                                        /* 2 */
    tr_peerIoReadBytes(handshake->io, inbuf, pstr, 20);
    if(memcmp(pstr, "\023BitTorrent␣protocol", 20))              /* 4 */
        return tr_handshakeDone(handshake, false);
    tr_peerIoReadBytes(handshake->io, inbuf, reserved, sizeof(reserved));
    tr_peerIoReadBytes(handshake->io, inbuf, hash, sizeof(hash));
    /* ... parsing of handshake and sending reply omitted */
    handshake->state =  AWAITING_PEER_ID;                         /* 5 */
    return READ_NOW;                                              /* 6 */
}

static int
readPeerId(tr_handshake * handshake, struct evbuffer * inbuf) {
    uint8_t peer_id[PEER_ID_LEN];

    if(evbuffer_get_length(inbuf) < PEER_ID_LEN)                  /* 8 */
        return READ_LATER;
    tr_peerIoReadBytes(handshake->io, inbuf, peer_id, PEER_ID_LEN);
    /* ... parsing of peer id omitted */
    return tr_handshakeDone(handshake, true);                     /* 9 */
}
```

For readability reasons, the code has been greatly simplified (in particular, encryption support has been omitted completely).

This function (not shown) is responsible for invoking the callback `handshake->doneCB` and then deallocating the `handshake` structure. This is another example of the multiple layers of callbacks mentioned above.

If the first part of the handshake completes without error, `canRead` then dispatches the buffer to `readPeerId` which completes the handshake (7). Just like `readHandshake`, it returns `READ_LATER` if the second part of the handshake has not arrived yet (8) and finally calls `tr_handshakeDone` with `true` to indicate that the handshake has been successfully completed (9).

In the original code, ten additional states are used to deal with the various steps of negotiating encryption keys. The last of these steps finally rolls back the state to `AWAITING_HANDSHAKE` and the keys are used by the function `tr_peerIoReadBytes` to decrypt the rest of the exchange transparently. The state machine approach makes the code slightly more readable than using pure callbacks.

1.2.2 Data flow

Since each callback function performs only a small part of the whole computation, the event-loop programmer needs to store temporary data required to carry on the computation in heap-allocated data structures, whereas stack-allocated variables would sometimes seem more natural in threaded style. The content of these data structures depends heavily on the program being developed but we can characterise some common patterns.

Event loops generally provide some means to specify a `void*` pointer when registering an event handler. When the expected event triggers, the pointer is passed as a parameter to the callback function, along with information about the event itself. This allows the programmer to store partial results in a structure of his choice, and recover it through the pointer without bothering to maintain the association between event handlers and data himself.

Coarse-grained, long lived data structures These data structures are usually large and coarse-grained. They correspond to some meaningful object in the context of the program, and are passed from callback to callback through a pointer. For instance, the `connection` structure used in Polipo (Figure 1.10) is allocated by `httpMakeConnection` when a connection starts and passed to the callbacks `httpTimeoutHandler` and `httpClientHandler` through the registering functions `scheduleTimeEvent` (5) and `do_stream_buf` (6). It lives as long as the HTTP connection it describes and contains no less than 22 fields (Fig. 1.12). The `tr_handshake` structure passed to `canRead` in Transmission (Fig. 1.11) is similarly large, with 18 fields (Fig. 1.13).

Some of these fields need to live for the whole connection (eg. `fd` which stores the file descriptor of the socket) but others are used only transiently (eg. `buf` which is filled only when sending a reply), or even not at all in some cases (eg. the structure `HTTPConnectionPtr` is used for both client and server connections, but the `pipelined` field is never used in the client case). Even if it wastes memory in some cases, it would be too much of a hassle for the programmer to track every possible data flow in the program and create ad-hoc data structures for each of them.

Figure 1.12: Data structure for HTTP connections in Polipo

```
typedef struct _HTTPConnection {
    int flags;
    int fd;
    char *buf;
    int len;
    int offset;
    HTTPRequestPtr request;
    HTTPRequestPtr request_last;
    int serviced;
    int version;
    int time;
    TimeEventHandlerPtr timeout;
    int te;
    char *reqbuf;
    int reqlen;
    int reqbegin;
    int reqoffset;
    int bodylen;
    int reqte;
    /* For server connections */
    int chunk_remaining;
    struct _HTTPServer *server;
    int pipelined;
    int connecting;
} HTTPConnectionRec, *HTTPConnectionPtr;
```

Figure 1.13: Data structure for handshakes in Transmission

```
struct tr_handshake {
    bool                haveReadAnythingFromPeer;
    bool                havePeerID;
    bool                haveSentBitTorrentHandshake;
    tr_peerIo *         io;
    tr_crypto *         crypto;
    tr_session *        session;
    uint8_t             mySecret[KEY_LEN];
    handshake_state_t   state;
    tr_encryption_mode  encryptionMode;
    uint16_t            pad_c_len;
    uint16_t            pad_d_len;
    uint16_t            ia_len;
    uint32_t            crypto_select;
    uint32_t            crypto_provide;
    uint8_t             myReq1[SHA_DIGEST_LENGTH];
    handshakeDoneCB     doneCB;
    void *              doneUserData;
    struct event       * timeout_timer;
};
```

```
typedef struct _AcceptRequest {
    int fd;
    int (*handler)(int, FdEventHandlerPtr, struct _AcceptRequest*);
    void *data;
} AcceptRequestRec, *AcceptRequestPtr;
```

Figure 1.14: Data structure for accept loop in Polipo

Minimal, short-lived data structures In some simple cases, however, the event-loop programmer is able to allocate very small and short-lived data structures. These minimal data structures are allocated directly within an event handler and are deallocated when the associated callback returns. They might even be allocated on the stack by the programmer and copied inside the event-loop internals by the helper function registering the event handler. The overhead is therefore kept as low as possible.

For instance, the function `schedule_accept` passes a tiny, stack-allocated structure `request`, of type `AcceptRequestRec` (Fig. 1.14), to the helper function `registerFdEvent` (Fig. 1.10 (2)). This structure is copied by `registerFdEvent` in the event-loop data structure associated with the event, and freed automatically after the callback `do_scheduled_accept` has returned; it is as short-lived and (almost) as compact as possible.

As it turns out, creating truly minimal structures is hard: `AcceptRequestRec` could in fact be optimised to get rid off the fields `data`—which is always `NULL` in practice in Polipo—and `fd`—which is also present in the encapsulating `event` data structure. Finding every such redundancy in the data flow of a large event-driven program would be a daunting task, hence the spurious and redundant fields used to lighten the programmer's burden.

1.3 From threads to events through continuations

We have seen in Section 1.1.1 that the programmer sometimes cannot avoid event-driven style, although threaded style is more convenient to write concurrent programs. To keep the best of both worlds, it is then natural to want to translate threads into events automatically. That way the programmer writes his program using threads, which are easier to reason about, and delegates to the compiler the task of generating efficient but intricate event-driven code.

The translation of threads into events has been rediscovered many times [Dun+06; KKK07; FMM07; Key10]. Unfortunately, these many implementations often lack a formal description of the transformation steps involved, let alone a proof of correctness. It is our thesis that these translations are in fact a few classical transformation techniques, studied extensively in the context of functional languages, and adapted—sometimes unknowingly—to imperative languages by programmers trying to solve the issue of writing events in a threaded style.

In this section, we first review existing solutions for writing event-driven programs in a threaded style. We then present the use of continuations to implement concurrency in functional languages. Finally, we give a brief overview of the techniques that we identified in existing threads-to-events translators, and used in the CPC translator: CPS conversion, lambda lifting and splitting.

1.3.1 Events in a threaded style

There has been a lot of research to provide efficient threads, as well as to make event-driven programming easier [Beh+03; WCB01; PDZ99; CK05; Dab+02]. We do not intend to resurrect an old debate on which model is better [Ous96; MY98; BCB03]: we have seen above that each of them has its advantages and drawbacks, without any decisive argument for either. We focus instead on results involving a transformation between threads and events, or building bridges between them. Most of these results consist in generating event-driven code from a threaded description, automating partially the work of saving flow of control and local state.

We would first like to clarify what we believe is a widespread misconception. It is often reported that threads and events are computationally equivalent, and this result is attributed to Lauer and Needham [LN79]. But their paper is not about threads and events: it is about the equivalence of procedure-oriented and message-oriented systems.[5] They map processes to monitors, sending messages to forking, and dispatch loops to locks. While message-oriented systems bear some similarity to event-driven programs, they also have fundamental differences: preemptive scheduling, no shared state, no global lock, communication and synchronisation done exclusively through message-passing. Whether threads and events are equivalent, for some definition of equivalence, remains an unstudied problem in the literature.

Adya et al. [Ady+02] introduce the notion of *stack ripping*: it describes the need, when a formerly atomic function needs to be made cooperative, to split every function (transitively) leading to it in the call graph into two new functions, to manually encode the call stack into chained callbacks at the point of cooperation. The authors single out this issue as the "the primary drawback to manual stack management". They advocate (but do not implement) static check that cooperative functions are only called from other cooperative functions. They present adaptors between event-driven and threaded code to write hybrid programs mixing both styles.

Duff introduces a technique, known as *Duff's device* [Duf83], to express general loop unrolling directly in C, using the `switch` statement. Since then, this technique has been employed multiple times to express state machines and event-driven programs in a threaded style.[6] For instance, it is used by Tatham to implement coroutines in C [Tat00]. Other C libraries later expanded this idea, such as *protothreads* [Dun+06] and *FairThreads*' automata [Bou06]. These libraries help keep a clearer flow of control but they provide no automatic handling of local variables: the programmer is expected to save them manually in his own data structures, just like in event-driven style.

Tame [KKK07] is a C++ language extension and library which exposes events to the programmer but does not impose event-driven style: it generates state machines to avoid the

[5] The first to cite Lauer and Needham in the threads-versus-events debate seem to be Adya et al. [Ady+02]. However they mention explicitly that "[Lauer and Needham's] comparison is decidedly not between the models we associate with multithreaded and event-driven programming." For some reason this sentence seems to have been overlooked in other papers which spread the misconception.

[6] This abuse was already envisioned by Duff in 1983: "I have another revolting way to use switches to implement interrupt driven state machines but it's too horrid to go into." [Duf83]

stack ripping issue and retain a thread-like feeling. The programmer needs to annotate local variables that must be saved across context switches.

TaskJava [FMM07] implements the same idea as Tame, in Java, but preserves local variables automatically, storing them in a state record. *Kilim* [SM08] is a message-passing framework for Java providing actor-based, lightweight threads. It is also implemented by a partial CPS conversion performed on annotated functions, but contrary to TaskJava, it works at the JVM bytecode level.

AC [Har+11] is a set of language constructs for composable asynchronous I/O in C and C++. Harris et al. introduce `do..finish` and `async` operators to write asynchronous requests in a synchronous style, and give an operational semantics. The language constructs are somewhat similar to those of Tame but the implementation is very different, using LLVM code blocks or macros based on GCC's nested functions rather than source-to-source transformations.

Haller and Odersky [HO09] advocate unification of thread-based and event-based models through actors, with the `react` and `receive` operators provided by the *Scala Actors* library; suspended actors are represented by continuations.

With Javascript becoming increasingly popular for web development in the last few years, multiple control-flow libraries have been written to automate recurrent asynchronous patterns. Among dozens of others, some of the most popular ones seem to be *Step*[7], *FuturesJS*[8], and *Seq*[9]. They have in common to use Javascript's closures and prototype-based objects to dynamically store the call stack, but the API and implementation details vary a lot. Unfortunately, these implementations are rarely described in the literature, and the main source of documentation is blog posts using non-standard terminology scattered around the web. *Arrowlets* by Khoo et al. [Kho+09] is a notable exception: it provides arrows and arrow combinators, borrowed from the Haskell community [Hug00], and the paper describes explicitly the implementation in terms of continuation-passing style.

1.3.2 Continuations and concurrency

We have seen that continuations offer a natural framework to implement concurrency systems in functional languages, since 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 [Rep93], 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 [CFW85; HFW86; DH89] (previously known as `catch` [Wan80]). Danvy et al. [DSZ09, Section 4.4] use continuations provided by the Rhino Javascript implementation to define Landin's J operator, which they then use to implement coroutines. *Stackless Python* [Tis00] uses first-class continuations to

[7] `https://github.com/creationix/step/`.
[8] `http://coolaj86.info/futures/`.
[9] `https://github.com/substack/node-seq/`.

implement generators, which are in turn used to implement concurrency primitives. Scala also uses first-class continuations, through the `shift` and `reset` operators, to implement concurrency primitives and asynchronous I/O [RMO09].

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 [Sch95], and extended by Claessen [Cla99] to a monad transformer yielding a concurrent version of existing monads. Li and Zdancewic [LZ07] use a continuation monad to lazily translate the thread abstraction exposed to the programmer into events scheduled by an event loop. In OCaml, Vouillon's *lwt* [Vou08] provides a lightweight alternative to native threads, with the ability to execute blocking tasks in a separate thread pool. The asynchronous model in F# is implemented with a localized continuation-passing translation of control-flow and a heap-based allocation of the closures, using three continuations for success, exceptions and cancellation [SPL11].

1.3.3 Transformation techniques

The three main techniques used in this thesis to compile threads into events—CPS conversion, lambda lifting and splitting—are fairly standard techniques for compiling functional languages, or at least languages providing inner functions.

The conversion into continuation-passing style, or CPS conversion, has been discovered and studied many times in different contexts [Plo75; SW74; Rey93]. It is used for instance to compile Scheme (Rabbit [Ste78]) and ML (SML/NJ [App92]), both of them exposing continuations to the programmer.

Lambda lifting, also called closure-conversion, is a standard technique to remove free variables. It is introduced by Johnsson [Joh85] and made more efficient by Danvy and Schultz [DS04]. Fischbach and Hannan prove its correctness for a call-by-name language[FH03]. Although environments are a more common way to handle free variables, some implementations use lambda lifting; for instance, the *Twobit* Scheme-to-C compiler [Cli98].

We call *splitting* the conversion of a complex flow of control into mutually recursive function calls. Van Wijngaarden is the first to describe such a transformation, in a source-to-source translation for Algol 60 [Wij66]. The idea is then used by Landin to formalise a translation between Algol and the lambda calculus [Lan65], and by Steele and Sussman to express gotos in applicative languages such as LISP or Scheme [SJS76]. Thielecke adapts van Wijngaarden's transformation to the C language, albeit in a restrictive way [Thi99].

We are aware of several implementations of splitting, but none of them has been described precisely in the literature. *Weave* is an unpublished tool used at IBM around the year 2000 to write firmware and drivers for SSA-SCSI RAID storage adapters [Key10]. It translates annotated Woven-C code, written in threaded style, into C code hooked into the underlying event-driven kernel. *TameJS*[10], by the authors of the C++ *Tame* library [KKK07], is a similar tool for Javascript where event-driven programming is made mandatory by the lack

[10] `http://tamejs.org/`.

of concurrency primitives.[11] *FlapJax* [Mey+09], a functional reactive programming library for Javascript, provides a compiler which performs an "implicit lifting"; despite the name, it seems that this transformation is a restricted form of splitting, but the implementation is not detailed. *MapJAX* [Mye+07] also compiles a conservative extension of Javascript to plain Javascript using splitting and CPS conversion.[12] Interestingly enough, the authors note that, in spite of Javascript's support for nested functions, they need to perform "function denesting" for performance reasons;[13] they store free variables in environments ("closure objects") rather than using lambda lifting.

In Chapter 7, we propose to generate state machines by combining splitting, mentioned above, and defunctionalisation, a technique introduced by Reynolds [Rey72]. Another approach to transform an arbitrary program into an automaton is the Böhm-Jacopini theorem [BJ66], which states that "every flowchart is equivalent to a **while**-program with one occurrence of **while**-**do**, provided additional variables are allowed". This formulation is due to Harel, who provides an impressively thorough analysis of the literature about this result, and shows how this "folk theorem" has in fact been rediscovered many times by independent researchers [Har80].

[11] Note that, contrary to TameJS, the original Tame implementation in C++ does not use splitting but a state machine with switches.

[12] "When the compiler encounters one of the blocking method calls, it computes the continuation of the call, packages that code as a continuation function, and adds that function as an extra argument to the call." [Mye+07]

[13] "We found that access to variables declared in a nesting hierarchy was considerably slower than access to variables declared in a top-level function." [Mye+07]

Chapter 2

Programming with CPC

Continuation-Passing C (CPC) is an extension of the C programming language with concurrency primitives which is implemented by a series of source-to-source transformations. After this series of transformations, the CPC translator yields a program in hybrid style, most of it event-driven, but using the native threads of the underlying operating system wherever this has been specified by the CPC programmer, for example in order to call blocking APIs or distribute CPU-bound code over multiple cores or processors.

The CPC language was initially designed by Chroboczek, who implemented a first prototype in Lisp [Chr05]. During his PhD, the author has rewritten the CPC translator from scratch in OCaml: full support of the C language, support for extruded variables (Section 3.2.2), increased modularity. The CPC runtime has also been extended with the ability to use native threads (`cpc_link`), and rewritten to some extent.

This chapter presents the CPC language. In Section 2.1, we give an introduction to CPC programming through excerpts from the code of Hekate, a BitTorrent server written with CPC. It aims at giving a first taste of CPC without diving too deeply into technical details. Then, in Section 2.2, we give a more systematic presentation of CPC concepts and primitives, along with their limitations. Finally, in Section 2.3, we present several synchronisation constructs built upon the CPC primitives and provided by the CPC standard library.

2.1 An introduction to CPC

The most significant program written in CPC is *Hekate*, a *BitTorrent seeder*, a massively concurrent network server designed to efficiently handle tens of thousands of simultaneously connected peers [KC11b; AC09]. It was developed by two undergraduate students, supervised by Chroboczek and the author. Because we wanted to evaluate to evaluate the usability of CPC for casual users, we chose students who had never worked with CPC before, and were purposefully kept unaware of CPC internals. This ensured that they used a wide range of CPC features, and some patterns that were not covered in our previous tests; writing Hekate

turned out to be a powerful means to iron bugs out of CPC. It was also highly motivating to write a truly useful piece of software with CPC.

In this section, we give an overview of the CPC language through several programming idioms that we discovered while working on Hekate with our students.

2.1.1 Cooperative CPC threads

The extremely lightweight, cooperative threads of CPC lead to a "threads are everywhere" feeling that encourages a somewhat unusual programming style.

Lightweight threads Contrary to the common model of using one thread per client, Hekate spawns at least three threads for every connecting peer: a reader, a writer, and a timeout thread. Spawning several CPC threads per client is not an issue, especially when only a few of them are active at any time, because idle CPC threads entail virtually no overhead.

The first thread reads incoming requests and manages the state of the client. The BitTorrent protocol defines two states for interested peers: *"unchoked,"* where a peer might perform requests and receive chunks of data, and *"choked"* which is a waiting state. Hekate maintains 90 % of its peers in *choked* state, and *unchokes* them in a round-robin fashion.

The second thread is in charge of actually sending the chunks of data requested by the peer. It usually sleeps on a condition variable, and is woken up by the first thread when needed. Because these threads are scheduled cooperatively, the list of pending chunks is manipulated by the two threads without need for a lock.

Every read on a network interface is guarded by a timeout, and a peer that has not been involved in any activity for a period of time is disconnected. Earlier versions of Hekate which did not include this protection would end up clogged by idle peers, which prevented new peers from connecting.

In order to simplify the protocol-related code, timeouts are implemented in the buffered read function, which spawns a new timeout thread on each invocation. This temporary third thread sleeps for the duration of the timeout, and aborts the I/O if it is still pending (the implementation is detailed in Section 2.3.2). Because most timeouts do not expire and are stopped as soon as I/O has completed, this solution relies on the efficiency of spawning and context-switching short-lived CPC threads (see Chapter 6 for experimental results).

Native and CPS functions CPC threads might execute two kinds of code: *native* functions and *CPS* functions (annotated with the `cps` keyword). Intuitively, CPS functions are interruptible and native functions are not: 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 switch to another scheduler (Section 2.1.3). Note that the `cps` keyword does not mean that the function is written in continuation-passing style, but rather that it is to be CPS-converted by the CPC translator. Native code, on the other hand, is "atomic": if a sequence of native code is executed in cooperative mode, it must be completed before anything else is allowed to run in the same scheduler. As we shall see in Chapter 3, from a more technical point of view,

```
cps void
listening(hashtable * table) {
  /* ... */
  while(1) {
    cpc_io_wait(socket_fd, CPC_IO_IN);
    client_fd = accept(socket_fd, ...);
    cpc_spawn client(table, client_fd);
  }
}
```

Figure 2.1: Accepting connections and spawning threads

CPS functions are compiled by performing a transformation into Continuation-Passing Style (CPS), while native functions execute on the native stack.

There is a global constraint on the call graph of a CPC program: a CPS function may only be called by a CPS function; equivalently, a native function can only call native functions—but a CPS function may call a native function. This means that at any point in time, the dynamic chain consists of a "CPS stack" of cooperating functions, and a "native stack" of regular C functions called by the most recent CPS function. Since context switches are forbidden in native functions, only the CPS stack needs to be saved and restored when a thread cooperates.

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 by another CPS function.

Figure 2.1 shows an example of a CPS function: the function `listening` calls the primitive `cpc_io_wait` to wait for the file descriptor `socket_fd` to be ready, before accepting incoming connections with the native function `accept` and spawning a new thread for each of them. The `cpc_spawn` keyword can be used, in both native and CPS functions, to create a new thread executing a CPS function (`client` in this example).

The CPC language provides five CPS primitive functions to suspend and synchronise threads on some events. The simplest one is `cpc_yield`, which yields control to the next thread to be executed, as determined by the scheduler. The primitives `cpc_io_wait` and `cpc_sleep` suspend the current thread until a given file descriptor has data available or some time has elapsed, respectively. A thread can wait on some condition variable [Hoa74] with `cpc_wait`; threads suspended on a condition variable are woken with the (non-CPS) functions `cpc_signal` and `cpc_signal_all`. To perform early interruptions, `cpc_io_wait` and `cpc_sleep` also accept an optional condition variable which, if signalled, will wake up the waiting thread.

The fifth CPS primitive, `cpc_link`, is used to control how threads are scheduled. We give more details about it in Section 2.1.3.

We have found that these five primitives are enough to build more complex synchronisation constructs and CPS functions, such as barriers or retriggerable timeouts. Some of these

generally useful functions, written in CPC and built above the CPC primitives, are distributed with CPC and form the CPC standard library (Section 2.3).

2.1.2 Comparison with event-driven programming

Code readability Hekate's code is much more readable than its event-driven equivalents. Consider for instance the BitTorrent handshake, a message exchange occurring just after a connection is established. As we have seen in Section 1.2.1, the handshake in Transmission is a complex piece of code, spanning over a thousand lines in a dedicated file. By contrast, Hekate's handshake is a single function of less than fifty lines including error handling.

While some of Transmission's complexity is explained by its support for encrypted connections, Transmission's code is intrinsically much more messy due to the use of callbacks and a state machine to keep track of the progress of the handshake (see Fig. 1.11 on page 40 for the actual code). This results in an obfuscated flow of control, scattered through a dozen functions (excluding encryption-related functions), typical of event-driven code.

Expressivity Surprisingly enough, CPC threads turn out to naturally express some idioms that are more commonly associated with event-driven style. Consider for instance the case of buffer allocation for reading data from the network. When a native thread performs a blocking read, it needs to allocate the buffer before the `read` system call; when many threads are blocked waiting for a read, these buffers add up to a significant amount of storage. In an event-driven program, it is possible to delay allocating the buffer until after an event indicating that data is available has been received. This is done for instance by the web proxy Polipo [Chr08].

The same technique is not only possible, but actually natural in CPC. For instance in Hekate, we define a data structure `cpc_buffer` that is allocated by `cpc_buffer_get` and filled by `cpc_buffer_read`. However, the function `cpc_buffer_get` initialises the field `buf` of this structure with a null pointer. The buffer is only allocated after `cpc_io_wait` has successfully returned the first time `cpc_buffer_read` is called (Fig. 2.2). This provides the reduced storage requirements of an event-driven program while retaining the linear flow of control of threads.

2.1.3 Detached threads

While cooperative, deterministically scheduled threads are less error-prone and easier to reason about than preemptive threads, there are circumstances in which native operating system threads are necessary. In traditional systems, this implies either converting the whole program to use native threads, or manually managing both kinds of threads.

A CPC thread can switch from cooperative to preemptive mode at any time by using the `cpc_link` primitive (inspired by FairThreads' `ft_thread_link` [Bou06]). A cooperative thread is said to be *attached* to an event loop, while a preemptive one is *detached*.

The `cpc_link` primitive takes a single argument, a scheduler, either an event loop (for cooperative scheduling) or a thread pool (for preemptive scheduling). It returns the thread's

Figure 2.2: Lazy buffer allocation

```
typedef struct cpc_buffer {
  int size;             /* Size of the buffer */
  int start;            /* Start of valid data */
  int end;              /* End of valid data */
  unsigned char *buf;   /* Actual buffer content */
} cpc_buffer;

cps int
cpc_buffer_read(int fd, cpc_buffer *b, int len)
{
    ssize_t rc;
    size_t pos = b->end - b->start;

    if(!b->buf) {                   /* Lazy allocation the first time */
        cpc_io_wait(fd, CPC_IO_IN)
        b->buf = malloc(b->size);
        goto first_read;            /* Skip the first cpc_io_wait */
    } else {
        ...                         /* Move data to the beginning of
                                       buffer */
    }

    while(pos < len) {
        cpc_io_wait(fd, CPC_IO_IN);
      first_read:
        rc = read(fd, b->buf + pos, b->size - pos);
        ...                         /* Error handling */
        pos += rc;
    }
    b->end = pos;
    return pos;
}
```

previous scheduler, which makes it possible to eventually restore the thread to its original state. Syntactic sugar is provided to execute a block of code in attached or detached mode (`cpc_attached`, `cpc_detached`).

Hekate is written in mostly non-blocking cooperative style; hence, Hekate's threads remain attached most of the time. There are a few situations, however, where the ability to detach a thread is needed.

Blocking OS interfaces Some operating system interfaces, like the `getaddrinfo` DNS resolver interface, may block for a long time (up to several seconds). Although there are several libraries which implement equivalent functionality in a non-blocking manner, in CPC we simply enclose the call to the blocking interface in a `cpc_detached` block (see Fig. 2.3a).

Figure 2.3b shows how `cpc_detached` is expanded by the CPC translator into two calls to the primitive `cpc_link`. Detaching a thread is done by linking it to a preemptive scheduler;

if the user does not specify such a scheduler explicitly, we use `cpc_default_threadpool`, a default thread pool created at the beginning of every CPC program. Note that CPC takes care to attach the thread before returning to the caller function, even though the `return` statement is within the `cpc_detached` block.

Figure 2.3: Expansion of `cpc_detached` in terms of `cpc_link`

```
cpc_detached {
    rc = getaddrinfo(name, ...)
    return rc;
}
```

(a)

```
cpc_scheduler *s =
    cpc_link(cpc_default_threadpool);
rc = getaddrinfo(name, ...)
cpc_link(s);
return rc;
```

(b)

Blocking library interfaces Hekate uses the *curl* library [1] to contact BitTorrent *trackers* over HTTP. Curl offers both a simple, blocking interface and a complex, asynchronous one. We decided to use the one interface that we actually understand, and therefore call the blocking interface from a detached thread.

Parallelism Detached threads enable code to be run on multiple processors or processor cores. Hekate does not use this feature, but a CPU-bound program would detach computationally intensive tasks and let the kernel schedule them on several processing units.

2.1.4 Hybrid programming

Most realistic event-driven programs are actually *hybrid* programs [PDZ99; WCB01]: they consist of a large event loop, and a number of threads (this is the case, by the way, of the *Transmission* BitTorrent client mentioned above). Such blending of native threads with event-driven code is made very easy by CPC, where switching from one style to the other is a simple matter of using the `cpc_link` primitive.

This ability is used in Hekate for dealing with disk reads. Reading from disk might block if the data is not in cache; however, if the data is already in cache, it would be wasteful to pay the cost of a detached thread. This is a significant concern for a BitTorrent seeder because the protocol allows chunks to be requested in random order, making kernel readahead heuristics inefficient.

The actual code is shown in Fig. 2.4: it sends a chunk of data from a memory-mapped disk file over a network socket. In this code, we first trigger an asynchronous read of the on-disk data (1), and immediately yield to threads servicing other clients (2) in order to give the kernel a chance to perform the read. When we are scheduled again, we check whether the read has completed (3); if it has, we perform a non-blocking write (7); if it has not, we yield one more time (4) and, if that fails again (5), delegate the work to a native thread which can block (6).

[1] `http://curl.haxx.se/libcurl/`.

```
    prefetch(source, length);                          /* (1) */
    cpc_yield();                                       /* (2) */
    if(!incore(source, length)) {                      /* (3) */
        cpc_yield();                                   /* (4) */
        if(!incore(source, length)) {                  /* (5) */
            cpc_detached {                             /* (6) */
                rc = cpc_write(fd, source, length);
            }
            goto done;
        }
    }
    rc = cpc_write(fd, source, length);                /* (7) */
done:
    ...
```

Figure 2.4: An example of hybrid programming (non-blocking read)

The functions `prefetch` *and* `incore` *are thin wrappers around the* `posix_madvise` *and* `mincore` *system calls.*

Note that this code contains a race condition: the prefetched block of data could have been swapped out before the call to `cpc_write`, which would stall Hekate until the write completes. Moreover, this race condition is even more likely to appear as load increases and on devices with constrained resources. To avoid this race condition and ensure non-blocking disk reads, one could use asynchronous I/O. However, while the Linux kernel does provide a small set of asynchronous I/O system calls, we found them scarcely documented, very restricted and difficult to use: they work only on some file systems and require the use of the flag `O_DIRECT`, which imposes alignment restrictions on the length and address of buffers and the file offset of I/Os, and disables the caching performed by the kernel.[2] We have therefore not experimented with them.

Note further that the call to `cpc_write` in the `cpc_detached` block (6) 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 primitives such as `cpc_io_wait` are designed to act sensibly in both attached and detached mode; this translates into more complex functions built upon them, and `cpc_write` simply behaves as `write` when invoked in detached mode. For simplicity, we choose to use the CPC wrappers throughout our code.

2.2 The CPC language

This section is a complete presentation of the CPC language, concepts and primitives. It also discusses the limitations, either fundamental to the CPC compilation technique or related to the current implementation, as well as possible improvements.

[2] "The thing that has always disturbed me about O_DIRECT is that the whole interface is just stupid, and was probably designed by a deranged monkey on some serious mind-controlling substances." Linus Torvald [Tor02]

CPC threads The main abstraction provided by CPC is a *CPC thread*. From the programmer's view, a CPC thread roughly corresponds to other programming languages' notion of *thread* or *lightweight process*, except that it has no identity: there is no *thread identifier* that can be used to kill or suspend a given CPC thread.

A CPC thread can be in one of two modes: *attached* to the CPC scheduler, or *detached*. At a given time, the set of all attached threads is scheduled cooperatively, and an attached thread can only be preempted by other attached threads because of explicit programmer action. A detached thread, on the other hand, is associated with a native operating-system thread, and is scheduled by the operating system asynchronously with respect to all other threads.

Structure of a CPC program Just like a plain C program, a CPC program is a set of functions. Functions in a CPC program are partitioned into "CPS" functions and "native" functions; a global constraint is that a CPS function can only ever be called by another CPS function, never by a native function. The precise set of contexts where a CPS function can be called is the set of *CPS contexts*, defined in Section 2.2.1.

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

Technically, native function calls are executed by using the machine's native stack. CPS function calls, on the other hand, are executed by using a lightweight stack-like structure known as a continuation (Section 3.2). This arrangement makes CPC context switches extremely fast; the trade-off is that a CPS function call is at least an order of magnitude slower than a native call (Section 6.1.2). Thus, computationally expensive code should be implemented in native code whenever possible.

Execution of a CPC program starts at a native function called `main`. This function usually starts by registering a number of threads with the CPC runtime (using `cpc_spawn`), and then passes control to the CPC runtime (by calling `cpc_main_loop`, Section 2.2.2).

Reserved words CPC is a conservative extension of the 1999 edition of the C programming language; thus, the syntax of CPC is defined as a set of productions to be added to the grammar defined in the ISO C99 standard [Int99]. In the rest of this section, we write

c99-rule ::= cpc-extension

to denote that the rule from C99 *c99-rule* is extended with the extension *cpc-extension* in CPC.

In addition to the reserved words in C99, CPC reserves the words `cps`, `cpc_spawn` and `cpc_linked`; the former is a function specifier, while the latter two are statements. Their semantics is detailed in Sections 2.2.1 to 2.2.3.

Fundamental limitations Not all legal C code is allowable in CPC.The following few limitations are fundamental to the implementation technique of CPC, unlikely to be lifted in a future version: the use of the `longjmp` library function, and its variants, is not allowed in CPC code, and the use of `alloca` in CPS context yields unpredictable behaviour;[3] both of these features modify directly the call stack. Although CPC conceals as much as possible to the programmer the "stack ripping" phenomenon that occurs when translating threads to events, it fails to preserve the illusion of an unaltered stack when these functions are involved.

The function `longjmp` works in pair with the function `setjmp`: the former saves a stack context and the latter jumps to it. They are used mainly as an exception mechanism, but have also been used to implement coroutines and even garbage collection of stack-allocated variables in the Scheme-to-C compiler Chicken [Bak95; Che70]. One fundamental reason `longjmp` cannot be used in CPC is because the function that performed the corresponding `setjmp` must not have returned—in other words, `longjmp` must be called in a child of the function that called `setjmp`. As a result, it cannot work in CPC where the call stack of CPS functions is flattened, transformed into successive calls to many event handlers by the event loop.

The function `alloca` allocates memory on top of the call stack; this memory is automatically freed when the current function returns. It cannot work with CPC because, after translation, the current function will become a short-lived event handler. The allocated memory will not be preserved until the function returns, only until the next call to a CPS function—and because the CPC translator introduces many CPS functions during its splitting pass (see Section 3.4), the programmer may not even rely on the "next call" being among cooperation points in his own code: it may happen much sooner in some cases, depending on how the splitting pass is implemented. The function `alloca` is therefore impossible to use reliably in CPC.

2.2.1 CPS functions

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

- the body of a CPS function (see below) is in CPS context,
- the body of a `cpc_spawn` statement is in CPS context (Section 2.2.2).

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

CPS functions

function-specifier ::= `cps`

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

[3] Some have argued that `alloca` is neither standard nor good style, and should therefore not be used anyway, even in plain C programs.

A call to a CPS function is called a *CPS call*. CPS function can only called in CPS context; a CPS call in native context causes a compilation error.

Limitation: in the current implementation, CPS function do not accept a variable number of parameters (`va_args` and its variants). Calling a CPS function through a function pointer is also forbidden, as it yields undefined (but likely damaging) behaviour; lifting this limitation would require to make `cps` part of the function type rather than a function specifier, to be able to identify statically every use of CPS function pointers.

Inner functions

block-item ::= function-definition

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

Free variables of inner functions are copies of the variables of the enclosing function; thus, a change to the value of the free variable is not visible in the enclosing function[4]. The free variables are initialized with a copy of the current value whenever the inner function is called; thus, their initial value does not depend on the location of the inner function within the outer one.

Limitation: in the current implementation, arrays cannot appear as free variables (because they cannot be passed by value), and variable-length arrays must not be used in CPS context. A solution might be to wrap arrays appearing as free variables in structures.

Primitive CPS functions CPC provides a set of five primitive CPS functions, allowing the programmer to schedule threads and wait for some events: `cpc_link`, `cpc_yield`, `cpc_sleep`, `cpc_wait` and `cpc_io_wait`. It is important to understand that these primitive functions could not be defined outside the CPC runtime: they must have access to the internals of the scheduler to operate. Since they are CPS functions, they are valid only in CPS context. These primitives are described in the following sections.

2.2.2 Bootstrapping

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

`cpc_main_loop` Since `main` is a native function and CPS functions can only be called by other CPS functions, some means is necessary to pass control to CPS code. The function `cpc_main_loop` invokes the CPC scheduler. This cooperative scheduler maintains a list of threads, executes them in a round-robin fashion and returns when the list is empty and every detached thread has been attached back (i.e. where there is nothing more to do). Before calling `cpc_main_loop`, the programmer populates the thread list with threads created by `cpc_spawn`.

statement ::= `cpc_spawn` statement

[4] Except if the variable is declared `static`.

`cpc_spawn` The `cpc_spawn` statement creates a new attached thread that executes the argument to `cpc_spawn` and places it at the end of the queue of runnable threads. If this argument contains free variables, they are handled like free variables in CPS functions (Section 2.2.1). Execution then proceeds after the `cpc_spawn` statement (control is *not* ceded to the main CPC loop). This statement is valid in arbitrary context: in the native `main` function before the calling `cpc_main_loop`, or later in any CPS or native function, either in attached or detached mode (in which case the new thread is started attached to the main loop, while the original one continues its detached execution).

Limitation: there is a single, implicit event loop to schedule cooperative threads in CPC and every new thread is created attached to this loop. Making the loop explicit would enable the creation of several loops; along with the ability to migrate threads from one loop to another, it would enable a SEDA style (staged event-driven architecture [WCB01]) that is expected to make more efficient use of multiple processors. On the other hand, multiple loops involve complex synchronisation issues and a clumsier API; we preferred to keep CPC simple and understandable. For long running computations and blocking operations, detached threads provide an alternative to multiple event loops.

2.2.3 Detaching and reattaching

A CPC thread can be scheduled to be run by a native thread (in a thread pool); intuitively, the thread "becomes" a native thread. When this happens, we say that the CPC thread has been *detached* from the CPC scheduler. The opposite operation is known as *attaching* a detached thread back to the CPC scheduler. It is of course possible to migrate a detached thread directly from one thread pool to another. Multiple thread pools are convenient to execute tasks of various duration: with a single thread pool, long-lasting tasks could end up clogging the thread pool and prevent short-lived tasks from executing.

The primitive function `cpc_link` is used to link the current CPC thread to a scheduler; schedulers are either the main CPC loop `cpc_default_scheduler` or a thread pool of native threads. Thread pools are created with the function `cpc_threadpool_get` and destroyed by calling the function `cpc_threadpool_release`; there is also a default thread pool `cpc_default_threadpool`. A CPC thread can determine its current scheduler by calling `cpc_current_scheduler`.

```
typedef cpc_sched;
cpc_sched *cpc_default_scheduler, *cpc_default_threadpool;

cpc_sched *cpc_threadpool_get(int);
int cpc_threadpool_release(cpc_sched *);

cps cpc_sched *cpc_link(cpc_sched *pool);
cpc_sched *cpc_current_scheduler();
```

`cpc_link` The `cpc_link` primitive function attaches the current thread to the given scheduler. If the scheduler is a thread pool, the next statements are executed in a dedicated native

thread, within the thread pool. If the scheduler is the special `cpc_default_scheduler`, the current thread is scheduled by the CPC scheduler. This function returns a pointer to the previous scheduler: this is needed, for instance, to write library functions that need to run code in detached mode but wish to return in the same mode as they have been called.[5]

`cpc_threadpool_get` This function returns a pointer to a new thread pool. It takes one argument: the maximum number of threads in the pool (threads are created dynamically). This argument is capped to `MAX_THREADS`[6] if it is outside of the interval [1; `MAX_THREADS`]. This function must not be called in detached mode (it is not thread-safe).

`cpc_threadpool_release` Conversely, the function `cpc_threadpool_release` releases a given thread pool. This function does not block. It returns 0 if the pool has been successfully released, −1 otherwise (most notably when it is still running detached threads and therefore cannot be released safely). In that case, the programmer should retry later. This function can be called in any mode and context, but it will always fail if a detached thread tries to release its own thread pool. Conversely, it should always succeed when `cpc_main_loop` has returned, since it means that all threads have completed their execution.

`cpc_default_threadpool` This global variable holds a pointer to the default thread pool, initialised during the call to `cpc_main_loop`.

`cpc_current_scheduler` This function returns the current scheduler. Although this is not implemented as a CPS function for efficiency reasons, it should be treated as such: this function is valid in CPS context only, and no pointer to it should be taken.

Syntactic sugar

Since calls to `cpc_link` are often balanced pairs around a few statements, CPC provides syntactic sugar, the `cpc_linked` statement, to execute a statement linked to some scheduler and then link the current thread back to its original scheduler. Some macros are also provided to link a thread to the main loop or the default thread pool.

`cpc_linked`

statement ::= `cpc_linked` (expression) statement

The body of a `cpc_linked` statement is run linked to the scheduler given as its argument: an implicit `s = cpc_link(expression)` is executed upon entering the body, and a

[5] GHC Haskell had a similar modularity issue with the `block` and `unblock` functions to handle asynchronous exceptions. Coincidently, this was solved in GHC 7 at about the same time `cpc_link` was introduced in CPC. GHC 7 uses a similar solution, deprecating `block` and `unblock` in favor of a new `mask` function that provides a way to restore the previous state.

[6] `MAX_THREADS` is an internal constant of the CPC runtime, currently set to 20.

`cpc_link(s)` is executed when the end of the statement is reached, as well as before any `return` statement within the body.

Limitation: some constructs are forbidden in the body of a `cpc_linked` statement because they disrupt the flow of control: `goto`, `break`, `continue` and labels. This is only a syntactic limitation: these statements can of course be used in attached and detached mode, but they cannot be used to jump directly in or out of a `cpc_linked` block.[7] Therefore, the programmer has to take care of such complex cases by himself, introducing calls to `cpc_link` at the right points of the code instead of using a `cpc_linked` block.

`cpc_is_detached, cpc_detach, cpc_detached` These three macros are used to test if the current thread is detached, to detach it if this is not already the case, and to detach the execution of a block of code (similarly to `cpc_linked`).

`cpc_attach, cpc_attached` These two macros are used to attach the current thread to the default loop if not already attached, and to attach the execution of a block of code (similarly to `cpc_linked`).

2.2.4 Synchronising with condition variables

Condition variables are synchronisation primitives introduced by Hoare [Hoa74]; a condition variable is a queue of waiting threads, woken up in a first-in first-out order (FIFO). In CPC, they are used both alone, to synchronise threads, and combined with other CPC primitives to provide a means of waking up threads waiting for an event (see the descriptions of `cpc_sleep` and `cpc_io_wait` in the next sections). For efficiency reasons, their use in CPC is restricted to attached (cooperative) mode; this lifts the need for locks, which are necessarily associated with condition variables in preemptively scheduled systems to avoid race conditions.

```
typedef struct cpc_condvar cpc_condvar;

void cpc_condvar_release(cpc_condvar*);
int cpc_condvar_count(cpc_condvar*);

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

`cpc_wait` The function `cpc_wait` places the current thread on the list of threads waiting on the condition variable passed as argument to `cpc_wait`. Control is passed back to the CPC loop. This function accepts two additional optional arguments, specifying a timeout in seconds and microseconds (implemented in terms of `cpc_sleep`, see below). This function returns `CPC_CONDVAR` or `CPC_TIMEOUT`, depending on the event which woke it up. It is only valid in attached mode.

[7] In fact, earlier versions of CPC used to allow these constructs, using a smart detection of jumps in and out of the body of `cpc_linked`. But this resulted in very weird semantics, more likely to trap the careless programmer than to be useful in any way, and was removed to simplify the implementation.

`cpc_signal` The function `cpc_signal` causes the first of the threads waiting on the condition variable passed as argument to be moved to the tail of the queue of runnable threads. Execution proceeds at the instruction following the call to `cpc_signal`. This statement is valid in arbitrary context, but only in attached mode.

`cpc_signal_all` The function `cpc_signal_all` causes all of the threads waiting on the condition variable passed as argument to be moved to the tail of the queue of runnable threads. This function guarantees that the threads will be run in the order in which they were suspended. This statement is valid in arbitrary context, but only in attached mode.

Limitation: with these primitives, it is impossible to implement a function `cpc_wait_2` that would take two condition variables and return as soon as one of them has been signalled. This would indeed imply two concurrent calls `cpc_wait`, the first to return interrupting the other, but there is no way to interrupt a specific call to `cpc_wait` without also waking up every other thread waiting on the same condition variable. In practice, this is rarely an issue but there are, for instance, some threads in Hekate that are left dangling longer than necessary when a timeout expires, because of this limitation.

In hindsight, the author believes that it would have been a better choice not to use condition variables to interrupt CPS primitives. For instance, we could have introduced a separate concept of *thread identifier* with the ability to interrupt a thread blocked in a CPC primitive—similar to the `interrupt` function in Java [Ora12]—and to use condition variables only for synchronising threads. With such a feature, `cpc_wait` would be interruptible like any other CPC primitive and it would be possible to implement an interruptible wait on an arbitrary number of variables. However, interruptibility introduces its own set of issues. For instance, library functions must deal with the possibility of being interrupted, yielding more verbose and error-prone code. The `Alert` function of Modula-2+ [Bir+87] solves this dilemma by offering both interruptible and non-interruptible primitives (`Wait` and `AlertWait` for instance). Another approach is the "cancellation token" used by Windows; "a token can be passed to an asynchronous IO operation when it is started, and cancelling a token cancels all asynchronous requests associated with it" [Har+11]. All these solutions decouple the notion of semaphore or condition variable from the notion of cancellation. In any case, such a mechanism would add a number of indirections and runtime overhead which should be carefully balanced with the increased flexibility.

2.2.5 Cooperating and sleeping

```
cps void cpc_yield();
cps int cpc_sleep(int sec, int usec, cpc_condvar *cond);
int cpc_gettimeofday(stuct timeval *tv);
time_t cpc_time(time_t *t);
```

`cpc_yield` The function `cpc_yield` causes the current thread to be suspended, and placed at the end of the queue of runnable threads. Control is passed back to the CPC main loop. This statement has no effect in detached mode.

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

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

The version of `cpc_condvar` with timeouts is in fact syntactic sugar using `cpc_sleep`: `cpc_condvar(c, s, ms)` = `cpc_sleep(s, ms, c)`.

Limitation: the form with a third argument is only valid in attached mode. Note that `cpc_sleep(0)` is not equivalent to `cpc_yield()` since the former does *not* place the current thread to the end of queue of runnable threads immediately: we prefer a consistent implementation where every sleeping thread goes through the same data structure, no matter how long it sleeps, than an "optimisation" where magic values are rooted in the semantics.

`cpc_gettimeofday, cpc_time` The function `cpc_gettimeofday` is a non-blocking equivalent of the native function `gettimeofday`. In detached mode it simply calls it. In attached mode, it returns the latest time measured by the scheduler. The function `cpc_time` is equivalent to `cpc_gettimeofday` but returns only the time in seconds, like the native `time` function. These functions are only valid in CPS context, and are useful when speed is more important than time accuracy.

2.2.6 Waiting for I/O

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

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

This function causes the current thread to be suspended until either the given file descriptor is available for I/O in the given direction, or the given condition variable is signalled, whichever happens first. It returns `CPC_IO_IN`, `CPC_IO_OUT` or `CPC_CONDVAR`, depending on what happened. It may also return `-1` if some error occurred (e.g. in detached mode, when the call to `poll` fails).

Limitation: the form with a third argument is only valid in attached mode.

`cpc_signal_fd` The function `cpc_signal_fd` wakes up threads waiting on the file descriptor `fd` for an event `direction`. *Limitation:* it must only be called from the CPC main loop (but not necessarily in CPS context) and will only wake up attached threads.

2.3 The CPC standard library

The CPC standard library is a collection of some generally useful functions, written in CPC and built above the CPC primitives, distributed with CPC to ease the development of CPC programs. In this section, we describe the implementation of two synchronisation constructs provided by the CPC library: barriers in Section 2.3.1 and retriggerable timeouts in Section 2.3.2.

Since these constructs are implemented above condition variables, they inherit their main limitation: barriers and timeouts cannot be used in detached mode. This reflects the general philosophy of CPC that threads spend most of their time in attached mode, using the cooperative scheduler as an implicit lock to sequentialize access to shared resources, and allowing very fast context switches and synchronisation operations.

2.3.1 Barriers

A barrier is a synchronisation construct that allows a set of threads to be woken up at the same time. A barrier is conceptually a queue of threads (implemented with the queue of a condition variable) and a count of threads remaining to wait for.

```
struct cpc_barrier {
    cpc_condvar *condition;
    int count;
};
```

The function `cpc_barrier_get` returns a new barrier initialised to wait for `count` threads. Its implementation, not shown here, is a straightforward sequence of allocations and initialisations.

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

Figure 2.5: Waiting for a barrier

```
cps void
cpc_barrier_await(cpc_barrier *barrier)
{
    assert(barrier->count > 0);
    barrier->count--;
    if(barrier->count > 0) {
        cpc_wait(barrier->condition);
        return;
    } else {
        cpc_signal_all(barrier->condition);
        cpc_condvar_release(barrier->condition);
        free(barrier);
        cpc_yield();
        return;
    }
}
```

Note that the function cpc_barrier_await guarantees that the threads are run in the order in which they were suspended. It relies in particular on the deterministic semantics of cpc_signal_all and cpc_yield to ensure that the last awaiting thread is also the last to execute. This also allows the condition variable to be freed, hence the barrier, as soon as the waiting threads have been signalled: there is no need for a cpc_barrier_release function.

2.3.2 Timeouts

Timeouts are data structures holding two condition variables: expire, which is signalled after some given amount of time has elapsed (secs seconds and usecs milliseconds), and cancel which is used to cancel a timeout before it expires. Flags are used to indicate that the timeout has expired (_TIMEOUT_EXPIRED) or has been cancelled (_TIMEOUT_CANCELLED).

Figure 2.6: CPC timeouts

```
struct cpc_timeout {
  cpc_condvar *cancel, *expire;
  int flags, secs, usecs;
};
```

The function cpc_timeout_get creates, initialises and starts a new timeout; its implementation is very similar to cpc_barrier_get shown above and is not detailed here.

To use a timeout t, the programmer should first check that it has not already expired (t->flags & _TIMEOUT_EXPIRED) then use the condition variable t->expire to wait for the timeout or interrupt some CPC primitive. Checking the flags first (and avoiding calls to CPC primitives between this check and the use of the condition variable) is essential to avoid race conditions, because the condition variable is only signalled once when the timeout expires.

Figure 2.7: Creating a timeout

```
cps cpc_timeout *
cpc_timeout_get(int secs, int usecs)
{
  cpc_timeout *timeout;
  /* snip allocation and initialisation */
  cpc_timeout_restart(timeout);
  return timeout;
}
```

The function cpc_timeout_restart restarts a timeout if it has not already expired. It creates a new thread (1) that sleeps for the timeout duration (2) before setting the expired flag (3) and signaling the condition variable expire (4); if the variable cancel is signalled, cpc_sleep returns CPC_CONDVAR instead of CPC_TIMEOUT (5) and this thread dies without doing anything.

Figure 2.8: Waiting for a timeout

```
cps void
cpc_timeout_restart(cpc_timeout *timeout)
{
  if(timeout->flags & _TIMEOUT_EXPIRED)
```

```
    return;

  cpc_spawn {                                          /* 1 */
    if(cpc_sleep(timeout->secs, timeout->usecs,        /* 2 */
                 timeout->cancel) == CPC_TIMEOUT) {    /* 5 */
      assert(!timeout->flags);
      timeout->flags |= _TIMEOUT_EXPIRED;              /* 3 */
      cpc_signal_all(timeout->expire);                 /* 4 */
    }
  }

  cpc_signal(timeout->cancel);                         /* 6 */
  timeout->flags = 0;
  cpc_yield();         /* Enter the sleep. */          /* 7 */
}
```

Then, the function kills the *previous* timeout thread (if any) by signaling `cancel` (6). This does not cancel the thread that was just created (1) because the semantics of `cpc_spawn` implies that it will not be started before the next turn of the main loop. Hence, the call to `cpc_sleep` is deferred too (2) and cannot be interrupted by `cpc_signal` (6). The final call to `cpc_yield` (7) ensures that the timeout thread is started and enters in the `cpc_sleep` before the function returns, in order to avoid race conditions where a timeout would be cancelled before it started.

The function `cpc_timeout_cancel` cancels a timeout that is not already cancelled or expired.

Figure 2.9: Canceling a timeout

```
void
cpc_timeout_cancel(cpc_timeout *timeout)
{
  if(!timeout->flags) {
    timeout->flags |= _TIMEOUT_CANCELLED;
    cpc_signal(timeout->cancel);
  }
}
```

Finally, the function `cpc_timeout_destroy` cancels the timeout and frees the associated data structures. Note that the deallocation can be safely performed immediately, since the cancellation of the timeout thread created by `cpc_timeout_restart` does not need to access the timeout.

Figure 2.10: Destroying a timeout

```
void
cpc_timeout_destroy(cpc_timeout *timeout)
{
  cpc_timeout_cancel(timeout);
  cpc_condvar_release(timeout->cancel);
  cpc_condvar_release(timeout->expire);
  free(timeout);
}
```

Chapter 3

The CPC compilation technique

The current implementation of CPC is structured into three parts: the CPC to C translator, implemented in OCaml [Ler+10] on top of the CIL framework [Nec+02], 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.

In this chapter, we present how the CPC translator turns a CPC program in threaded style into an equivalent C program written in continuation-passing style. Therefore, we only need to focus on the transformations applied to CPS functions, ignoring completely the notions of CPC thread or CPC primitive which are handled in the runtime part of CPC. We detail each step of the CPC compilation technique and the difficulties specifically related to the C language. In later chapters, we prove the correctness of the two most important translation passes performed by CPC: lambda lifting (Chapter 4) and CPS conversion with early evaluation (Chapter 5).

3.1 Translation passes

The CPC translator is structured in a series of source-to-source transformations which turn a threaded-style CPC program into an equivalent event-driven C program. This sequence of transformations consists of the following passes:

- *Lambda lifting*: preliminary pass to remove user-defined inner functions;
- *Boxing*: address-taken local variables need to be encapsulated in heap-allocated storage to ensure the correctness of the later passes;
- *Splitting*: the flow of control of each CPS function is split into a set of mutually recursive, tail-called, inner functions;
- *Lambda lifting*: free local variables are copied from one inner function produced by splitting to the next, yielding closed inner functions;
- *CPS conversion*: at this point, the program is in *CPS-convertible form*, a form simple enough to perform a one-pass partial conversion into continuation-passing style; the resulting continuations are used at runtime by the CPC scheduler.

The converted program is then compiled by a standard C compiler and linked to the CPC scheduler to produce the final executable.

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. The problem is that these transformations copy free variables: if the original variable is modified, the copy becomes out-of-date and the program yields a different result. This is not an issue for the first lambda-lifting pass, because user-defined inner functions have a copy semantics for free variables that is compatible with lambda lifting (Section 2.2.1), but the second lambda lifting, applied to inner functions introduced by the translator during the splitting pass, and the CPS conversion pass might be incorrect.

Copying is not the only way to handle free variables. When applying lambda lifting to a call-by-value language with mutable variables, the common solution is to box variables that are actually mutated, and to store them in environments. However, this solution turns out to be less efficient:[1] adding a layer of indirection hinders cache efficiency and disables a number of compiler optimisations. Therefore, CPC strives to avoid boxing as much as possible.

One key point of the efficiency of CPC is that we do not need to box every mutated variable for lambda lifting and CPS conversion to be correct. As we show in Chapter 4, even though C is an imperative language, lambda lifting is correct without boxing for most variables, provided that the lifted functions are called in tail position. As it turns out, functions issued from the splitting pass are always called in tail position, and it is therefore correct to perform lambda lifting in CPC while keeping most variables unboxed.

Only a small number of variables, whose addresses have been retained with the "address of" operator "&", needs to be boxed: we call them extruded variables. Our experiments with Hekate show that 50 % of local variables in CPS functions need to be *lifted* (ie. duplicated by the lambda-lifting pass). Of that number, only 10 % are extruded. In other words, in the largest program written in CPC so far, our translator manages to box only 5 % of the local variables in CPS functions.

Sections 3.2 to 3.6 present each pass, explain how they interact and why they are correct. Although CPS conversion is the last pass performed by the CPC translator, we present it first (Section 5.3) because it helps understanding the purpose of the other passes, which aim at translating the program into CPS-convertible form; the other passes are then presented chronologically. Also note that the correctness of the lambda-lifting pass depends on a theorem that is shown in Chapter 4, and that the correctness of CPS conversion is detailed in Chapter 5.

3.2 CPS conversion

Conversion into Continuation-Passing Style [SW74; Plo75], 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.

[1] See Chapter 7 for a presentation of eCPC, a variant of CPC using environments, and an evaluation of its efficiency.

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(□ + 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 $f^\star$ taking an extra argument, its *continuation*. Where f would return with value v, $f^\star$ 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 does not use the native call stack: the information that would normally be in the call stack (the dynamic chain) is encoded within the continuation.

3.2.1 Important properties

This translation has three well-known interesting properties, on which we rely in the implementation of CPC: CPS conversion need not be global; continuations are abstract data structures; and continuation transformers are linear.

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 [KY01].

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 (Section 6.1.2); 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. For instance, Plotkin [Plo75, p. 147] defines the call-by-value CPS conversion process as follows, where κ is a function representing the current continuation.

$$\begin{aligned}
\overline{a} &= \lambda\kappa.\kappa a \\
\overline{x} &= \lambda\kappa.\kappa x \\
\overline{\lambda x M} &= \lambda\kappa.\kappa(\lambda x.\overline{M}) \\
\overline{MN} &= \lambda\kappa.\overline{M}(\lambda\alpha.\overline{N}(\lambda\beta.\alpha\beta\kappa))
\end{aligned}$$

where a is a constant, x a variable, and M and N two arbitrary lambda-terms.

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

$$\begin{aligned}\overline{a} &= \lambda\kappa.\kappa a\\ \overline{x} &= \lambda\kappa.\kappa x\\ \overline{\lambda x M} &= \lambda\kappa.\kappa(\lambda x.\overline{M})\\ \overline{f\,N} &= \lambda\kappa.\overline{N}(\lambda\beta.f\beta\kappa)\end{aligned}$$

where f is a constant or a variable.

However, continuations are abstract data structures and functions are only one particular concrete representation of them. The CPS conversion process actually performs only two operations on continuations: calling a continuation (**invoke** below), and prepending a function application to the body of a continuation (**push** below). In the lambda calculus, **push** and **invoke** are defined as:

$$\begin{aligned}\mathbf{invoke}(\kappa, x) &= \kappa x\\ \mathbf{push}(\alpha, \kappa) &= \lambda\beta.\alpha\beta\kappa\end{aligned}$$

and the above example is rewritten as:

$$\begin{aligned}\overline{a} &= \lambda\kappa.\,\mathbf{invoke}(\kappa, a)\\ \overline{x} &= \lambda\kappa.\,\mathbf{invoke}(\kappa, x)\\ \overline{\lambda x M} &= \lambda\kappa.\,\mathbf{invoke}(\kappa, \lambda x\overline{M})\\ \overline{f\,N} &= \lambda\kappa.\overline{N}(\mathbf{push}(f, \kappa))\end{aligned}$$

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 does not have full first-class functions (closures), CPC uses this property to implement continuations as arrays of function pointers and parameters.

Note: the attentive reader might remark that our definition of **push** depends on the simplifying hypothesis made about function calls, and wonder what it would be in the general case for the lambda calculus. It is relatively straightforward to define using delimited continuations: defunctionalising continuations in the example above yields two versions of **push**. See for instance, Felleisen et al. [Fel+88]: they introduce the operators *Send* and *AddFrame*, similar to **invoke** and **push**, that operate on *abstract continuations* (or contexts) to define a semantics of control operators.

Continuation transformers are linear CPS conversion introduces linear (or "one-shot") continuations [DL92; Dan94; BWD96]: when a CPS-converted function receives a continuation, it will use it exactly once,[2] and never duplicate or discard it. Some control operators, such as *call/cc* or *shift* and *reset*, do not respect linearity but they do not arise from the CPS conversion itself, and need to be added explicitly to the source language.

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 continuations to be reliably reclaimed without the need for costly devices such as reference counting.

3.2.2 CPS-convertible form

CPS conversion is not defined for every C function; instead, we restrict ourselves to a subset of functions, which we call the *CPS-convertible* subset. As we shall see in Section 3.4, every C function can be converted to an equivalent function in CPS-convertible form.

The CPS-convertible form restricts the calls to CPS functions to make it straightforward to capture their continuation. In CPS-convertible form, a call to a CPS function `f` is either in tail position, or followed by a tail call to another CPS function whose parameters are *non-shared* variables, that cannot be modified by `f`. This restriction about shared variables is ensured by the *boxing* pass detailed in Section 3.3.

Definition 3.2.1 (Extruded and shared variables). *Extruded variables* are local variables (or function parameters) the address of which has been retained using the "address of" operator "`&`" (including arrays in particular).

We say *shared variables* for variables that are either extruded or global variables. □

Thus, the set of shared variables includes every variable that might be modified by several functions called in the same dynamic chain.

Definition 3.2.2 (CPS-convertible form in C). A function is in *CPS-convertible form* if every call to a CPS function that it contains matches one of the following patterns, where both `f` and `g` are CPS functions, e_1, ..., e_n are any C expressions and `x`, y_1, ..., y_n are non-shared variables:

```
                          return f(e1, ..., en);          (3.1)
x = f(e1, ..., en); return g(x, y1, ..., yn);            (3.2)
       f(e1, ..., en); return g(y1, ..., yn);             (3.3)
                        f(e1, ..., en); return;           (3.4)
      f(e1, ..., en); g(y1, ..., yn); return;             (3.5)
x = f(e1, ..., en); g(x, y1, ..., yn); return;            (3.6)
```

[2] This is assuming a full CPS conversion, or terminating functions; with a partial conversion, direct-style non-terminating functions might block the computation without ever using the continuation.

Note how each of these cases ends with a `return` statement, in order to ensure that CPS functions are only called in tail position (or immediately followed by a CPS tail call). The definition of CPS-convertible form might seem convoluted at first, but in fact Eqs. (3.3) to (3.6) are only necessary to handle a syntactic peculiarity of the C language about functions returning `void`. In the rest of this document, we ignore the cases where `f` or `g` return `void` and focus on the first two cases.

3.2.3 Early evaluation of non-shared variables

To understand why the definition of CPS-convertible form requires non-shared variables for the parameters of g, consider what happens when converting a CPS-convertible function. We use a partial CPS conversion, as explained above, focused on tail positions. In a functional language, we would define the CPS conversion as follows, where $f^\star$ is the CPS-converted form of `f` and `k` the current continuation:

```
return a;                          →  return k(a);
return f(a1, ..., an);             →  return f*(a1, ..., an, k);
x = f(a1, ..., an);                →  k' = λx. g*(x, y1, ..., yn, k);
  return g(x, y1, ..., yn);           return f*(a1, ..., an, k');
```

Note the use of the lambda-abstraction λ`x. g*(x, y1, ..., yn, k)` to represent the continuation of the call to $f^\star$ in the last case. In a call-by-value language, this continuation can be described as: *"get the return value of* $f^\star$*, bind it to* `x`*, evaluate the variables* y_1 *to* y_n *and call* $g^\star$ *with these parameters."*

Representing this continuation in C raises the problem of evaluating the values of y_1 to y_n after the call to $f^\star$ has completed: these variables are local to the enclosing CPS-converted function and, as such, allocated in a stack frame which will be discarded when it returns. To keep them available until the evaluation of the continuation, we would need to store them in an environment and garbage-collect them at some point. We want to avoid this approach as much as possible for performance reasons since it implies an extra layer of indirection and extra bookkeeping.

We use instead a technique that we call *early evaluation* of variables. It is based on the following property: if we can ensure that the variables y_i cannot be modified by the function `f`, then it is correct to commute their evaluation with the call to `f`. Because the CPC translator produces code where these variables are not shared, thanks to the boxing pass (Section 3.3), it is indeed guaranteed that they cannot be modified by a call to another function in the same dynamic chain. In CPC, the CPS conversion therefore evaluates these variables when the continuation is built, before calling `f`, and stores their values directly in the continuation.

We finally define the CPS conversion pass, using early evaluation and the fact mentioned above that continuations are abstract data structures mutated by two operators: `push`, which adds a function and its parameters to the continuation, and `invoke` which executes a continuation.

Definition 3.2.3 (CPS conversion in C). The *CPS conversion* translates the tail positions of every CPS-convertible term as follows, where `f`* is the CPS-converted form of `f` and `k` is the current continuation:

```
return a;                          →  return invoke(k, a);

return f(a1, ..., an);             →  push (k, f*, a1, ..., an);
                                      return invoke(k);

x = f(a1, ..., an);                   push (k, g*, y1, ..., yn);
  return g(x, y1, ..., yn);        →    push (k, f*, a1, ..., an);
                                        return invoke(k);
```

□

A proof of correctness of this conversion, including a proof that early evaluation is correct in the absence of shared variables, is detailed in Chapter 5.

3.2.4 Implementation details

From an implementation standpoint, the continuation `k` is a stack of function pointers and parameters, and `push` adds elements to this stack. The function `invoke` calls the first function of the stack, with the rest of the stack as its parameters. The form `invoke(k, a)` also takes care of passing the value `a` to the continuation `k`; it simply pushes `a` at the top of the continuation before calling its first function.

The current implementation of `push` grows continuations by a multiplicative factor when the array is too small to contain the pushed parameters, 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 that this case is rare enough not to justify complicating the implementation.

The function `push` does not align parameters 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 micro-benchmark with deliberately misaligned parameters. We have reconsidered this trade-off when porting CPC to MIPS, an architecture with no hardware support for unaligned accesses, and added a compilation option to align continuation frames. However this option is disabled by default, at least until we gather more experimental data to evaluate its efficiency.

There is one difference between Definition 3.2.3 and the implementation. In a language with proper tail calls, each function would simply invoke the continuation directly; but, because most C compilers do not ensure proper tail calls (even for cases, like invoking a continuation, where they could be performed safely), doing that leads to unbounded growth of the native call stack. Therefore, the tail call `return invoke(k)` cannot be used; we work around this issue by using a "trampolining" technique [GFW99]. Instead of calling `invoke`, each CPS function returns its continuation: `return k`. The main event loop iteratively receives

these continuations and invokes them until a CPC primitive returns NULL, which yields to the next CPC thread.

In the following sections, we show how the boxing, splitting and lambda-lifting passes translate any CPC program into CPS-convertible form.

3.3 Boxing

Boxing is a common, straightforward technique to encapsulate mutable variables. It is necessary to ensure the correctness of the CPS conversion and lambda-lifting passes (Sections 3.2 and 3.5). However, boxing induces an expensive indirection to access boxed variables. In order to keep this cost at an acceptably low level, we box only a subset of all variables—namely extruded variables, whose address is retained with the "address of" operator "&" (see Definition 3.2.1 on page 71).

Example Consider the following function.

Figure 3.1: Extruded variables

```
cps void f(int x) {
  int y = 0;
  int *p1 = &x, *p2 = &y;
  /* ... */
  return;
}
```

The local variables x and y are extruded, because their address is stored in the pointers p1 and p2. The boxing pass allocates them on the heap on entry to the function f and frees them before f returns. For instance, in the next program every occurrence of x has been replaced by a pointer indirection *px, and &x by the pointer px.

Figure 3.2: Boxing extruded variables

```
cps void f(int x) {
  int *px = malloc(sizeof(int));
  int *py = malloc(sizeof(int));
  *px = x;                  /* Initialise px */
  *py = 0;
  int *p1 = px, *p2 = py;
  /* ... with x and y replaced accordingly */
  free(px); free(py);
  return;
}
```

The extruded variables x and y are no longer used (except to initialise px). Instead, px and py are used; note that these variables, contrary to x and y, are not extruded: they hold the address of other variables, but their own address is not retained. After the boxing pass, there are no more extruded variables used in CPS functions.

Cost analysis The efficiency of CPC relies in great part on avoiding boxing as much as possible. Performance-wise, we expect boxing only extruded variables to be far less expensive than boxing every lifted variable (ie. variables that are duplicated by the lambda-lifting pass). Indeed, in a typical C program, 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. This result is obtained when compiling Hekate with the current CPC implementation. They take into account the fact that the CPC translator tries to be smart about which variables should be boxed or lifted: for instance, if the address of a variable is retained with the "address of" operator "&" but never used, this variable is not considered as extruded. Using a naive implementation, however, does not change the proportion of boxed variables: earlier versions of CPC lacking optimisations[3] used to box 29 variables out of 323 lifted parameters. In both cases, CPC boxes about 10 % of the lifted variables, which appears to be an acceptable overhead.

Interaction with other passes The boxing pass yields a program without the "address of" operator "&"; extruded variables are allocated on the heap, and only pointers to them are copied by the lambda-lifting and CPS-conversion passes rather than extruded variables themselves. One may wonder, however, 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 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 language does not ensure the elimination of tail-recursive calls anyway, as the stack frame should similarly be preserved in case of extruded variables, and C programmers are used not to rely on it.

3.4 Splitting

To transform a CPC program into CPS-convertible form, the CPC translator needs to ensure that every call to a CPS function is either in tail position or followed by a tail call to another CPS function. In the original CPC code, calls to CPS functions might, by chance, respect this

[3] In particular without the *percolating* pass described in Section 3.6.

property but, more often than not, they are followed by some direct-style (non-CPS) code. The role of the CPC translator is therefore to replace this direct-style chunk of code following a CPS call by a call to a CPS function which encapsulates this chunk. We call this pass *splitting* because it splits each original CPS function into many, mutually recursive, CPS functions in CPS-convertible form.

To reach CPS-convertible form, the splitting pass must introduce tail calls after every existing CPS call. This is done in two steps: we first introduce a `goto` after every existing CPS call (Section 3.4.1), then we translate these `goto` statements into tail calls (Section 3.4.2).

The first step consists in introducing a `goto` after every CPS call. Of course, to keep the semantics of the program unchanged, this inserted `goto` must jump to the statement following the tail call in the control-flow graph: it makes the control flow explicit, and prepares the second step which translates these `goto` statements into tail calls. In most cases, the control flow falls through linearly and inserting `goto` statements is trivial; as we shall see in Section 3.4.1, more care is required when the control flow crosses loops and labelled blocks. This step produces code which resembles CPS-convertible form, except that it uses `goto` instead of tail calls.

The second step is based on the observation that tail calls are equivalent to jumps. We convert each labelled block into an inner function, and each `goto` statement into a tail call to the associated function, yielding a program in CPS-convertible form.

We detail these two steps in the rest of this section.

3.4.1 Explicit flow of control

When a CPS call is not in tail position, or followed by a tail CPS call, the CPC translator adds a `goto` statement after it, to jump explicitly to the next statement in the control-flow graph. These inserted `goto` are to be converted into tail CPS calls in the next step.

In most cases, the control flow falls through linearly, and making it explicit is trivial.

Figure 3.3: CPS call with implicit control flow

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

We only need to add a `goto` after the first statement, and a label before the second one.

Figure 3.4: CPS call with explicit control flow

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

However, loops and conditional jumps require more care.

Figure 3.5: Loop with implicit control flow

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

This loop is converted into a labelled conditional block, with `goto` statements to continue looping and breaking outside.

```
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();
```

Figure 3.6: Loop with explicit control flow

More generally, when the flow of control after a CPS call goes outside of a loop (`for`, `while` or `do ...while`) or a `switch` statement, the CPC translator simplifies these constructs into `if` and `goto` statements, adding the necessary labels on the fly.

Although adding trivial `goto`, and making loops explicit, brings the code in a shape close to CPS-convertible form, we need to add some more `goto` statements for the second step to correctly encapsulate chunks of code into CPS functions. Consider for instance the following piece of code.

```
if(rc < 0) {
  cpc_yield(); rc = 0;
}
printf("rc␣=␣%d\n", rc);
return rc;
```

Figure 3.7: CPS call within a conditional block

With the rules described above a single `goto` would be added, after the CPS call to `cpc_yield`.

```
if(rc < 0) {
  cpc_yield(); goto l;
  l: { rc = 0; }
}
printf("rc␣=␣%d\n", rc);
return rc;
```

Figure 3.8: Labelled block with implicit control flow

This is not enough because the block labelled by `l` will be converted to a CPS function in the second step. But if we encapsulate only the `{rc = 0;}` part, we will miss the call to `printf` when `rc < 0`. Therefore, to ensure a correct encapsulation in the second step, we also need to make the flow of control explicit when exiting a labelled block. Thus, Fig. 3.7 is in fact translated into the following piece of code.

```
if(rc < 0) {
  cpc_yield(); goto l;
  l: { rc = 0; goto done; }
}
done:
  printf("rc␣=␣%d\n", rc);
  return rc;
```

Figure 3.9: Labelled block with explicit control flow

After this step, every CPS call is either in tail position or followed by a `goto statement`, and every labelled block exits with either a `return` or a `goto` statement.

3.4.2 Eliminating gotos

It is a well-known fact in the compiler folklore that a tail call is equivalent to a goto. It is perhaps less known that a goto is equivalent to a tail call [SJS76; Wij66]: 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 Fig. 3.4, the label `l` yields a function `l()`.

Figure 3.10: CPS call in CPS-convertible form

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

We see that the first line is now in CPS-convertible form. Note again that we use `return` to mark tail calls explicitly in the C code.

Applying the same conversion to loops yields mutually recursive functions. For instance, the `while` loop in Fig. 3.6 is converted into `while_label()`, `l()` and `break_label()`.

Figure 3.11: Loop translated into CPS-convertible form

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

When a label, like `while_label` above, is reachable not only through a `goto`, 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.

Splitting may introduce free variables; for instance, in Fig. 3.11, `rc` is free in the function `l`. In this intermediate C-like code, inner functions are considered just like any other C block when it comes to the scope and lifespan of variables: each variable lives as long as the block or function which defines it, and it can be read and modified by any instruction within this block, including inner functions. There is in particular no copy of free variables in inner functions; variables are shared within their block.

This *sharing* semantics is different from the *copying* semantics used when defining inner CPS functions in the source of CPC program (Section 2.2.1). The reason for this discrepancy is that *sharing* is the expected semantics for splitting to be correct; on the other hand, *copying* is the actual semantics implemented by the lambda-lifting algorithm used by CPC (Section 3.5). In Chapter 4, we show that lambda lifting is correct in the case of CPC; but this only means that the *copying* and *sharing* semantics are indistinguishable in the very case of functions introduced during the splitting pass. Since we cannot enforce the appropriate invariants on user-defined inner functions, they are compiled with the *copying* semantics inherited from lambda lifting.

In Fig. 3.11, the lifespan of the variable `rc` is the `if` block, including the call to the function `l` which reads the free variable allocated in its enclosing function `while_label`.

Figure 3.9 shows the importance of having explicit flow of control at the end of labelled blocks. After `goto` elimination, it is indeed in CPS-convertible form.

Figure 3.12: Conditional block in CPS-convertible form

```
if(rc < 0) {
  cpc_yield(); return l();
  cps int l() { rc = 0; return done(); }
}
cps int done() {
  printf("rc = %d\n", rc);
  return rc;
}
return done();
```

Note the tail call to done at the end of `l` to ensure that the `printf` is executed when `rc < 0`, and again the call on the last line to execute it otherwise.

3.4.3 Implementation details

The actual implementation of the CPC translator implements splitting on an as-needed basis: a given subtree of the abstract syntax tree (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 CPUs and the GCC compiler are optimised for human-written code.

To perform splitting, the CPC translator iterates over the AST, checking whether the CPS functions are in CPS-convertible form and interleaving the two steps described above to reach CPS-convertible form incrementally. On each pass, when the translator finds a CPS call it dispatches on the statement following it:

- in the case of a `return`, the fragment is already CPS-convertible and the translator continues;
- in the case of a `goto`, it is converted into a tail call, with the corresponding label turned into an inner function (Section 3.4.2), and the translator starts another pass;
- for any other statement, a `goto` is added to make the flow of control explicit, converting enclosing loops if necessary (Section 3.4.1). The translator then starts another pass and will eventually convert the introduced `goto` into a tail call.

At the end of the splitting pass, the translated program is in CPS-convertible form. However, it is not quite ready for CPS conversion because we introduced inner functions, which makes it invalid C. In particular, these functions may contain free variables. The next pass, lambda lifting, takes care of these free variables to get a valid C program in CPS-convertible form, suitable for the CPS conversion pass described in Section 3.2.

3.5 Lambda lifting

Lambda lifting [Joh85] is a standard technique to eliminate free variables in functional languages. It proceeds in two phases [DS04]. 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 top level.

3.5.1 An example of lambda lifting

Coming back to Fig. 3.12 on the preceding page, we add an enclosing function `f` to define `rc` and make the fragment self-contained.

Figure 3.13: CPS-convertible code with inner functions

```
cps int f(int rc) {
  if(rc < 0) {
    cpc_yield(); return l();
    cps int l() { rc = 0; return done(); }
  }
  cps int done() {
    printf("rc␣=␣%d\n", rc);
    return rc;
  }
  return done();
}
```

The function `f` contains two inner functions, `l` and `done`. The local variable `rc` is used as a free variable in both of these functions.

Parameter lifting consists in adding the free variable as a parameter of every inner function.

Figure 3.14: Parameter lifting and closed functions

```
cps int f(int rc) {
  if(rc < 0) {
    cpc_yield(); return l(rc);
    cps int l(int rc) { rc = 0; return done(rc); }
  }
  cps int done(int rc) {
    printf("rc␣=␣%d\n", rc);
    return rc;
  }
  return done(rc);
}
```

Note that `rc` is now a parameter of `l` and `done`, and has been added accordingly whenever these functions where called. There are, now, three copies of `rc`; alpha conversion makes this more obvious.

Figure 3.15: Parameter lifting and alpha conversion

```
cps int f(int rc1) {
  if(rc1 < 0) {
    cpc_yield(); return l(rc1);
    cps int l(int rc2) { rc2 = 0; return done(rc2); }
  }
  cps int done(int rc3) {
    printf("rc␣=␣%d\n", rc3);
    return rc3;
  }
  return done(rc1);
}
```

Once the parameter lifting step has been performed, there are no free variables any longer and the block floating step extracts these inner, closed functions at top-level.

Figure 3.16: Block floating

```
cps int f(int rc1) {
  if(rc1 < 0) {
    cpc_yield(); return l(rc1);
  }
  return done(rc1);
}
cps int l(int rc2) {
  rc2 = 0;
  return done(rc2);
}
cps int done(int rc3) {
  printf("rc = %d\n", rc3);
  return rc3;
}
```

It is easy to see that applying boxing, splitting and lambda lifting always yields CPS-convertible programs:

- every call to a CPS function is either a tail call (not affected by the transformations) or followed by a tail CPS call (introduced in the splitting pass),
- the parameters of this second CPS call are local variables, since they are introduced by the lambda-lifting pass,
- these parameters are not shared because they are neither global (local variables) nor extruded (the boxing pass ensures that there are no more extruded variables in the program).

3.5.2 Lambda lifting in imperative languages

There is one major issue with applying lambda lifting to C extended with inner functions: this transformation is in general not correct in a call-by-value language with mutable variables.

Consider what would happen if splitting were to yield the following code.

Figure 3.17:

CPS-convertible code with non-tail calls

```
cps void f(int rc) {
  cps void set() { rc = 0; return; }
  cps void done() {
    printf("rc = %d\n", rc);
    return;
  }
  set(); done(); return;
}
```

This code, which is in CPS-convertible form, prints out `rc = 0` whatever the original value of `rc` was: the call to `set` sets `rc` to 0, because of the sharing semantics, and the call to done displays it.

Once lambda-lifted, the result changes.

Figure 3.18: Lifted CPS-convertible code with non-tail calls

```
cps void f(int rc1) {
  set(rc1); done(rc1); return;
}
cps void set(int rc2) {
  rc2 = 0;
  return;
}
cps void done(int rc3) {
  printf("rc␣=␣%d\n", rc3);
  return;
}
```

The code now displays the original value of `rc` passed to `f` rather than 0. The reason why lambda lifting is incorrect in that case is because `set` and `done` work on two separate copies of `rc`, `rc2` and `rc3`, whereas in the original code there was only one instance of `rc` shared by all inner functions.

This issue is triggered by the fact that the function `set` is not called in tail position. This non-tail call allows us to observe the incorrect value of `rc3` after `set` has modified the copy `rc2` and returned. If `set` were called in tail position instead, the fact that it operates on a copy of `rc1` would remain unnoticed.

3.5.3 Lambda lifting and tail calls

In fact, although lambda lifting is not correct in general in a call-by-value language with mutable variables, it becomes correct once restricted to functions called in tail position in a language without extruded variables. More precisely, in the absence of extruded variables, it is safe to lift a parameter provided every inner function where this parameter is lifted is only called in tail position. We show this result in Chapter 4 (Theorem 4.1.9 on page 92).

Inner functions in CPC are the result of `goto` elimination during the splitting step. As a result, they are always called in tail position. Moreover, as explained in Section 3.3, the boxing pass ensures that extruded variables have been eliminated at this point of the transformation. Hence, lambda lifting is correct in the case of CPC.

3.5.4 Implementation details

To lift as few variables as possible, lambda lifting is implemented incrementally. There are two ways to implement lambda lifting: either iterate on inner functions, closing them by lifting all of their free variables, or iterate on variables, lifting each of them in every function where it is free. The CPC translator uses the former: it looks for free variables to be lifted in a CPS function and adds them as parameters at call points; this creates new free variables, and the translator repeats the procedure until it reaches a fixed point.

This implementation might be further optimised with a liveness analysis, which would in particular avoid lifting uninitialised parameters. The current translator performs a very limited analysis: only variables which are used (hence alive) in a single function are not lifted (see Section 3.6 for more details).

3.5.5 Experimental results

A straightforward technique to use lambda lifting in an imperative language is to box every mutated variable, in order to duplicate pointers to these variables instead of the variables themselves. To quantify the amount of boxing avoided by our technique of lambda lifting tail-called functions, we used a modified version of CPC which blindly boxes every lifted parameter and measure 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 functions.[4] 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 translator boxes only 5 % of the variables in CPS functions. In other words, 90 % of the lifted variables in Hekate are safely left unboxed, keeping the overhead associated with boxing to a reasonable level.

3.6 Optimisation passes

In addition to the passes described above, the CPC translator performs two simple optimisation passes. They are not necessary for the correctness of the translation but they reduce the overhead generated by the other translation passes.

3.6.1 Percolating

Between the splitting and lambda-lifting passes, the CPC translator performs a *percolating* pass. Percolating consists in identifying local variables that are used in only one inner function and binding them to this function. It is a very simplified form of liveness analysis, which does not make any inter-procedural analysis but helps to reduce the number of lifted variables during the lambda-lifting pass.

Consider for instance the following function before lambda lifting.

```
cps void f() {
    int tmp;
    cps void g() {
        cps void h() {
            ...
            tmp = 2;
            ...
        }
        h();
    }
    cps void l() {
        g();
```

Figure 3.19: Local variable used in a single inner function

[4] These numbers leave out direct-style functions, which do not need to be converted, and around 200 unused temporary variables introduced by a single pathological macro-expansion in the *curl* library.

```
    }
    ...
    l();
}
```

The variable `tmp` is used only in the inner function `h`. The percolating pass produces the following code, which is invariant by lambda lifting.

Figure 3.20: Percolating of local variable

```
cps void f() {
    cps void g() {
        cps void h() {
            int tmp;
            ...
            tmp = 2;
            ...
        }
        h();
    }
    cps void l() {
        g();
    }
    ...
    l();
}
```

Without the percolating pass, the variable `tmp` would be lifted in every inner function during the lambda-lifting pass.

Figure 3.21: Lambda lifting without percolating

```
cps void f() {
    int tmp;
    cps void g(int tmp) {
        cps void h(int tmp) {
            ...
            tmp = 2;
            ...
        }
        h(tmp);
    }
    cps void l(int tmp) {
        g(tmp);
    }
    ...
    l(tmp);
}
```

This would introduce several useless copies of the (yet uninitialised) variable `tmp`. Remember that inner functions arise in particular from the translation of loops during splitting; in these cases, percolating is even more important since the intermediate functions such as `l` might be called recursively a very high number of times, each call adding a useless copy of `tmp`.

However crude it might seem, the percolating pass has a dramatic impact on the number of lifted variables. For instance in Hekate, it reduces their number by more than 61 %, from

323 down to 125. The CPC translator introduces many temporary variables when it simplifies the flow of control. These variables are typically used only in two consecutive statements, which are not split apart during the splitting pass.

Although a complete liveness analysis would be preferable, since it could reduce even further the number of lifted variables, the percolating is therefore a key optimisation pass of the CPC translator.

3.6.2 Inlining

The inlining optimisation pass is performed between the lambda-lifting and CPS-conversion passes. It consists in identifying CPS functions that perform only a single call to another CPS function, with exactly the same parameters, and replacing them by a call to the latter function wherever they are used. It is deliberately kept very simple because more opportunities for more complex optimisations, such as permutation of function parameters, do not seem to happen in practice.

For instance, in the following example, `f` merely performs a call to `g` and `g` a call to `h`.

```
cps int f(int x, int y) { return g(x,y); }
cps int g(int a, int b) { return h(a,b); }
cps int h(int p, int q) { return (p+q)/2; }

f(1, 2) + g(3, 4);
```

Figure 3.22: Trivial CPS functions

Inlining replaces the calls accordingly.

```
cps int f(int x, int y) { return h(x,y); }
cps int g(int a, int b) { return h(a,b); }
cps int h(int p, int q) { return (p+q)/2; }

h(1, 2) + h(3, 4);
```

Figure 3.23: Inlining trivial functions

Note that inlining is iterated until it reaches a fix-point: in this example, `f` is replaced by `g` and then further replaced by `h`.

In Hekate, 18 trivial CPS functions are inlined out of 113 inner functions. This proportion (16 %) of trivial functions is due to unoptimised splitting operations: translating loops to goto statements and then turning them into tail calls creates a number of pathological cases. These are much easier to optimise in a separate inlining step than trying to refine the set of complex splitting rules.

Chapter 4

Lambda lifting in an imperative language

In this chapter, we prove that lambda lifting is correct in an imperative, call-by-value language for functions that are called in tail position. In order to do this proof, we do not reason directly on CPC programs, 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. Therefore, we define a simple language with restricted values, expressions and terms, that captures the features that we are most interested in (Section 4.1).

The big-step reduction rules for this language (Section 4.1.1) use a simplified memory model without pointers and enforce that local variables are not accessed outside of their scope, as ensured by our boxing pass. This is necessary because lambda lifting is not correct in general in the presence of extruded variables.

It turns out that the "naive" reduction rules defined in Section 4.1.1 do not provide invariants strong enough to prove this correctness theorem by induction, mostly because we represent memory with a store that is not invariant with respect to lambda lifting. Therefore, in Section 4.2, we define an equivalent, "optimised" set of reduction rules which enforces more regular stores and closures.

The proof of correctness is then carried out in Section 4.4 using these optimised rules. We first define the invariants needed for the proof and formulate a strengthened version of the correctness theorem (Theorem 4.4.6, Section 4.4.1). A comprehensive overview of the proof is then given in Section 4.4.2. The proof is fully detailed in Section 4.4.5, with the help of a number of lemmas to keep the main proof shorter (Sections 4.4.3 and 4.4.4).

The main limitation of this proof is that Theorems 4.1.9 and 4.4.6 are implications, not equivalences: we do not prove that lambda lifting preserves non-reducibility. For instance, this proof does not ensure that infinite loops remain infinite once lambda-lifted.

4.1 Definitions

In this section, we define the terms (Definition 4.1.1), the reduction rules (Section 4.1.1) and the lambda-lifting transformation itself (Section 4.1.2) for our small imperative language. With these preliminary definitions, we are then able to characterise *liftable parameters* (Definition 4.1.8) and state the main correctness theorem (Theorem 4.1.9, Section 4.1.3).

Definition 4.1.1 (Values, expression and terms). Values are either boolean and integer constants or **1**, a special value for functions returning `void`.

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

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

$$e ::= \quad v \mid x \mid \ldots$$

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

$$\begin{aligned} T ::= \quad & e \mid x := T \mid \mathbf{if}\ T\ \mathbf{then}\ T\ \mathbf{else}\ T \mid T\,;\,T \\ & \mid \mathbf{letrec}\ f(x_1, \ldots, x_n) = T\ \mathbf{in}\ T \mid f(T, \ldots, T) \end{aligned}$$

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—and that is, in fact, how the splitting pass translates them. Since lambda lifting happens after the splitting pass, our language needs to include inner functions (although they are not part of the C language), but it can safely exclude `goto` statements.

One important simplification of this language compared to C is the lack of pointers. However, remember that we are lifting only local, stack-allocated variables, and that these variables cannot be accessed outside of their scope, as ensured by our boxing pass. Since we get rid of the "address of" operator `&`, pointers remaining in CPC code after boxing always point to the heap, never to the stack. Adding a heap and pointers to our language would only make it larger without changing the proof of correctness.

4.1.1 Naive reduction rules

Environments and stores Handling inner functions requires explicit closures in the reduction rules. We need environments, written ρ, to bind variables to locations, and a store, written s, to bind locations to values.

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

Definition 4.1.2 (Modification of a partial function). The modification (or extension) f' of a partial function f, written $f' = f\{x \mapsto y\}$, is defined as follows:

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

Definition 4.1.3 (Variable and function environments). Variable environments are defined inductively by

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

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

Function environments associate function names to closures:

$$\mathcal{F}\colon \{f, \dots\} \to \{[\lambda x_1, \dots, x_n.T,\ \rho,\ \mathcal{F}]\}$$

where the closure $[\lambda x_1, \dots, x_n.T,\ \rho,\ \mathcal{F}]$ captures the function f, the parameters of which are the variables x_i and the body of which is the term T, in the context of a variable environment ρ and a function environment $\mathcal{F}$.

Note that although we have a notion of locations, which correspond roughly to memory addresses in C, there is no way to copy, change or otherwise manipulate a location directly in the syntax of our language. This is deliberate, since adding this possibility would make lambda lifting incorrect: it translates the fact, ensured by the boxing pass in the CPC translator, that there are no extruded variables in the lifted terms.

Reduction rules We use classical big-step reduction rules for our language (Fig. 4.1). The judgment $\langle T,\ s\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v,\ s'\rangle$ means that, in the context of a variable environment ρ, a function environment $\mathcal{F}$, and a store s, the term T reduces to a value v with a store s'. In the (call) rule, we need to introduce *fresh* locations for the parameters of the called function. This means that we must choose locations that are not already in use, in particular in the environments ρ' and $\mathcal{F}$. To express this choice, we define two ancillary functions, Envs and Locs, to extract the environments and locations contained in the closures of a given function environment $\mathcal{F}$.

Definition 4.1.4 (Set of environments, set of locations). The set of environments Envs is the least solution for *envs* to the following equation:

$$\mathit{envs}(\mathcal{F}) = \bigcup_{[\lambda x_1,\dots,x_n.M,\ \rho,\ \mathcal{F}'] \in \mathrm{Im}(\mathcal{F})} \{\rho\} \cup \mathit{envs}(\mathcal{F}').$$

The set of locations Locs is defined as:

$$\mathrm{Locs}(\mathcal{F}) = \bigcup \{\mathrm{Im}(\rho) \mid \rho \in \mathrm{Envs}(\mathcal{F})\}.$$

A location l is said to *appear* in $\mathcal{F}$ if and only if l belongs to $\mathrm{Locs}(\mathcal{F})$.

Figure 4.1: "Naive" reduction rules

$$\text{(val)}\ \frac{}{\langle v,\ s\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v,\ s\rangle} \qquad \text{(var)}\ \frac{\rho\ x = l \in \operatorname{dom} s}{\langle x,\ s\rangle \xrightarrow[\mathcal{F}]{\rho} \langle s\ l,\ s\rangle}$$

$$\text{(assign)}\ \frac{\langle a,\ s\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v,\ s'\rangle \qquad \rho\ x = l \in \operatorname{dom} s'}{\langle x \coloneqq a,\ s\rangle \xrightarrow[\mathcal{F}]{\rho} \langle \mathbf{1},\ s'\{l \mapsto v\}\rangle}$$

$$\text{(seq)}\ \frac{\langle a,\ s\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v,\ s'\rangle \qquad \langle b,\ s'\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v',\ s''\rangle}{\langle a\ ;\ b,\ s\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v',\ s''\rangle}$$

$$\text{(if-t.)}\ \frac{\langle a,\ s\rangle \xrightarrow[\mathcal{F}]{\rho} \langle \mathbf{true},\ s'\rangle \qquad \langle b,\ s'\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v,\ s''\rangle}{\langle \mathbf{if}\ a\ \mathbf{then}\ b\ \mathbf{else}\ c,\ s\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v,\ s''\rangle}$$

$$\text{(if-f.)}\ \frac{\langle a,\ s\rangle \xrightarrow[\mathcal{F}]{\rho} \langle \mathbf{false},\ s'\rangle \qquad \langle c,\ s'\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v,\ s''\rangle}{\langle \mathbf{if}\ a\ \mathbf{then}\ b\ \mathbf{else}\ c,\ s\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v,\ s''\rangle}$$

$$\text{(letrec)}\ \frac{\begin{array}{c}\langle b,\ s\rangle \xrightarrow[\mathcal{F}']{\rho} \langle v,\ s'\rangle \\ \mathcal{F}' = \mathcal{F}\{f \mapsto [\lambda x_1, \ldots, x_n.a,\ \rho,\ \mathcal{F}]\}\end{array}}{\langle \mathbf{letrec}\ f(x_1, \ldots, x_n) = a\ \mathbf{in}\ b,\ s\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v,\ s'\rangle}$$

$$\text{(call)}\ \frac{\begin{array}{c}\mathcal{F}\,f = \left[\lambda x_1, \ldots, x_n.b,\ \rho',\ \mathcal{F}'\right] \qquad \rho'' = (x_1, l_1) \cdot \ldots \cdot (x_n, l_n) \qquad l_i \text{ fresh and distinct} \\ \forall i, \langle a_i,\ s_i\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v_i,\ s_{i+1}\rangle \qquad \langle b,\ s_{n+1}\{l_i \mapsto v_i\}\rangle \xrightarrow[\mathcal{F}'\{f \mapsto \mathcal{F}\,f\}]{\rho''\cdot\rho'} \langle v,\ s'\rangle\end{array}}{\langle f(a_1, \ldots, a_n),\ s_1\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v,\ s'\rangle}$$

It turns out that our reduction rules do not build circular function environments, so the previous equation for *envs* can in fact be read directly as a definition for Envs. We then define fresh locations.

Definition 4.1.5 (Fresh location). In the (call) rule, a location is *fresh* when:

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

Note that the second condition implies in particular that l does not appear in either the function environment $\mathcal{F}$ or the variable environment ρ'.

4.1.2 Lambda lifting

As explained in Section 3.5, lambda lifting [Joh85] is a standard technique to eliminate free variables; it proceeds in two passes [DS04]. 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 top level.

Parameter lifting can be performed iteratively. It consists in finding a free variable and adding it as a parameter of every inner function where it appears free, in order to transform it into a bound variable. This step is repeated until every variable is bound in every function. Once all functions are closed, they no longer interact with each other any longer and block floating can be performed safely: inner functions are syntactically extracted and copied at top level.

We will focus only on parameter lifting here, since the correctness of block floating is trivial [DS04]. Note that although the whole transformation is called lambda lifting, we do not focus on a single function and try to lift all of its free variables. On the contrary, we define the lifting of a single parameter x in every function where it is free. We will show the correctness of this elementary step, which implies the correctness of lambda lifting as a whole by iterating it until every variable is bound.

There are various smart lambda-lifting algorithms in the literature that strive to minimize the number of lifted variables [DS04; MS07]. Such is not our concern here: to keep the proof as simple as possible, parameters are lifted in every function where they might potentially be free.

Definition 4.1.6 (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, to avoid name clashes with lifted parameters.
Then the *lifted form* $(M)_*$ of the term M with respect to x is defined inductively as follows:

$$(\mathbf{1})_* = \mathbf{1} \qquad (n)_* = n$$

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

4.1.3 Correctness condition

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

We first define tail positions as usual [Cli98]:

Definition 4.1.7 (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" and "$M\ ;\ N$" and "**letrec** $f(x_1,\ldots,x_n) = M$ **in** N".

A parameter x defined in a function g is liftable if every inner function in g is called exclusively in tail position.

Definition 4.1.8 (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.

Our main theorem states that performing parameter lifting on a liftable parameter preserves the result of the reduction:

Theorem 4.1.9 (Correctness of lambda lifting). *If x is a liftable parameter in M, then*

$$\exists t, \langle M,\ \varepsilon\rangle \xrightarrow[\varepsilon]{\varepsilon} \langle v,\ t\rangle \text{ implies } \exists t', \langle (M)_*,\ \varepsilon\rangle \xrightarrow[\varepsilon]{\varepsilon} \langle v,\ t'\rangle .$$

Note that the resulting store t' changes because lambda lifting introduces new variables, hence new locations in the store, and changes the values associated with lifted variables; Section 4.4 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.

4.2 Optimised reduction rules

The naive reduction rules (Section 4.1.1) are not well-suited to prove the correctness of lambda lifting. Indeed, the proof is by induction and requires a number of invariants on the structure of stores and environments. Rather than having a dozen lemmas to ensure these invariants during the proof of correctness, we translate them into constraints in the reduction rules.

To this end, we introduce two optimisations—minimal stores (Section 4.2.1) and compact closures (Section 4.2.2)—which lead to the definition of an optimised set of reduction rules (Fig. 4.2, Section 4.2.3). The equivalence between optimised and naive reduction rules is shown in Section 4.3.

4.2.1 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 for locally, at the rule level.

Consider for instance the simplest possible case of lambda lifting:

$$\textbf{letrec } g(x) = (\textbf{letrec } h() = x \textbf{ in } h()) \textbf{ in } g(\mathbf{1}) \qquad \text{(original)}$$
$$\textbf{letrec } g(x) = (\textbf{letrec } h(y) = y \textbf{ in } h(x)) \textbf{ in } g(\mathbf{1}) \qquad \text{(lifted)}$$

At the end of the reduction, the store for the original term is $\{l_x \mapsto \mathbf{1}\}$ whereas the store for the lifted term is $\{l_x \mapsto \mathbf{1}; l_y \mapsto \mathbf{1}\}$. More complex terms would yield even larger stores, with many out-of-date copies of lifted variables.

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 x from the store once we are certain that it will never be used again, i.e. as soon as the term in tail position in the function which defines x has been evaluated. This mechanism is analogous to the deallocation of a stack frame when a function returns. We call *minimal* a store that does not contain such unreachable locations.

To track the variables whose location can be safely reclaimed after the reduction of some term M, we introduce *split environments*. Split environments are written $\rho_T ; \rho$, where ρ_T is called the *tail environment* and ρ the non-tail one; only the variables belonging to the tail environment may be safely reclaimed. The reduction rules build environments so that a variable x belongs to ρ_T if and only if the term M is in tail position in the current function f and x is a parameter of f. In that case, it is safe to discard the locations associated with all of the parameters of f, including x, after M has been reduced because we are sure that the evaluation of f is completed (and there are no first-class functions in the language to keep references to variables beyond their scope of definition).

We also define a *cleaning* operator, $\cdot \setminus \cdot$, to remove a set of variables from the store.

Definition 4.2.1 (Store cleaning). The store s cleaned with respect to the variables in ρ, written $s \setminus \rho$, is defined as $s \setminus \rho = s|_{\mathrm{dom}(s) \setminus \mathrm{Im}(\rho)}$.

4.2.2 Compact closures

Another source of complexity with the naive reduction rules is the inclusion of useless variables in closures. It is safe to remove from the variable environment contained in closures the variables that are also parameters of the function: when the function is called, and the environment restored, these variables will be masked 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 these captured values will never be used since calling the function adds fresh parameters with the same names. We introduce *compact closures* in the optimised reduction rules to avoid dealing with this hiding mechanism in the proof of lambda lifting.

A compact closure is a closure that does not capture any variable which would be masked when the closure is called because of function parameters having the same name.

Definition 4.2.2 (Compact closure and environment). A closure $[\lambda x_1, \ldots, x_n.M,\ \rho,\ \mathcal{F}]$ is *compact* if $\forall i, x_i \notin \mathrm{dom}(\rho)$ 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 4.2.3 (Canonical compact environment). The *canonical compact environment* $\mathcal{F}_*$ is the unique environment with the same domain as $\mathcal{F}$ such that

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

4.2.3 Optimised reduction rules

Combining both optimisations yields the *optimised* reduction rules (Fig. 4.2), used in Section 4.4 for the proof of lambda lifting. The judgment $\langle T,\ s\rangle \xRightarrow[\mathcal{F}]{\rho_T;\rho} \langle v,\ s'\rangle$ means that, in the context of a tail variable environment ρ_T, a non-tail variable environment ρ, a function environment $\mathcal{F}$, and a store s, the term T reduces to a value v with a store s'. Consider for instance the rule (seq).

$$(\textsc{seq})\ \frac{\langle a,\ s\rangle \xRightarrow[\mathcal{F}]{\varepsilon;\rho_T\cdot\rho} \langle v,\ s'\rangle \qquad \langle b,\ s'\rangle \xRightarrow[\mathcal{F}]{\rho_T;\rho} \langle v',\ s''\rangle}{\langle a\ ;\ b,\ s\rangle \xRightarrow[\mathcal{F}]{\rho_T;\rho} \langle v',\ s''\rangle}$$

The variable environment is split into the tail environment, ρ_T, and the non-tail one, ρ. This means that $a\ ;\ b$ is in tail position in a function whose parameters are the variables of ρ_T. When we reduce the left part of the sequence, a, we track the fact that it is not in tail position

in this function by moving ρ_T to the non-tail environment: $\langle a,\ s\rangle \overset{\varepsilon\,;\,\rho_T\cdot\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s'\rangle$. On the other hand, when we reduce b, we are in the tail of the term and the environment stays split.

In Section 4.2.1, we have introduced split environments in order to ensure minimal stores. Stores are kept minimal in the rules corresponding to tail positions, ie. the leaves of the reduction tree: (val), (var) and (assign). In these three rules, variables that appear in the tail environment are cleaned from the resulting store: $s \setminus \rho_T$.

Finally, the (letrec) and (call) rules are modified to introduce compact closures and split environments, respectively. Compact closures are built in the (letrec) rule by removing the parameters of f from the captured environment ρ'. In the (call) rule, environments are split in a tail part, which contains local variables of the called function, and a non-tail part, which contains captured variables; only the former are cleaned when the tail instruction of the function is reduced.

Theorem 4.2.4 (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 naive reduction:*

$$\langle M,\ \varepsilon\rangle \overset{\varepsilon\,;\,\varepsilon}{\underset{\varepsilon}{\Longrightarrow}} \langle v,\ \varepsilon\rangle \quad \textit{iff} \quad \exists s, \langle M,\ \varepsilon\rangle \overset{\varepsilon}{\underset{\varepsilon}{\Rightarrow}} \langle v,\ s\rangle$$

We prove this theorem in Section 4.3.

4.3 Equivalence of optimised and naive reduction rules

This section is devoted to the proof of equivalence between the optimised naive reduction rules (Theorem 4.2.4).

To clarify the proof, we introduce intermediate reduction rules (Fig. 4.3), with only one of the two optimisations: minimal stores, but not compact closures. The judgment $\langle T,\ s\rangle \overset{\rho_T\,;\,\rho}{\underset{\mathcal{F}}{\Rightarrow}} \langle v,\ s'\rangle$ means that, in the context of a tail variable environment ρ_T, a non-tail variable environment ρ, a function environment $\mathcal{F}$, and a store s, the term T reduces to a value v with a store s'.

The proof then consists in proving that optimised and intermediate rules are equivalent (Lemmas 4.3.2 and 4.3.3, Section 4.3.1), then that naive and intermediate rules are equivalent (Lemmas 4.3.8 and 4.3.9, Section 4.3.2).

$$\text{Naive rules} \underset{\text{Lemma 4.3.8}}{\overset{\text{Lemma 4.3.9}}{\rightleftarrows}} \text{Intermediate rules} \underset{\text{Lemma 4.3.3}}{\overset{\text{Lemma 4.3.2}}{\rightleftarrows}} \text{Optimised rules}$$

Figure 4.2: Optimised reduction rules

$$\text{(val)}\ \frac{}{\langle v,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s \setminus \rho_T\rangle} \qquad \text{(var)}\ \frac{(\rho_T \cdot \rho)\ x = l \in \operatorname{dom} s}{\langle x,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle s\ l,\ s \setminus \rho_T\rangle}$$

$$\text{(assign)}\ \frac{\langle a,\ s\rangle \overset{\varepsilon;\rho_T\cdot\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s'\rangle \qquad (\rho_T \cdot \rho)\ x = l \in \operatorname{dom} s'}{\langle x \coloneqq a,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle \mathbf{1},\ s'\{l \mapsto v\} \setminus \rho_T\rangle}$$

$$\text{(seq)}\ \frac{\langle a,\ s\rangle \overset{\varepsilon;\rho_T\cdot\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s'\rangle \qquad \langle b,\ s'\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v',\ s''\rangle}{\langle a\ ;\ b,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v',\ s''\rangle}$$

$$\text{(if-t.)}\ \frac{\langle a,\ s\rangle \overset{\varepsilon;\rho_T\cdot\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle \mathbf{true},\ s'\rangle \qquad \langle b,\ s'\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s''\rangle}{\langle \mathbf{if}\ a\ \mathbf{then}\ b\ \mathbf{else}\ c,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s''\rangle}$$

$$\text{(if-f.)}\ \frac{\langle a,\ s\rangle \overset{\varepsilon;\rho_T\cdot\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle \mathbf{false},\ s'\rangle \qquad \langle c,\ s'\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s''\rangle}{\langle \mathbf{if}\ a\ \mathbf{then}\ b\ \mathbf{else}\ c,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s''\rangle}$$

$$\text{(letrec)}\ \frac{\begin{array}{c}\langle b,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}'}{\Longrightarrow}} \langle v,\ s'\rangle \\ \rho' = \rho_T \cdot \rho|_{\operatorname{dom}(\rho_T\cdot\rho)\setminus\{x_1,\ldots,x_n\}} \qquad \mathcal{F}' = \mathcal{F}\{f \mapsto [\lambda x_1, \ldots, x_n.a,\ \rho',\ \mathcal{F}]\}\end{array}}{\langle \mathbf{letrec}\ f(x_1, \ldots, x_n) = a\ \mathbf{in}\ b,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s'\rangle}$$

$$\text{(call)}\ \frac{\begin{array}{c}\mathcal{F}\ f = [\lambda x_1, \ldots, x_n.b,\ \rho',\ \mathcal{F}'] \qquad \rho'' = (x_1, l_1) \cdot \ldots \cdot (x_n, l_n) \qquad l_i \text{ fresh and distinct} \\ \forall i, \langle a_i,\ s_i\rangle \overset{\varepsilon;\rho_T\cdot\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v_i,\ s_{i+1}\rangle \qquad \langle b,\ s_{n+1}\{l_i \mapsto v_i\}\rangle \overset{\rho'';\rho'}{\underset{\mathcal{F}'\{f\mapsto\mathcal{F}\ f\}}{\Longrightarrow}} \langle v,\ s'\rangle\end{array}}{\langle f(a_1, \ldots, a_n),\ s_1\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s' \setminus \rho_T\rangle}$$

Figure 4.3: Intermediate reduction rules

$$(\textsc{val})\ \frac{}{\langle v,\ s\rangle \overset{\rho_T;\rho}{=\!=\!\underset{\mathcal{F}}{=\!=}\!\Rightarrow} \langle v,\ s \setminus \rho_T\rangle} \qquad (\textsc{var})\ \frac{(\rho_T \cdot \rho)\ x = l \in \mathrm{dom}\ s}{\langle x,\ s\rangle \overset{\rho_T;\rho}{=\!=\!\underset{\mathcal{F}}{=\!=}\!\Rightarrow} \langle s\ l,\ s \setminus \rho_T\rangle}$$

$$(\textsc{assign})\ \frac{\langle a,\ s\rangle \overset{\varepsilon;\rho_T\cdot\rho}{=\!=\!\underset{\mathcal{F}}{=\!=}\!\Rightarrow} \langle v,\ s'\rangle \qquad (\rho_T \cdot \rho)\ x = l \in \mathrm{dom}\ s'}{\langle x \coloneqq a,\ s\rangle \overset{\rho_T;\rho}{=\!=\!\underset{\mathcal{F}}{=\!=}\!\Rightarrow} \langle \mathbf{1},\ s'\{l \mapsto v\} \setminus \rho_T\rangle}$$

$$(\textsc{seq})\ \frac{\langle a,\ s\rangle \overset{\varepsilon;\rho_T\cdot\rho}{=\!=\!\underset{\mathcal{F}}{=\!=}\!\Rightarrow} \langle v,\ s'\rangle \qquad \langle b,\ s'\rangle \overset{\rho_T;\rho}{=\!=\!\underset{\mathcal{F}}{=\!=}\!\Rightarrow} \langle v',\ s''\rangle}{\langle a\ ;\ b,\ s\rangle \overset{\rho_T;\rho}{=\!=\!\underset{\mathcal{F}}{=\!=}\!\Rightarrow} \langle v',\ s''\rangle}$$

$$(\textsc{if-t.})\ \frac{\langle a,\ s\rangle \overset{\varepsilon;\rho_T\cdot\rho}{=\!=\!\underset{\mathcal{F}}{=\!=}\!\Rightarrow} \langle \mathbf{true},\ s'\rangle \qquad \langle b,\ s'\rangle \overset{\rho_T;\rho}{=\!=\!\underset{\mathcal{F}}{=\!=}\!\Rightarrow} \langle v,\ s''\rangle}{\langle \mathbf{if}\ a\ \mathbf{then}\ b\ \mathbf{else}\ c,\ s\rangle \overset{\rho_T;\rho}{=\!=\!\underset{\mathcal{F}}{=\!=}\!\Rightarrow} \langle v,\ s''\rangle}$$

$$(\textsc{if-f.})\ \frac{\langle a,\ s\rangle \overset{\varepsilon;\rho_T\cdot\rho}{=\!=\!\underset{\mathcal{F}}{=\!=}\!\Rightarrow} \langle \mathbf{false},\ s'\rangle \qquad \langle c,\ s'\rangle \overset{\rho_T;\rho}{=\!=\!\underset{\mathcal{F}}{=\!=}\!\Rightarrow} \langle v,\ s''\rangle}{\langle \mathbf{if}\ a\ \mathbf{then}\ b\ \mathbf{else}\ c,\ s\rangle \overset{\rho_T;\rho}{=\!=\!\underset{\mathcal{F}}{=\!=}\!\Rightarrow} \langle v,\ s''\rangle}$$

$$(\textsc{letrec})\ \frac{\begin{array}{c}\langle b,\ s\rangle \overset{\rho_T;\rho}{=\!=\!\underset{\mathcal{F}'}{=\!=}\!\Rightarrow} \langle v,\ s'\rangle \\ \rho' = \rho_T \cdot \rho \qquad \mathcal{F}' = \mathcal{F}\{f \mapsto [\lambda x_1, \ldots, x_n.a,\ \rho,\ \mathcal{F}]\}\end{array}}{\langle \mathbf{letrec}\ f(x_1, \ldots, x_n) = a\ \mathbf{in}\ b,\ s\rangle \overset{\rho_T;\rho}{=\!=\!\underset{\mathcal{F}}{=\!=}\!\Rightarrow} \langle v,\ s'\rangle}$$

$$(\textsc{call})\ \frac{\begin{array}{c}\mathcal{F}\ f = \left[\lambda x_1, \ldots, x_n.b,\ \rho',\ \mathcal{F}'\right] \qquad \rho'' = (x_1, l_1) \cdot \ldots \cdot (x_n, l_n) \qquad l_i \text{ fresh and distinct} \\ \forall i, \langle a_i,\ s_i\rangle \overset{\varepsilon;\rho_T\cdot\rho}{=\!=\!\underset{\mathcal{F}}{=\!=}\!\Rightarrow} \langle v_i,\ s_{i+1}\rangle \qquad \langle b,\ s_{n+1}\{l_i \mapsto v_i\}\rangle \overset{\rho'';\rho'}{=\!=\!\underset{\mathcal{F}'\{f \mapsto \mathcal{F}\ f\}}{=\!=}\!\Rightarrow} \langle v,\ s'\rangle\end{array}}{\langle f(a_1, \ldots, a_n),\ s_1\rangle \overset{\rho_T;\rho}{=\!=\!\underset{\mathcal{F}}{=\!=}\!\Rightarrow} \langle v,\ s' \setminus \rho_T\rangle}$$

4.3.1 Optimised and intermediate reduction rules equivalence

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

$$\text{Intermediate rules} \underset{\text{Lemma 4.3.3}}{\overset{\text{Lemma 4.3.2}}{\rightleftharpoons}} \text{Optimised rules}$$

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 environments when we know for sure that they will be masked by freshly instantiated parameters, since their associated locations will be unreachable in the closure. The following lemma formalises this intuition.

Lemma 4.3.1 (Masked variables elimination).

$$\forall l, l', \langle M,\ s\rangle \overset{\rho_T\cdot(x,l)\,;\,\rho}{\underset{\mathcal{F}}{=\!\Rightarrow}} \langle v,\ s'\rangle \quad \textit{iff} \quad \langle M,\ s\rangle \overset{\rho_T\cdot(x,l)\,;\,(x,l')\cdot\rho}{\underset{\mathcal{F}}{=\!\Rightarrow}} \langle v,\ s'\rangle$$

$$\forall l, l', \langle M,\ s\rangle \overset{\rho_T\cdot(x,l)\,;\,\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s'\rangle \quad \textit{iff} \quad \langle M,\ s\rangle \overset{\rho_T\cdot(x,l)\,;\,(x,l')\cdot\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s'\rangle$$

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

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

$$\langle a,\ s\rangle \overset{\varepsilon\,;\,\rho_T\cdot(x,l)\cdot(x,l')\cdot\rho}{\underset{\mathcal{F}}{=\!\Rightarrow}} \langle v,\ s'\rangle \quad \text{iff} \quad \langle a,\ s\rangle \overset{\varepsilon\,;\,\rho_T\cdot(x,l)\cdot\rho}{\underset{\mathcal{F}}{=\!\Rightarrow}} \langle v,\ s'\rangle$$

Moreover, by the induction hypotheses,

$$\langle b,\ s'\rangle \overset{\rho_T\cdot(x,l)\,;\,(x,l')\cdot\rho}{\underset{\mathcal{F}}{=\!\Rightarrow}} \langle v',\ s''\rangle \quad \text{iff} \quad \langle b,\ s'\rangle \overset{\rho_T\cdot(x,l)\,;\,\rho}{\underset{\mathcal{F}}{=\!\Rightarrow}} \langle v',\ s''\rangle$$

Hence,

$$\langle a\ ;\ b,\ s\rangle \overset{\rho_T\cdot(x,l)\,;\,(x,l')\cdot\rho}{\underset{\mathcal{F}}{=\!\Rightarrow}} \langle v',\ s''\rangle \quad \text{iff} \quad \langle a\ ;\ b,\ s\rangle \overset{\rho_T\cdot(x,l)\,;\,\rho}{\underset{\mathcal{F}}{=\!\Rightarrow}} \langle v',\ s''\rangle$$

The other cases are similar.

(val)

$$\langle v,\ s\rangle \overset{\rho_T\cdot(x,l)\,;\,\rho}{\underset{\mathcal{F}}{=\!\Rightarrow}} \langle v,\ s \setminus \rho_T \cdot (x, l)\rangle \quad \text{iff} \quad \langle v,\ s\rangle \overset{\rho_T\cdot(x,l)\,;\,(x,l')\cdot\rho}{\underset{\mathcal{F}}{=\!\Rightarrow}} \langle v,\ s \setminus \rho_T \cdot (x, l)\rangle$$

(var) $\rho_T \cdot (x,l) \cdot \rho = \rho_T \cdot (x,l) \cdot (x,l') \cdot \rho$ so, with $l'' = \rho_T \cdot (x,l) \cdot \rho\ y$,

$$\langle y,\ s\rangle \overset{\rho_T\cdot(x,l)\,;\,\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle s\ l'',\ s \setminus \rho_T \cdot (x,l)\rangle \quad \text{iff} \quad \langle y,\ s\rangle \overset{\rho_T\cdot(x,l)\,;\,(x,l')\cdot\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle s\ l'',\ s \setminus \rho_T \cdot (x,l)\rangle$$

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

$$\langle a,\ s\rangle \overset{\varepsilon\,;\,\rho_T\cdot(x,l)\cdot(x,l')\cdot\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s'\rangle \quad \text{iff} \quad \langle a,\ s\rangle \overset{\varepsilon\,;\,\rho_T\cdot(x,l)\cdot\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s'\rangle$$

Hence, with $l'' = \rho_T \cdot (x,l) \cdot \rho\ y$,

$$\langle y \coloneqq a,\ s\rangle \overset{\rho_T\cdot(x,l)\,;\,\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle \mathbf{1},\ s'\{l'' \mapsto v\} \setminus \rho_T \cdot (x,l)\rangle \quad \text{iff}$$

$$\langle y \coloneqq a,\ s\rangle \overset{\rho_T\cdot(x,l)\,;\,(x,l')\cdot\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle \mathbf{1},\ s'\{l'' \mapsto v\} \setminus \rho_T \cdot (x,l)\rangle$$

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

(letrec) $\rho_T \cdot (x,l) \cdot \rho = \rho_T \cdot (x,l) \cdot (x,l') \cdot \rho = \rho'$. Moreover, by the induction hypotheses,

$$\langle b,\ s\rangle \overset{\rho_T\cdot(x,l)\,;\,(x,l')\cdot\rho}{\underset{\mathcal{F}'}{\Longrightarrow}} \langle v,\ s'\rangle \quad \text{iff} \quad \langle b,\ s\rangle \overset{\rho_T\cdot(x,l)\,;\,\rho}{\underset{\mathcal{F}'}{\Longrightarrow}} \langle v,\ s'\rangle$$

Hence,

$$\langle \textbf{letrec}\ f(x_1,\ldots,x_n) = a\ \textbf{in}\ b,\ s\rangle \overset{\rho_T\cdot(x,l)\,;\,(x,l')\cdot\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s'\rangle \quad \text{iff}$$

$$\langle \textbf{letrec}\ f(x_1,\ldots,x_n) = a\ \textbf{in}\ b,\ s\rangle \overset{\rho_T\cdot(x,l)\,;\,\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s'\rangle$$

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

$$\forall i, \langle a_i,\ s_i\rangle \overset{\varepsilon\,;\,\rho_T\cdot(x,l)\cdot(x,l')\cdot\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v_i,\ s_{i+1}\rangle \quad \text{iff} \quad \langle a_i,\ s_i\rangle \overset{\varepsilon\,;\,\rho_T\cdot(x,l)\cdot\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v_i,\ s_{i+1}\rangle$$

Hence,

$$\langle f(a_1,\ldots,a_n),\ s_1\rangle \overset{\rho_T\cdot(x,l)\,;\,(x,l')\cdot\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s' \setminus \rho_T \cdot (x,l)\rangle \quad \text{iff}$$

$$\langle f(a_1,\ldots,a_n),\ s_1\rangle \overset{\rho_T\cdot(x,l)\,;\,\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \langle v,\ s' \setminus \rho_T \cdot (x,l)\rangle\,.$$

□

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

Lemma 4.3.2 (Intermediate implies optimised).

$$\textit{If } \langle M,\ s\rangle \overset{\rho_T\,;\rho}{\underset{\mathcal{F}}{=\!\Rightarrow}} \langle v,\ s'\rangle \textit{ then } \langle M,\ s\rangle \overset{\rho_T\,;\rho}{\underset{\mathcal{F}_*}{\Longrightarrow}} \langle v,\ s'\rangle .$$

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

(letrec) By the induction hypotheses,

$$\langle b,\ s\rangle \overset{\rho_T\,;\rho}{\underset{\mathcal{F'}_*}{\Longrightarrow}} \langle v,\ s'\rangle$$

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:

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

hence:

$$\langle \textbf{letrec } f(x_1, \ldots, x_n) = a \textbf{ in } b,\ s\rangle \overset{\rho_T\,;\rho}{\underset{\mathcal{F}_*}{\Longrightarrow}} \langle v,\ s'\rangle$$

(call) By the induction hypotheses,

$$\forall i, \langle a_i,\ s_i\rangle \overset{\varepsilon\,;\rho_T\cdot\rho}{\underset{\mathcal{F}_*}{\Longrightarrow}} \langle v_i,\ s_{i+1}\rangle$$

and

$$\langle b,\ s_{n+1}\{l_i \mapsto v_i\}\rangle \overset{\rho''\,;\rho'}{\underset{(\mathcal{F'}\{f\mapsto\mathcal{F}f\})_*}{\Longrightarrow}} \langle v,\ s'\rangle$$

Lemma 4.3.1 allows us to remove masked variables, which leads to

$$\langle b,\ s_{n+1}\{l_i \mapsto v_i\}\rangle \overset{\rho''\,;\rho'_{|\mathrm{dom}(\rho')\setminus\{x_1,\ldots,x_n\}}}{\underset{(\mathcal{F'}\{f\mapsto\mathcal{F}f\})_*}{\Longrightarrow}} \langle v,\ s'\rangle$$

Besides,

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

and

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

Hence

$$\langle f(a_1, \ldots, a_n),\ s_1\rangle \overset{\rho_T\,;\rho}{\underset{\mathcal{F}_*}{\Longrightarrow}} \langle v,\ s' \setminus \rho_T\rangle .$$

(val) $\langle v,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}_*}{\Longrightarrow}} \langle v,\ s \setminus \rho_T\rangle$

(var) $\langle x,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}_*}{\Longrightarrow}} \langle s\ l,\ s \setminus \rho_T\rangle$

(assign) By the induction hypotheses, $\langle a,\ s\rangle \overset{\varepsilon;\rho_T\cdot\rho}{\underset{\mathcal{F}_*}{\Longrightarrow}} \langle v,\ s'\rangle$. Hence,

$$\langle x \coloneqq a,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}_*}{\Longrightarrow}} \left\langle \mathbf{1},\ s'\{l \mapsto v\} \setminus \rho_T\right\rangle$$

(seq) By the induction hypotheses,

$$\langle a,\ s\rangle \overset{\varepsilon;\rho_T\cdot\rho}{\underset{\mathcal{F}_*}{\Longrightarrow}} \left\langle v,\ s'\right\rangle \qquad\qquad \left\langle b,\ s'\right\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}_*}{\Longrightarrow}} \left\langle v',\ s''\right\rangle$$

Hence,

$$\langle a\ ;\ b,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}_*}{\Longrightarrow}} \left\langle v',\ s''\right\rangle$$

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

Lemma 4.3.3 (Optimised implies intermediate).

$$\textit{If } \langle M,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Longrightarrow}} \left\langle v,\ s'\right\rangle \textit{ then } \forall \mathcal{G} \textit{ such that } \mathcal{G}_* = \mathcal{F}, \langle M,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{G}}{\Rightarrow}} \left\langle v,\ s'\right\rangle.$$

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

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

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

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

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

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

By the induction hypotheses,

$$\langle b,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{G}'}{\Rightarrow}} \left\langle v,\ s'\right\rangle$$

Hence,

$$\langle \textbf{letrec}\ f(x_1, \ldots, x_n) = a\ \textbf{in}\ b,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{G}}{\Rightarrow}} \left\langle v,\ s'\right\rangle$$

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

$$\forall i, \langle a_i,\ s_i\rangle \overset{\varepsilon\,;\rho_T\cdot\rho}{\underset{\mathcal{G}}{=\!=\!\Rightarrow}} \langle v_i,\ s_{i+1}\rangle$$

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

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

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,

$$\langle b,\ s_{n+1}\{l_i \mapsto v_i\}\rangle \overset{\rho''\,;\rho'}{\underset{\mathcal{G}'\{f\mapsto\mathcal{G}\ f\}}{=\!=\!\Rightarrow}} \left\langle v,\ s'\right\rangle$$

Lemma 4.3.1 leads to

$$\langle b,\ s_{n+1}\{l_i \mapsto v_i\}\rangle \overset{\rho''\,;(x_i,l_i)\cdot\ldots\cdot(x_j,l_j)\rho'}{\underset{\mathcal{G}'\{f\mapsto\mathcal{G}\ f\}}{=\!=\!\Rightarrow}} \left\langle v,\ s'\right\rangle$$

Hence,

$$\left\langle f(a_1, \ldots, a_n),\ s_1\right\rangle \overset{\rho_T\,;\rho}{\underset{\mathcal{G}}{=\!=\!\Rightarrow}} \left\langle v,\ s' \setminus \rho_T\right\rangle.$$

(val) $\forall\mathcal{G}$ such as ${\mathcal{G}}_* = \mathcal{F}, \langle v,\ s\rangle \overset{\rho_T\,;\rho}{\underset{\mathcal{G}}{=\!=\!\Rightarrow}} \langle v,\ s'\rangle$

(var) $\forall\mathcal{G}$ such as ${\mathcal{G}}_* = \mathcal{F}, \langle x,\ s\rangle \overset{\rho_T\,;\rho}{\underset{\mathcal{G}}{=\!=\!\Rightarrow}} \langle s\ l,\ s \setminus \rho_T\rangle$

(assign) Let $\mathcal{G}$ such as ${\mathcal{G}}_* = \mathcal{F}$. By the induction hypotheses, $\langle a,\ s\rangle \overset{\varepsilon\,;\rho_T\cdot\rho}{\underset{\mathcal{G}}{=\!=\!\Rightarrow}} \langle v,\ s'\rangle$. Hence,

$$\langle x \coloneqq a,\ s\rangle \overset{\rho_T\,;\rho}{\underset{\mathcal{G}}{=\!=\!\Rightarrow}} \left\langle \mathbf{1},\ s'\{l \mapsto v\} \setminus \rho_T\right\rangle$$

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

$$\langle a,\ s\rangle \overset{\varepsilon\,;\rho_T\cdot\rho}{\underset{\mathcal{G}}{=\!=\!\Rightarrow}} \left\langle v,\ s'\right\rangle \qquad\qquad \left\langle b,\ s'\right\rangle \overset{\rho_T\,;\rho}{\underset{\mathcal{G}}{=\!=\!\Rightarrow}} \left\langle v',\ s''\right\rangle$$

Hence

$$\langle a\ ;\ b,\ s\rangle \overset{\rho_T\,;\rho}{\underset{\mathcal{G}}{=\!=\!\Rightarrow}} \left\langle v',\ s''\right\rangle$$

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

4.3.2 Intermediate and naive reduction rules equivalence

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

$$\text{Naive rules} \underset{\text{Lemma 4.3.8}}{\overset{\text{Lemma 4.3.9}}{\rightleftharpoons}} \text{Intermediate rules}$$

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 4.3.4 (Store extension).

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

Property 4.3.5. *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'). □

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 4.3.6 (Alpha conversion). *If $\langle M,\ s\rangle \overset{\rho_T\,;\,\rho}{\underset{\mathcal{F}}{\Rightarrow}} \langle v,\ s'\rangle$ then, for all l, for all l' appearing neither in s nor in $\mathcal{F}$ nor in $\rho \cdot \rho_T$,*

$$\left\langle M,\ s[l'/l]\right\rangle \overset{\rho_T[l'/l]\,;\,\rho[l'/l]}{\underset{\mathcal{F}[l'/l]}{\Rightarrow}} \left\langle v,\ s'[l'/l]\right\rangle.$$

Moreover, both derivations have the same height.

Notation: in $F[l'/l]$, alpha conversion is applied inductively to every closure of every element of $\mathcal{F}$.

Proof. By induction on the height of the derivation. For the (call) case, we 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'.

We start with two preliminary elementary remarks. First, provided l' appears neither in ρ or ρ_T, nor in s,

$$(s \setminus \rho)[l'/l] = (s[l'/l]) \setminus (\rho[l'/l])$$

and

$$(\rho_T \cdot \rho)[l'/l] = \rho_T[l'/l] \cdot \rho[l'/l].$$

Moreover, if $\langle M,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{=\!\Rightarrow}} \langle v,\ s'\rangle$, then $\mathrm{dom}(s') = \mathrm{dom}(s) \setminus \rho_T$ (straightforward by induction). This leads to: $\rho_T = \varepsilon \Rightarrow \mathrm{dom}(s') = \mathrm{dom}(s)$.

By induction on the height of the derivation, because the induction hypothesis must be applied twice in the case of the (call) rule.

(call) $\forall i, \mathrm{dom}(s_i) = \mathrm{dom}(s_{i+1})$. Thus, $\forall i, l' \notin \mathrm{dom}(s_i)$. This leads, by the induction hypotheses, to

$$\forall i, \left\langle a_i,\ s_i[l'/l]\right\rangle \overset{\varepsilon;(\rho_T\cdot\rho)[l'/l]}{\underset{\mathcal{F}}{=\!\Rightarrow}} \left\langle v_i,\ s_{i+1}[l'/l]\right\rangle \mathcal{F}[l'/l]$$

Moreover , $\mathcal{F}'$ is part of $\mathcal{F}$. As a result, since l' does not appear in $\mathcal{F}$, it does not appear in $\mathcal{F}'$, nor in $\mathcal{F}'\{f \mapsto \mathcal{F}\, f\}$. It does not appear in ρ' either (since ρ' is part of $\mathcal{F}'$). On the other hand, there might be some j such that $l_j = l'$, so l' might appear in ρ''. In that case, we apply the induction hypotheses a first time to rename l_j in some $l'_j \neq l'$. One can chose l'_j such that it does not appear in s_{n+1}, $\mathcal{F}'\{f \mapsto \mathcal{F}\, f\}$ nor in $\rho'' \cdot \rho$. As a result, l'_j is fresh. Since l_j is fresh too, and does not appear in $\mathrm{dom}(s')$ (because of our preliminary remarks), this leads to a mere substitution in ρ'':

$$\left\langle b,\ s_{n+1}\{l_i[l'_j/l_j] \mapsto v_i\}\right\rangle \overset{\rho''[l'_j/l_j];\rho'}{\underset{\mathcal{F}'\{f\mapsto\mathcal{F}\,f\}}{=\!\Rightarrow}} \left\langle v,\ s'\right\rangle$$

Once this (potentially) disturbing l_j has been renamed (we ignore it in the rest of the proof), we apply the induction hypotheses a second time to rename l to l':

$$\left\langle b,\ (s_{n+1}\{l_i \mapsto v_i\})[l'/l]\right\rangle \overset{\rho''[l'/l];\rho'[l'/l]}{\underset{\mathcal{F}'\{f\mapsto\mathcal{F}\,f\}}{=\!\Rightarrow}} \left\langle v,\ s'[l'/l]\right\rangle$$

Now, $(s_{n+1}\{l_i \mapsto v_i\})[l'/l] = s_{n+1}[l'/l]\{l_i \mapsto v_i\}$. Moreover,

$$\mathcal{F}[l'/l]\ f = \left[\lambda x_1, \ldots, x_n.b,\ \rho'[l'/l],\ \mathcal{F}'[l'/l]\right]$$

and

$$(\mathcal{F}'\{f \mapsto \mathcal{F}\, f\})[l'/l] = \mathcal{F}'[l'/l]\{f \mapsto \mathcal{F}[l'/l]\ f\}$$

Finally, $\rho''[l'/l] = \rho''$. Hence:

$$\left\langle f(a_1, \ldots, a_n),\ s_1[l'/l]\right\rangle \overset{\rho_T[l'/l];\rho[l'/l]}{\underset{\mathcal{F}[l'/l]}{=\!\Rightarrow}} \left\langle v,\ s'[l'/l] \setminus \rho_T[l'/l]\right\rangle.$$

(val) $\langle v,\ s[l'/l]\rangle \xrightarrow[\mathcal{F}[l'/l]]{\rho_T[l'/l];\rho[l'/l]} \langle v,\ s[l'/l] \setminus \rho_T[l'/l]\rangle$

(var) $s[l'/l](\rho_T[l'/l] \cdot \rho[l'/l]\ x) = s((\rho_T \cdot \rho)\ x) = v$ implies

$$\left\langle x,\ s[l'/l]\right\rangle \xrightarrow[\mathcal{F}[l'/l]]{\rho_T[l'/l];\rho[l'/l]} \left\langle v,\ s[l'/l] \setminus \rho_T[l'/l]\right\rangle$$

(assign) By the induction hypotheses,

$$\left\langle a,\ s[l'/l]\right\rangle \xrightarrow[\mathcal{F}[l'/l]]{\varepsilon;(\rho_T\cdot\rho)[l'/l]} \left\langle v,\ s'[l'/l]\right\rangle$$

Let $s'' = s'\{(\rho_T \cdot \rho)\ x \mapsto v\}$. Then,

$$s'[l'/l]\{(\rho_T \cdot \rho)[l'/l]\ x \mapsto v\} = s''[l'/l]$$

Hence

$$\left\langle x \coloneqq a,\ s[l'/l]\right\rangle \xrightarrow[\mathcal{F}[l'/l]]{\rho_T[l'/l];\rho[l'/l]} \left\langle \mathbf{1},\ s''[l'/l] \setminus \rho_T[l'/l]\right\rangle$$

(seq) By the induction hypotheses,

$$\left\langle a,\ s[l'/l]\right\rangle \xrightarrow[\mathcal{F}[l'/l]]{\varepsilon;(\rho_T\cdot\rho)[l'/l]} \left\langle v,\ s'[l'/l]\right\rangle$$

Besides, $\mathrm{dom}(s') = \mathrm{dom}(s)$, therefore $l' \notin \mathrm{dom}(s')$. Then, by the induction hypotheses,

$$\left\langle b,\ s'[l'/l]\right\rangle \xrightarrow[\mathcal{F}[l'/l]]{\rho_T[l'/l];\rho[l'/l]} \left\langle v',\ s''[l'/l]\right\rangle$$

Hence

$$\left\langle a\ ;\ b,\ s[l'/l]\right\rangle \xrightarrow[\mathcal{F}[l'/l]]{\rho_T[l'/l];\rho[l'/l]} \left\langle v',\ s''[l'/l]\right\rangle$$

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

(letrec) Since l' appears neither in ρ' nor in $\mathcal{F}$, it does not appear in $\mathcal{F}'$ either. By the induction hypotheses,

$$\left\langle b,\ s[l'/l]\right\rangle \xrightarrow[\mathcal{F}'[l'/l]]{\rho_T[l'/l];\rho[l'/l]} \left\langle v,\ s'[l'/l]\right\rangle$$

Moreover,

$$\mathcal{F}'[l'/l] = \mathcal{F}[l'/l]\{f \mapsto \left[\lambda x_1, \ldots, x_n.a,\ \rho'[l'/l],\ \mathcal{F}\right]\}$$

Hence

$$\langle \textbf{letrec } f(x_1, \ldots, x_n) = a \textbf{ in } b,\ s\rangle \overset{\rho_T[l'/l];\rho[l'/l]}{\underset{\mathcal{F}[l'/l]}{=\!=\!\Rightarrow}} \langle v,\ s'\rangle$$

□

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 4.3.7 (Extending a store in a derivation).

Given the reduction $\langle M,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \langle v,\ s'\rangle$, *then* $\forall t \sqsupseteq s, \exists t' \sqsupseteq s', \langle M,\ t\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \langle v,\ t'\rangle$.

Moreover, both derivations have the same height.

Proof. By induction on the height of the derivation. The most interesting case is (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 4.3.5; (seq), (if-true), (if-false) and (letrec) are straightforward by the induction hypotheses.

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

$$\exists t_2 \sqsupseteq s_2, \langle a_1,\ t_1\rangle \overset{\varepsilon;\rho_T\cdot\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \langle v_1,\ t_2\rangle$$

$$\exists t_{i+1} \sqsupseteq s_{i+1}, \langle a_i,\ t_i\rangle \overset{\varepsilon;\rho_T\cdot\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \langle v_i,\ t_{i+1}\rangle$$

$$\exists t_{n+1} \sqsupseteq s_{n+1}, \langle a_n,\ t_n\rangle \overset{\varepsilon;\rho_T\cdot\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \langle v_n,\ t_{n+1}\rangle$$

The locations l_i might belong to $\mathrm{dom}(t_{n+1})$ and thus not be fresh. By alpha conversion (Lemma 4.3.6), we chose fresh l_i' (not in $\mathrm{Im}(\rho')$ and $\mathrm{dom}(s')$) such that

$$\left\langle b,\ s_{n+1}\{l_i' \mapsto v_i\}\right\rangle \overset{(l_i',v_i);\rho'}{\underset{\mathcal{F}'\{f\mapsto \mathcal{F}f\}}{=\!=\!\Rightarrow}} \langle v,\ s'\rangle$$

and the height of the derivation is preserved. By Property 4.3.5, $t_{n+1}\{l_i' \mapsto v_i\} \sqsupseteq s_{n+1}\{l_i' \mapsto v_i\}$. By the induction hypotheses,

$$\exists t' \sqsupseteq s', \left\langle b,\ t_{n+1}\{l_i' \mapsto v_i\}\right\rangle \overset{(l_i',v_i);\rho'}{\underset{\mathcal{F}'\{f\mapsto \mathcal{F}f\}}{=\!=\!\Rightarrow}} \langle v,\ t'\rangle$$

and the height of the derivation is preserved. Moreover, $t' \setminus \rho_T \sqsupseteq s' \setminus \rho_T$. Hence,

$$\langle f(a_1, \ldots, a_n),\ t_1\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \langle v,\ t' \setminus \rho_T\rangle.$$

(var) Let $t \sqsupseteq s$. $\langle v,\ t\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Rightarrow}} \langle v,\ t \setminus \rho_T\rangle$ and $\exists t' = t \setminus \rho_T \sqsupseteq s \setminus \rho_T = s'$ (Property 4.3.5).

(val) Let $t \sqsupseteq s$. $\langle x,\ t\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Rightarrow}} \langle t\ l,\ t \setminus \rho_T\rangle$ and $\exists t' = t \setminus \rho_T \sqsupseteq s \setminus \rho_T = s'$ (Property 4.3.5). Moreover, $t\ l = s\ l$ because $l \in \mathrm{dom}(s)$ and $t|_{\mathrm{dom}(s)} = s$.

(assign) Let $t \sqsupseteq s$. By the induction hypotheses,

$$\exists t' \sqsupseteq s', \langle a,\ t\rangle \overset{\varepsilon;\rho_T\cdot\rho}{\underset{\mathcal{F}}{\Rightarrow}} \langle v,\ t'\rangle$$

and the height of the derivation is preserved. Hence,

$$\langle x \coloneqq a,\ t\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Rightarrow}} \langle \mathbf{1},\ t'\{l \mapsto v\} \setminus \rho_T\rangle$$

concludes, since $t'\{l \mapsto v\} \setminus \rho_T \ \sqsupseteq\ t'\{l \mapsto v\} \setminus \rho_T$ (Property 4.3.5).

(seq) Let $t \sqsupseteq s$. By the induction hypotheses,

$$\exists t' \sqsupseteq s', \langle a,\ t\rangle \overset{\varepsilon;\rho_T\cdot\rho}{\underset{\mathcal{F}}{\Rightarrow}} \langle v,\ t'\rangle$$

$$\exists t'' \sqsupseteq s'', \langle b,\ t'\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Rightarrow}} \langle v',\ t''\rangle$$

and the height of the derivation is preserved. Hence,

$$\exists t'' \sqsupseteq s'', \langle a\ ;\ b,\ t\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Rightarrow}} \langle v',\ t''\rangle$$

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

(letrec) Let $t \sqsupseteq s$. By the induction hypotheses,

$$\exists t' \sqsupseteq s', \langle b,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}'}{\Rightarrow}} \langle v,\ s'\rangle$$

and the height of the derivation is preserved. Hence,

$$\exists t' \sqsupseteq s', \langle \mathbf{letrec}\ f(x_1,\ldots,x_n) = a\ \mathbf{in}\ b,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{\Rightarrow}} \langle v,\ t'\rangle$$

□

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

Lemma 4.3.8 (Intermediate implies naive).

$$\textit{If } \langle M,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \langle v,\ s'\rangle \textit{ then } \exists t' \sqsupseteq s', \langle M,\ s\rangle \xrightarrow[\mathcal{F}]{\rho_T\cdot\rho} \langle v,\ t'\rangle .$$

Proof. By induction on the height of the derivation, because some stores are modified during the proof. The interesting cases are (seq) and (call), where Lemma 4.3.7 is used to extend intermediary stores. Other cases are straightforward by Property 4.3.5 and the induction hypotheses.

(seq) By the induction hypotheses,

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

Moreover,

$$\langle b,\ s'\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \langle v',\ s''\rangle .$$

Since $t' \sqsupseteq s'$, Lemma 4.3.7 leads to:

$$\exists t \sqsupseteq s'', \langle b,\ t'\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \langle v',\ t\rangle$$

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

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

Hence, since $\sqsubseteq$ is transitive (Property 4.3.5),

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

(call) Similarly to the (seq) case, we apply the induction hypotheses and Lemma 4.3.7:

$$\exists t_2 \sqsupseteq s_2, \langle a_1,\ s_1\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v_1,\ t_2\rangle \qquad \text{(Induction)}$$

$$\exists t'_{i+1} \sqsupseteq s_{i+1}, \langle a_i,\ t_i\rangle \overset{\varepsilon;\rho_T\cdot\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \langle v_i,\ t'_{i+1}\rangle \qquad \text{(Lemma 4.3.7)}$$

$$\exists t_{i+1} \sqsupseteq t'_{i+1} \sqsupseteq s_{i+1}, \langle a_i,\ t_i\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v_i,\ t_{i+1}\rangle \qquad \text{(Induction)}$$

$$\exists t'_{n+1} \sqsupseteq s_{n+1}, \langle a_n,\ t_n\rangle \overset{\varepsilon;\rho_T\cdot\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \langle v_n,\ t'_{n+1}\rangle \qquad \text{(Lemma 4.3.7)}$$

$$\exists t_{n+1} \sqsupseteq t'_{n+1} \sqsupseteq s_{n+1}, \langle a_n,\ t_n\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v_n,\ t_{n+1}\rangle \qquad \text{(Induction)}$$

The locations l_i might belong to $\mathrm{dom}(t_{n+1})$ and thus not be fresh. By alpha conversion (Lemma 4.3.6), we choose a set of fresh l'_i that also do not appear in $\mathrm{Im}(\rho')$ and $\mathrm{dom}(s')$, such that

$$\left\langle b,\ s_{n+1}\{l'_i \mapsto v_i\}\right\rangle \xRightarrow[\mathcal{F}'\{f \mapsto \mathcal{F}\, f\}]{(l'_i, v_i);\rho'} \left\langle v,\ s'\right\rangle.$$

By Property 4.3.5, $t_{n+1}\{l'_i \mapsto v_i\} \sqsupseteq s_{n+1}\{l'_i \mapsto v_i\}$. Lemma 4.3.7 leads to,

$$\exists t \sqsupseteq s', \left\langle b,\ t_{n+1}\{l'_i \mapsto v_i\}\right\rangle \xRightarrow[\mathcal{F}'\{f \mapsto \mathcal{F}\, f\}]{(l'_i, v_i);\rho'} \langle v,\ t\rangle.$$

By the induction hypotheses,

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

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

$$\langle f(a_1, \ldots, a_n),\ s_1\rangle \xRightarrow[\mathcal{F}]{\rho} \left\langle v,\ t' \setminus \rho_T\right\rangle.$$

(val) $\langle v,\ s\rangle \xRightarrow[\mathcal{F}]{\rho} \langle v,\ t'\rangle$ with $t' = s \sqsupseteq s \setminus \rho_T = s'$.

(var) $\langle x,\ s\rangle \xRightarrow[\mathcal{F}]{\rho} \langle s\ l,\ s''\rangle$ with $t' = s \sqsupseteq s \setminus \rho_T = s'$.

(assign) By the induction hypotheses,

$$\exists s'' \sqsupseteq s', \langle a,\ s\rangle \xRightarrow[\mathcal{F}]{\rho} \left\langle v,\ t'\right\rangle$$

Hence,

$$\langle x \coloneqq a,\ s\rangle \xRightarrow[\mathcal{F}]{\rho} \left\langle \mathbf{1},\ t'\{l \mapsto v\}\right\rangle$$

concludes since $t'\{l \mapsto v\} \sqsupseteq s'\{l \mapsto v\}$ (Property 4.3.5).

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

(letrec) By the induction hypotheses,

$$\exists t' \sqsupseteq s', \langle b,\ s\rangle \xrightarrow[\mathcal{F}']{\rho} \langle v,\ s'\rangle .$$

Hence,

$$\exists t' \sqsupseteq s', \langle \textbf{letrec } f(x_1,\ldots,x_n) = a \textbf{ in } b,\ s\rangle \xrightarrow[\mathcal{F}]{\rho} \langle v,\ t'\rangle .$$

□

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:

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

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

Lemma 4.3.9 (Naive implies intermediate). *Assume* $\mathrm{Im}(\rho_T \cdot \rho) \cup \mathrm{Locs}(\mathcal{F}) \subset \mathrm{dom}(s)$. *Then,* $\langle M,\ s\rangle \xrightarrow[\mathcal{F}]{\rho_T \cdot \rho} \langle v,\ s'\rangle$ *implies*

$$\forall t \sqsubseteq s \textit{ such that } \mathrm{Im}(\rho_T \cdot \rho) \cup \mathrm{Locs}(\mathcal{F}) \subset \mathrm{dom}(t), \quad \langle M,\ t\rangle \overset{\rho_T ; \rho}{\underset{\mathcal{F}}{\Rightarrow}} \left\langle v,\ s'|_{\mathrm{dom}(t) \setminus \mathrm{Im}(\rho_T)}\right\rangle .$$

Proof. By induction on the structure of the derivation.

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

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

Hence,

$$\langle v,\ t\rangle \overset{\rho_T ; \rho}{\underset{\mathcal{F}}{\Rightarrow}} \langle v,\ t \setminus \rho_T\rangle .$$

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

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

Hence,

$$\langle x,\ t\rangle \overset{\rho_T ; \rho}{\underset{\mathcal{F}}{\Rightarrow}} \langle t\ l,\ t \setminus \rho_T\rangle .$$

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

$$\langle a,\ t\rangle \overset{\varepsilon;\rho_T\cdot\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \left\langle v,\ s'|_{\mathrm{dom}(t)}\right\rangle$$

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

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

Hence,

$$\langle x \coloneqq a,\ s\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \left\langle \mathbf{1},\ (s'|_{\mathrm{dom}(t)}\{l \mapsto v\}) \setminus \rho_T\right\rangle.$$

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

$$\langle a,\ t\rangle \overset{\varepsilon;\rho_T\cdot\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \left\langle v,\ s'|_{\mathrm{dom}(t)}\right\rangle$$

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

$$\left\langle b,\ s'|_{\mathrm{dom}(t)}\right\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \left\langle v',\ s''|_{\mathrm{dom}(s'|_{\mathrm{dom}(t)})\setminus\mathrm{Im}(\rho_T)}\right\rangle.$$

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

$$\langle a\ ;\ b,\ t\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \left\langle v',\ s''|_{\mathrm{dom}(t)\setminus\mathrm{Im}(\rho_T)}\right\rangle.$$

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

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

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

Then, by the induction hypotheses,

$$\langle b,\ t\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}'}{=\!=\!\Rightarrow}} \left\langle v,\ s'|_{\mathrm{dom}(t)\setminus\mathrm{Im}(\rho_T)}\right\rangle.$$

Hence,

$$\langle \textbf{letrec } f(x_1,\ldots,x_n) = a \textbf{ in } b,\ t\rangle \overset{\rho_T;\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \left\langle v,\ s'|_{\mathrm{dom}(t)\setminus\mathrm{Im}(\rho_T)}\right\rangle.$$

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

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

By the induction hypotheses, they yield:

$$\langle a_1,\ t\rangle \overset{\varepsilon\,;\,\rho_T\cdot\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \langle v_1,\ s_2|_{\mathrm{dom}(t)}\rangle$$

$$\langle a_2,\ s_2|_{\mathrm{dom}(t)}\rangle \overset{\varepsilon\,;\,\rho_T\cdot\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \langle v_1,\ s_3|_{\mathrm{dom}(t)}\rangle$$

$$\forall i, \langle a_i,\ s_i|_{\mathrm{dom}(t)}\rangle \overset{\varepsilon\,;\,\rho_T\cdot\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \langle v_i,\ s_{i+1}|_{\mathrm{dom}(t)}\rangle$$

Moreover, $s_{n+1}|_{\mathrm{dom}(t)} \sqsubseteq s_{n+1}$ implies $s_{n+1}|_{\mathrm{dom}(t)}\{l_i \mapsto v_i\} \sqsubseteq s_{n+1}\{l_i \mapsto v_i\}$ (Property 4.3.5) and:

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

Then, by the induction hypotheses,

$$\langle b,\ s_{n+1}|_{\mathrm{dom}(t)}\{l_i \mapsto v_i\}\rangle \overset{\rho''\,;\,\rho'}{\underset{\mathcal{F}'\{f\mapsto\mathcal{F}\,f\}}{=\!=\!\Rightarrow}} \langle v,\ s'|_{\mathrm{dom}(s_{n+1}|_{\mathrm{dom}(t)}\{l_i\mapsto v_i\})\setminus\mathrm{Im}(\rho'')}\rangle$$

Finally,

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

Hence,

$$\langle f(a_1, \ldots, a_n),\ t\rangle \overset{\rho_T\,;\,\rho}{\underset{\mathcal{F}}{=\!=\!\Rightarrow}} \langle v,\ (s' \setminus \rho_T)|_{\mathrm{dom}(t)\setminus\mathrm{Im}(\rho_T)}\rangle\,.$$

□

4.4 Correctness of lambda lifting

In this section, we prove the correctness of lambda lifting (Theorem 4.1.9 on page 92) by induction on the height of the optimised reduction.

Section 4.4.1 defines stronger invariants and rewords the correctness theorem with them. Section 4.4.2 gives an overview of the proof. Sections 4.4.3 and 4.4.4 prove a few lemmas needed for the proof. Section 4.4.5 contains the actual proof of correctness.

4.4.1 Strengthened hypotheses

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*. They yield a strengthened version of liftable parameters (Definition 4.4.3). We then define lifted environments (Definition 4.4.4) to mirror the effect of lambda lifting in lifted terms captured in closures, and finally reformulate the correctness of lambda lifting in Theorem 4.4.6 with hypotheses strong enough to be provable directly by induction.

Definition 4.4.1 (Aliasing). A set of environments $\mathcal{E}$ is *aliasing-free* when:

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

By extension, an function environment $\mathcal{F}$ is aliasing-free when $\mathrm{Envs}(\mathcal{F})$ is aliasing-free.

The notion of aliasing-free environments is not an artifact of our small language, but translates a fundamental property of the C semantics: distinct function parameters or local variables are always bound to distinct memory locations (Section 6.2.2, paragraph 6 in ISO/IEC 9899 [Int99]).

A local position is any position in a term except inner functions. Local positions are used to distinguish functions defined directly in a term from deeper nested functions, because we need to enforce Invariant 3 (Definition 4.4.3) on the former only.

Definition 4.4.2 (Local position). *Local positions* are defined inductively as follows:

1. M is in local position in "M" and "$x \coloneqq M$" and "$M\ ;\ M$" and "**if** M **then** M **else** M" and "$f(M, \ldots, M)$".
2. N is in local position in "**letrec** $f(x_1, \ldots, x_n) = M$ **in** N".

We extend the notion of liftable parameter (Definition 4.1.8 on page 92) to enforce invariants on stores and environments. It is no longer enough to define liftability in a term M: we need to define it in a 4-tuple made of a term M, a function environment $\mathcal{F}$, a tail variable environment ρ_T and a non-tail variable environment ρ.

Definition 4.4.3 (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, $x \in \mathrm{dom}(\rho_T \cdot \rho) \Leftrightarrow \exists i, f = h_i$,
4. moreover, if h_i is called in tail position in M, then $x \in \mathrm{dom}(\rho_T)$,
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 $\mathrm{Envs}(\mathcal{F}) \cup \{\rho, \rho_T\}$ is aliasing-free.

We also extend the definition of lambda lifting (Definition 4.1.6 on page 91) to environments, in order to reflect changes in lambda-lifted parameters captured in closures.

Definition 4.4.4 (Lifted form of an environment).

$$\text{If } \mathcal{F}\, f = \left[\lambda x_1, \ldots, x_n.b,\ \rho',\ \mathcal{F}'\right] \qquad \text{then}$$

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

Lifted environments are defined such that a liftable parameter never appears in them. This property will be useful during the proof of correctness.

Lemma 4.4.5 (Liftable parameters in lifted environments). *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 $(\mathcal{F})_*$. □

These invariants and definitions lead to a correctness theorem with stronger hypotheses.

Theorem 4.4.6 (Strengthened correctness of lambda lifting). *If x is a liftable parameter in $(M, \mathcal{F}, \rho_T, \rho)$, then*

$$\langle M,\ s\rangle \xrightarrow[\mathcal{F}]{\rho_T;\rho} \langle v,\ s'\rangle \text{ implies } \langle (M)_*,\ s\rangle \xrightarrow[(\mathcal{F})_*]{\rho_T;\rho} \langle v,\ s'\rangle$$

Since naive and optimised reductions rules are equivalent (Theorem 4.2.4), the proof of Theorem 4.1.9 is a direct corollary of this theorem.

Corollary 4.4.7. *If x is a liftable parameter in M, then*

$$\exists t, \langle M,\ \varepsilon\rangle \xrightarrow{\varepsilon} \langle v,\ t\rangle \text{ implies } \exists t', \langle (M)_*,\ \varepsilon\rangle \xrightarrow{\varepsilon} \langle v,\ t'\rangle .$$

4.4.2 Overview of the proof

With the enhanced liftability definition, we have invariants strong enough to perform a proof by induction of the correctness theorem. This proof is detailed in Section 4.4.5.

The proof is not by structural induction but by induction on the height of the derivation. This is necessary because, even with the stronger invariants, we cannot apply the induction hypotheses directly to the premises in the case of the (call) rule: we have to change the stores and environments, which means rewriting the whole derivation tree, before using the induction hypotheses.

To deal with this most difficult case, we distinguish between calling one of the lifted functions ($f = h_i$) and calling another function (either g, where x is defined, or any other

function outside of g). Only the former requires rewriting; the latter follows directly from the induction hypotheses.

In the (call) rule with $f = h_i$, issues arise when reducing the body b of the lifted function. During this reduction, indeed, the store contains a new location l' bound by the environment to the lifted variable x, but also contains the location l which contains the original value of x. Our goal is to show that the reduction of b implies the reduction of $(b)_*$, with store and environments fulfilling the constraints of the (call) rule.

To obtain the reduction of the lifted body $(b)_*$, we modify the reduction of b in a series of steps, using several lemmas:

- the location l of the free variable x is moved to the tail environment (Lemma 4.4.8);
- the resulting reduction meets the induction hypotheses, which we apply to obtain the reduction of the lifted body $(b)_*$;
- however, this reduction does not meet the constraints of the optimised reduction rules because the location l is not fresh: we rename it to a fresh location l' to hold the lifted variable (Lemma 4.4.9);
- finally, since we renamed l to l', we need to reintroduce a location l to hold the original value of x (Lemmas 4.4.10 and 4.4.11).

The rewriting lemmas used in the (call) case are shown in Section 4.4.3.

For every other case, the proof consists in checking thoroughly that the induction hypotheses apply, in particular that x is liftable in the premises. These verifications consist in checking Invariants 1 and 6 of the extended liftability definition (Definition 4.4.3). To keep the main proof as compact as possible, the most difficult cases of liftability, related to aliasing, are proven in some preliminary lemmas (Section 4.4.4).

One last issue arises during the induction when one of the premises does not contain the lifted variable x. In that case, the invariants do not hold, since they assume the presence of x. But it turns out that in this very case, the lifting function is the identity (since there is no variable to lift) and lambda lifting is trivially correct.

4.4.3 Rewriting lemmas

Calling a lifted function has an impact on the resulting store. New locations are introduced for the lifted parameters, and the original locations can no longer be modified. Because of these changes, the induction hypotheses do not apply directly in the case of the (call) rule for a lifted function h_i. We use the following four lemmas to obtain, through several rewriting steps, a reduction of lifted terms meeting the induction hypotheses.

- Lemma 4.4.8 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. It is used to transform the original free variable x (in the non-tail environment) to its lifted copy (which is a parameter of h_i, hence in the tail environment).
- Lemma 4.4.9 handles alpha conversion in stores and is used when choosing a fresh location.

- Lemmas 4.4.10 and 4.4.11 finally add into the store and the environment a fresh location, bound to an arbitrary value. It is used to reintroduce the location containing the original value of x, after it has been alpha converted to l'.

Lemma 4.4.8 (Switching to tail environment). *For all* $x \notin \mathrm{dom}(\rho_T)$, $\langle M,\ s\rangle \xRightarrow[\mathcal{F}]{\rho_T;(x,l)\cdot\rho} \langle v,\ s'\rangle$ *implies* $\langle M,\ s\rangle \xRightarrow[\mathcal{F}]{\rho_T\cdot(x,l);\rho} \left\langle v,\ s'|_{\mathrm{dom}(s')\setminus\{l\}}\right\rangle$. *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 $s \setminus \rho_T \cdot (x,l) = s'|_{\mathrm{dom}(s')\setminus\{l\}}$ when $s' = s \setminus \rho_T$.

(val) $\langle v,\ s\rangle \xRightarrow[\mathcal{F}]{\rho_T\cdot(x,l);\rho} \langle v,\ s \setminus \rho_T \cdot (x,l)\rangle$ and $s \setminus \rho_T \cdot (x,l) = s'|_{\mathrm{dom}(s')\setminus\{l\}}$ with $s' = s \setminus \rho_T$.

(var) $\langle y,\ s\rangle \xRightarrow[\mathcal{F}]{\rho_T\cdot(x,l);\rho} \langle s\ l',\ s \setminus \rho_T \cdot (x,l)\rangle$ and $s \setminus \rho_T \cdot (x,l) = s'|_{\mathrm{dom}(s')\setminus\{l\}}$, with $l' = \rho_T \cdot (x,l) \cdot \rho\ y$ and $s' = s \setminus \rho_T$.

(assign) By hypothesis,

$$\langle a,\ s\rangle \xRightarrow[\mathcal{F}]{\varepsilon;\rho_T\cdot(x,l)\cdot\rho} \langle v,\ s'\rangle$$

hence

$$\langle y \coloneqq a,\ s\rangle \xRightarrow[\mathcal{F}]{\rho_T\cdot(x,l);\rho} \langle \mathbf{1},\ s'\{l' \mapsto v\} \setminus \rho_T \cdot (x,l)\rangle$$

and $s'\{l' \mapsto v\} \setminus \rho_T \cdot (x,l) = s'|_{\mathrm{dom}(s')\setminus\{l\}}$ with $l' = \rho_T \cdot (x,l) \cdot \rho\ y$ and $s' = s'\{l' \mapsto v\} \setminus \rho_T$.

(seq) By hypothesis,

$$\langle a,\ s\rangle \xRightarrow[\mathcal{F}]{\varepsilon;\rho_T\cdot(x,l)\cdot\rho} \langle v,\ s'\rangle$$

and, by the induction hypotheses,

$$\langle b,\ s'\rangle \xRightarrow[\mathcal{F}]{\rho_T\cdot(x,l);\rho} \left\langle v,\ s''|_{\mathrm{dom}(s'')\setminus\{l\}}\right\rangle$$

hence

$$\langle a\ ;\ b,\ s\rangle \xRightarrow[\mathcal{F}]{\rho_T\cdot(x,l);\rho} \left\langle v,\ s''|_{\mathrm{dom}(s'')\setminus\{l\}}\right\rangle.$$

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

(letrec) By the induction hypotheses,

$$\langle b,\ s\rangle \xRightarrow[\mathcal{F}']{\rho_T\cdot(x,l)\,;\rho} \left\langle v,\ s'|_{\mathrm{dom}(s')\setminus\{l\}}\right\rangle$$

hence

$$\langle \textbf{letrec } f(x_1,\ldots,x_n) = a \textbf{ in } b,\ s\rangle \xRightarrow[\mathcal{F}]{\rho_T\cdot(x,l)\,;\rho} \left\langle v,\ s'|_{\mathrm{dom}(s')\setminus\{l\}}\right\rangle$$

(call) The hypotheses do not change, and the conclusion becomes:

$$\langle f(a_1,\ldots,a_n),\ s_1\rangle \xRightarrow[\mathcal{F}]{\rho_T\cdot(x,l)\,;\rho} \left\langle v,\ s'\setminus\rho_T\cdot(x,l)\right\rangle$$

as expected, since $s'\setminus\rho_T\cdot(x,l) = s''|_{\mathrm{dom}(s'')\setminus\{l\}}$ with $s'' = s'\setminus\rho_T$ □

Lemma 4.4.9 (Alpha conversion). *If* $\langle M,\ s\rangle \xRightarrow[\mathcal{F}]{\rho_T\,;\rho} \langle v,\ s'\rangle$ *then, for all* l*, for all* l' *appearing neither in* s *nor in* $\mathcal{F}$ *nor in* $\rho\cdot\rho_T$*,*

$$\left\langle M,\ s[l'/l]\right\rangle \xRightarrow[\mathcal{F}[l'/l]]{\rho_T[l'/l]\,;\rho[l'/l]} \left\langle v,\ s'[l'/l]\right\rangle$$

Moreover, both derivations have the same height.

Proof. See Lemma 4.3.6 on page 103. □

Lemma 4.4.10 (Spurious location in store). *If* $\langle M,\ s\rangle \xRightarrow[\mathcal{F}]{\rho_T\,;\rho} \langle v,\ s'\rangle$ *and* k *does not appear in either* s*,* $\mathcal{F}$ *or* $\rho_T\cdot\rho$*, then, for all value* u*,* $\langle M,\ s\{k\mapsto u\}\rangle \xRightarrow[\mathcal{F}]{\rho_T\,;\rho} \langle v,\ s'\{k\mapsto u\}\rangle$*. Moreover, both derivations have the same height.*

Proof. By induction on the height of the derivation. 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 that $l_j = k$. In that case, we need to rename l_j to some fresh variable $l'_j \neq k$ (by alpha conversion) before applying the induction hypotheses.

(call) By the induction hypotheses,

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

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

$$k \notin \mathrm{Locs}(\mathcal{F}'\{f \mapsto \mathcal{F}\, f\}) \subset \mathrm{Locs}(\mathcal{F})$$

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 4.4.9). 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,

$$\langle b,\ s_{n+1}\{l_i \mapsto v_i\}\{k \mapsto u\}\rangle \xrightarrow[\mathcal{F}'\{f\mapsto\mathcal{F}\,f\}]{\rho''\,;\rho'} \left\langle v,\ s'\{k \mapsto u\}\right\rangle$$

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

$$\langle f(a_1, \ldots, a_n),\ s_1\{k \mapsto u\}\rangle \xrightarrow[\mathcal{F}]{\rho_T\,;\rho} \left\langle v,\ s'\{k \mapsto u\} \setminus \rho_T\right\rangle.$$

(val) $\langle v,\ s\{k \mapsto u\}\rangle \xrightarrow[\mathcal{F}]{\rho_T\,;\rho} \langle v,\ s\{k \mapsto u\} \setminus \rho_T\rangle$ and $s\{k \mapsto u\} \setminus \rho_T = s \setminus \rho_T\{k \mapsto u\}$ since k does not appear in ρ_T.

(var) $\langle x,\ s\{k \mapsto u\}\rangle \xrightarrow[\mathcal{F}]{\rho_T\,;\rho} \langle (s\{k \mapsto u\})\ l,\ s\{k \mapsto u\} \setminus \rho_T\rangle$, with $s\{k \mapsto u\} \setminus \rho_T = s \setminus \rho_T\{k \mapsto u\}$ since k does not appear in ρ_T, and $(s\{k \mapsto u\})\ l = s\ l$ since $k \neq l$ (k does not appear in s).

(assign) By the induction hypotheses, $\langle a,\ s\{k \mapsto u\}\rangle \xrightarrow[\mathcal{F}]{\varepsilon\,;\rho_T\cdot\rho} \langle v,\ s'\{k \mapsto u\}\rangle$. And $k \neq l$ (since k does not appear in s) then $s'\{k \mapsto u\}\{l \mapsto v\} = s'\{l \mapsto v\}\{k \mapsto u\}$. Moreover, k does not appear in ρ_T then $s'\{l \mapsto v\}\{k \mapsto u\} \setminus \rho_T = s'\{l \mapsto v\} \setminus \rho_T\{k \mapsto u\}$. Hence

$$\langle x \coloneqq a,\ s\{k \mapsto u\}\rangle \xrightarrow[\mathcal{F}]{\rho_T\,;\rho} \left\langle \mathbf{1},\ s'\{l \mapsto v\} \setminus \rho_T\{k \mapsto u\}\right\rangle$$

(seq) By the induction hypotheses,

$$\langle a,\ s\{k \mapsto u\}\rangle \xrightarrow[\mathcal{F}]{\varepsilon\,;\rho_T\cdot\rho} \left\langle \mathbf{true},\ s'\{k \mapsto u\}\right\rangle$$

$$\left\langle b,\ s'\{k \mapsto u\}\right\rangle \xrightarrow[\mathcal{F}]{\rho_T\,;\rho} \left\langle v',\ s''\{k \mapsto u\}\right\rangle$$

Hence

$$\langle a\,;\ b,\ s\{k \mapsto u\}\rangle \xrightarrow[\mathcal{F}]{\rho_T\,;\rho} \left\langle v',\ s''\{k \mapsto u\}\right\rangle$$

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

(letrec) The location k does not appear in $\mathcal{F}'$, because it does not appear in either $\mathcal{F}$ or $\rho' \subset \rho_T \cdot \rho$ ($\mathcal{F}' = \mathcal{F}\{f \mapsto [\lambda x_1, \ldots, x_n.a,\ \rho',\ \mathcal{F}]\}$). Then, by the induction hypotheses,

$$\langle b,\ s\{k \mapsto u\}\rangle \xRightarrow[\mathcal{F}']{\rho_T;\rho} \langle v,\ s'\{k \mapsto u\}\rangle$$

Hence

$$\langle \textbf{letrec}\ f(x_1, \ldots, x_n) = a\ \textbf{in}\ b,\ s\{k \mapsto u\}\rangle \xRightarrow[\mathcal{F}]{\rho_T;\rho} \langle v,\ s'\{k \mapsto u\}\rangle.$$

□

Lemma 4.4.11 (Spurious variable in environments).

$$\forall l, l', \langle M,\ s\rangle \xRightarrow[\mathcal{F}]{\rho_T\cdot(x,l);\rho} \langle v,\ s'\rangle \quad \textit{iff} \quad \langle M,\ s\rangle \xRightarrow[\mathcal{F}]{\rho_T\cdot(x,l);(x,l')\cdot\rho} \langle v,\ s'\rangle$$

Moreover, both derivations have the same height.

Proof. See Lemma 4.3.1 on page 98. □

4.4.4 Aliasing lemmas

We need three lemmas to show that environments remain aliasing-free during the proof by induction in Section 4.4.5. 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 4.4.3) holds in the context of the (call) and (letrec) rules, respectively.

Lemma 4.4.12 (Concatenation). *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

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

given that

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

If $\rho_1 \in \mathcal{E}$ and $\rho_2 \in \mathcal{E}$, immediate. If $\rho_1 = \rho \cdot \rho'$, $\rho_1\, x = \rho\, x$ 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. □

Lemma 4.4.13 (Aliasing in (call) rule). *Assume that, in a (call) rule,*

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

Then $\mathrm{Envs}(\mathcal{F}'\{f \mapsto \mathcal{F}\, f\}) \cup \{\rho', \rho''\}$ *is also aliasing-free.*

Proof. Let $\mathcal{E} = \mathrm{Envs}(\mathcal{F}'\{f \mapsto \mathcal{F}\, f\}) \cup \{\rho'\}$. We know that $\mathcal{E} \subset \mathrm{Envs}(\mathcal{F})$ so $\mathcal{E}$ is aliasing-free We want to show that adding fresh and distinct locations from ρ'' preserves this lack of aliasing. More precisely, we want to show that

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

given that

$$\forall \rho_1, \rho_2 \in \mathcal{E}, \forall x \in \mathrm{dom}(\rho_1), \forall y \in \mathrm{dom}(\rho_2),\ \rho_1\, x = \rho_2\, y \Rightarrow x = y.$$

We reason by checking of all cases. If $\rho_1 \in \mathcal{E}$ and $\rho_2 \in \mathcal{E}$, immediate. If $\rho_1 = \rho_2 = \rho''$ then $\rho''\, x = \rho''\, y \Rightarrow x = y$ holds because the locations of ρ'' are distinct. If $\rho_1 = \rho''$ and $\rho_2 \in \mathcal{E}$ then $\rho_1\, x = \rho_2\, y \Rightarrow x = y$ holds because $\rho_1\, x \neq \rho_2\, y$ (by freshness hypothesis). □

Lemma 4.4.14 (Aliasing in (letrec) rule). *If* $\mathrm{Envs}(\mathcal{F}) \cup \{\rho, \rho_T\}$ *is aliasing free, then, for all* x_i,

$$\mathrm{Envs}(\mathcal{F}) \cup \{\rho, \rho_T\} \cup \left\{\rho_T \cdot \rho\ |_{\mathrm{dom}(\rho_T \cdot \rho) \setminus \{x_1, \ldots, x_n\}}\right\}$$

is aliasing free.

Proof. Let $\mathcal{E} = \mathrm{Envs}(\mathcal{F}) \cup \{\rho, \rho_T\}$ and $\rho'' = \rho_T \cdot \rho|_{\mathrm{dom}(\rho_T \cdot \rho) \setminus \{x_1, \ldots, x_n\}}$. Adding ρ'', a restricted concatenation of ρ_T and ρ, to $\mathcal{E}$ preserves the lack of aliasing, as in the proof of Lemma 4.4.12. 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. □

4.4.5 Proof of correctness

We finally recall and show Theorem 4.4.6.

Theorem 4.4.6. *If* x *is a liftable parameter in* $(M, \mathcal{F}, \rho_T, \rho)$*, then*

$$\langle M,\ s\rangle \xrightarrow[\mathcal{F}]{\rho_T\,;\rho} \langle v,\ s'\rangle \text{ implies } \langle (M)_*\,,\ s\rangle \xrightarrow[(\mathcal{F})_*]{\rho_T\,;\rho} \langle v,\ s'\rangle$$

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 $\langle M,\ s\rangle \xrightarrow[\mathcal{F}]{\rho_T\,;\rho} \langle v,\ s'\rangle$.

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

Indeed, the invariants of Definition 4.4.3 hold:

- Invariant 1: x is defined as a parameter of g in $\mathcal{F}$ (because h_i is an inner function of g so x cannot be defined in $h_i(a_1, \ldots, a_n)$).
- Invariant 2: since a_i is not in tail position in $h_i(a_1, \ldots, a_n)$, it does not contain any call to a function h_j; calls in $\mathcal{F}$ respect the invariant by the induction hypotheses.
- Invariant 3: By the definition of a local position, every f defined in local position in a_i is in local position in $h_i(a_1, \ldots, a_n)$, hence the expected property by the induction hypotheses.
- Invariant 4: immediate since the premise does not hold : since the a_i are not in tail position in $h_i(a_1, \ldots, a_n)$, they cannot feature calls to h_i (by Invariant 2).
- Invariant 5: immediate by the induction hypotheses.
- Invariant 6: immediate for the compact closures by the induction hypotheses. Lack of aliasing is guaranteed by Lemma 4.4.12.

By the induction hypotheses, we get

$$\langle (a_i)_*,\ s_i \rangle \xrightarrow[(\mathcal{F})_*]{\varepsilon\,;\,\rho_T \cdot \rho} \langle v_i,\ s_{i+1} \rangle .$$

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

On the other hand, we have $x \in \mathrm{dom}(\rho')$: since, by hypothesis, x is a liftable parameter in $(h_i(a_1, \ldots, a_n), \mathcal{F}, \rho_T, \rho)$, 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: $\rho' = (x, l) \cdot \rho'''$. It is then possible to move (x, l) to the tail environment, according to Lemma 4.4.8:

$$\langle b,\ s_{n+1}\{l_i \mapsto v_i\} \rangle \xrightarrow[\mathcal{F}'\{f \mapsto \mathcal{F}\, f\}]{\rho''(x,l)\,;\,\rho'''} \left\langle v,\ s'|_{\mathrm{dom}(s') \setminus \{l\}} \right\rangle$$

If x is not defined in b or $\mathcal{F}'\{f \mapsto \mathcal{F}\, f\}$, then $(\cdot)_*$ is the identity function and can trivially be applied to the reduction of b.[1] Otherwise, the rewriting performed above ensures that x is a liftable parameter in $(b, \mathcal{F}'\{f \mapsto \mathcal{F}\, f\}, \rho'' \cdot (x, l), \rho''')$.

Indeed, the invariants of Definition 4.4.3 hold. Assume that x is defined as a parameter of some function g, in either b or $\mathcal{F}'\{f \mapsto \mathcal{F}\, f\}$:

- Invariant 1: by hypothesis.
- Invariant 2: since both b and the terms appearing in $\mathcal{F}'\{f \mapsto \mathcal{F}\, f\}$ also appear in $\mathcal{F}$, immediate by the induction hypotheses.

[1] In fact, this case never happens because h_i is an inner function of g so g appears in $\mathcal{F}'$. It is easier to assume it could happen than to prove it cannot, though.

- Invariant 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 $x \in \mathrm{dom}(\rho'' \cdot (x, l) \cdot \rho''')$.
- Invariant 4: immediate since $x \in \mathrm{dom}(\rho'' \cdot (x, l) \cdot \rho''')$.
- Invariant 5: immediate since $\mathcal{F}'$ is included in $\mathcal{F}$.
- Invariant 6: immediate for the compact closures by the induction hypotheses. Lack of aliasing is guaranteed by Lemma 4.4.13.

By the induction hypotheses,

$$\langle (b)_*,\ s_{n+1}\{l_i \mapsto v_i\}\rangle \xRightarrow[(\mathcal{F}'\{f\mapsto \mathcal{F}\,f\})_*]{\rho''(x,l);\rho'''} \langle v,\ s'|_{\mathrm{dom}(s')\setminus\{l\}}\rangle$$

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 with l does not change (since l' is modified instead of l).

Lemma 4.4.5 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. As a consequence, lack of aliasing implies by Definition 4.4.1 that the location l, associated with x, does not appear in $(\mathcal{F}'\{f \mapsto \mathcal{F}\,f\})_*$ either, so

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

Moreover, l does not appear in $s'|_{\mathrm{dom}(s')\setminus\{l\}}$. By alpha conversion (Lemma 4.4.9, since l' does not appear in the store or the environments of the reduction, we rename l to l':

$$\left\langle (b)_*,\ s_{n+1}[l'/l]\{l_i \mapsto v_i\}\right\rangle \xRightarrow[(\mathcal{F}'\{f\mapsto \mathcal{F}\,f\})_*]{\rho''(x,l');\rho'''} \left\langle v,\ s'|_{\mathrm{dom}(s')\setminus\{l\}}\right\rangle.$$

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 $\rho''(x, l') \cdot \rho'''$. Thus, by Lemma 4.4.10,

$$\left\langle (b)_*,\ s_{n+1}[l'/l]\{l_i \mapsto v_i\}\{l \mapsto v_x\}\right\rangle \xRightarrow[(\mathcal{F}'\{f\mapsto \mathcal{F}\,f\})_*]{\rho''(x,l');\rho'''} \left\langle v,\ s'|_{\mathrm{dom}(s')\setminus\{l\}}\{l \mapsto v_x\}\right\rangle.$$

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 $s'|_{\mathrm{dom}(s')\setminus\{l\}}\{l \mapsto v_x\} = s'\{l \mapsto v_x\}$, we finish the rewriting by Lemma 4.4.11,

$$\left\langle (b)_*,\ s_{n+1}\{l_i \mapsto v_i\}\{l' \mapsto v_x\}\right\rangle \xRightarrow[(\mathcal{F}'\{f\mapsto \mathcal{F}\,f\})_*]{\rho''(x,l');(x,l)\cdot\rho'''} \left\langle v,\ \ s'\{l \mapsto v_x\}\right\rangle.$$

Hence the result:

$$\frac{\begin{array}{c}(\mathcal{F})_* \ h_i = \left[\lambda x_1, \ldots, x_n x.\,(b)_*,\ \rho',\ \left(\mathcal{F}'\right)_*\right] \\ \rho'' = (x_1, l_1) \cdot \ldots \cdot (x_n, l_n)(x, \rho_T\ x) \qquad l' \text{ and } l_i \text{ fresh and distinct} \\ \forall i, \langle (a_i)_*,\ s_i\rangle \xRightarrow[(\mathcal{F})_*]{\varepsilon;\rho_T\cdot\rho} \langle v_i,\ s_{i+1}\rangle \qquad \langle (x)_*,\ s_{n+1}\rangle \xRightarrow[(\mathcal{F})_*]{\varepsilon;\rho_T\cdot\rho} \langle v_x,\ s_{n+1}\rangle \\ \left\langle (b)_*,\ s_{n+1}\{l_i \mapsto v_i\}\{l' \mapsto v_x\}\right\rangle \xRightarrow[(\mathcal{F}'\{f\mapsto\mathcal{F}\,f\})_*]{\rho''(x,l');\rho'} \left\langle v,\ s'\{l \mapsto v_x\}\right\rangle\end{array}}{\langle (h_i(a_1, \ldots, a_n))_*,\ s_1\rangle \xRightarrow[(\mathcal{F})_*]{\rho_T;\rho} \left\langle v,\ s'\{l \mapsto v_x\} \setminus \rho_T\right\rangle}\ (\textsc{call})$$

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

(call)—second case We now consider the case where f is not one of the h_i. The variable x is a liftable parameter in $(f(a_1, \ldots, a_n), \mathcal{F}, \rho_T, \rho)$.

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

Indeed, the invariants of Definition 4.4.3 hold. Assume that x is defined as a parameter of some function g, in either a_i or $\mathcal{F}$:

- Invariant 1: by hypothesis.
- Invariant 2: since a_i is not in tail position in $f(a_1, \ldots, a_n)$, it does not contain any call to a function h_i; calls in g and the inner functions h_j respect the invariant by the induction hypotheses.
- Invariant 3: By the definition of a local position, every f defined in local position in a_i is in local position in $f(a_1, \ldots, a_n)$, hence the expected property by the induction hypotheses.
- Invariant 4: immediate since the premise does not hold : the a_i are not in tail position in $f(a_1, \ldots, a_n)$ so they cannot feature calls to h_i (by Invariant 2:).
- Invariant 5: immediate by the induction hypotheses.
- Invariant 6: Lemma 4.4.12.

By the induction hypotheses, we get

$$\langle (a_i)_*,\ s_i\rangle \xRightarrow[(\mathcal{F})_*]{\varepsilon;\rho_T\cdot\rho} \langle v_i,\ s_{i+1}\rangle,$$

and, by Definition 4.1.6,

$$(f(a_1, \ldots, a_n))_* = f((a_1)_*, \ldots, (a_n)_*).$$

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

Indeed, the invariants of Definition 4.4.3 hold. Assume that x is defined as a parameter of some function g, in either b or $\mathcal{F}$:

- Invariant 1: by hypothesis.
- Invariant 2: since both b and the terms appearing in $\mathcal{F}'\{f \mapsto \mathcal{F}\, f\}$ also appear in $\mathcal{F}$, immediate by the induction hypotheses.
- Invariant 3: We have to distinguish the cases where $f = g$ (with $x \in \mathrm{dom}(\rho'')$) and $f \neq g$ (with $x \notin \mathrm{dom}(\rho'')$ and $x \notin \mathrm{dom}(\rho')$). In both cases, the result is immediate by the induction hypotheses.
- Invariant 4: If $f \neq g$, the premise cannot hold (by the induction hypotheses, Invariant 2). If $f = g$, $x \in \mathrm{dom}(\rho'')$ (by the induction hypotheses, Invariant 2).
- Invariant 5: immediate since $\mathcal{F}'$ is included in $\mathcal{F}$.
- Invariant 6: immediate for the compact closures. The lack of aliasing is guaranteed by Lemma 4.4.13.

By the induction hypotheses,

$$\left\langle (b)_*,\ s_{n+1}\{l_i \mapsto v_i\} \right\rangle \xRightarrow[(\mathcal{F}'\{f \mapsto \mathcal{F}\, f\})_*]{\rho'';\rho'} \left\langle v,\ s' \right\rangle$$

hence:

$$\text{(CALL)}\ \frac{\begin{array}{c}(\mathcal{F})_*\ f = \left[\lambda x_1, \ldots, x_n.\, (b)_*,\ \rho',\ \left(\mathcal{F}'\right)_*\right] \\ \rho'' = (x_1, l_1) \cdot \ldots \cdot (x_n, l_n) \qquad l_i \text{ fresh and distinct} \\ \forall i, \left\langle (a_i)_*,\ s_i \right\rangle \xRightarrow[(\mathcal{F})_*]{\varepsilon;\rho_T \cdot \rho} \left\langle v_i,\ s_{i+1} \right\rangle \qquad \left\langle (b)_*,\ s_{n+1}\{l_i \mapsto v_i\} \right\rangle \xRightarrow[(\mathcal{F}'\{f \mapsto \mathcal{F}\, f\})_*]{\rho'';\rho'} \left\langle v,\ s' \right\rangle\end{array}}{\left\langle (f(a_1, \ldots, a_n))_*,\ s_1 \right\rangle \xRightarrow[(\mathcal{F})_*]{\rho_T;\rho} \left\langle v,\ s' \setminus \rho_T \right\rangle}$$

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

Indeed, the invariants of Definition 4.4.3 hold:

- Invariant 1: by the induction hypotheses; note that when $f = g$, x is then defined in $\mathcal{F}'$ rather than b.
- Invariant 2: by the induction hypotheses; note that when $f = h_i$, the invariant is also preserved.
- Invariants 3 and 4: immediate by the induction hypotheses and the definition of tail and local positions.
- Invariant 5: By the induction hypotheses, Invariant 3 (x is to appear in the new closure if and only if $f = h_i$).
- Invariant 6: Lemma 4.4.14.

By the induction hypotheses, we get

$$\left\langle (b)_*,\ s \right\rangle \xRightarrow[(\mathcal{F}')_*]{\rho_T;\rho} \left\langle v,\ s' \right\rangle.$$

If $f \neq h_i$,

$$(\textbf{letrec } f(x_1, \ldots, x_n) = a \textbf{ in } b)_* = \textbf{letrec } f(x_1, \ldots, x_n) = (a)_* \textbf{ in } (b)_*$$

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

$$\text{(LETREC)}\ \frac{\rho' = \rho_T \cdot \rho|_{\mathrm{dom}(\rho_T \cdot \rho) \setminus \{x_1, \ldots, x_n\}} \qquad (\mathcal{F}')_* = (\mathcal{F})_* \{f \mapsto [\lambda x_1, \ldots, x_n.\, (a)_*,\ \rho',\ \mathcal{F}]\}}{\langle (\textbf{letrec } f(x_1, \ldots, x_n) = a \textbf{ in } b)_*,\ s\rangle \xRightarrow[(\mathcal{F})_*]{\rho_T ; \rho} \langle v,\ s'\rangle} \quad \frac{\langle (b)_*,\ s\rangle \xRightarrow[(\mathcal{F}')_*]{\rho_T ; \rho} \langle v,\ s'\rangle}{}$$

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

$$(\textbf{letrec } f(x_1, \ldots, x_n) = a \textbf{ in } b)_* = \textbf{letrec } f(x_1, \ldots, x_n x) = (a)_* \textbf{ in } (b)_*$$

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

$$\text{(LETREC)}\ \frac{\rho' = \rho_T \cdot \rho|_{\mathrm{dom}(\rho_T \cdot \rho) \setminus \{x_1, \ldots, x_n x\}} \qquad (\mathcal{F}')_* = (\mathcal{F})_* \{h_i \mapsto [\lambda x_1, \ldots, x_n x.\, (a)_*,\ \rho',\ \mathcal{F}]\}}{\langle (\textbf{letrec } h_i(x_1, \ldots, x_n) = a \textbf{ in } b)_*,\ s\rangle \xRightarrow[(\mathcal{F})_*]{\rho_T ; \rho} \langle v,\ s'\rangle} \quad \frac{\langle (b)_*,\ s\rangle \xRightarrow[(\mathcal{F}')_*]{\rho_T ; \rho} \langle v,\ s'\rangle}{}$$

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

$$\text{(VAL)}\ \frac{}{\langle (v)_*,\ s\rangle \xRightarrow[(\mathcal{F})_*]{\rho_T ; \rho} \langle v,\ s \setminus \rho_T\rangle}$$

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

$$\text{(VAR)}\ \frac{(\rho_T \cdot \rho)\ y = l \in \mathrm{dom}\ s}{\langle (y)_*,\ s\rangle \xRightarrow[(\mathcal{F})_*]{\rho_T ; \rho} \langle s\ l,\ s \setminus \rho_T\rangle}$$

(assign) The parameter x is liftable in $(y \coloneqq a, \mathcal{F}, \rho_T, \rho)$ so in $(a, \mathcal{F}, \varepsilon, \rho_T \cdot \rho)$ too.
Indeed, the invariants of Definition 4.4.3 hold:

- Invariant 1: immediate by the induction hypotheses.
- Invariant 2: immediate by the induction hypotheses.
- Invariant 3: immediate by the induction hypotheses since local positions in a and in $y \coloneqq a$ are the same.
- Invariant 4: h_i cannot be called in tail position in a by Invariant 2 since a is not in tail position in $y \coloneqq a$.
- Invariant 5: immediate by the induction hypotheses.

- Invariant 6: Lemma 4.4.12.

By the induction hypotheses, we get

$$\langle (a)_*\,,\ s\rangle \xRightarrow[(\mathcal{F})_*]{\varepsilon\,;\rho_T\cdot\rho} \langle v,\ s'\rangle\,.$$

Moreover

$$(y \coloneqq a)_* = y \coloneqq (a)_*\,,$$

so :

$$(\textsc{assign})\ \frac{\langle (a)_*\,,\ s\rangle \xRightarrow[(\mathcal{F})_*]{\varepsilon\,;\rho_T\cdot\rho} \langle v,\ s'\rangle \qquad (\rho_T\cdot\rho)\ y = l \in \operatorname{dom} s'}{\left\langle (y \coloneqq a)_*\,,\ s\right\rangle \xRightarrow[(\mathcal{F})_*]{\rho_T\,;\rho} \left\langle \mathbf{1},\ s'\{l \mapsto v\} \setminus \rho_T\right\rangle}$$

(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 $(\cdot)_*$ is the identity function and can trivially be applied to the reduction of a. Otherwise, x is a liftable parameter in $(a, \mathcal{F}, \varepsilon, \rho_T\cdot\rho)$.

Indeed, the invariants of Definition 4.4.3 hold. Assume that x is defined as a parameter of some function g, in either a or $\mathcal{F}$:

- Invariant 1: by hypothesis.
- Invariant 2: immediate by the induction hypotheses.
- Invariant 3: immediate by the induction hypotheses since local positions in a and in $a\ ;\ b$ are the same.
- Invariant 4: h_i cannot be called in tail position in a by Invariant 2 since a is not in tail position in $a\ ;\ b$.
- Invariant 5: immediate by the induction hypotheses.
- Invariant 6: Lemma 4.4.12.

If x is not defined in b or $\mathcal{F}$, then $(\cdot)_*$ is the identity function and can trivially be applied to the reduction of b. Otherwise, x is a liftable parameter in $(b, \mathcal{F}, \rho_T, \rho)$. Indeed, the invariants of Definition 4.4.3 hold (the proof is the same as for the liftability in a above, except for Invariant 6 which is immediate by the induction hypotheses).

By the induction hypotheses, we get $\langle (a)_*\,,\ s\rangle \xRightarrow[(\mathcal{F})_*]{\varepsilon\,;\rho_T\cdot\rho} \langle v,\ s'\rangle$ and $\langle (b)_*\,,\ s'\rangle \xRightarrow[(\mathcal{F})_*]{\rho_T\,;\rho} \langle v',\ s''\rangle$.
Moreover,

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

hence:

$$(\textsc{seq})\ \frac{\langle (a)_*\,,\ s\rangle \xRightarrow[(\mathcal{F})_*]{\varepsilon\,;\rho_T\cdot\rho} \langle v,\ s'\rangle \qquad \langle (b)_*\,,\ s'\rangle \xRightarrow[(\mathcal{F})_*]{\rho_T\,;\rho} \langle v',\ s''\rangle}{\langle (a\ ;\ b)_*\,,\ s\rangle \xRightarrow[(\mathcal{F})_*]{\rho_T\,;\rho} \langle v',\ s''\rangle}$$

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

Chapter 5

CPS conversion in an imperative language

In this chapter, we prove the correctness of the CPS conversion pass performed by the CPC translator, using the small imperative language defined in Chapter 4. As explained in Section 3.2.2, our CPS conversion is defined only on a subset of C programs that we call *CPS-convertible terms*. In Section 5.1, we formalise the notion of CPS-convertible terms in our small language. Then, in Section 5.2, we show that the *early evaluation* (Section 3.2.3) of function parameters in CPS-convertible terms is correct.

In Section 5.3, we define the *CPS conversion* and its image, *CPS terms*. Since CPS terms feature two new operators—`push` and `invoke`—to build and execute continuations, we also define the associated reduction rules. The proof of correctness of CPS conversion (Theorem 5.4.1) is finally carried out in Section 5.4.

5.1 CPS-convertible form

CPS conversion is not defined for every C function; instead, we restrict ourselves to a subset of functions, which we call the *CPS-convertible* subset (Section 3.2.2). The CPS-convertible form restricts the calls to CPS functions to make it straightforward to capture their continuation. In CPS-convertible form, a call to a CPS function `f` is either in tail position, or followed by a tail call to another CPS function whose parameters are *non-shared* variables that cannot be modified by `f`.

In Section 3.2.3 (Definition 3.2.3 on page 73), we defined the CPS-convertible form as follows for C programs:[1]

Definition 5.1.1 (CPS-convertible form). A function is in *CPS-convertible form* if every call to a CPS function that it contains matches one of the following patterns, where both `f` and

[1] We skip here the variants needed to take into account the syntactical peculiarity of functions returning `void` in C.

g are CPS functions, `e`$_1$`, ..., e`$_n$ are any C expressions and `x, y`$_1$`, ..., y`$_n$ are distinct, non-shared variables:

```
                    return f(e1, ..., en);
x = f(e1, ..., en); return g(x, y1, ..., yn);
```

CPS-convertible terms To prove the correctness of CPS conversion, we need to express this definition in our small imperative language. This is done by defining *CPS-convertible terms*, which are a subset of the terms introduced in Section 4.1 (Definition 4.1.1 on page 88). 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.

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 sequence of nested function calls in a highly restricted form: their parameters are (side-effect free) expressions, except possibly for the first one, which can be another function call of the same form. Values and expressions are left unchanged (remember that our definition of expressions does not include function calls).

Definition 5.1.2 (CPS-convertible terms).

$$
\begin{array}{rcll}
v & ::= & \mathbf{1} \mid \mathbf{true} \mid \mathbf{false} \mid n \in \mathbf{N} & \text{(values)} \\
e & ::= & v \mid x \mid \ldots & \text{(expressions)} \\
F & ::= & f(e_1, \ldots, e_n) \mid g(F, e_1, \ldots, e_n) & \text{(tail)} \\
T & ::= & e \mid x \coloneqq e\,;\ T \mid \mathbf{if}\ e\ \mathbf{then}\ T\ \mathbf{else}\ T \mid F & \text{(head)}
\end{array}
$$

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.

The tails are a generalisation of Definition 5.1.1, which will be useful for the proof of correctness of CPS conversion. Note that `x = f(e`$_1$`, ..., e`$_n$`); return g(x, y`$_1$`, ..., y`$_n$`)` is represented by $g(f(e_1, \ldots, e_n), y_1, \ldots, y_n)$: this translation is correct because, contrary to C, our language guarantees a left-to-right evaluation of function parameters.

Note that, compared to our original language, there is no letrec construct any longer since every function is defined at top-level. We have also restricted assignments, conditions and function parameters of f to expressions, to ensure that function calls only appear in tail position. Finally, there is no need to forbid shared variables explicitly in the parameters of g because they are ruled out of our language by design.

Reduction rules To evaluate CPS-convertible terms, we do not need to keep an explicit function environment since there are no inner functions any more. The reduction rules are adapted accordingly (Fig. 5.1). Note that we split the (call) rule to match the constrained form

of function calls in tails: (call) is limited to function calls with expressions as parameters, while (tail) handles the case of nested function calls. We also make explicit the fact that stores are not modified when evaluating expressions.

Figure 5.1: Reduction rules for CPS-convertible terms

$$(\textsc{val})\ \langle v,\ s\rangle \xrightarrow{\rho} \langle v,\ s\rangle$$

$$(\textsc{var})\ \langle x,\ s\rangle \xrightarrow{\rho} \langle s\ (\rho\ x),\ s\rangle$$

$$(\textsc{assign})\ \frac{\langle e,\ s\rangle \xrightarrow{\rho} \langle v,\ s\rangle \qquad \rho\ x = l \in \mathrm{dom}\ s}{\langle x \coloneqq e,\ s\rangle \xrightarrow{\rho} \langle \mathbf{1},\ s\{l \mapsto v\}\rangle}$$

$$(\textsc{seq})\ \frac{\langle a,\ s\rangle \xrightarrow{\rho} \langle v,\ s'\rangle \qquad \langle b,\ s'\rangle \xrightarrow{\rho} \langle v',\ s''\rangle}{\langle a\ ;\ b,\ s\rangle \xrightarrow{\rho} \langle v',\ s''\rangle}$$

$$(\textsc{if-t.})\ \frac{\langle e,\ s\rangle \xrightarrow{\rho} \langle \mathbf{true},\ s\rangle \qquad \langle a,\ s\rangle \xrightarrow{\rho} \langle v,\ s'\rangle}{\langle \mathbf{if}\ e\ \mathbf{then}\ a\ \mathbf{else}\ b,\ s\rangle \xrightarrow{\rho} \langle v,\ s'\rangle}$$

$$(\textsc{if-f.})\ \frac{\langle e,\ s\rangle \xrightarrow{\rho} \langle \mathbf{false},\ s\rangle \qquad \langle b,\ s\rangle \xrightarrow{\rho} \langle v,\ s'\rangle}{\langle \mathbf{if}\ e\ \mathbf{then}\ a\ \mathbf{else}\ b,\ s\rangle \xrightarrow{\rho} \langle v,\ s'\rangle}$$

$$(\textsc{call})\ \frac{f = \lambda x_1, \ldots, x_n.a \qquad \rho' = (x_1, l_1) \cdot \ldots \cdot (x_n, l_n) \qquad l_i \text{ fresh and distinct} \qquad \forall i, \langle e_i,\ s\rangle \xrightarrow{\rho} \langle v_i,\ s\rangle \qquad \langle a,\ s\{l_i \mapsto v_i\}\rangle \xrightarrow{\rho'} \langle v,\ s'\rangle}{\langle f(e_1, \ldots, e_n),\ s\rangle \xrightarrow{\rho} \langle v,\ s'\rangle}$$

$$(\textsc{tail})\ \frac{\langle F,\ s\rangle \xrightarrow{\rho} \langle v_0,\ s'\rangle \qquad \forall i, \langle e_i,\ s'\rangle \xrightarrow{\rho} \langle v_i,\ s'\rangle \qquad \langle g(v_0, \ldots, v_n),\ s'\rangle \xrightarrow{\rho} \langle v,\ s''\rangle}{\langle g(F, e_1, \ldots, e_n),\ s\rangle \xrightarrow{\rho} \langle v,\ s''\rangle}$$

5.2 Early evaluation

In this section, we prove that correctness of *early evaluation*, ie. evaluating the expressions $expr$ before F when reducing $f(F, expr, \ldots, expr)$ in a tail. This result is necessary to show

the correctness of the CPS conversion, because function parameters are evaluated before any function call when building continuations (as explained in Section 3.2).

More precisely, we prove here that it is correct to commute the evaluation of the tail F and the evaluation of the expressions e_i in the (tail) rule.

Theorem 5.2.1 (Correctness of early evaluation). *Let F be a tail such that $\langle F,\ s\rangle \xrightarrow{\rho} \langle v,\ s'\rangle$. Then $\langle e_i,\ s'\rangle \xrightarrow{\rho} \langle v_i,\ s'\rangle$ if and only if $\langle e_i,\ s\rangle \xrightarrow{\rho} \langle v_i,\ s\rangle$.*

We first show that the evaluation of a CPS-convertible term cannot modify the locations other than those of the current function, accessible through the current environment.

Lemma 5.2.2 (Effect of CPS-convertible terms on stores). *Let M be a CPS-convertible term. Then,*

$$\langle M,\ s\rangle \xrightarrow{\rho} \langle v,\ s'\rangle$$

implies

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

Proof. By induction on the structure of the reduction.

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

(assign) Since $l \in \mathrm{Im}(\rho)$, $s|_{\mathrm{dom}(s)\setminus \mathrm{Im}(\rho)} = s\{l \mapsto v\}|_{\mathrm{dom}(s)\setminus \mathrm{Im}(\rho)}$.

(seq) By the induction hypotheses,

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

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

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

Hence,

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

(if-true) and (if-false) immediate by the induction hypotheses.

(call) By the induction hypotheses,

$$(s\{l_i \mapsto v_i\})|_{\mathrm{dom}(s\{l_i\mapsto v_i\})\setminus \mathrm{Im}(\rho')} = s'|_{\mathrm{dom}(s\{l_i\mapsto v_i\})\setminus \mathrm{Im}(\rho')}.$$

Since $\mathrm{Im}(\rho') = \{l_i\}$ and $\mathrm{dom}(s) \cap \{l_i\} = \varnothing$ (by freshness),

$$(s\{l_i \mapsto v_i\})|_{\mathrm{dom}(s)} = s'|_{\mathrm{dom}(s)}$$

hence $s = s'|_{\mathrm{dom}(s)}$.
Since $\mathrm{dom}(s) \setminus \mathrm{Im}(\rho) \subset \mathrm{dom}(s)$,

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

(tail) By the induction hypotheses,

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

Hence,

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

□

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

Corollary 5.2.3. *Let F be a tail of function calls such that $\langle F,\ s\rangle \xrightarrow{\rho} \langle v,\ s'\rangle$. Then $s \sqsubseteq s'$.*

Proof. Remember that *store extension* (written $\sqsubseteq$) is a partial order over stores (Property 4.3.5) defined in Section 4.3.2 as follows: $s \sqsubseteq s' \quad \text{iff} \quad s'|_{\mathrm{dom}(s)} = s$.

The proof is by induction on the structure of the derivation.

(call) Lemma 5.2.2 implies:

$$(s\{l_i \mapsto v_i\})|_{\mathrm{dom}(s\{l_i\mapsto v_i\})\setminus\mathrm{Im}(\rho')} = s'|_{\mathrm{dom}(s\{l_i\mapsto v_i\})\setminus\mathrm{Im}(\rho')}.$$

Since $\mathrm{Im}(\rho') = \{l_i\}$ and $\mathrm{dom}(s) \cap \{l_i\} = \varnothing$ (by freshness),

$$(s\{l_i \mapsto v_i\})|_{\mathrm{dom}(s)} = s'|_{\mathrm{dom}(s)}.$$

As a result, $s = s'|_{\mathrm{dom}(s)}$ hence $s \sqsubseteq s'$.

(tail) By the induction hypotheses, $s \sqsubseteq s'$ and $s' \sqsubseteq s''$. Since $\sqsubseteq$ is a partial order, by transitivity, $s \sqsubseteq s''$.

□

This leads to our correctness theorem for early evaluation.

Theorem 5.2.1. *Let F be a tail such that $\langle F,\ s\rangle \xrightarrow{\rho} \langle v,\ s'\rangle$. Then $\langle e_i,\ s'\rangle \xrightarrow{\rho} \langle v_i,\ s'\rangle$ if and only if $\langle e_i,\ s\rangle \xrightarrow{\rho} \langle v_i,\ s\rangle$.*

Proof. If the expressions e_i are constant values, the theorem is trivial. We assume that the expressions e_i are variables: $e_i = y_i$. By the rule (var), $v_i = s'\ (\rho\ y_i)$ (respectively, $v_i = s'\ (\rho\ y_i)$). By Corollary 5.2.3, $s \sqsubseteq s'$. Since $\rho\ y_i \in \mathrm{dom}(s)$, $s'\ (\rho\ y_i) = s\ (\rho\ y_i)$. Hence $\langle y_i,\ s\rangle \xrightarrow{\rho} \langle v_i,\ s\rangle$. (respectively, $\langle y_i,\ s'\rangle \xrightarrow{\rho} \langle v_i,\ s'\rangle$). □

5.3 CPS conversion

In this section, we define our CPS conversion, its target language—*CPS terms*—as well as continuations and the associated reduction rules.

CPS terms In classical CPS conversion techniques [Plo75], continuations are expressed as functions, and there is no distinction between CPS terms and continuations: the image of CPS conversion is lambda terms which represent continuations, and need to be applied to the identity function to yield a value. In our case, however, continuations are abstract data types, built and executed by our CPS terms. 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.

Except for the introduction of **push** and **invoke**, the structure of CPS terms is very close to the structure of CPS-convertible terms.

Definition 5.3.1 (CPS terms).

$$
\begin{array}{rll}
v ::= & \mathbf{1} \mid \mathbf{true} \mid \mathbf{false} \mid n \in \mathbf{N} & \text{(values)} \\
e ::= & v \mid x \mid \ldots & \text{(expressions)} \\
F ::= & \mathbf{push}\ f(e_1, \ldots, e_n)\ ;\ \mathbf{invoke} & \\
\mid & \mathbf{push}\ g(e_1, \ldots, e_n)\ ;\ F & \text{(tail)} \\
T ::= & \mathbf{invoke}\ e \mid x \coloneqq e\ ;\ T \mid \mathbf{if}\ e\ \mathbf{then}\ T\ \mathbf{else}\ T \mid F & \text{(head)}
\end{array}
$$

The operators **push** and **invoke** act on *continuations*. A continuation is a finite sequence of nested function calls to be performed, with already evaluated parameters. The empty continuation is noted ε.

Definition 5.3.2 (Continuations). Continuations are defined inductively as follows:

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

For instance, the continuation $f(1,2) \cdot g(3)$ means that, if the current computation yields some value v, the program should return $g(f(v,1,2),3)$.

Reduction rules The reduction rules for CPS terms (Fig. 5.2) are the same as the rules for CPS-convertible terms, except that the rules for heads and tails carry the current continuation k[2]. We add reduction rules for the **push** and **invoke** operators, which replace the (tail) rule. Note that the (invoke) rule is not defined when the current continuation is empty; this case is forbidden by the structure of CPS terms, where **invoke** is always preceded by a **push** operator in the tail.

[2] The rules (var) and (val) are not changed because evaluating an expression does not require a continuation.

Figure 5.2: Reduction rules for CPS terms

$$(\textsc{push})\ \frac{\forall i, \langle e_i,\ s\rangle \xrightarrow{\rho} \langle v_i,\ s\rangle}{\langle \textbf{push}\ f(e_1,\ldots,e_n),\ s,\ k\rangle \xrightarrow{\rho} \langle \textbf{1},\ s,\ f(v_1,\ldots,v_n)\cdot k\rangle}$$

$$(\textsc{inv.})\ \frac{\langle f(v_1,\ldots,v_n),\ s,\ k\rangle \xrightarrow{\rho} \langle v,\ s',\ k'\rangle}{\langle \textbf{invoke},\ s,\ f(v_1,\ldots,v_n)\cdot k\rangle \xrightarrow{\rho} \langle v,\ s',\ k'\rangle}$$

$$(\textsc{inv.-}\varepsilon)\ \frac{\langle e,\ s\rangle \xrightarrow{\rho} \langle v,\ s\rangle}{\langle \textbf{invoke}\ e,\ s,\ \varepsilon\rangle \xrightarrow{\rho} \langle v,\ s,\ \varepsilon\rangle}$$

$$(\textsc{inv.-e})\ \frac{\langle e,\ s\rangle \xrightarrow{\rho} \langle v_0,\ s\rangle \qquad \langle f(v_0,\ldots,v_n),\ s,\ k\rangle \xrightarrow{\rho} \langle v,\ s',\ k'\rangle}{\langle \textbf{invoke}\ e,\ s,\ f(v_1,\ldots,v_n)\cdot k\rangle \xrightarrow{\rho} \langle v,\ s',\ k'\rangle}$$

$$(\textsc{assign})\ \frac{\langle e,\ s\rangle \xrightarrow{\rho} \langle v,\ s\rangle \qquad \rho\ x = l \in \mathrm{dom}\ s}{\langle x \coloneqq e,\ s,\ k\rangle \xrightarrow{\rho} \langle \textbf{1},\ s\{l \mapsto v\},\ k\rangle}$$

$$(\textsc{seq})\ \frac{\langle a,\ s,\ k\rangle \xrightarrow{\rho} \langle v,\ s',\ k'\rangle \qquad \langle b,\ s',\ k'\rangle \xrightarrow{\rho} \langle v',\ s'',\ k''\rangle}{\langle a\ ;\ b,\ s,\ k\rangle \xrightarrow{\rho} \langle v',\ s'',\ k''\rangle}$$

$$(\textsc{if-t.})\ \frac{\langle e,\ s\rangle \xrightarrow{\rho} \langle \textbf{true},\ s\rangle \qquad \langle a,\ s,\ k\rangle \xrightarrow{\rho} \langle v,\ s',\ k'\rangle}{\langle \textbf{if}\ e\ \textbf{then}\ a\ \textbf{else}\ b,\ s,\ k\rangle \xrightarrow{\rho} \langle v,\ s',\ k'\rangle}$$

$$(\textsc{if-f.})\ \frac{\langle e,\ s\rangle \xrightarrow{\rho} \langle \textbf{false},\ s\rangle \qquad \langle b,\ s,\ k\rangle \xrightarrow{\rho} \langle v,\ s',\ k'\rangle}{\langle \textbf{if}\ e\ \textbf{then}\ a\ \textbf{else}\ b,\ s,\ k\rangle \xrightarrow{\rho} \langle v,\ s',\ k'\rangle}$$

$$(\textsc{call})\ \frac{f = \lambda x_1,\ldots,x_n.a \qquad \rho' = (x_1,l_1)\cdot\ldots\cdot(x_n,l_n) \qquad l_i \text{ fresh and distinct} \qquad \forall i, \langle e_i,\ s\rangle \xrightarrow{\rho} \langle v_i,\ s\rangle \qquad \langle a,\ s\{l_i \mapsto v_i\},\ k\rangle \xrightarrow{\rho'} \langle v,\ s',\ k'\rangle}{\langle f(e_1,\ldots,e_n),\ s,\ k\rangle \xrightarrow{\rho} \langle v,\ s',\ k'\rangle}$$

CPS conversion The CPS conversion, written $\cdot^\star$, is a straightforward bijection between CPS-convertible terms and CPS terms. It generalises the translation given for C programs in Section 3.2 (Definition 3.2.3 on page 73).

Definition 5.3.3 (CPS conversion). The CPS conversion $\cdot^\star$ is defined inductively as follows on the CPS-convertible terms, where $f^\star$ is the function f after CPS conversion:

$$\begin{aligned}
e^\star &= \textbf{invoke}\ e \\
(x \coloneqq e\ ;\ T)^\star &= x \coloneqq e\ ;\ T^\star \\
(\textbf{if}\ e\ \textbf{then}\ T_1\ \textbf{else}\ T_2)^\star &= \textbf{if}\ e\ \textbf{then}\ T_1^\star\ \textbf{else}\ T_2^\star \\
(f(e_1, \ldots, e_n))^\star &= \textbf{push}\ f^\star(e_1, \ldots, e_n)\ ;\ \textbf{invoke} \\
(g(F, e_1, \ldots, e_n))^\star &= \textbf{push}\ g^\star(e_1, \ldots, e_n)\ ;\ F^\star
\end{aligned}$$

One might wonder why we defined CPS conversion for C programs in a restricted case only. One reason is that the tails produced by the splitting pass are limited to two function calls; there is no point considering arbitrary long tails in practice. Earlier versions of the CPC compiler tried nonetheless to find longer tails in the original source code and to preserve them during the splitting pass. But even if such longer tails were found—which in fact never happens except in contrived examples—the compiler must then ensure that the function parameters of these functions are not shared—which once again is automatically the case after splitting and lambda lifting, but not necessarily in the original source code. We finally decided that the general case made the whole translation more complex and fragile for no practical benefit, and switched to the restricted version presented here.

5.4 Proof of correctness

In this section, we prove the correctness of CPS conversion.

Theorem 5.4.1 (Correctness of CPS conversion). *Let M be a CPS-convertible term. Then*

$$\langle M,\ s\rangle \xrightarrow{\rho} \langle v,\ s'\rangle \Leftrightarrow \langle M^\star,\ s,\ \varepsilon\rangle \xrightarrow{\rho} \langle v,\ s',\ \varepsilon\rangle\,.$$

Proof. The theorem is a direct corollary of Lemma 5.4.2 below, with $k = \varepsilon$. □

To prove the correctness of CPS conversion by induction, we add to the hypotheses that invoking the current continuation k with the result of the evaluation of the CPS-convertible term M yields the same result than evaluating the CPS-converted term $M^\star$.

Lemma 5.4.2 (Correctness of CPS conversion). *Let M be a CPS-convertible term, k a continuation, v_1 a value, s_0 and s_1 two stores, and ρ an environment. The following two propositions are equivalent:*

1. $\langle M^\star,\ s_0,\ k\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle$

2. *there are a value v_2 and a store s_2 such that*
   a) $\langle M,\ s_0\rangle \xrightarrow{\rho} \langle v_2,\ s_2\rangle$ *and*
   b) $\langle \textbf{invoke}\ v_2,\ s_2,\ k\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle$.

We prove both implications by induction. The proof is tedious but mostly straightforward. The most interesting case is (tail), where we use the early evaluation theorem and need to apply the induction hypotheses twice—once for the evaluation of the tail and once for the function call (or continuation invocation) that uses the result of this evaluation.

(2) ⇒ (1) By induction on the structure of the derivation (2a).

1. If M is an expression e, we want to show that

$$\langle \textbf{invoke}\ e,\ s_0,\ k\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle .$$

   Consider cases for k:
   - If k is the empty continuation ε, the reduction (2b) yields $v_2 = v_1$ and $s_1 = s_2$ by the (invoke-ε) rule. The conclusion follows by the (invoke-ε) rule.
   - Otherwise, $k = f(w_1, \ldots, w_n) \cdot k'$. By the (invoke-e) rule, the reduction (2b) implies

$$\left\langle f(v_2, w_1, \ldots, w_n),\ s_2,\ k'\right\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle . \tag{5.1}$$

     Moreover, the reduction (2a) yields $s_0 = s_2$, by the (val) or the (var) rule. Hence, by the (invoke-e) rule, the reductions (2a) and (5.1) lead to the conclusion:

$$\left\langle \textbf{invoke}\ e,\ s_0,\ f(w_1, \ldots, w_n) \cdot k'\right\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle .$$

2. If M is a tail of the form $g(F, e_1, \ldots, e_n)$, we want to show that

$$\langle \textbf{push}\ g^\star(e_1, \ldots, e_n)\ ;\ F^\star,\ s_0,\ k\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle .$$

   By the (tail) rule, the reduction (2a) yields

$$\langle F,\ s_0\rangle \xrightarrow{\rho} \left\langle v_0,\ s'\right\rangle \tag{5.2}$$

$$\forall i, \left\langle e_i,\ s'\right\rangle \xrightarrow{\rho} \left\langle w_i,\ s'\right\rangle \tag{5.3}$$

$$\left\langle g(v_0, w_1, \ldots, w_n),\ s'\right\rangle \xrightarrow{\rho} \langle v_2,\ s_2\rangle . \tag{5.4}$$

   By early evaluation (Theorem 5.2.1), the reductions (5.2) and (5.3) yield

$$\forall i, \langle e_i,\ s_0\rangle \xrightarrow{\rho} \langle w_i,\ s_0\rangle .$$

By the (push) rule, this leads to

$$\langle \mathbf{push}\ g^\star(e_1,\ldots,e_n),\ s_0,\ k\rangle \xrightarrow{\rho} \langle \mathbf{1},\ s_0,\ g^\star(w_1,\ldots,w_n)\cdot k\rangle\,. \tag{5.5}$$

By the induction hypotheses, the reductions (5.4) and (2b) lead to

$$\big\langle \mathbf{push}\ g^\star(v_0,w_1,\ldots,w_n)\ ;\ \mathbf{invoke},\ s',\ k\big\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle$$

hence, by the (seq), (push) and (invoke) rules,

$$\big\langle g^\star(v_0,w_1,\ldots,w_n),\ s',\ k\big\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle\,.$$

By the (invoke-e) rule, this leads to

$$\big\langle \mathbf{invoke}\ v_0,\ s',\ g^\star(w_1,\ldots,w_n)\cdot k\big\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle\,. \tag{5.6}$$

By the induction hypotheses, the reductions (5.2) and (5.6) lead to

$$\langle F,\ s_0,\ g^\star(v_1,\ldots,v_n)\cdot k\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle\,. \tag{5.7}$$

Finally, by the (seq) rule, the reductions (5.5) and (5.7) lead to the conclusion.

3. If M is a tail call of the form $f(e_1,\ldots,e_n)$, we want to show that

$$\langle \mathbf{push}\ f^\star(e_1,\ldots,e_n)\ ;\ \mathbf{invoke},\ s_0,\ k\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle\,.$$

By the (call) rule, the reduction (2a) yields

$$\forall i, \langle e_i,\ s_0\rangle \xrightarrow{\rho} \langle w_i,\ s_0\rangle$$
$$\langle a,\ s_0\{l_i \mapsto w_i\}\rangle \xrightarrow{\rho'} \langle v_2,\ s_2\rangle\,.$$

By the induction hypotheses,

$$\langle a^\star,\ s_0\{l_i \mapsto w_i\},\ k\rangle \xrightarrow{\rho'} \langle v_1,\ s_1,\ \varepsilon\rangle\,.$$

By the (call) rule, this leads to

$$\langle f^\star(e_1,\ldots,e_n),\ s_0,\ k\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle\,.$$

The conclusion follows by the (seq), (push) and (invoke) rules.

4. If M is an assignment or a conditional, immediate by the induction hypotheses.

(1) ⇒ (2) By induction on the structure of the derivation (1).

1. If M is an expression e, $M^\star$ is **invoke** e and we want to show that there is a value v_2 and a store s_2 such that

$$\langle e,\ s_0\rangle \xrightarrow{\rho} \langle v_2,\ s_2\rangle$$
$$\langle \textbf{invoke}\ v_2,\ s_2,\ k\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle$$

   We distinguish on the size of k.
   - If k is the empty continuation ε, the reduction (1) yields $v_2 = v_1$ and $s_1 = s_2$ by the (invoke-ε) rule. The conclusion follows by the (var) or (val) rule for (2a) and by the (invoke-ε) rule for (2b).
   - Otherwise, $k = f(w_1, \ldots, w_n)\cdot k'$. By the (invoke-e) rule, the reduction (1) implies

$$\langle e,\ s_0\rangle \xrightarrow{\rho} \langle v_2,\ s_0\rangle \tag{5.8}$$
$$\left\langle f(v_2, w_1, \ldots, w_n),\ s_0,\ k'\right\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle \tag{5.9}$$

     The reduction (5.8) implies the reduction (2a) with $s_2 = s_0$. The reduction (5.9) yields the reduction (2b) by the (invoke-e) rule.
2. If M is a tail of the form $g(F, e_1, \ldots, e_n)$, then $M^\star$ is **push** $g^\star(e_1, \ldots, e_n)\ ;\ F^\star$ and we want to show that there is a value v_2 and a store s_2 such that

$$\langle g(F, e_1, \ldots, e_n),\ s_0\rangle \xrightarrow{\rho} \langle v_2,\ s_2\rangle$$
$$\langle \textbf{invoke}\ v_2,\ s_2,\ k\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle\,.$$

   By the (seq) and (push) rules, the reduction (1) yields

$$\forall i, \langle e_i,\ s_0\rangle \xrightarrow{\rho} \langle w_i,\ s_0\rangle \tag{5.10}$$
$$\langle F^\star,\ s_0,\ g^\star(w_1, \ldots, w_n)\cdot k\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle \tag{5.11}$$

   By the induction hypotheses, the reduction (5.11) leads to the existence of a value v and a store s such that

$$\langle F,\ s_0\rangle \xrightarrow{\rho} \langle v,\ s\rangle \tag{5.12}$$
$$\langle \textbf{invoke}\ v,\ s,\ g^\star(w_1, \ldots, w_n)\cdot k\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle\,. \tag{5.13}$$

   By early evaluation (Theorem 5.2.1), the reductions (5.12) and (5.10) yield

$$\forall i, \langle e_i,\ s\rangle \xrightarrow{\rho} \langle w_i,\ s\rangle\,. \tag{5.14}$$

By the (invoke-e) rule, the reduction (5.13) implies

$$\langle g(v, w_1, \ldots, w_n),\ s,\ k\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle\,.$$

hence, by the induction hypotheses, there is a value v_2 and a store s_2 such that

$$\langle g(v, w_1, \ldots, w_n),\ s\rangle \xrightarrow{\rho} \langle v_2,\ s_2\rangle \tag{5.15}$$

$$\langle \textbf{invoke}\ v_2,\ s_2,\ k\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle\,. \tag{5.16}$$

By the (call) rule, the reductions (5.12), (5.14) and (5.15) lead to the reduction (2a). We conclude with the reduction (5.16), which is the expected reduction (2b).

3. If M is a tail call of the form $f(e_1, \ldots, e_n)$, then $M^\star$ is **push** $f^\star(e_1, \ldots, e_n)$; **invoke** and we want to show that there is a value v_2 and a store s_2 such that

$$\langle f(e_1, \ldots, e_n),\ s_0\rangle \xrightarrow{\rho} \langle v_2,\ s_2\rangle$$

$$\langle \textbf{invoke}\ v_2,\ s_2,\ k\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle\,.$$

By the (seq), (push) and (invoke) rules, the reduction (1) yields

$$\forall i, \langle e_i,\ s_0\rangle \xrightarrow{\rho} \langle w_i,\ s_0\rangle$$

$$\langle f(w_1, \ldots, w_n),\ s_0,\ k\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle\,.$$

By the induction hypotheses, there is a value v_2 and a store s_2 such that

$$\langle f(w_1, \ldots, w_n),\ s_0\rangle \xrightarrow{\rho} \langle v_2,\ s_2\rangle$$

$$\langle \textbf{invoke}\ v_2,\ s_2,\ k\rangle \xrightarrow{\rho} \langle v_1,\ s_1,\ \varepsilon\rangle\,.$$

The conclusion follows by the (call) rule.

4. If M is an assignment or a conditional, immediate by the induction hypotheses. □

Chapter 6

Experimental results

Translating threaded code into event-driven style is useful, regardless of performance, when the target system or language does not support threads at all. But it is even more interesting if the generated code is able to retain most of the space and time efficiency associated with hand-written event-driven code. This chapter presents the results of our experiments to assess the efficiency of CPC programs, from micro-benchmarks of individual CPC primitives to a global evaluation of Hekate's performance.

The CPC language provides no more than half a dozen 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. In Section 6.1, we show the results of benchmarking individual CPC primitives against other thread libraries. In these benchmarks, CPC turns out to be comparable to the fastest thread libraries available to us.

In the absence of reliable information on the relative dynamic frequency of CPS function calls and thread synchronisation primitives in CPC programs, it is not clear what the performance of individual primitives implies about the performance of a complete program. In Section 6.2, we present the results of benchmarking a set of naive web servers written in CPC and in a number of thread libraries, as well as a set of reference full-fledged web servers.

Finally, in Section 6.3, we present a number of insights that we gained by working with Hekate, our BitTorrent seeder written in CPC.

6.1 CPC threads and primitives

In this section, we measure the space utilisation of CPC threads (Section 6.1.1) and the efficiency of CPC primitives (Section 6.1.2). We compare CPC with the following thread libraries: *nptl*, the native thread library in GNU libc 2.13 (or μClibc 0.9.32 for our tests on embedded hardware) [DM05]; *GNU Pth* version 2.0.7 [Eng06; Eng00]; *State Threads (ST)* version 1.9 [SA06]. *Nptl* is a kernel thread library, while *GNU Pth* and *ST* are cooperative user-space thread libraries.

Table 6.1: Number of threads possible with 4 GB of memory

Library	Number of threads
nptl	32 330
Pth	700 000 (est.) [a]
ST	934 600
ST (4 kB stacks)	961 400
CPC	50 190 000

[a] *Pth* never completed because thread creation rate decreases as the number of threads increases: 6 800 threads in 1 s, 9 800 in 2 s, 21 500 in 10 s and only 30 900 in 60 s. The given value is an estimation based on the average memory used per thread and the total amount of available memory.

6.1.1 Space utilisation

On a 64-bit machine with 4 GB of physical memory and no swap space, CPC can handle up to 50 million simultaneous threads, which implies an average memory usage of roughly 82 bytes per continuation. This figure compares very favourably to both kernel and user-space thread libraries (see Table 6.1), 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.

This benchmark consists in creating as many threads as possible until the machine runs out of memory. The fact that CPC is able to create an order of magnitude more threads than the other thread libraries is mainly due to the fact that it allocates small continuations by default and resizes them as needed; a CPC program might be limited to less threads in the same conditions if many of them have a deep call stack. Remember however that one of the strengths of CPC is its ability to create a lot of very short-lived threads with an almost flat call stack (see for instance the implementation of timeouts in Section 2.3.2). Some thread libraries designed for embedded operating systems, such as Nano-Qplus, also provide resizable shared-stack threads [Gu+07] but we could not include them in our benchmark since they run on hardware we did not have access to.

6.1.2 Speed of CPC primitives

Table 6.2 on the facing page presents timings of CPC primitives on three different processors: the *Core 2 Duo*, a superscalar out-of-order 64-bit processor, the *Pentium-M*, a slightly less advanced out-of-order 32-bit processor, and the *MIPS 4Kc*, a simple in-order 32-bit RISC chip with a short pipeline.

Table 6.2: Speed of thread primitives on various architectures

Library	loop	call[a]	CPS-call	switch	cond	spawn
nptl (1 core)	2	2		439	1 318	4 000 000 [b]
nptl (2 cores)	2	2		135	1 791	11 037
Pth	2	2		2 130	2 602	6 940
ST	2	2		170	25	411
CPC	2	2	20	16	36	74
eCPC [c]	2	2	49	27	41	162

x86-64: Intel Core 2 Duo at 3.17 GHz, Linux 2.6.39

Library	loop	call	CPS-call	switch	cond	spawn
nptl	2	4		691	2 426	24 575
ST	2	4		1 003	96	2 293
CPC	2	4	54	46	91	205
eCPC [c]	2	4	120	77	101	626

x86-32: Intel Pentium M at 1.7 GHz, Linux 2.6.38

Library	loop	call	CPS-call	switch	cond	spawn
nptl	58	63		6 310	63 305	933 689
CPC	58	63	2 018	1 482	3 268	8 519
eCPC [c]	58	63	5 888	2 119	2 962	13 544

MIPS-32: MIPS 4KEc at 184 MHz, Linux 2.6.37

All times are in nanoseconds per loop iteration, averaged over millions of runs (smaller is better). The columns are as follows:
loop: time per loop iteration (reference);
call: time per loop iteration calling a direct-style function;
CPS-call: time per loop iteration calling a CPS-converted function;
switch: context switch;
cond: context switch on a condition variable;
spawn: thread creation.

[a] On x86-64, a direct-style function call adds no measurable delay.

[b] The behaviour of the system call `sched_yield` has changed in Linux 2.6.23, where it has the side-effect of reducing the priority of the calling thread. Until Linux 2.6.38, setting the kernel variable `sched_compat_yield` restored the previous behaviour, which we have done in our x86-32 and MIPS-32 benchmarks. This knob having been removed in Linux 2.6.39, the *nptl* results on x86-64 are highly suspicious.

[c] The eCPC translator is an alternative implementation of CPC using environments instead of lambda lifting (Chapter 7). The benchmark results are discussed in Section 7.2.2.

Function calls Modern processors include specialised hardware for optimising function calls; obviously, this hardware is not able to optimise CPS calls, which are therefore dramatically slower than direct-style calls. In our very simple benchmark (two embedded loops containing a function call), the *Core 2 Duo* was able to completely hide the cost of a trivial function call[1], while the function call costs three processor cycles on the *Pentium-M* and just one on the MIPS processor; we do not claim to understand the hardware-level magic that makes such results possible.

With CPS calls, on the other hand, our benchmarking loop takes on the order of 100 processor cycles per iteration (400 on MIPS). This apparently poor result is mitigated by the fact that after goto elimination and lambda lifting, the loop consists of four CPS function calls; hence, on the *Pentium-M*, a CPS function call is just 6 times slower than a direct-style call, a surprisingly good result considering that our continuations are allocated on the heap [MR94]. On the MIPS chip the slowdown is closer to a factor of 100, more in line with what we expected.

Context switches and thread creation The "real" primitives are pleasantly fast in CPC. Context switches have similar cost to function calls, which is not surprising since they compile to essentially the same code. While the benchmarks make them appear to be much faster than in any other implementation, including *ST*, this result is skewed by a flaw in the *ST* library, which goes through the event loop and performs a system call on every `st_yield` operation. Context switches on a condition variable are roughly two times slower than pure context switches, which yields timings comparable to those of *ST*.

Thread creation is similarly fast, since it consists in simply allocating a new continuation and registering it with the event loop. This yields results that are ten times faster than *ST*, which must allocate a new stack, and a hundred times faster than the kernel implementation.

6.2 Web-server comparison

As we have seen above, the speed of CPC primitives ranges from similar to dramatically faster than that of their equivalents in traditional implementations. However, a CPC program incurs the overhead of the CPS translation, which introduces CPS function calls which are much slower than their direct-style equivalents. As it is difficult to predict the number of CPS calls in a CPC program, it is not clear whether this performance increase will be reflected in realistic programs.

Our goal being to compare implementations of concurrency, rather than to provide realistic benchmarks of concurrent programs, we have chosen to use a simple, well-understood and repeatable benchmark rather than a realistic one. We have benchmarked a number of HTTP servers written in different programming styles, serving a single short (305 bytes) page to varying numbers of simultaneous clients. The servers measured include on the one hand a

[1] Disassembly of the binary shows that the function call is present.

number of high-quality production servers, and on the other hand a number of "toy" servers written for this exercise.

In Section 6.2.1 we describe the experiment that we performed, as well as the moderate amount of tuning we had to apply to our setup in order to obtain reproducible results. In Section 6.2.2 we describe the web servers that we have measured. In Section 6.2.3 we give the experimental results that we obtained, and discuss them in detail.

6.2.1 Experimental approach

We used a simple, easily reproducible setup which would show how well a number of web server implementations handle multiple concurrent requests. Achieving reproducibility is not as straightforward as it might seem, however, because benchmarking a web server involves an impressive number of parameters, each of them introducing a potential bottleneck and altering the measurements.

Experimental setup The server is a Pentium-M laptop downclocked to 600 MHz to ensure that it is slower than the client. CPU usage was close to 100 % during all of the tests. The client is a standard workstation using an AMD Athlon 64 at 2.2 GHz, with power-saving features turned off. The server and the client were running Linux kernel 2.6.24. In none of our tests did the CPU usage rise above 20 %. Both machines have Gigabit Ethernet interfaces, and were connected through a dedicated Gigabit Ethernet switch. We used the standard Ethernet MTU of 1500 bytes.

We have used the *ApacheBench* client, included with Apache 2.2, which is designed to generate a constant load on a web server. Given a parameter l known as the *concurrency level* or simply *load*, ApacheBench tries to generate a self-clocking stream of requests timed so that the number of open connections at any given time is exactly l.

In practice, however, we have found that ApacheBench needs a period of time to "ramp up"; for this reason, we patched ApacheBench to display a detailed timing of every request and discarded the first and last 1,000 samples from our measurements.

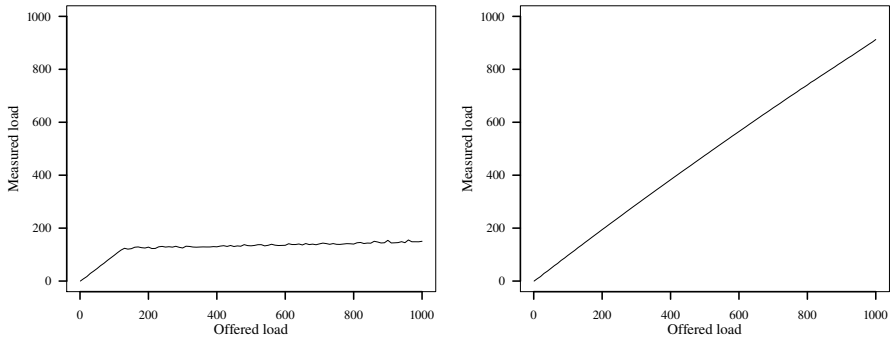
Tuning We were rather surprised by our first batch of experimental data. Below 128 simultaneous clients, the median and the average latency coincided and were linear in the number of simultaneous clients, just as expected. Above that threshold, however, the median latency remained at a constant value of 50 ms, while the average value became irregular and irreproducible. These results indicated a small number of extreme outliers—requests that were taking 200 ms or more to serve.

In order to understand the phenomenon, we came up with the notion of *measured load* of a benchmark run. Consider an ideal run, in which there is no idle time: the number of in-flight requests at any given time is exactly l. Writing t for the total run time of the test, n the total number of requests serviced, and t_r the average servicing time for a request, we have $t = n \cdot t_r / l$. We therefore define the *measured load* l_m as $l_m = n \cdot t_r / t$; this value is always

lower than the desired load l, and we would hope that it is very close to l in a run in which the client is able to saturate the server.

Plotting the measured load against the offered load (Fig. 6.1a) nevertheless showed us that, however large the offered load, the effective load never exceeded roughly 130; obviously, something in our setup was limiting the number of connections to 130.

Figure 6.1: Measured load against offered load, before and after tuning



(a) somaxconn = 128 (b) somaxconn = 1024

It turned out that the bottleneck was the kernel variable somaxconn, the value of which defaults to 128 [BD99]. The `listen` system call, which is used to establish a passive ("listening") socket, takes a parameter that indicates the length of the associated "accept queue"; when this queue becomes full, e.g. because the server does not accept connections fast enough, new requests are discarded, and will be resent by the client after a timeout. The variable somaxconn limits the size of the accept queue: the parameter to `listen` is silently limited to the value of somaxconn.

Raising somaxconn to 1024 solves the problem as shown on Fig. 6.1b.

Other potential bottlenecks In order to ensure that our results apply more generally and are not specific to our particular setup, we repeated our benchmarks while varying other parameters, and found that they had no significant impact on the results. In particular, using different network cards and removing the switch between the client and the server yielded no measurable difference—hence, no hardware queues were being overflown. Using different computers (a faster client, a slower server) yielded slightly different figures, but did not change the general conclusions. Finally, preloading the served file into memory and changing its size (up to 100 kB) only caused a very slight additive difference.

6.2.2 Benchmarked implementations

We benchmarked four production web servers, and a set of "toy" web servers that were written for this particular purpose.

Full-fledged web servers Apache [Apa08] is the most widely deployed web server in the Internet today; hence, a benchmark of web servers must include it as a comparison point. One of the claimed advantages of Apache 2 is its ability to run with different concurrency models; we measured two of Apache's concurrency models, the process-pool model *(prefork)* and the thread-pool model *(worker)*.

Thttpd [Pos03] is a small event-driven server which was considered as one of the fastest open-source web servers in the late 1990s. It uses a standard event-driven model, with one minor twist: connections are accepted eagerly, and kept in a user-space queue of accepted connections until they are serviced.

Polipo [Chr08] is an HTTP proxy written by Chroboczek that can also serve as a web server. It uses a fairly standard event-driven model.

Lighttpd [Kne08] is more recent, highly optimised event-driven web server.

We used Apache 2.2, Thttpd 2.25b, Polipo 1.0.4 and Lighttpd 1.4.19.

Toy servers We have written a set of toy 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 one is accepted; our servers do not use any clever implementation techniques, such as thread pools. Because of this simple structure, these servers can be directly compared, and we are able to benchmark the underlying implementation of concurrency rather than the implementation of the web server.

One of these web servers uses heavyweight Unix processes, created using the `fork` system call. One is written using *NPTL*, the native thread library used in Linux version 2.6. Two are written using standard user-space thread libraries, called respectively *Pth* [Eng06] and *ST* [SA06]. We used the versions ST 1.7 and Pth 2.0.7. Finally one uses *CPC* [CK12].

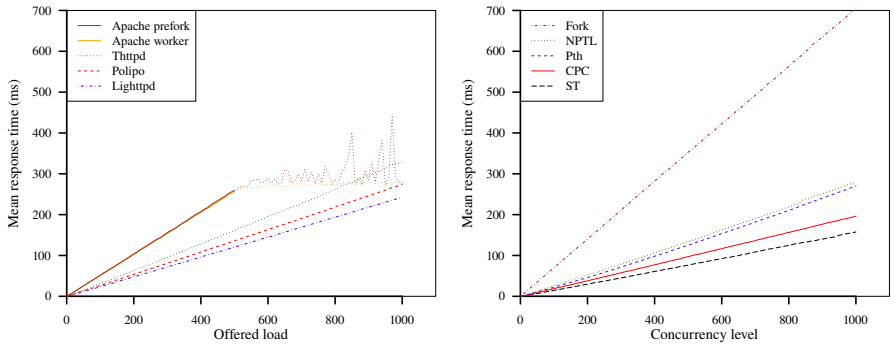
6.2.3 Experimental results

Figure 6.2 presents the results of our experiment. It plots the average serving time per request against the desired load; a smaller slope indicates a faster server. Figure 6.2a presents full-fledged production servers, while Fig. 6.2b presents the set of toy servers. With the exception of Apache, the curves are extremely regular (in each case, the correlation coefficient between the mean response time and the offered load exceeds 0.999).

Discussion Apache artificially limits the size of the accept queue to 512; hence, its results for more than 512 simultaneous requests should not be taken into account. Apache turns out to be the slowest amongst the production servers that we considered; moreover, we found that the process-pool *(prefork)* and the thread-pool *(worker)* implementations performed equally poorly.

All three event-driven production servers were significantly faster than Apache, and their performance was roughly similar. Thttpd was somewhat slower than Polipo, and Lighttpd very slightly faster; we believe that the discrepancy is due to different amounts of micro-optimisation rather than to any fundamental difference between the three servers. Incidentally,

Figure 6.2: Web servers comparison



(a) Full-fledged web servers

(b) Toy web servers

results when the accept queue did overflow (not shown) were much more regular for thttpd than for the other servers, which shows the effect of a user-space accept queue.

The production servers were generally slower than the toy servers, because they do more than just serving pages, like performing security checks, checking for changes in user configuration, etc.

The implementation using full-fledged processes, created using the `fork` system call, was slower than any other of the implementations that we benchmarked; hence, even in a modern virtual-memory Unix system, there is a non-negligible cost associated with creating many short-lived heavyweight processes.

The version implemented using NPTL, the native (kernel) threads of the Linux operating system, turned out to be surprisingly fast, especially when compared to the version using `fork`, which indicates that the poor performance of the latter is due to virtual memory rather than scheduling overhead. Its performance is unexpectedly close to that of the poorer user-space thread libraries.

The user-space threading libraries, Pth and ST, behaved quite differently. Pth yielded results similar to those of NPTL, while ST yielded excellent results; a cursory examination of ST's sources indicates that it uses some rather clever data structures—such as heaps for timeout events—and a number of micro-optimisations, some of which could easily be reused in other user-space implementations of threads to improve their performance.

Finally, the version implemented using *CPC* gave results that were better than all of the other implementations save that implemented using ST: CPC took 25 % more time than ST. Since the implementation techniques of ST and CPC are very different, there is no obvious single explanation to the slightly better results of ST; in this particular benchmark, the overhead of CPS function calls in CPC probably exceeds the cost of allocating and context-switching call stacks in ST.

This benchmark shows that, despite using high-level source-to-source transformations, CPC produces code that is only 25 % slower than the fastest thread library available to us, and faster than every other while keeping a much smaller memory footprint. It strongly suggests that the techniques used by CPC are efficient enough to be used in the development of large, real-world software.

6.3 Hekate

To complete our evaluation of CPC's performance, we decided to measure the efficiency of Hekate (Section 2.1). Benchmarking a BitTorrent seeder is a difficult task because it relies either on a real-world load, which is hard to control and only provides seeder-side information, or on an artificial testbed, which might fail to accurately reproduce real-world behaviour. We have however been able to collect enough empirical evidence to show that Hekate is an efficient implementation of BitTorrent, lightweight enough to be run on embedded hardware.

Real-world workload To benchmark the ability of Hekate to sustain a real-world load, we need popular torrents with many requesting peers over a long period of time. Updates for Blizzard's game *World of Warcraft* (WoW), distributed over BitTorrent, meet these conditions: each of the millions of WoW players around the world runs a hidden BitTorrent client, and at any time many of them are looking for the latest update.

We have run an instance of Hekate seeding WoW updates without interruption for weeks. We saw up to 1,000 connected peers (800 on average) and a throughput of up to 10 MB/s (around 5 MB/s on average). Hekate never used more than 10 % of the 3.16 GHz dual core CPU of our benchmarking machine, and was bottlenecked either by the available bandwidth during peaks of requests (10 MB/s), or by the mere fact that we did not gather enough peers to saturate the link. Unfortunately, a few weeks later, Blizzard switched to a modified, incompatible version of BitTorrent and we have not been able to find such a fruitful source of peers ever since.

Stress-test on embedded hardware We have ported Hekate to *OpenWrt*[2], a Linux distribution for embedded devices. Hekate runs flawlessly on an Asus WL-500G Premium, a cheap home router with a 266 MHz *MIPS 4Kc* CPU, 32 MB of RAM and a 100 Mbps network card. The torrent files were kept on a USB flash drive. Because Hekate maps every served file into memory using the `mmap` system call, we were limited to 2 GB of data on this 32-bit architecture; to simplify the benchmark, we decided to serve a single large torrent to many clients.

Our stress-test consists in 1,000 simultaneous clients (implemented as a single CPC program[3] running on a fast computer directly connected to the device over 100 Mbps Ethernet) requesting random chunks of a 1.2 GB torrent. In these conditions, Hekate was able to serve data at a rate of 2.9 MB/s. The CPU load was pegged at 100%, with most of the CPU time (60 %) spent servicing software interrupt requests, notably from the USB subsystem (the usb-storage kernel module using up to 25 % of CPU). These data indicate that even on a slow MIPS processor, Hekate's performance is fairly good, and that the performance could be much improved by using a device on which mass storage traffic does not go over the USB bus.

This benchmark confirms our intuition that the CPC compilation technique is well-suited to the development of lightweight concurrent programs, in particular network servers for

[2] `http://openwrt.org/`.

[3] We wrote a modified version of Hekate to act as a random client instead of a seeder.

embedded systems. It is also consistent with the measurements of the speed of CPC primitives on a MIPS architecture reported in Section 6.1.2.

Chapter 7

Alternative event-driven styles

This chapter is joint work with Matthieu Boutier [Bou11].

Not all event-driven programs look the same; several styles and implementations exist, depending on the programmer's taste. Since event-driven programming consist in handling manually the control flow and data flow of each task, a tedious and error-prone activity, the programmer often choses a style based on some trade-off between (his idea of) efficiency and code-readability, and then sticks with it in the whole program. Even if the representation of control or data turns out to be suboptimal, changing it requires a complete refactoring of the program, not likely to be undertaken for an uncertain performance gain.

The CPC translator, for instance, produces only one specific kind of event-driven code: data flow is handled with lambda lifting, and control flow with CPS conversion. We made these choices because of our intuitions about cache-efficiency, and because of the challenge presented by applying these well-known techniques in the hostile context of the C language. Most hand-written programs, however, use simpler approaches; an automatic translation tool such as CPC offers the opportunity to generate several variants of the same program and compare their efficiency, provided they can be expressed in terms of source-to-source program transformations.

We saw in Section 1.2 several examples of event-driven styles found in real-world programs. We distinguished two typical kinds of control-flow and data-flow encodings: callbacks and state machines for the flow of control (Section 1.2.1), coarse-grained and minimal data structures for the data flow (Section 1.2.2). In this chapter, we propose automatic translation steps to generate these various styles from a threaded description: defunctionalisation for the control flow, and shared environments for the data flow (Section 7.1). We then detail the implementation of eCPC, a variant of CPC using environments instead of lambda lifting to handle the data flow in the generated programs, and compare its efficiency to the original CPC translator (Section 7.2).

7.1 Generating alternative event-driven styles

As we have seen in Section 3.4, the existing translation from a program with CPC threads to an event-driven C program starts by a splitting pass. It yields a program where each CPS

function is split in several mutually recursive, atomic functions, very similar to event handlers. Remember that the tail positions of these inner functions might only be:

- a `return` statement,
- a tail call to another (external or inner) CPS function,
- a call to an external CPS function followed by a call to an inner CPS function.

We recognise the typical patterns of an event-driven program that we studied in Section 1.2: returning a value to the upper layer (Fig. 1.10 (4)), calling a function to carry on the current computation (Fig. 1.11 (1)), or calling a function with a callback to resume the computation once it has returned (Fig. 1.10 (2)).

The CPC translator then makes the data flow explicit with a lambda-lifting pass (Section 3.5). Lambda lifting yields a program where the data is copied from function to function, each copy living as long as the associated handler. If some piece of data is no longer needed during the computation, it will not be copied in the subsequent handlers: lambda lifting produces short-lived, almost minimal data structures (as mentioned previously, uninitialised data is not treated as efficiently).

Finally, the control flow is made explicit with a CPS conversion (Section 3.2). The continuations store callbacks and their parameters in a regular stack-like structure. It implements as efficiently and systematically as possible the layered callback scheme described in the previous section.

In this section, we propose two alternative translation passes to produce different event-driven styles. We show that *defunctionalising* inner function yields state machines (Section 7.1.1), and encapsulating local variables in *shared environments* yields larger, heap-allocated, long-lived data structures with full context (Section 7.1.2).

7.1.1 Defunctionalising inner functions

Defunctionalisation is a compilation technique introduced by Reynolds to translate higher-order programs into first-order ones [Rey72]. It maps every first-class function to a first-order structure that contains both an index representing the function, and the values of its free variables. These data structures are usually a constructor, whose parameters store the free variables. Function calls are then performed by a dedicated function that dispatches on the constructor, restores the content of the free variables and executes the code of the relevant function.

Consider the following small ML program, an example of defunctionalisation by Danvy and Nielsen [DN01, Section 1.1].

```
(* aux : (int -> int) -> int *)
fun aux f
    = f 1 + f 10

(* main : int * int * bool -> int *)
fun main (x, y, b)
    = aux (fn z => x + z) *
      aux (fn z => if b then y + z else y - z)
```

This program contains two first-class functions, `fn z => x + z` and `fn z => if b then y + z else y - z`, both of them applied to the function `aux`. To translate this higher-order program into a first-order one, we replace these two function abstractions by two constructors, respectively `LAM1` and `LAM2`, and a dispatch function `apply` that executes the code of the proper abstraction depending on its parameter.

Figure 7.1: Defunctionalisation

```
datatype lam = LAM1 of int
             | LAM2 of int * bool

(* apply : lam * int -> int *)
fun apply (LAM1 x, z)
    = x + z
  | apply (LAM2 (y, b), z)
    = if b then y + z else y - z

(* aux_def : lam -> int *)
fun aux_def f
    = apply (f, 1) + apply (f, 10)

(* main_def : int * int * bool -> int *)
fun main_def (x, y, b)
    = aux_def (LAM1 x) * aux_def (LAM2 (y, b))
```

The functions `aux` and `main` are modified accordingly to use `apply` instead of invoking the abstractions.

The dispatch function introduced by defunctionalisation is very close to a state automaton. It is therefore not surprising that defunctionalising inner functions in CPC yields an event-driven style similar to state machines presented in Section 1.2.1.

Defunctionalisation of CPC programs Usually, defunctionalisation contains an implicit lambda-lifting pass, to make free variables explicit and store them in constructors. For example, in Fig. 7.1, the constructor `LAM1` carries a copy of the free variable `x`.

In this discussion, we wish to decouple this data-flow transformation from the translation of the control flow into a state machine. Therefore, we define the dispatch function as an inner function which merges the content of the other inner functions but still contains free variables. This is possible because the splitting pass does not create any closure: it introduces inner functions with free variables, but these are always called directly, not stored as first-class values whose free variables must be captured.

Consider the following function, which counts down and finally prints `BOOM`.

Figure 7.2: Count-down in threaded style

```
cps int countdown(int x) {
    cps int count() {
        if(x == 0) return boom();
        printf("%d\n", x--);
        cpc_sleep(1);
        return count();
    }
    cps int boom() {
```

```
        printf("BOOM!\n");
        return -1;
    }
    if(x < 0)
        return boom();
    else
        return count();
}
```

Once defunctionalised, it contains a single inner function `dispatch` that dispatches on an enumeration representing the former inner function `count` and `boom`.

Figure 7.3: Defunctionalised count-down

```
enum state { COUNT, BOOM };

cps int countdown(int x) {
    cps int dispatch(enum state s) {
        switch(s) {
            case COUNT:
            {
                if(x == 0) return dispatch(BOOM);
                printf("%d\n", x--);
                cpc_sleep(1);
                return dispatch(COUNT);
            }
            case BOOM:
            {
                printf("BOOM!\n");
                return -1;
            }
        }
    }

    if(x < 0)
        return dispatch(COUNT);
    else
        return dispatch(BOOM);
}
```

As an optimisation, the first tail recursive call to `dispatch` can be replaced by a `goto` statement. However, we cannot replace the call that follows the CPS function `cpc_sleep(1)`, because we will need to provide `dispatch` as a callback to `cpc_sleep` during CPS conversion, to avoid blocking.

Figure 7.4: Optimisation of tail calls to `dispatch`

```
cps int countdown(int x) {
    cps int dispatch(enum state s) {
        switch(s) {
            case COUNT: COUNT_LABEL:
            {
                if(x == 0) goto boom_label;
                printf("%d\n", x--);
                cpc_sleep(1);
                return dispatch(COUNT);
```

```
            }
            case BOOM: boom_label:
            {
                printf("BOOM!\n");
                return -1;
            }
        }
    }

    if(x < 0)
        return dispatch(COUNT);
    else
        return dispatch(BOOM);
}
```

We must then eliminate free variables, with either lambda lifting or shared environments (Section 7.1.2). Note that lambda lifting is correct because defunctionalisation does not break the required invariant on tail calls.

Figure 7.5: Lifted dispatch loop

```
cps int dispatch(enum state s, x) {
    switch(s) {
        case COUNT: COUNT_LABEL:
        {
            if(x == 0) goto boom_label;
            printf("%d\n", x--);
            cpc_sleep(1);
            return dispatch(COUNT, x);
        }
        case BOOM: boom_label:
        {
            printf("BOOM!\n");
            return -1;
        }
    }
}
cps int countdown(int x) {
    if(x < 0)
        return dispatch(COUNT, x);
    else
        return dispatch(BOOM, x);
}
```

We reach code that is similar in style to the state machine shown in Fig. 1.11.

If we ignore the `switch`—which serves mainly as an entry point to the dispatch function, à la Duff's device [Duf83]—we recognise the intermediate code generated during the first step of splitting, as having an explicit control flow using gotos but without inner functions (Section 3.4.1). In retrospect, the second step of splitting, which translates gotos to inner functions (Section 3.4.2), can be considered as a form a *refunctionalisation*, the left inverse of defunctionalisation [DN01].

An actual implementation could merge the splitting and defunctionalisation passes to limit the number of code transformations and generate a state automaton directly from the threaded

description. Without an efficient liveness analysis pass, it is however more convenient to keep two distinct passes, with a percolating optimisation after splitting to identify the temporary variables that can be made local to the dispatching function. For instance, the Tame framework for writing events in C++ generates a state automaton directly from threaded style, but it does not handle free variables automatically: the programmer is responsible for choosing which variables must be saved and restored between context switches [KKK07].

Benefits The translation presented here is in fact a *partial* defunctionalisation: each CPS function in the original program gets its own dispatch function, and only inner functions are defunctionalised. A global defunctionalisation would imply a whole program analysis, would break modular compilation, and would probably not be very efficient because C compilers are optimised to compile hand-written, reasonably-sized functions rather than a giant state automaton with hundreds of states. On the other hand, since it is only partial, this translation does not eliminate the need for a subsequent CPS conversion step to translate calls to external CPS functions into operations on continuations.

Despite adding a new translation step while keeping the final CPS conversion, this approach has several advantages over the CPS conversion of many smaller, mutually recursive functions performed by the current CPC translator. First, we do not pay the cost of a CPS call for inner functions. This might bring significant speed-ups in the case of tight loops or complex control flows. Moreover, it leaves many more optimisation opportunities for the C compiler, for instance to store certain variables in registers, and reduces the number of operations on the continuations. It also makes debugging easier, avoiding numerous hops through ancillary CPS functions.

Return values If one studies the previous example carefully, it turns out that it is not in CPS-convertible form. More precisely, this fragment is not a valid tail:

```
cpc_sleep(1); return dispatch(COUNT, x);
```

because the integer value returned by `cpc_sleep` is not passed to `dispatch`. It should have the following form:

```
int ignored = cpc_sleep(1); return dispatch(COUNT, x, ignored);
```

But consider what happens when more external CPS functions are called in `dispatch`: it becomes impossible to keep a single dispatching function that receives return values as its last parameter. Our CPS conversion does not work in that case: the dispatching function alone does not represent the continuation, and as such cannot receive the result of the previous computation directly; just like the state of the automaton, the return value must be passed by another means.

This issue can be solved by adding another layer of indirection.[1] The solution is to pass to every CPS function a pointer containing the address where it should write its return value; the

[1] Not surprisingly, since "any problem in computer science can be solved by another level of indirection" (attributed to David Wheeler by Butler Lampson during his Turing Award Lecture [Lam93]).

caller (`dispatch`) is responsible for allocating the required memory, and to pass its address to the callee (`cpc_sleep` in our example above). This technique, which is also used in eCPC, is described in greater detail in Section 7.2.

7.1.2 Shared environments

There are two main compilation techniques to handle free variables: lambda lifting, already discussed extensively in Chapter 4, and *environments*. An environment is a data structure used to capture every free variable of a first-class function when it is defined; when the function is later applied, it accesses its variables through the environment. Environments add a layer of indirection, but contrary to lambda lifting they do not require free variables to be copied on every function call.

In most functional languages, each environment represents the free variables of a single function; a pair of a function pointer and its environment is called a closure. However, nothing prevents in principle an environment from being *shared* between several functions, provided they have the same free variables. We use this technique to allocate a single environment shared by inner functions, containing all local variables and function parameters.

An example of shared environments Consider again the countdown example from the previous section (Fig. 7.2 on page 151).

```
cps int countdown(int x) {
    cps int count() {
        if(x == 0) return boom();
        printf("%d\n", x--);
        cpc_sleep(1); return count();
    }
    cps int boom() {
        printf("BOOM!\n");
        return -1;
    }
    if(x < 0) return boom();
    else      return count();
}
```

We introduce an environment to contain the local variables of `countdown` (only `x` here).

Figure 7.6: Environments to save local variables

```
struct env_countdown { int x };

cps int countdown(int x) {
    struct env_countdown *e =
          malloc(sizeof(struct env_countdown));          /* 1 */
    e->x = x;                                            /* 2 */

    cps int count(struct env_countdown *e) {
        if(e->x == 0) return boom(e);                    /* 3 */
        printf("%d\n", e->x--);                          /* 4 */
        cpc_sleep(1); return count(e);                   /* 5 */
```

```
    }
    cps int boom(struct env_countdown *e) {
        printf("BOOM!\n");
        free(e);                                          /* 6 */
        return -1;
    }
    if(x < 0) return boom(e);
    else      return count(e);
}
```

The environment is allocated (1) and initialised (2) when the function `countdown` is entered. The inner functions access `x` through the environment, either to read (3) or to write it (4). A pointer to the environment is passed from function to function (5); hence all inner functions share the same environment. Finally, the environment is deallocated just before the last inner function exits (6). In fact, environments are very similar to boxing (Section 3.3), but applied to every local variable instead of a subset of them only.

The resulting code is similar in style to hand-written event-driven code, with a single, heap-allocated data structure sharing the local state between a set of callbacks. Note that inner functions have no remaining free variables and can therefore be lambda-lifted trivially.

Benefits Encapsulating local variables in environments avoids having to copy them back and forth between the continuation and the native call stack. However, it does not necessarily mean that the generated programs are faster; in fact, lambda-lifted programs are often more efficient (Section 7.2). Another advantage of environments is that they make programs easier to debug, because the local state is always fully available, whereas in a lambda-lifted program "useless" variables are discarded as soon as possible, even though they might be useful to understand what went wrong before the program crashed.

7.2 eCPC

In this section, we describe the implementation of eCPC, a CPC variant using shared environments instead of lambda lifting to encapsulate the local state of CPS functions. We also compare the efficiency of programs generated with eCPC and CPC. The eCPC translator has been designed and implemented together with Matthieu Boutier, and this section is mostly based on the material of his undergraduate thesis [Bou11].

7.2.1 Translation passes

One of the design goal of eCPC was to reuse as much of the existing CPC infrastructure as possible. The eCPC translator consists of the following passes:

- *Lambda lifting*: preliminary pass to remove user-defined inner functions (unchanged);
- *Preparing environments*: creation of an environment pointer, deallocation of environments on return points and introduction of a pointer for return values;

- *Splitting*: the flow of control of each CPS function is split into a set of mutually recursive, tail-called, inner functions (unchanged);
- *Generating environments*: creation, allocation and initialisation of environments; local variables are then accessed through the environment pointer and removed from inner functions;
- *Lambda lifting*: this pass becomes a trivial block floating step, since every inner function is already closed (unchanged);
- *CPS conversion*: last pass to produce the final event-driven program (unchanged).

It introduces two new passes, the first of which replaces boxing, and requires only limited changes to the runtime system, to handle return values. The optimisation passes are performed as usual (percolating after splitting, inlining after the second lambda lifting).

Preparing environments Environments must be introduced before the splitting pass for two reasons. First, it is easier to identify the exit points of CPS functions, where the environments must be deallocated, before they are split into multiple, mutually recursive, inner functions. Furthermore, these environment deallocations occur in tail position, and have therefore an impact on the splitting pass itself.

Although deallocation points are introduced before splitting, neither allocation nor initialisation or indirect memory accesses are performed at this stage. Environments introduced during this preparatory pass are empty shells, of type `void*`, that merely serve to mark the deallocation points. This is necessary because not all temporary variables have been introduced at this stage (the splitting pass will generate more of them), and the percolating optimisation cannot be performed before the splitting pass. Deciding which variables will be stored in environments is delayed to a later pass (Fig. 7.7, function `g`).

This preparatory pass also needs to modify how return values are handled. Recall that, in CPC, return values are written directly in the continuation when the returning function invokes its continuation. This is made possible by the convention that the return value of a CPS function is the last parameter of its continuation, hence at a fixed position on the continuation stack. Such is not the case in eCPC, where function parameters are kept in the environment rather than copied into the continuation.

In eCPC, the caller function passes a pointer to the callee, indicating the address where the callee must write its return value. The preparatory pass transforms every CPS function returning a type T (different from `void`) into a function returning `void` with an additional parameter of type $T*$; call and return points are modified accordingly (Fig. 7.7, function `f`). The implementation of CPC primitives in the CPC runtime is also modified to reflect this change.

Generating environments After the splitting and percolating pass, the eCPC translator allocates and initialises environments, and replaces variables by their counterpart in the environment.

First, it collects local variables (except the environment pointer itself) and function parameters and generates the layout of the associated environment. Then, it allocates the environment

Figure 7.7: Environments in eCPC (first step)

Initial code

```
cps int f(int x)
{
    int tmp;
    tmp = x + 3;
    return tmp;
}
```

```
cps void g(void)
{
    int x;
    x = f(4);
    printf ("%d\n", x);
    return;
}
```

Preparatory pass

```
cps void f(int *retval, int x)
{
    void *env; int tmp;
    tmp = x + 3;
    if (retval != NULL)
        *retval = tmp;
    free(env); return;
}
```

```
cps void g(void)
{
    void *env; int x;
    f(&x, 4);
    printf ("%d\n", x);
    free(env); return;
}
```

and initialises the fields corresponding to the function parameters. Because this initialisation is done at the very beginning of the translated function, it does not affect the tails, thus preserving CPS-convertibility. Finally, every use of variables is replaced by its counterpart in the environment, and local variables are discarded.

Figure 7.8 shows the same example than Fig. 7.7, after splitting and generation of environments. The translation is not optimal: environments could be avoided when there is no inner function, or when a variable is not used in any of the inner functions (for instance `tmp` in the function `f`).

Note further that inner functions are modified to receive the environment as a parameter. This last bit of transformation is not strictly necessary, since the following lambda-lifting pass would do it just as well, but it was trivial to implement at this stage and we preferred a self-contained pass for environments. As a result, the lambda-lifting pass consists in trivially floating closed functions to top-level. The CPS conversion is also kept unchanged: the issue of return values has been dealt with completely in the preparatory pass and every CPS function returns `void` at this stage.

7.2.2 Benchmark results

To compare CPC and eCPC, we ran the set of benchmarks that we formerly used to compare CPC to other thread libraries. The experimental setup and the complete results of each benchmark have been detailed in Chapter 6. In this section, we focus on the relative speed of CPC and eCPC, on various architectures.

Split functions

```
cps void f(int *retval, int x)
{
    void *env; int tmp;
    tmp = x + 3;
    if (retval != NULL)
        *retval = tmp;
    free(env); return;
}
```

```
cps void g(void)
{
    void *env; int x;
    f(&x, 4);
    g_aux(); return;

    cps void
    g_aux(void)
    {
        printf ("%d\n", x);
        free(env); return;
    }
}
```

Environments

```
struct f_env { int x; int tmp; };
cps void f(int *retval, int x)
{
    f_env *env;
    env = malloc(sizeof(f_env));
    env->x = x;
    env->tmp = env->x + 3;
    if (retval != NULL)
        *retval = env->tmp;
    free(env); return;
}
```

```
struct g_env { int x; };
cps void g(void)
{
    struct g_env *env;
    f(&(g_env->x), 4);
    g_aux(env); return;

    cps void
    g_aux(struct g_env *env)
    {
        printf ("%d\n", env->x);
        free(env); return;
    }
}
```

Figure 7.8: Environments in eCPC (second step)

Primitive operations We first compare the time of individual CPC primitives. Table 7.1 shows the relative speed of CPC and eCPC, based on the experimental setup and benchmark results given in Section 6.1.2.

The slowest primitive operation in CPC is a CPS function call (around 100 times slower than a direct call), mostly because of the multiple layers of indirection introduced by continuations. This overhead is even larger in the case of eCPC: performing a CPS function call is 2 to 3 times slower than with CPC.

This difference of cost for CPS function calls probably has an impact on the other benchmarks, making them more difficult to interpret. Context switches are around 50 % slower on every architecture, which is surprisingly high since they involve almost no manipulation of environments. Thread creation varies a lot across architectures: more than 3 times slower on the Pentium M, but only 59 % slower on a MIPS embedded processor. Finally, condition variables are even more surprising: not much slower on x86 and x86-64, and even 9 % faster

Table 7.1: Relative speed of CPC and eCPC (primitives)

Architecture	CPS-call	switch	condvar	spawn
Core 2 Duo (x86-64)	2.45	1.67	1.13	2.18
Pentium M (x86)	2.35	1.75	1.08	3.12
MIPS-32 4KEc	2.92	1.43	0.91	1.59

Execution time of eCPC compared to CPC ($t_{\text{eCPC}}/t_{\text{CPC}}$); lower is better. The experimental setup is described in Section 6.1.2, including individual benchmark times in Table 6.2.

on MIPS. We could reproduce this last result consistently, but it remains unclear which combination of factors leads eCPC to outperform CPC on this particular benchmark only: we believe that the larger number of registers[2] helps to limit the number of memory accesses, but we were not able to quantify this effect precisely.

These benchmarks of CPC primitives show that the allocation of environments slows down eCPC in most cases, and confirms our intuition that avoiding boxing as much as possible is very important in CPC.

Tic-tac-toe generator Benchmarking individual CPC primitives unfortunately gives little information on the performance of a whole program, because their cost might be negligible compared to other operations. To get a better understanding of the performance behaviour of eCPC, we needed an additional test, a highly concurrent program with intensive memory operations. Boutier wrote a tic-tac-toe generator that generates every 3×3 grid of tic-tac-toe (including partially filled and invalid grids); this benchmark does not actually play tic-tac-toe, it is only a small program designed to use a lot of threads and memory accesses.

The generator explores the space of grids. It creates three threads at each step, each one receiving a copy of the current grid and filling the next cell with either **X**, **O**, or nothing. This creates $3^9 = 19\,683$ threads and as many copies of the grid. We implemented two variants of the code, to test different schemes of memory usage. The former in a manual scheme that allocates copies of the grids with `malloc` before creating the associated threads, and frees each of them in one of the "leaf" threads, once the grid is completed. The latter is an automatic scheme that declares the grids as local variables. The grids are in fact also allocated on the heap in that case, but this is done automatically by the translator: in CPC, they are encapsulated in the boxing pass (since they are arrays); in eCPC, they are stored in the environment.

To ensure that the grids are not deallocated prematurely in the automatic scheme, the threads are synchronised using the barriers from the CPC standard library (Section 2.3.1). The manual scheme is more optimised, hence faster that the automatic one, but in terms of comparing CPC and eCPC it turned out to make no difference.

We launched an increasing number of generator tasks simultaneously, each one generating the 19 683 grids and threads mentioned above; we ran up to 100 tasks, ie. almost 2 000 000 threads in total. The slowest benchmark (eCPC with barriers) took around 3 seconds to complete on an Intel Centrino 1,87 Ghz, downclocked to 800 MHz (Fig. 7.9).

[2] MIPS has 32 integer registers, compared to 8 registers on x86 and 16 on x86-64.

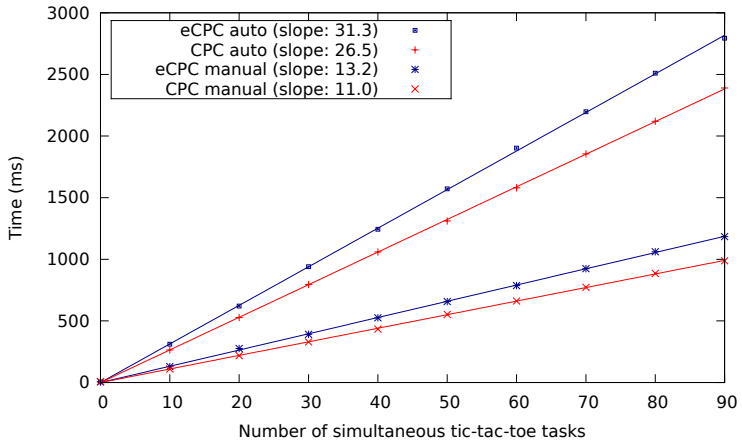


Figure 7.9: Tic-tac-toe benchmark

Finally, we compute the mean time per tic-tac-toe task. This ratio turns out to be independent of the number of simultaneous tasks: both CPC and eCPC scale linearly in this benchmark. We measured that eCPC is 20 % slower than CPC in the case of manual allocation (13.2 vs. 11.0 ms per task), and 18 % slower in the automatic case (31.3 vs. 26.5 ms per task). This benchmark confirms that environments add a significant overhead in programs performing a lot a memory accesses, although it is not as important as in benchmarks of CPC primitives.

Web servers We reproduced the web server benchmark described in Section 6.2. The results are shown in Fig. 7.10. Note that although we used the same server, the performance of CPC is slightly improved compared to Fig. 6.2b, most probably because the CPC translator and runtime had been completely rewritten in the meantime; a newer Linux kernel in the latest benchmarks might also play a role in this discrepancy.

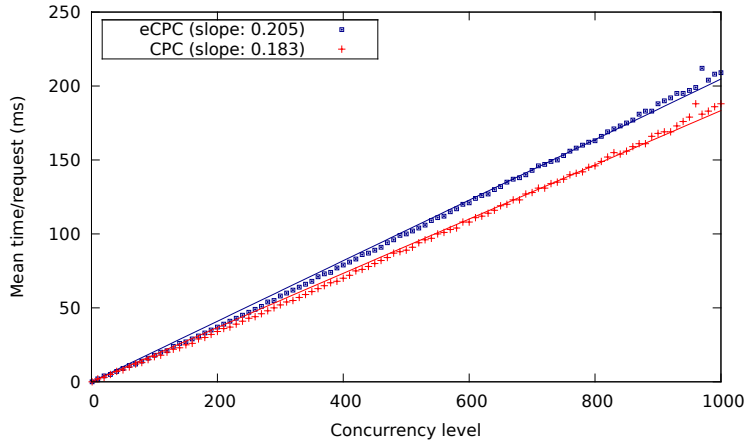


Figure 7.10: Web server benchmark

In this benchmark, the web server compiled with eCPC is 12 % slower than the server compiled with CPC. Even on programs that spend most of their time performing network

I/O, the overhead of environments remains measurable. Unfortunately, we did not have time during Boutier's internship to reproduce the benchmark of Hekate on embedded hardware using eCPC (Section 6.3), since it would have shown if this slowdown is still present in real-world programs.

Conclusions

> The events in our lives happen in a sequence in time, but in their significance to ourselves they find their own order [...]: the continuous thread of revelation.
>
> *One writer's beginnings*
> Eudora Welty

In this dissertation, we have shown that concurrent C programs written in threaded style can be translated automatically into efficient, equivalent event-driven programs through a series of source-to-source transformations, in particular lambda lifting and conversion into continuation-passing style.

Our study focused on building and experimenting with Continuation-Passing C, an extension of the C programming language that provides lightweight threads. The programmer decides at any point of the code whether a given thread is to be executed cooperatively or preemptively: CPC threads introduce a new abstraction layer, distinct from the notion of kernel threads. The CPC translator performs a series of source-to-source transformation, from CPC to C, to compile these threads either as cooperatively-scheduled event handlers, or as native threads scheduled preemptively by the operation system.

To convert threads into events, the CPC translator makes the control flow and data flow explicit through several source-to-source transformations: boxing, splitting, lambda lifting and CPS conversion. These transformations are commonly used to compile functional programs, but CPC is to the best of our knowledge the first example of combining them to compile imperative programs.

Correctness of translation passes

Working with an imperative language raises issues related to shared variables and, in the particular case of the C language, extruded variables—respectively variables that can be

modified by several functions called in the same dynamic chain and local variables whose address is retained through a pointer. Extruded variables are very difficult to control: they must not be copied by the transformation passes because it would invalidate the pointers that contain their addresses. Since lambda lifting and CPS conversion need to copy local variables, the CPC translator encapsulate every extruded variable into an environment during a preliminary boxing pass. As a side-effect, this also eliminates most shared variables; the only remaining shared variables are global and static variables, which are not affected by the CPC transformations.

We have shown that our transformation passes are correct. Despite removing extruded and shared variables, the correctness of lambda lifting and CPS conversion is not straightforward because these passes copy local variables which might be later modified, the outdated copy would then yield incorrect results. We have shown that in the specific case of CPC, once a variable is copied, its original is dead (never used anymore) and can be discarded. For lambda lifting, this stems from the fact that lifted functions are only called in tail-position; for CPS conversion, from the fact that it is possible to commute the evaluation of non-shared variables with function calls. Hence, even though local variables are copied from one event handler to another instead of being allocated on the call stack, copies are used linearly and this duplication does not affect the behaviour of the program.

To the best of our knowledge, the correctness of lambda lifting for functions called in tail-position in an imperative call-by-value language without shared variables is a novel result. There are few formal studies of lambda lifting in the literature and none of them focuses on imperative languages. Our result might also apply to the compilation of functional programs, for instance to reduce the number of boxed mutable variables in continuation-based compilers using lambda lifting rather than environments. Due to the big-step reduction rules used, the current proof is not quite complete—it does not show that non-termination is preserved by lambda lifting, but there is no reason to believe that this property does not hold.

The correctness of our CPS transform in an imperative language is probably a novel result too. However, it is intimately tied to the specific CPS conversion performed by CPC, which stores copies of free variables directly in continuations. The proof is an important part of this dissertation since it shows the theoretical correctness of CPC and helped discover several bugs in early implementations, but it is not obvious whether it will be useful outside of this context.

Evaluation

We evaluated the CPC language and translator on several levels. We defined metrics to evaluate the impact of translations and optimisations on generated code: for instance the number of lifted variables, or the number of inner CPS functions resulting from the splitting pass. We also performed a series of benchmarks to evaluate the amount of memory consumed by CPC threads, and the efficiency of CPC primitives. We compared CPC with other thread libraries, in particular by writing small web servers more representative of typical network programs than benchmarks of individual primitives. Finally, we wrote Hekate, a BitTorrent

seeder aimed at scaling to thousands of peers simultaneously, with students who had no prior knowledge of CPC. This allowed us to evaluate the usability of the language and translator.

Benchmarking To evaluate the speed and memory occupation of CPC programs, and compare it to other thread libraries, we wrote a number of benchmarking programs. We first ran a set of micro-benchmarks on several architectures, measuring the number of threads that can be created, and the speed of common thread management and synchronisation operations: thread creation, context-switching, and waiting for a condition variable. However, comparing CPC to other thread libraries is not straightforward, because CPC introduces a new kind of function call, CPS calls, which we benchmarked to be one to two orders of magnitude slower than native function calls. In the absence of information about the typical number of CPS calls in a CPC program, it is impossible to make direct conclusions about CPC efficiency.

Then, to evaluate the overhead associated with CPS calls in larger programs, we wrote a series of nearly-identical web servers differing only in the concurrency library that they used. This benchmark did not seek to measure directly the cost of CPS calls, but rather to determine whether they had a significant impact on the efficiency of CPC network server performing a large amount of I/O. It focused on one of the supposedly strong aspects of CPC, writing network servers, comparing it to state-of-the-art concurrency libraries and full-fledged, production servers.

These benchmarks have shown that CPC makes it possible to create at least an order of magnitude more threads than other thread libraries, while reaching similar or even better performance. Individual benchmarks of CPC primitives indicate that the CPC runtime is implemented efficiently, and larger programs have shown that the speed resulting from event-driven style surpasses the overhead introduced by CPC transformations. The CPC translator manages to keep the number of boxed variables and inner functions at a level low enough not to hinder efficiency.

Lambda lifting and environments We validated our choice of using lambda lifting by comparing CPC to eCPC, a variant of CPC using environments implemented together with Matthieu Boutier. Environments are more commonly used than lambda lifting to implement free variables, and they circumvent the theoretical issues of lambda lifting and CPS conversion in an imperative language. However, we believed that the fewer memory indirections of lambda lifting would yield better performance.

Even though the transformations performed by eCPC are simpler, easier to prove correct and involve less copying of variables, our benchmarks have shown that generated programs were slower in almost all cases. This seems to confirm our intuition that memory allocation and indirect memory accesses are more costly than copying variables which are generally in the cache, or even in registers, of the CPU. Systematic lambda lifting is too tedious for an event-loop programmer to perform by hand, but automatic translation made it both possible to experiment with and worthwhile performance-wise.

Usability The Hekate BitTorrent seeder has been a means to teach ourselves how to program with the unified lightweight threads provided by CPC, as well as a sample large program to evaluate the CPC translator. Our experience with Hekate shows that it is very convenient to program a network server using cheap cooperative threads, translated automatically to efficient event-driven code, while retaining the ability to switch on the fly to heavier preemptive threads when arises the need to perform blocking operations. It has validated the usability of CPC by showing that programmers unaware of its internals (including third-year undergraduate students) can use CPC to write an efficient network server.

We ran Hekate on torrents from the *World of Warcraft* game, to gather as many peers as possible and evaluate its behaviour facing a real-world workload. In this context, Hekate saturated the network link while keeping a low memory and CPU footprint. Performing stress-tests of Hekate on embedded hardware with very limited resources (a 266 MHz MIPS router with 32 MB of RAM) also confirmed that CPC threads are well-suited to writing network servers for constrained systems, where event-driven code would usually be preferred.

Understanding events

In the last chapter of this thesis, we described various styles of event-driven programming and studied how to obtain them from a common threaded description. We have shown that control flow can be handled either with multiple callbacks, generated by splitting, or with state machines, generated by defunctionalisation. We have also shown that local state can be saved either in large long-lived data structures, generated by environments, or in smaller short-lived ones, generated by lambda lifting and more efficient in most cases.

We have not yet implemented defunctionalisation to generate state machines with the CPC translator. A variant of the CPC translator with this feature would allow us to measure the relative efficiency of both techniques, and yield continuations closer to an actual call stack, thus easier to understand and debug. It might even turn out that the ideal choice between lambda lifting and environments, and between splitting and refunctionalisation, depends on the function being converted: the complexity of the control flow, the number of local variables, etc. Being able to mix several translation techniques would potentially allow the compiler to use the most effective one in each case, although it might be difficult to decide statically which technique to use.

Further work

The current implementation of CPC could probably be made more efficient. In particular, the implementation of lambda lifting could be further improved by the use of liveness analysis to avoid lifting uninitialised or dead variables. This is a very practical concern when programming with CPC, because GCC issues warnings about the use of uninitialised variables when they are copied into continuations. These spurious warnings tend to hide relevant warnings about programing mistakes.

Our proofs of correctness could be completed, mechanised and applied to other programming languages. Having a complete proof of correctness of the CPC translation passes would further enhance the confidence in the generated code. We assumed that boxing and splitting are "obviously correct" because they are made of a series of very elementary transformations, but formalising this on paper is difficult since it is not clear which fragment of the C language and reduction rules to consider. It would be a huge step forward to rewrite CPC as part of a certified C compiler, to prove their correctness alongside the implementation. However, having very little experience with computer-assisted proofs, we have no idea how (in)tractable the more difficult proofs of correctness of lambda lifting and CPS conversion would become in such a formal framework.

As mentioned above, the correctness of lambda lifting could probably be used in functional languages to reduce the number of boxed mutable variables; benchmarking such an implementation would provide one more data-point to validate our results with eCPC on the efficiency of lambda lifting as compared to environments.

The next step would be to study in more depth the relationship between splitting, state machines, defunctionalisation and refunctionalisation. Given a more complete understanding of the transformations between threaded and event-driven styles, one might not only translate threads into events, but also reconstruct a linear control flow from event-driven code. Ad-hoc debugging tools exist to reconstruct the call-graph of callback-based systems, but they require the programmer to use some specific event-loop library. A more generic tool, recognising and reversing most common event-driven patterns, would be invaluable to help to debug, maintain, and possibly refactor legacy pieces of software.

The transformation techniques used in the CPC translator are mostly independent of the runtime system that we have implemented. With little or no change to the translator, it should be possible to enhance the current runtime, for instance by adding support for multiple event loops and other primitives to use multiple processors. It should even be possible to use a different paradigm, for instance providing preemptive coroutines communicating over synchronous channels instead of cooperative threads.

Finally, the current version of CPC is focused on system programming for Unix. It uses the Unix model of non-blocking synchronous I/O which is sometimes limiting, for instance for disk I/O (Section 2.1.4). It would be interesting to port CPC to languages and systems which provide asynchronous, inherently event-driven interfaces, such as Javascript or the dominant operating system Windows.

Full contents

List of Figures

List of Tables

List of Definitions

List of Lemmas

List of Theorems

Index

Bibliography

[Abe+98] Harold Abelson, R. Kent Dybvig, Christopher T. Haynes, Guillermo Juan Rozas, Norman I. Adams IV, Daniel P. Friedman, Eugene E. Kohlbecker, Guy L. Steele Jr., David H. Bartley, Robert H. Halstead Jr., Don Oxley, Gerald J. Sussman, Gary Brooks, Chris Hanson, Kent M. Pitman, and Mitchell Wand. "Revised^5 report on the algorithmic language scheme". In: *Higher-Order and Symbolic Computation* 11 (1 1998), pp. 7–105. DOI: 10.1023/A:1010051815785 (cit. on p. 21).

[AC09] Pejman Attar and Yoann Canal. "Réalisation d'un seeder BitTorrent en CPC". Undergraduate thesis. Laboratoire PPS, Université Paris Diderot, June 2009. URL: http://www.pps.univ-paris-diderot.fr/~jch/software/hekate/hekate-attar-canal.pdf (cit. on p. 49).

[Ady+02] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. "Cooperative task management without manual stack management". In: *USENIX Annual Technical Conference, General Track*. Ed. by Carla Schlatter Ellis. USENIX, 2002, pp. 289–302. URL: http://www.usenix.org/publications/library/proceedings/usenix02/adyahowell.html (cit. on pp. 15, 36, 44).

[Aho+88] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison Wesley, 1988. ISBN: 0321428900 (cit. on p. 30).

[Apa08] Apache Software Foundation. *Apache HTTP server version 2.2 documentation*. Version 2.2.9. June 2008. URL: http://httpd.apache.org/docs/2.2/ (cit. on p. 145).

[App87] Andrew W. Appel. "Garbage collection can be faster than stack allocation". In: *Information Processing Letters* 25.4 (June 1987), pp. 275–279. DOI: 10.1016/0020-0190(87)90175-X.

[App92] Andrew W. Appel. *Compiling with continuations*. Cambridge University Press, 1992. ISBN: 0521416957 (cit. on p. 46).

[Bak95] Henry G. Baker. "CONS should not CONS its arguments, part II: Cheney on the M.T.A." In: *SIGPLAN Notices* 30.9 (1995), pp. 17–20. DOI: 10.1145/214448.214454 (cit. on p. 57).

[BC00] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel. from I/O ports to process management*. O'Reilly Media, Oct. 2000. ISBN: 0596000022 (cit. on p. 30).

[BCB03] J. Robert von Behren, Jeremy Condit, and Eric A. Brewer. "Why events are a bad idea (for high-concurrency servers)". In: *HotOS*. Ed. by Michael B. Jones. USENIX, 2003, pp. 19–24 (cit. on p. 44).

[BD99] Gaurav Banga and Peter Druschel. "Measuring the capacity of a web server under realistic loads". In: *World Wide Web* 2.1-2 (1999), pp. 69–83. DOI: 10.1023/A:1019292504731 (cit. on p. 144).

[Beh+03] J. Robert von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric A. Brewer. "Capriccio: scalable threads for internet services". In: *SOSP*. Ed. by Michael L. Scott and Larry L. Peterson. ACM, 2003, pp. 268–281. DOI: 10.1145/945445.945471 (cit. on pp. 18, 31, 44).

[Bel73] James R. Bell. "Threaded code". In: *Commun. ACM* 16.6 (1973), pp. 370–372. DOI: 10.1145/362248.362270.

[Bir+87] Andrew Birrell, John V. Guttag, James J. Horning, and Roy Levin. "Synchronization primitives for a multiprocessor: a formal specification". In: *SOSP*. 1987, pp. 94–102. DOI: 10.1145/41457.37509 (cit. on p. 62).

[BJ66] Corrado Böhm and Giuseppe Jacopini. "Flow diagrams, Turing machines and languages with only two formation rules". In: *Commun. ACM* 9.5 (May 1966), pp. 366–371. DOI: 10.1145/355592.365646 (cit. on p. 47).

[Bou06] Frédéric Boussinot. "FairThreads: mixing cooperative and preemptive threads in C". In: *Concurrency and Computation: Practice and Experience* 18.5 (2006), pp. 445–469. DOI: 10.1002/cpe.919 (cit. on pp. 44, 52).

[Bou11] Matthieu Boutier. "Sauvegarde de variables locales en CPC". Undergraduate thesis. Laboratoire PPS, Université Paris Diderot, June 2011. URL: http://www.pps.univ-paris-diderot.fr/~jch/research/rapport-m1-boutier-2011.pdf (cit. on pp. 8, 149, 156).

[BW73] Daniel G. Bobrow and Ben Wegbreit. "A model and stack implementation of multiple environments". In: *Commun. ACM* 16.10 (1973), pp. 591–603. DOI: 10.1145/362375.362379 (cit. on p. 31).

[BWD96] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. "Representing control in the presence of one-shot continuations". In: *PLDI*. Ed. by Charles N. Fischer. ACM, 1996, pp. 99–107. DOI: 10.1145/231379.231395 (cit. on p. 71).

[CFW85] William Clinger, Daniel P. Friedman, and Mitchell Wand. "A scheme for a higher-level semantic algebra". In: *Algebraic Methods in Semantics*. Ed. by John Reynolds and Maurice Nivat. Cambridge University Press, 1985, pp. 237–250 (cit. on p. 45).

[Che70] Chris J. Cheney. "A nonrecursive list compacting algorithm". In: *Commun. ACM* 13.11 (1970), pp. 677–678. DOI: 10.1145/362790.362798 (cit. on p. 57).

[Chr05] Juliusz Chroboczek. *Continuation-passing for C: a space-efficient implementation of concurrency*. Tech. rep. PPS, Université Paris 7, 2005. OAI: hal.archives-ouvertes.fr:hal-00135160 (cit. on p. 49).

[Chr08] Juliusz Chroboczek. *The Polipo manual*. Version 1.0.4. Jan. 2008. URL: http://www.pps.univ-paris-diderot.fr/~jch/software/polipo/manual/ (cit. on pp. 37, 52, 145).

[CK05] Ryan Cunningham and Eddie Kohler. "Making events less slippery with eel". In: *HotOS*. USENIX Association, 2005. URL: http://www.usenix.org/events/hotos05/final_papers/full_papers/cunningham/cunningham.pdf (cit. on p. 44).

[CK12] Juliusz Chroboczek and Gabriel Kerneis. *The CPC manual*. Preliminary edition. Jan. 2012. URL: http://www.pps.univ-paris-diderot.fr/~kerneis/software/files/cpc-manual.pdf (cit. on p. 145).

[Cla99] Koen Claessen. "A poor man's concurrency monad". In: *Journal of Functional Programming* 9.3 (1999), pp. 313–323. DOI: 10.1017/S0956796899003342 (cit. on pp. 21, 46).

[Cli98] William D. Clinger. "Proper tail recursion and space efficiency". In: *PLDI*. Ed. by Jack W. Davidson, Keith D. Cooper, and A. Michael Berman. ACM, 1998, pp. 174–185. DOI: 10.1145/277650.277719 (cit. on pp. 46, 92).

[Dab+02] Frank Dabek, Nickolai Zeldovich, M. Frans Kaashoek, David Mazières, and Robert Morris. "Event-driven programming for robust software". In: *ACM SIGOPS European Workshop*. Ed. by Gilles Muller and Eric Jul. ACM, 2002, pp. 186–189. DOI: 10.1145/1133373.1133410 (cit. on p. 44).

[Dan94] Olivier Danvy. "Back to direct style". In: *Science of Computer Programming* 22.3 (1994). A preliminary version was presented at the Fourth European Symposium on Programming (ESOP 1992), pp. 183–195. DOI: 10.1016/0167-6423(94)00003-4 (cit. on p. 71).

[Dei90] Harvey M. Deitel. *An introduction to operating systems*. 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990. ISBN: 0201180383 (cit. on p. 30).

[DH89] R. Kent Dybvig and Robert Hieb. "Engines from continuations". In: *Comput. Lang.* 14.2 (1989), pp. 109–123. DOI: 10.1016/0096-0551(89)90018-0 (cit. on pp. 21, 45).

[DL92] Olivier Danvy and Julia L. Lawall. "Back to direct style II: first-class continuations". In: *Proceedings of the 1992 ACM conference on LISP and functional programming*. LFP '92. New York, NY, USA: ACM, 1992, pp. 299–310. DOI: 10.1145/141471.141564 (cit. on p. 71).

[DM05] Ulrich Drepper and Ingo Molnar. "The Native POSIX Thread Library for Linux". Feb. 2005. URL: http://people.redhat.com/drepper/nptl-design.pdf (cit. on p. 139).

[DN01] Olivier Danvy and Lasse R. Nielsen. "Defunctionalization at work". In: *PPDP*. ACM, 2001, pp. 162–174. DOI: 10.1145/773184.773202 (cit. on pp. 150, 153).

[DS04] Olivier Danvy and Ulrik Pagh Schultz. "Lambda-lifting in quadratic time". In: *Journal of Functional and Logic Programming* 2004 (2004). DOI: 10.1007/3-540-45788-7_8 (cit. on pp. 46, 79, 91).

[DSZ09] Olivier Danvy, Chung-chieh Shan, and Ian Zerny. "J is for JavaScript: a direct-style correspondence between Algol-like languages and JavaScript using first-class continuations". In: *DSL*. Ed. by Walid Mohamed Taha. Vol. 5658. Lecture Notes in Computer Science. Springer, 2009, pp. 1–19. DOI: 10.1007/978-3-642-03034-5_1 (cit. on p. 45).

[Duf83] Tom Duff. *Duff's device*. Electronic mail to R. Gomes, D. M. Ritchie and R. Pike. Nov. 1983. URL: http://www.lysator.liu.se/c/duffs-device.html (cit. on pp. 44, 153).

[Dun+06] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. "Protothreads: simplifying event-driven programming of memory-constrained embedded systems". In: *SenSys*. Ed. by Andrew T. Campbell, Philippe Bonnet, and John S. Heidemann. ACM, 2006, pp. 29–42. DOI: 10.1145/1182807.1182811 (cit. on pp. 43, 44).

[Ecm09] Ecma International. *ECMA-262 "ECMAScript"*. Dec. 2009 (cit. on p. 32).

[Eng00] Ralf S. Engelschall. "Portable multithreading-the signal stack trick for user-space thread creation". In: *USENIX Annual Technical Conference, General Track*. USENIX, 2000, pp. 239–250. URL: http://www.usenix.org/publications/library/proceedings/usenix2000/general/engelschall.html (cit. on p. 139).

[Eng06] Ralf S. Engelschall. *The GNU Portable Threads manual*. Version 2.0.7. June 2006. URL: http://www.gnu.org/software/pth/pth-manual.html (cit. on pp. 139, 145).

[Fel+88] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. "Abstract continuations: a mathematical semantics for handling full jumps". In: *LISP and Functional Programming*. 1988, pp. 52–62. DOI: 10.1145/62678.62684 (cit. on p. 70).

[FH03] Adam Fischbach and John Hannan. "Specification and correctness of lambda lifting". In: *Journal of Functional Programming* 13.3 (2003), pp. 509–543. DOI: 10.1017/S0956796802004604 (cit. on p. 46).

[FMM07] Jeffrey Fischer, Rupak Majumdar, and Todd D. Millstein. "Tasks: language support for event-driven programming". In: *PEPM*. Ed. by G. Ramalingam and Eelco Visser. ACM, 2007, pp. 134–143. DOI: 10.1145/1244381.1244403 (cit. on pp. 43, 45).

[GFW99] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. "Trampolined style". In: *ICFP*. Ed. by Didier Rémy and Peter Lee. ACM, 1999, pp. 18–27. DOI: 10.1145/317636.317779 (cit. on p. 73).

[GRA89] James Gosling, David S. H. Rosenthal, and Michele J. Arden. *The NeWS book: an introduction to the network/extensible window system (SUN Technical Reference Library)*. New York, NY, USA: Springer-Verlag, 1989. ISBN: 0387969152.

[Gu+07] Boncheol Gu, Yongtae Kim, Junyoung Heo, and Yookun Cho. "Shared-stack cooperative threads". In: *SAC*. Ed. by Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo. ACM, 2007, pp. 1181–1186. DOI: 10.1145/1244002.1244258 (cit. on pp. 18, 31, 140).

[Har+11] Tim Harris, Martín Abadi, Rebecca Isaacs, and Ross McIlroy. "AC: composable asynchronous IO for native languages". In: *OOPSLA*. Ed. by Cristina Videira Lopes and Kathleen Fisher. ACM, 2011, pp. 903–920. DOI: 10.1145/2048066.2048134 (cit. on pp. 45, 62).

[Har80] David Harel. "On folk theorems". In: *Commun. ACM* 23.7 (July 1980), pp. 379–389. DOI: 10.1145/358886.358892 (cit. on p. 47).

[HFW86] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. "Obtaining coroutines with continuations". In: *Computer Languages* 11.3/4 (1986), pp. 143–153. DOI: 10.1016/0096-0551(86)90007-X (cit. on pp. 21, 45).

[HO09] Philipp Haller and Martin Odersky. "Scala actors: unifying thread-based and event-based programming". In: *Theor. Comput. Sci.* 410.2-3 (2009), pp. 202–220. DOI: 10.1016/j.tcs.2008.09.019 (cit. on p. 45).

[Hoa74] C. A. R. Hoare. "Monitors: an operating system structuring concept". In: *Commun. ACM* 17.10 (1974), pp. 549–557. DOI: 10.1145/355620.361161 (cit. on pp. 51, 61).

[Hug00] John Hughes. "Generalising monads to arrows". In: *Sci. Comput. Program.* 37.1-3 (2000), pp. 67–111. DOI: 10.1016/S0167-6423(99)00023-4 (cit. on p. 45).

[Int99] International Organization for Standardization. *ISO/IEC 9899:1999 "Programming Languages – C"*. Dec. 1999 (cit. on pp. 56, 113).

[Joh85] Thomas Johnsson. "Lambda lifting: transforming programs to recursive equations". In: *FPCA*. 1985, pp. 190–203. DOI: 10.1007/3-540-15975-4_37 (cit. on pp. 46, 79, 91).

[JR81] Stephen C. Johnson and Dennis M. Ritchie. *The C language calling sequence*. Computing Science Technical Report 102. Bell Laboratories, Sept. 1981. URL: http://cm.bell-labs.com/cm/cs/who/dmr/clcs.html (cit. on p. 37).

[KBD98] Sanjeev Kumar, Carl Bruggeman, and R. Kent Dybvig. "Threads yield continuations". In: *Lisp and Symbolic Computation* 10.3 (1998), pp. 223–236. DOI: 10.1023/A:1007782300874 (cit. on p. 21).

[KC09] Gabriel Kerneis and Juliusz Chroboczek. *Are events fast?* Tech. rep. PPS, Université Paris 7, Jan. 2009. OAI: hal.archives-ouvertes.fr:hal-00434374.

[KC11a] Gabriel Kerneis and Juliusz Chroboczek. "Continuation-Passing C, compiling threads to events through continuations". In: *Higher-Order and Symbolic Computation* 24 (3 2011), pp. 239–279. DOI: 10.1007/s10990-012-9084-5.

[KC11b] Gabriel Kerneis and Juliusz Chroboczek. "CPC: programming with a massive number of lightweight threads". In: *Proceedings of the Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software*. 2011, pp. 30–34. OAI: hal.archives-ouvertes.fr:hal-00563369 (cit. on p. 49).

[KC12] Gabriel Kerneis and Juliusz Chroboczek. "Lambda-lifting and CPS conversion in an imperative language". In: *CoRR* abs/1202.3247 (2012). arXiv:1202.3247v1 [cs.PL].

[Key10] Andy Key. *Weave: translated threaded source (with annotations) to fibers with context passing*. As used within RAID-1 code in IBM SSA RAID adapters (ca. 1995–2000). Personal communication. Nov. 2010 (cit. on pp. 32, 43, 46).

[Kho+09] Yit Phang Khoo, Michael Hicks, Jeffrey S. Foster, and Vibha Sazawal. "Directing JavaScript with arrows". In: *DLS*. Ed. by James Noble. ACM, 2009, pp. 49–58. DOI: 10.1145/1640134.1640143 (cit. on p. 45).

[KKK07] Maxwell N. Krohn, Eddie Kohler, and M. Frans Kaashoek. "Events can make sense". In: *USENIX Annual Technical Conference*. USENIX, 2007, pp. 87–100. URL: http://www.usenix.org/events/usenix07/tech/krohn.html (cit. on pp. 43, 44, 46, 154).

[Kne08] Jan Kneschke. *The Lighttpd manual*. Version 1.4.19. Mar. 2008. URL: http://trac.lighttpd.net/trac/wiki/Docs/ (cit. on p. 145).

[Knu74] Donald E. Knuth. "Structured programming with go to statements". In: *ACM Computing Surveys* 6.4 (Dec. 1974), pp. 261–301. DOI: 10.1145/356635.356640 (cit. on p. 7).

[KY01] Jung-taek Kim and Kwangkeun Yi. "Interconnecting between CPS terms and non-CPS terms". In: *Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW'01)*. Ed. by Amr Sabry. Published as Technical Report TR545, Computer Science Program, Indiana University. Jan. 2001, pp. 7–16. URL: http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR545 (cit. on p. 69).

[Lam93] Butler Lampson. *Principles for Computer System Design*. Turing Award lecture. 1993. URL: http://research.microsoft.com/en-us/um/people/blampson/slides/turinglectureabstract.htm (cit. on p. 154).

[Lan65] Peter J. Landin. "Correspondence between ALGOL 60 and Church's lambda-notation: part I". In: *Commun. ACM* 8.2 (1965), pp. 89–101. DOI: 10.1145/363744.363749 (cit. on p. 46).

[LCB11] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. "Dthreads: efficient deterministic multithreading". In: *SOSP*. Ed. by Ted Wobber and Peter Druschel. ACM, 2011, pp. 327–336. DOI: 10.1145/2043556.2043587 (cit. on p. 31).

[Leh10] Marc Lehmann. *The libev manual*. 2010. URL: http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod.

[Ler+10] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective-Caml system*. June 2010. URL: http://caml.inria.fr/ (cit. on p. 67).

[LN79] Hugh C. Lauer and Roger M. Needham. "On the duality of operating system structures". In: *Operating Systems Review* 13.2 (1979), pp. 3–19. DOI: 10.1145/850657.850658 (cit. on p. 44).

[LZ07] Peng Li and Steve Zdancewic. "Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives". In: *PLDI*. Ed. by Jeanne Ferrante and Kathryn S. McKinley. ACM, 2007, pp. 189–199. DOI: 10.1145/1250734.1250756 (cit. on p. 46).

[Mas11] Massachusetts Institute of Technology. *MIT/GNU Scheme 9.1*. 2011. URL: http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-user/ (cit. on p. 36).

[Mey+09] Leo A. Meyerovich, Arjun Guha, Jacob P. Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. "Flapjax: a programming language for Ajax applications". In: *OOPSLA*. Ed. by Shail Arora and Gary T. Leavens. ACM, 2009, pp. 1–20. DOI: 10.1145/1640089.1640091 (cit. on p. 47).

[MR94] James S. Miller and Guillermo J. Rozas. *Garbage Collection is Fast, but a Stack is Faster*. AI Memo 1462. Cambridge, MA, USA: Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Mar. 1994. HDL: 1721.1/6622 (cit. on p. 142).

[MS07] Marco T. Morazán and Ulrik Pagh Schultz. "Optimal lambda lifting in quadratic time". In: *IFL*. Ed. by Olaf Chitil, Zoltán Horváth, and Viktória Zsók. Vol. 5083. Lecture Notes in Computer Science. Springer, 2007, pp. 37–56. DOI: 10.1007/978-3-540-85373-2_3 (cit. on p. 91).

[MY98] Shivakant Mishra and Rongguang Yang. "Thread-based vs event-based implementation of a group communication service". In: *IPPS/SPDP*. 1998, pp. 398–402. DOI: 10.1109/IPPS.1998.669947 (cit. on p. 44).

[Mye+07] Daniel S. Myers, Jennifer N. Carlisle, James A. Cowling, and Barbara Liskov. "MapJAX: data structure abstractions for asynchronous web applications". In: *USENIX Annual Technical Conference*. USENIX, 2007, pp. 101–114. URL: http://www.usenix.org/events/usenix07/tech/myers.html (cit. on p. 47).

[Nec+02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. "CIL: intermediate language and tools for analysis and transformation of C programs". In: *CC*. Ed. by R. Nigel Horspool. Vol. 2304. Lecture Notes in Computer Science. Springer, 2002, pp. 213–228. DOI: 10.1007/3-540-45937-5_16 (cit. on p. 67).

[Ode+04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. *An Overview of the Scala Programming Language*. Tech. rep. IC/2004/64. École Polytechnique Fédérale de Lausanne, 2004. URL: http://infoscience.epfl.ch/record/52656 (cit. on p. 21).

[Ora12] Oracle. *Java Platform Standard Edition, class Thread, method interrupt*. 7th ed. 2012. URL: http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupt() (cit. on p. 62).

[Ous96] John Ousterhout. "Why threads are a bad idea (for most purposes)". In: *USENIX 1996 Technical Conference*. Invited talk. June 1996. URL: http://www.stanford.edu/~ouster/cgi-bin/papers/threads.pdf (cit. on p. 44).

[PDZ99] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. "Flash: an efficient and portable web server". In: *USENIX Annual Technical Conference, General Track*. USENIX, 1999, pp. 199–212. URL: http://www.usenix.org/events/usenix99/full_papers/pai/pai.pdf (cit. on pp. 31, 44, 54).

[Pik90] Rob Pike. "The implementation of Newsqueak". In: *Softw., Pract. Exper.* 20.7 (1990), pp. 649–659. DOI: 10.1002/spe.4380200703.

[Plo75] Gordon D. Plotkin. "Call-by-name, call-by-value and the lambda-calculus". In: *Theoretical Computer Science* 1.2 (1975), pp. 125–159. DOI: 10.1016/0304-3975(75)90017-1 (cit. on pp. 22, 46, 68, 69, 132).

[Pos03] Jef Poskanzer. *The thttpd man page*. Version 2.25b. Dec. 2003. URL: http://www.acme.com/software/thttpd/thttpd_man.html (cit. on p. 145).

[Rep93] John H. Reppy. "Concurrent ML: design, application and semantics". In: *Functional Programming, Concurrency, Simulation and Automated Reasoning*. Ed. by Peter E. Lauer. Vol. 693. Lecture Notes in Computer Science. Springer, 1993, pp. 165–198. DOI: 10.1007/3-540-56883-2_10 (cit. on p. 45).

[Rey72] John C. Reynolds. "Definitional interpreters for higher-order programming languages". In: 2.4 (Aug. 1972), pp. 717–740. DOI: 10.1023/A:1010027404223. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [Rey98] (cit. on pp. 47, 150).

[Rey93] John C. Reynolds. "The discoveries of continuations". In: *Lisp and Symbolic Computation* 6.3-4 (1993), pp. 233–248. DOI: 10.1007/BF01019459 (cit. on p. 46).

[Rey98] John C. Reynolds. "Definitional interpreters revisited". In: *Higher-Order and Symbolic Computation* 11.4 (1998), pp. 355–361. DOI: 10.1023/A:1010075320153 (cit. on p. 193).

[RMO09] Tiark Rompf, Ingo Maier, and Martin Odersky. "Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform". In: *ICFP*. Ed. by Graham Hutton and Andrew P. Tolmach. ACM, 2009, pp. 317–328. DOI: 10.1145/1596550.1596596 (cit. on pp. 21, 46).

[RN11] Jeffrey Richter and Christophe Nasarre. *Windows via C/C++*. Microsoft Press, 2011. ISBN: 0735624240 (cit. on p. 32).

[SA06] Gene Shekhtman and Mike Abbott. *The State Threads Library Documentation*. Version 1.7. May 2006. URL: http://state-threads.sourceforge.net/docs/ (cit. on pp. 139, 145).

[Sch95] Enno Scholz. *A Concurrency Monad Based on Constructor Primitives. or, Being First-Class is not Enough*. Tech. rep. B 95-01. Berlin, Germany: Fachbereich Mathematik und Informatik, Freie Universität Berlin, Jan. 1995. URL: http://www.inf.fu-berlin.de/inst/pubs/tr-b-95-01.abstract.html (cit. on p. 46).

[SJS76] Guy L. Steele Jr. and Gerald J. Sussman. *Lambda, the Ultimate Imperative*. AI Memo 353. Cambridge, MA, USA: Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Mar. 1976. HDL: 1721.1/5790 (cit. on pp. 46, 78).

[SM08] Sriram Srinivasan and Alan Mycroft. "Kilim: isolation-typed actors for Java". In: *ECOOP*. Ed. by Jan Vitek. Vol. 5142. Lecture Notes in Computer Science. Springer, 2008, pp. 104–128. DOI: 10.1007/978-3-540-70592-5_6 (cit. on p. 45).

[SPL11] Don Syme, Tomas Petricek, and Dmitry Lomov. "The F# asynchronous programming model". In: *PADL*. Ed. by Ricardo Rocha and John Launchbury. Vol. 6539. Lecture Notes in Computer Science. Springer, 2011, pp. 175–189. DOI: 10.1007/978-3-642-18378-2_15 (cit. on p. 46).

[Ste78] Guy L. Steele Jr. "RABBIT: A Compiler for SCHEME". Technical report AI-TR-474. MA thesis. Cambridge, MA, USA: Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 1978. HDL: 1721.1/6913 (cit. on p. 46).

[SW74] Christopher Strachey and Christopher P. Wadsworth. *Continuations: A Mathematical Semantics for Handling Full Jumps*. Technical Monograph PRG-11. Oxford, England: Oxford University Computing Laboratory, Programming Research Group, 1974. Reprinted in Higher-Order and Symbolic Computation 13(1/2):135–152, 2000, with a foreword [Wad00] (cit. on pp. 22, 46, 68).

[Tat00] Simon Tatham. *Coroutines in C*. 2000. URL: http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html (cit. on p. 44).

[Tay11] Ian Lance Taylor. *Split stacks in GCC*. 2011. URL: http://gcc.gnu.org/wiki/SplitStacks (cit. on p. 31).

[Thi99] Hayo Thielecke. "Continuations, functions and jumps". In: *SIGACT News* 30.2 (1999), pp. 33–42. DOI: 10.1145/568547.568561 (cit. on p. 46).

[Tis00] Christian Tismer. "Continuations and stackless Python". In: *Proceedings of the 8th International Python Conference*. Python Software Foundation, 2000. URL: http://python.org/workshops/2000-01/proceedings/papers/tismers/spcpaper.pdf (cit. on p. 45).

[Tor02] Linus Torvald. *O_DIRECT performance impact on 2.4.18*. Electronic message to the linux-kernel mailing list. Message-ID: <fa.m6umeav.15008gs@ifi.uio.no>. May 2002. URL: http://yarchive.net/comp/linux/o_direct.html (cit. on p. 55).

[Vou08] Jérôme Vouillon. "Lwt: a cooperative thread library". In: *ML*. Ed. by Eijiro Sumii. ACM, 2008, pp. 3–12. DOI: 10.1145/1411304.1411307 (cit. on pp. 21, 46).

[Wad00] Christopher P. Wadsworth. "Continuations revisited". In: *Higher-Order and Symbolic Computation* 13.1/2 (2000), pp. 131–133 (cit. on p. 194).

[Wan80] Mitchell Wand. "Continuation-based multiprocessing". In: (1980), pp. 19–28. DOI: 10.1023/A:1010093700911. Reprinted in Higher-Order and Symbolic Computation 12(3):285–299, 1999, with a foreword [Wan99] (cit. on p. 45).

[Wan99] Mitchell Wand. "Continuation-based multiprocessing revisited". In: *Higher-Order and Symbolic Computation* 12.3 (1999), p. 283. DOI: 10.1023/A:1010049917750 (cit. on p. 194).

[WCB01] Matt Welsh, David E. Culler, and Eric A. Brewer. "SEDA: an architecture for well-conditioned, scalable internet services". In: *SOSP*. ACM, 2001, pp. 230–243. DOI: 10.1145/502034.502057 (cit. on pp. 31, 44, 54, 59).

[Wij66] Adriaan van Wijngaarden. "Recursive definition of syntax and semantics". In: *Formal Language Description Languages for Computer Programming*. Amsterdam, Netherlands: North-Holland Publishing Company, 1966, pp. 13–24 (cit. on pp. 46, 78).

The original source for this PhD thesis was created with LaTeX, built on top of Donald Knuth's TeX typesetting system by Leslie Lamport. It was output as PDF by Hàn Thế Thành's pdfTeX. The body text is typeset in 11pt with Minion Pro, designed by Robert Slimbach, and figure captions in 10pt with Myriad Pro, by Carol Twombly and Robert Slimbach. Code listings are typeset in 9pt with Inconsolata, designed by Raph Levien. Micro-typographic details were fine-tuned with Robert Schlicht's microtype package. The layout is a golden-section typeblock on an ISO page, proposed by Robert Bringhurst in *The Elements of Typographic Style*, and implemented by Peter Wilson in the memoir class. The diagrams were created with R and gnuplot, and the tables were enhanced with the booktabs package by Simon Fear.

La plupart des programmes informatiques sont concurrents : ils doivent effectuer plusieurs tâches en même temps. Les *threads* et les événements sont deux techniques usuelles d'implémentation de la concurrence. Les événements sont généralement plus légers et efficaces que les *threads*, mais aussi plus difficiles à utiliser. De plus, ils sont souvent trop limités ; il est alors nécessaire d'écrire du code hybride, encore plus complexe, utilisant à la fois des *threads* ordonnancés préemptivement et des événements ordonnancés coopérativement.

Nous montrons dans cette thèse que des programmes concurrents écrits dans un style à *threads* sont traduisibles automatiquement en programmes à événements équivalents et efficaces par une suite de transformations source-source prouvées.

Nous proposons d'abord Continuation-Passing C, une extension du langage C pour l'écriture de systèmes concurrents qui offre des *threads* très légers et unifiés (coopératifs et préemptifs). Les programmes CPC sont transformés par le traducteur CPC pour produire du code à événements séquentialisé efficace, utilisant des *threads* natifs pour les parties préemptives. Nous définissons et prouvons ensuite la correction de ces transformations, en particulier le lambda lifting et la conversion CPS, pour un langage impératif. Enfin, nous validons la conception et l'implémentation de CPC en le comparant à d'autres bibliothèques de *threads* et en exhibant notre *seeder* BitTorrent Hekate. Nous justifions aussi notre choix du lambda lifting en implémentant eCPC, une variante de CPC utilisant les environnements, et en comparant ses performances à celles de CPC.

Most computer programs are concurrent ones: they need to perform several tasks at the same time. Threads and events are two common techniques to implement concurrency. Events are generally more lightweight and efficient than threads, but also more difficult to use. Additionally, they are often not powerful enough; it is then necessary to write hybrid code, that uses both preemptively-scheduled threads and cooperatively-scheduled event handlers, which is even more complex.

In this dissertation, we show that concurrent programs written in threaded style can be translated automatically into efficient, equivalent event-driven programs through a series of proven source-to-source transformations.

We first propose Continuation-Passing C, an extension of the C programming language for writing concurrent systems that provides very lightweight, unified (cooperative and preemptive) threads. CPC programs are processed by the CPC translator to produce efficient sequentialized event-loop code, using native threads for the preemptive parts. We then define and prove the correctness of these transformations, in particular lambda lifting and CPS conversion, for an imperative language. Finally, we validate the design and implementation of CPC by comparing it to other thread librairies, and by exhibiting our Hekate BitTorrent seeder. We also justify the choice of lambda lifting by implementing eCPC, a variant of CPC using environments, and comparing its performances to CPC.